UNIVERSITY OF TWENTE.

Formal Methods and Tools

Master's Thesis

A comparison of state space reduction techniques in SCOOP

Ferry Olthuis, BSc

Supervisors:
Dr. M.I.A. Stoelinga
Dr. Ir. M. Timmer
Prof. Dr. J.C. van de Pol

ABSTRACT

In quantitative model checking we are concerned with verifying both quantitative and qualitative properties over models. Markov Automata are what we use to model these systems. In these transition systems nondeterminism, probabilistic behaviour and behaviour over time can all be modelled. SCOOP is a tool that uses several techniques, including confluence reduction and dead variable reduction, to reduce the state space of a Markov Automaton.

In this work, we present a research in which we compare the effect of the reduction techniques used by SCOOP in different case studies. This is relevant in two aspects. For potential users of SCOOP we provide concrete proof that SCOOP and its reduction techniques can be a solution to their problems. It also gives more insights in why SCOOP does not work in some situations and provide directions to where the implementation of SCOOP could be improved.

We modelled a set of case studies in MAPA (the process algebra used by SCOOP), in order to show the strengths and weaknesses in SCOOP. The first case study is a Gossip Protocol. This is a means by which computer systems can communicate in a way similar to how gossip spreads through a community. In the second game we model a trade market, originating from an electroinic strategy game called Gazillionaire. The third and last case study describes a means by which a group of robots comes to consensus of the best out of two options. They do so in a way similar to how ants seek the shortest path to a food source.

We show that there are huge differences in the effect of the reduction techniques on these 3 models. We show that confluence reduction has no impact at all on the gossip model and that it only slightly decreases the state space size of the other three models. We could come nowhere near the numbers shown in previous case studies. This is partly due to the density of relevant communication and rates in the model. We also show that if we extract the core from the model and show that if the basic sequence of actions forming gets large, this negatively impacts the effect of confluence reduction. For dead variable reduction we show that this only works for the gossip model (from our 3 models) because this is a phased type model.

ACKNOWLEDGEMENTS

First, I would like to thank my three supervisors, Marielle Stoelinga, Mark Timmer and Jaco van de Pol. All of you have been very helpful during this project and I am very grateful for that. I couldn't have done all this without your advice, useful comments and motivational support. In particular I would like to thank Mark for his amazing help, especially in the early phases of this project. Thanks for reviewing every chapter, section and word of this thesis and for always making time to help me, even when you were very busy with finishing your own project.

Second, I want to thank my fellow students from MTV: Daan, Paul, Harold, Freark, Ruud, Jeroen, Bjorn and Harold. Thanks for the pleasant working environment and the moments of fun we had in between the periods of hard work. I specifically want to thank Daan for the many projects we have worked on together over the last five years. I honestly don't think that I could have finished this study so fast without your help and motivation. We make a strong team!

Finally, I want to thank my family for always being there for me. You all have supported me and were there for me when I needed it. Thanks to you all!

CONTENTS

DOCUMENT STRUCTURE

This document is divided in four parts. Part I serves as an introduction of the performed research. In Chapter 1 we informally introduce some background of the research. Chapter 2 introduces the problem. This directly leads to the research goal and main questions to be answered.

In Part II we introduce the theoretical background we need for the research. Chapter 3 introduces the mathematical background and introduces Markov Automata and their behaviour. MAPA, the input language for SCOOP, will be described in Chapter 4. Then in Chapters 5 and 6 we will show how the major reduction techniques, confluence and dead variable reduction work for Markov Automata. The tool that we use for verifying some properties of our models, IMCA, will be introduced in Chapter 7. In Chapter 8 we show some related work. This includes several state space reduction techniques that are similar to the ones performed by SCOOP. We also show how Markov Automata can serve as a semantic model for other languages, and which case studies have already been done on the same topic.

Part III is concerned with the conducted case studies. We start with a global description of what a case study looks like and what process we use to execute them. This is done in Chapter 9. Then in Chapters 10-12 we describe the four different case studies. We give some background, show the design and provide the results. We directly describe interesting results for each case study, but extend a real analysis (combining the results of all case studies) to Part IV.

In Part IV we conclude our research. We analyse the results from the case studies in Chapter 13 and answer the research questions in Chapter 14. In this chapter we also provide directions for future work.

Part I.

Prologue

CHAPTER 1

INTRODUCTION

50 years ago there were hardly any computers in our society. However, even imagining a life without computers nowadays is close to impossible. Almost every little aspect of our society depends on computers and on their correct behaviour. Whenever we launch a rocket into space, no fault may be present in the controller. If there is a fault, the ship might explode or not reach its destination for example. If a computer system in a hospital fails, people may die. If the controller of a nuclear reactor contains an error, it may explode. Lots of such examples can be thought of in which it is essential that computer programs do not contain any errors.

Two of several major research directions that aim at getting your code correct are testing and model checking. In this research we are concerned with model checking. To be able to perform model checking we first need a model of the software. On this model we check whether certain requirements, specified in some sort of logic, hold and prove that the system behaves correctly under all circumstances. In traditional model checking we are only interested in functional aspects of the software, which basically means that we verify whether or not certain states are reachable from the initial state. Timing does not play any role here. Examples of functional aspects are that the system never deadlocks, that no two processes are editing some critical piece of data at the same time (critical section problem) or that an ATM gives you your money whenever you request it. Verifying such properties requires a model that describes everything that the system is able to do. Properties such as the time it takes to do some action or the probability that something happens are not taken into account. If we want to verify that functional requirements hold we do an exhaustive search of the state space and check whether the properties can be violated.

However, one might also be interested in quantitative aspects such as "the system is unreachable only 1% of the time" or "whenever I request money from an ATM it will respond within 3 seconds". To be able to verify these kinds of properties we need models that describe how the system behaves over time and what their probabilistic behaviour is. A model that incorporates these aspects is what we call a *probabilistic model.* We used Markov Automata during this research and these are introduced in Chapter 3. It is convenient to specify such a model in an abstract language, as a graphical representation can grow very large. MAPA [1], which is formally introduced in Chapter 4, is such a language. It allows for three key ingredients of a realistic model. *Nondeterminism* is a way to let the system behave differently in the same state. *Markovian transitions* describe the behaviour of the system over time and *probabilistic transitions* allow the system to behave

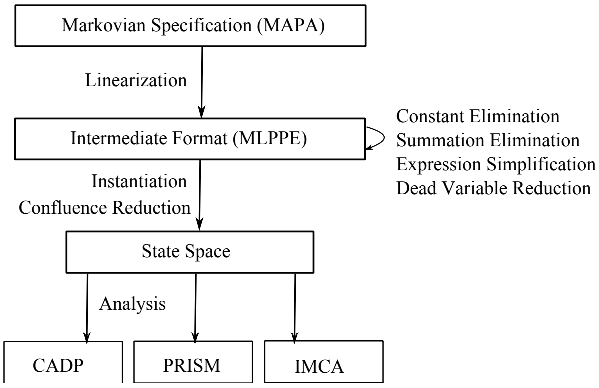


Figure 1.1.: Overview of SCOOP

in a certain way with a certain probability. With these three ingredients we can define a lot of real-world computer systems and measure both qualitative and quantitative properties.

Model checking suffers from a major problem called the state space explosion problem. Basically, this means that whenever the complexity of the model increases slightly, there is an exponential increase in the number of states. That is why many state space reduction techniques have been developed for both traditional models and for probabilistic models. SCOOP [2] is a tool that foresees in several state space reduction techniques. It has MAPA as its input language and transforms MAPA-specifications into a standard format, called a Markovian linear probabilistic process equation (MLPPE). This transformation, called linearisation, will be briefly described in Chapter 4. On an MLPPE, SCOOP applies several techniques which simplify the MLPPE specification. These techniques often decreases the state space generation time but do not change the size of the state space.

On the simplified MLPPE two major reduction techniques are applied. With *dead variable reduction*, which will be described in Chapter 6, we analyse under which circumstances the value of certain variables is irrelevant. We replace references to irrelevant variables with a constant value, often reducing but never increasing the state space. This technique is applied directly on the MAPA specification, so prior to state space instantiation. *Confluence reduction*, which will be described in Chapter 5 is a technique that reduces the state space during state space generation. The general idea behind confluence is that often it does not matter in which order certain transitions take place. Confluence detects whenever this is the case and gives one interleaving priority over the others. The result is a reduced state space that can be analysed using probabilistic model checking tools such as IMCA [3], CADP [4] and PRISM [5].

Figure 1.1, adapted from [2], shows a general overview of SCOOP.

Contributions The tool SCOOP and its reduction techniques were already developed prior to this research. The main contribution of this research is that we show under which circumstances the reduction techniques performed by SCOOP work well. This has been done using a couple of case studies in which real world problems are modelled using the MAPA language. We measure the impact of the reduction technique by comparing the original state space with the reduced state space. The case studies we have done are a railway safety system, a gossip protocol, a business game (Gazillionaire) and a consensus algorithm. In Part III we show all the details concerning the case studies. In Part IV we analyse interesting results, answer the research questions and provide directions for future research in this field.

CHAPTER 2

PROBLEM DESCRIPTION

In this chapter we analyse the research problem and formulate the goal of this thesis. We also formulate the main question that is answered during this project. The main question is divided into several sub-questions. For each question, we give a brief preview of what can be expected.

2.1. Research Goal

Much research has been going on in the field of model checking. Efficient algorithms have been developed and powerful tools applying these algorithms exist. For probabilistic model checking far less algorithms and tools have been developed. (Probabilistic) model checking suffers from several problems. The most famous one is the state space explosion problem. Assume we would have a model containing 1000 states and we extend this model with one more variable that can have 2 values. Potentially, the number of states doubles with only one extra variable. So with a few more variables this can quickly explode to infeasible amounts of states. Another problem that is more specific to probabilistic model checking is the restricted treatment of data, meaning not many tools are capable of handling large amounts of data. SCOOP is a tool that contributes to solving these problems. It comes with a process algebra (MAPA) to symbolically represent a system. MAPA treats data as a first class citizen. Also, nondeterminism, probabilistic choice and behaviour over time can be represented in MAPA. SCOOP uses powerful state space reduction techniques to contribute to reducing the state space explosion problem. The tool and its reduction techniques have been implemented and tested. Also, several small case studies have been conducted to show that the solutions work.

Large differences in reductions have been shown in the different case studies that have been performed prior to this research. In this research we show which aspects of case studies cause these differences and how we can predict whether SCOOP will work well for certain case studies. This both gives more evidence to potential users that SCOOP is a good solution for their problems and provide directions for future research.

2.2. Research Questions

We know that SCOOP is a tool that can help solving the state space explosion problem in some situations. We should ask ourselves why the impact of SCOOP's reduction techniques differs so much between these situations. This leads to our main question: In which application scenarios do the reduction techniques performed by SCOOP work best? We can subdivide this question in the following four sub-questions:

1. How do the reduction techniques used by SCOOP work?
2. What is the impact of the reduction techniques and how can we measure this?
3. What differences between reduction techniques exist and how can we explain these?
4. How can we improve SCOOP such that it works better in specific scenario's?

Question 1 has been answered in Chapters 3-6. This has been done by a literature study, as the reduction techniques used by SCOOP were already developed prior to writing this thesis. However, as the theory behind SCOOP is fundamental for the reader to understand the rest of this research, it will be described in detail. The theory consists of four major parts. We first describe Markov Automata and how we can analyse their behaviour. We continue with a description of the MAPA language and its linear format, the MLPPE. In the chapter about MAPA we also describe several simplification techniques that are used by SCOOP to make state space generation easier and quicker. In the chapters about Confluence and Dead Variable Reduction (Chapters 5 and 6), we provide the reader with the theory behind these techniques. We describe how we can detect and apply the reduction techniques directly on the MAPA-specification.

The answer to question 2 is basically a description of the approach we want to take in order to answer question 3. This is described in Chapter 9. First, we need something to measure, so we identify gossip protocols, a trade market and a consensus algorithm as interesting case studies and model these in MAPA. We run each reduction technique independently on each case to show the impact of the reduction techniques. In order to make the case studies easily repeatable we developed some simple tooling to automate the case study process.

In Chapters 10-12 we show that large differences exist between the case studies. Dead variable reduction only effects the state space for the gossip model and confluence only has a small effect on the other two models. We also show large differences in effect of the simplification techniques. We show that the differences are likely caused by the density of communication, density of rates in the core of the model and the length of the basic action sequence of this model. In the analysis chapter of this thesis we will describe the meaning of these terms (Chapter 13).

We were not really able to identify concrete points for improvements to SCOOP. However, we have given some directions for future research that is most likely to lead to improvements on the current theory. A short explanation of what this further research could look like then is our answer to question 4.

Part II.

Background

CHAPTER 3

PRELIMINARIES

This chapter introduces the basic concepts that are used throughout this thesis. In Section 3.1 we explain the basics of probability distribution, equivalence relations and the exponential distribution. In Section 3.2 we define Labelled Transition Systems and Probabilistic Automata. In Section 3.3 we extend Probabilistic Automata with delayed transitions, resulting in Markov Automata. We first give a formal definition of a Markov Automaton and then show how to analyse its behaviour and give policies for whenever there are is more than one possible behaviour. We conclude this chapter with explaining some behavioural equivalences. These state that when we reduce the size of the Markov Automata [6, 7], some properties are still preserved.

3.1. Mathematical Basics

In this section we provide the mathematical basics and notation used in this thesis. It is concerned with introducing equivalence relations, which partition elements of a set into a number of disjoint subsets, called equivalence classes. We also explain the exponential distribution. In order to introduce equivalence relations, we need the notions of probability distributions and powersets.

Definition 3.1 (Probability Distributions)
A probability distribution over a countable set S is a function $\mu\colon S \to [0,1]$ such that $\sum_{s\in S} \mu(s) = 1$. Whenever $S' \subseteq S$ we write $\mu(S') = \sum_{s\in S'} \mu(s)$. The set of all probability distributions over S is denoted by $\mathrm{Distr}(S)$. We use $\mu = \mathbb{1}_s$ to denote that with a probability of 1 we end up in state s. This is what is called a Dirac Distribution.

Definition 3.2 (Powerset)
The powerset of a set S (the set of all its subsets) is denoted by $\mathcal{P}(S)$.

3.1.1. Equivalence Relations

A relation R on a set S is an equivalence relation if and only if R is reflexive, symmetric and transitive. Formally, this is defined as follows.

Definition 3.3 (Equivalence Relation)
Given a set S, a relation $R \subseteq S \times S$ is an equivalence relation if and only if R is:

- Reflexive: $\forall a \in S\colon (a, a) \in R$
- Symmetric: $\forall a, b \in S\colon (a, b) \in R \to (b, a) \in R$
- Transitive: $\forall a, b, c \in S\colon (a, b) \in R \wedge (b, c) \in R \to (a, c) \in R$

Given an equivalence relation R we introduce equivalence classes. An equivalence class induced by a state s, denoted by $[s]_R = \{s' \in S \mid (s, s') \in R\}$, contains all elements in S that are equivalent with s. Two probability distributions μ and μ' over a set S are equivalent under R, denoted by $\mu \equiv_R \mu'$, if and only if they yield the same value for every equivalence class, i.e. $\mu([s]_R) = \mu'([s]_R)$ for every $s \in S$.

A Partial Equivalence Relation (PER) is a relation that is only symmetric and transitive.

3.1.2. Exponential Distribution

The exponential distribution [8] is the continuous time variant of the geometric distribution and is a probability distribution that is often used in queueing models. For example, the length of a telephone call and the waiting time at the bakery are exponentially distributed. In the exponential distribution, the cumulative density function (cdf) is a function over a random variable X that calculates the probability that X does not happen before time t. The exponential distribution has one parameter λ, which represents the rate at which the cdf approaches 1. The cdf is given as $f(x) = 1 - e^{-\lambda t}$. In Figure 3.1 an example is shown of a (negative) exponential distribution with rate $\lambda = 0.25$. We will now give an example of an exponentially distributed real world problem.

Example 1 (The Bank Clerk)
Let X denote the amount of time that the bank clerk spends with a random customer. The average time is known to be 4 minutes. The exact time spend is known to have an exponential distribution, i.e. $X \sim \text{Exp}(\lambda)$. In an exponential distribution, the expected time to finish is $\frac{1}{\lambda}$. Since we spend 4 minutes on average, the rate λ here is 0.25. The cumulative density function is $1 - e^{-\lambda t}$, which means that the probability that the clerk is finished before time t, i.e. $P[X \leq t]$ is calculated with $1 - e^{-0.25t}$.

The exponential distribution is memoryless. This means that the future evolution of the model is independent of what has happened in the past. The Bank Clerk example illustrates this. Whether he has already spend 1 minute or 1 hour with a customer, the expected remaining time is the same. However, note that the probability that he will be finished within time t will increase whenever this t grows larger. This can be seen from the chart in Figure 3.1.

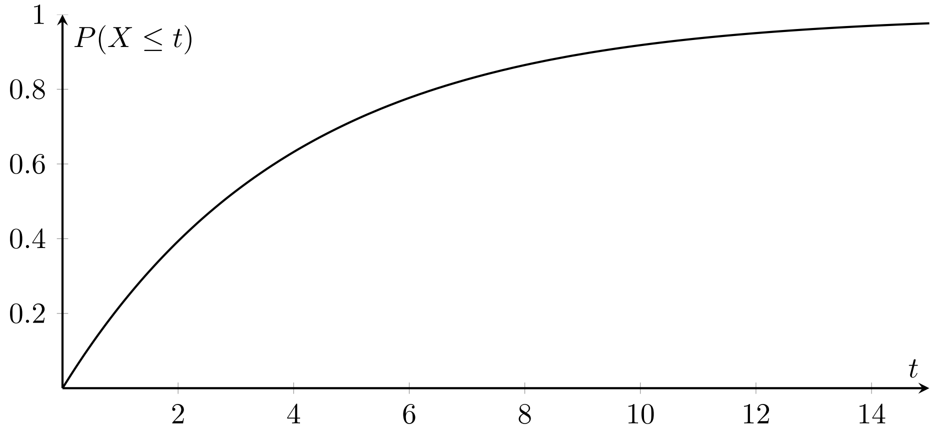


Figure 3.1.: Exponential Distribution with $\lambda = 0.25$

3.2. Automata Theory

In this section we introduce Labelled Transition Systems (LTS) which are a commonly used model for traditional model checking. An LTS consists of states and transitions between states [9]. A state is a representation of how the system looks at a certain time. Transitions model the actions that the system performs (which possibly change the state of the system). An LTS allows for nondeterminism, which means that in a state multiple transitions may be enabled. Thus, in the same state the system might not always behave in the same way. We distinguish between external actions, which the user can observe, and internal actions, denoted by τ.

Definition 3.4 (Labelled Transition System (LTS))
An LTS is a 4-tuple $A = \langle S, s^0, \mathit{Act}, T\rangle$ where S is a countable, non-empty set of states of which s^0 is initial. Act is a set of action labels and $T \subseteq S \times (\mathit{Act} \cup \{\tau\}) \times S$ with $\tau \notin \mathit{Act}$ is the transition relation. Whenever $(p, a, q) \in T$ we write $p \xrightarrow{a} q$. This denotes that from state p we can execute the action a and then go to state q. An action τ represents an internal action.

We want to incorporate probabilistic choice in the model. These often occur in practice, for example whenever a component in the system might break. A Probabilistic Automaton (PA) incorporates these probabilistic transitions, Every action that can be executed has a certain probability distribution function μ which determines with what probability we move to each state.

Definition 3.5 (Probabilistic Automaton (PA))
A Probabilistic Automaton is a 4-tuple $A = \langle S, s^0, \mathit{Act}, T\rangle$ where S, s^0 and Act are defined as for an LTS and $T \subseteq S \times \mathcal{P}(\mathit{Act} \times \mathrm{Distr}(S))$ is the probabilistic transition relation. Transitions are written in a similar way as for LTSs, so whenever $(s, a, \mu) \in T$ we write $s \xrightarrow{a} \mu$ to denote that in state p we can do an action a after which the probability to move to a state $q \in S$ is $\mu(q)$. We define $\mathrm{spt}(\mu) = \{s \in S \mid \mu(s) > 0\}$ to be the support (possible next states) of μ.

3.3. Adding delays

Markov Automata (MA) extend probabilistic automata with a continuous time domain. A new type of transitions is added, called Markovian transitions. Markovian transitions are delayed with an exponential distribution. In probabilistic automata we assumed that all transitions occur instantaneously. This is not true for Markovian transitions.

3.3.1. Markov Automata

Markovian transitions contain a fixed parameter λ that represents the delay with which the transition happens. Whenever there is a Markovian transition λ from state s to state s' this means that the probability that within time t this transition takes place is exponentially distributed with parameter λ. This probability is calculated with $1 - e^{-\lambda t}$, as explained in Section 3.1. Probabilistic transitions are also present in MAs and are called *interactive transitions* from now on. States that can only execute Markovian transitions are called *Markovian states* and states that can only execute interactive transitions are called *interactive states.* States that can perform both types of transitions are called *hybrid states.* If in a hybrid state a τ-transition can happen, then it will always take precedence over Markovian transitions. This is because τ-transitions can happen immediately and Markovian transitions cannot. Therefore, the Markovian transition may as well be removed. This is often referred to as the *maximal progress property* [1]. Note that this does not hold for interactive non-τ transitions as they may be subject to synchronization from other components. How this synchronization works is explained later in this chapter.

Definition 3.6 (Markov Automaton)
A Markov Automaton is a 5-tuple $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow \rangle$, where S, s^0 and Act are defined as for LTSs and PAs. We define $\hookrightarrow \subseteq S \times Act \cup \{\tau\} \times \text{Distr}(S)$ as the interactive transition relation representing the probabilistic transitions and $\rightsquigarrow \subseteq S \times \mathbb{R}^+ \times S$ as the Markovian transition relation. Interactive transitions are represented in the same way as for PAs. For Markovian transitions we write $p \overset{\lambda}{\rightsquigarrow} q$ to denote that there exists a Markovian transition from state p to state q with rate λ.

The rate between two states is $\text{rate}(s, s') = \sum_{(s,\lambda,s') \in \rightsquigarrow} \lambda$ and the outgoing rate of a state s, denoted by $\text{rate}(s) = \sum_{s' \in S} \text{rate}(s, s')$ is the combined rate of all outgoing Markovian transitions. If $\text{rate}(s) > 0$ the branching probability distribution after this delay is denoted by $\mathbb{P}_s$ and defined by $\mathbb{P}_s(s') = \frac{rate(s,s')}{rate(s)}$ for every $s' \in S$

Example 2 (Markov Automaton)
Consider a very simple (and stupid) coffee machine. One of the components of the machine produces the coffee. The component receives a request to produce d cups of coffee. The production time per cup of coffee is exponentially distributed with rate 5. After the production has been completed, the machine gives a finish signal. Afterwards an inspection takes place whether the system still behaves correctly. Since the machine is not very stable it breaks with probability $\frac{1}{3}$ and continues to work correctly with probability $\frac{2}{3}$. In the last case it goes back to the initial state in which it waits for a new request.

In Figure 3.2 the Markov Automaton of the coffee producer is shown. Formally it can be given by the following tuple:

- $S = \{s_0, s_1, s_2, s_3, s_4\}$
- $s^0 = s_0$
- $A = \{\text{request, fin, break}\}$
- $\hookrightarrow = \{s_0 \xrightarrow{request} \mu_1, s_2 \xrightarrow{fin} \mu_2, s_3 \xrightarrow{break} \mu_3\}$ with $\mu_1(s_1) = \mu_3(s_4) = 1$, $\mu_2(s_3) = \frac{1}{3}$, $\mu_2(s_0) = \frac{2}{3}$ and $\mu_i(s_j) = 0$ for every other combination of i and j.

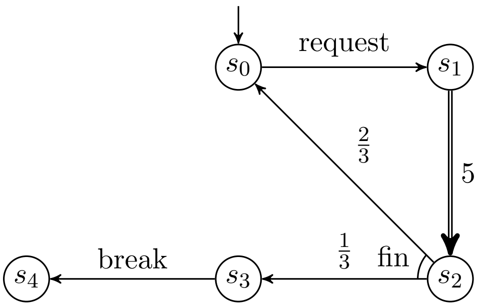


Figure 3.2.: Markov Automaton for the coffee producer

- $\rightsquigarrow = \{s_1 \overset{5}{\rightsquigarrow} s_2\}$

In the picture the arrow above s_0 denotes that this is the initial state. The single lined arrows between states are represent interactive transitions. The double lined arrow represents the Markovian transition.

We want to introduce another component to our example and would like to analyse the correct behaviour of the combined behaviour of the two components. This is where we calculate the parallel composition of two Markov Automata M and M'. The parallel composition allows for multiple automata to communicate over a set of actions A that are present in both MAs, such that whenever states in M and M' can execute these actions, then these actions may only happen in both systems simultaneously. Formally:

Definition 3.7 (Parallel Composition)
Whenever we have two Markov Automata $M = \langle S_1, s_1^0, Act_1, \hookrightarrow_1, \rightsquigarrow_1\rangle$ and $M' = \langle S_2, s_2^0, Act_2, \hookrightarrow_2, \rightsquigarrow_2\rangle$ we define the parallel composition $M||_A M'$ as a 5-tuple $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow\rangle$ with:

- $S = S_1 \times S_2$
- $s^0 = s_1^0 \times s_2^0$
- $Act = Act_1 \cup Act_2$
- $\hookrightarrow = \{(s_1, s_2 \overset{a}{\hookrightarrow} t_1, s_2) \mid (s_1, a, t_1) \in \hookrightarrow_1 \wedge a \notin A\} \cup$
  $\{(s_1, s_2 \overset{a}{\hookrightarrow} s_1, t_2) \mid (s_2, a, t_2) \in \hookrightarrow_2 \wedge a \notin A\} \cup$
  $\{(s_1, s_2 \overset{a}{\hookrightarrow} t_1, t_2) \mid (s_1, a, t_1) \in \hookrightarrow_1 \wedge (s_2, a, t_2) \in \hookrightarrow_2 \wedge a \in A\}$
- $\rightsquigarrow = \{(s_1, s_2 \overset{\lambda}{\rightsquigarrow} t_1, s_2) \mid (s_1, \lambda, t_1) \in \rightsquigarrow_1\} \cup$
  $\{(s_1, s_2 \overset{\lambda}{\rightsquigarrow} s_1, t_2) \mid (s_2, \lambda, t_2) \in \rightsquigarrow_2\}$

We call an MA open whenever it is subject to communication with other MAs. A system without external actions is called closed. Unless stated otherwise, we assume we are dealing with closed Markov Automata in the remainder of this work.

3.3.2. Executions and policies

An execution in an MA is an alternating sequence of states and actions. We call these sequences paths. We also define some special classes of paths in this section, namely finite and maximal paths. Finally, we introduce a policy on how to handle nondeterministic choice. This is done using schedulers.

A (time-abstract) path describes a possible behaviour of the MA over time. It describes which states it visits and which actions it executes at which time. The trace of a path is an external behaviour of the system, so only what the user sees. States and internal actions are omitted.

Definition 3.8 (Paths)
Given an MA $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow\rangle$ we define a path π of M to be either a finite sequence $\pi^{\text{fin}} = s_0 \overset{a_1,u_1}{\rightarrow} s_1 \overset{a_2,u_2}{\rightarrow} ... \overset{a_n,u_n}{\rightarrow} s_n$ or an infinite sequence $\pi^{\text{inf}} = s_0 \overset{a_1,u_1}{\rightarrow} s_1 \overset{a_2,u_2}{\rightarrow} ...$ with $s_i \in S$ for all $0 \leq i \leq n$ and all $0 \leq i$ respectively.

Definition 3.9 (Traces)
Given an MA M and a path $\pi = s_0 \overset{a_1,u_1}{\rightarrow} s_1 \overset{a_2,u_2}{\rightarrow} ... \overset{a_n,u_n}{\rightarrow} s_n$ we define the trace of π, denoted trace(π), as $a_1a_2a_3...a_n$ while omitting all τ-actions. ϵ denotes the empty trace.

An initial path-fragment of length i is denoted with prefix(π, i). One transition on a path is a step, denoted by step(π, i). For finite paths, we define $|\pi| = n$ and last$(\pi) = s_n$. We use finpaths$_M$ for the set of all finite paths of M and finpaths$_M(s)$ for all finite paths with $s_0 = s$.

The system behaves under the maximal progress property. This means that no Markovian transition can happen in a state in which also τ-transitions may happen. However, Markovian transitions still might happen whenever an external action a takes place because it might wait for a synchronizing action in another process. The set of actions that can indeed happen in the system is what is referred to as the extended transition set. Intuitively these are the interactive transitions and the Markovian transitions from states that cannot perform a τ-action. We denote a Markovian transition with rate r as $\chi(r)$.

Definition 3.10 (Extended Transition Set)
Let $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow\rangle$ be a Markov Automata. Then the extended action set of M is given by $A^\chi = Act \cup \{\chi(r) \mid r \in \mathbb{R}_{>0}\}$. Given a state $s \in S$ and an action $\alpha \in A^\chi$ we write $s \overset{\alpha}{\rightarrow} \mu$ if either

- $\alpha \in Act$ and $s \overset{\alpha}{\hookrightarrow} \mu$ or
- $\alpha = \chi(\text{rate}(s)), \text{rate}(s) > 0, \mu = \mathbb{P}_s$ and there is no μ' such that $s \overset{\tau}{\hookrightarrow} \mu'$

In any of these cases a transition $s \overset{\alpha}{\rightarrow} \mu$ is called an extended transition.

So χ denotes a set of Markovian transitions from one state and the extended action set consists of normal probabilistic transitions and Markovian transitions with a positive rate. A Markovian transition in a state where also a τ is enabled can never happen due to the maximal progress property, and thus is not part of the extended transition set.

In order to compute the probability of a certain path occurring, we need to determine which transitions may happen in which state and with what probability. We assume we are working with a closed MA, so transitions are not subject to synchronization. We have our extended action set, so we know which actions may take place. So we are left with computing the likelihood of each choice. We have 3 cases between which we can distinguish.

1. States that can execute both interactive and Markovian transitions
2. States that can execute multiple interactive transitions.
3. States that can execute multiple Markovian transitions.

In a closed MA interactive transitions always have priority over Markovian transitions, due to the maximal progress property. This resolves case 1. For handling nondeterministic choice between two interactive transitions (case 2) we introduce a scheduler, which is a policy that assigns a certain probability to every possible continuation of a path. Note that it might be dependent of the past, i.e. the probability to execute a certain transition in state s might be different whenever s is reached by different paths. Also note that a scheduler will only assign a probability distribution to possible next states after a finite path.

Definition 3.11 (Schedulers)
Given an MA $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow \rangle$ and $\rightarrow$ its set of extended transitions. Then a scheduler C for M assigns to every finite path π a probability distribution over all next states:

$$C \colon \text{finpaths}_M \rightarrow \text{Distr}(\{\perp\} \cup \rightarrow)$$

It requires that for all finite paths π it holds that $\mu_{\pi[i]}(\pi[i+i]) > 0$, so all transitions should indeed be possible in the MA. We use $\perp$ to denote that no transition is chosen by the scheduler, i.e. the execution is finished.

Resolving conflicts between Markovian transitions (case 3) is resolved using races. Consider a state s with two outgoing transitions λ_1 to state q and λ_2 to state r. Then the probability to move to q is $\lambda_1/\text{rate}(s)$ and the probability to move to r is $\lambda_2/\text{rate}(s)$

3.3.3. Analysis

For the analysis of the system we are often interested in paths that terminate. The scheduler C determines which paths are enabled. This is what we refer to as finite paths under C. Whenever the system might terminate in the last state of a path, i.e. $\perp$ is possible for the scheduler, we call this a maximal path.

Definition 3.12 (Finite and Maximal Paths)
Let M be an MA and C a scheduler for M. Then the set of finite and maximal paths of M under C are given by

$$\begin{aligned}\text{finpaths}_M^C &= \{\pi \in \text{finpaths}_M \mid \forall 0 \leq i < |\pi| \cdot C(\text{prefix}(\pi, i)(\text{step}(\pi, i+1)) > 0\}\\ \text{maxpaths}_M^C &= \{\pi \in \text{finpaths}_M^C \mid C(\pi)(\perp) > 0\}\end{aligned}$$

$\text{maxpaths}_M^C(s)$ and $\text{finpaths}_M^C(s)$ are defined as the sets of all maximal and finite paths starting in state s.

The probability that a certain path will be taken is calculated by multiplying all probabilities of the transitions on this path.

Definition 3.13 (Path Probabilities)
Let M be a Markov Automata, C a scheduler for M and s an arbitrary state in M. Now the path probability function $P_M^C, s \colon \text{finpaths}_M(s) \rightarrow [0, 1]$ is defined by

$$P_{M,s}^C(s) = 1;$$

$$P_{M,s}^R(\pi \overset{a,\mu}{\rightarrow} t) = P_{M,s}^C(\pi) \cdot C(\pi)(\text{last}(\pi), a, \mu) \cdot \mu(t)$$

The probability to take a path $\pi = s$ whenever you are already in s is obviously 1. Now consider a longer path ending in a state t, denoted by $\pi \overset{a,\mu}{\to} t$. The probability is defined as the probability that we take the initial part π of the path, we multiply this by the probability that the scheduler picks transition a (leading to μ) and multiply this again by the probability that μ will indeed lead to state t.

When we have a starting state s and a scheduler C in a Markov Automaton M we want to define the probability mass that defines in what states we can end up with what probability, denoted by $F_M^C(s)$. So we do the following calculation for every state s'

$$F_M^C(s) = \{s' \mapsto \sum_{\pi \in \text{maxpaths}_M^C(s) \wedge \text{last}(\pi) = s'} P_{m,s}^C(\pi) \cdot C(\pi)(\perp) \mid s' \in S\}$$

3.4. Behavioural Equivalences

In the abstractions techniques that will be presented in this chapter we inevitably lose some details of the model. However, this is our goal since we are often not interested in some of the details. In Chapter 3.1 we have defined equivalence relations, which is a relation over the states of a system. In this chapter we will present some of these equivalence relations. For the reduction and simplification techniques we can proof that they preserve some of these equivalence relations. We will explain Isomorphism, Strong Bisimulation and Branching Bisimulation respectively. They have an ascending reduction power, but the more the system is reduced, the less properties will be preserved.

Isomorphism is the strongest equivalence relation of the three. Whenever two MAs are isomorphic they can only be distinguished by state name. They have the same internal and external behaviour.

Definition 3.14 (Isomorphism)
Consider two MAs $M = \langle S_1, s_1^0, Act_1, \hookrightarrow_1, \rightsquigarrow_1\rangle$ and $M' = \langle S_2, s_2^0, A_2, \hookrightarrow_2, \rightsquigarrow_2\rangle$. Then M and M' are isomorphic (denoted by $M \equiv M'$) if and only if there exists a bijection $f\colon S_1 \to S_2$ such that $\forall s \in S_1$ and $a \in A^\chi$ it holds that $s \xrightarrow{a} \mu(S_1) \Leftrightarrow f(s) \xrightarrow{a} \mu(f(S_1))$.

When we want to reduce the state space, we can never preserve isomorphism, since for two systems to be isomorphic we require the number of states in both systems to be the same. Strong bisimulation is an equivalence relation that doesn't have this requirement. However, it is still very strict since it requires for all states in the relation to have exactly the same behaviour. Steps cannot be delayed by internal transitions.

Definition 3.15 (Strong Bisimulation for MAs)
Let $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow\rangle$ be a Markov Automaton. Then, an equivalence relation $R \subseteq S \times S$ is a strong bisimulation for M if for all $(s,t) \in R$ and every extended transition $s \xrightarrow{a} \mu$ it holds that $t \xrightarrow{a} \mu'$ such that $\mu \equiv_R \mu'$. We say that two states p and q are strongly bisimilar, denoted by $p \leftrightarrow_{str} q$ if there exists a strong bisimulation R for M such that $(p,q) \in R$. Two MAs M and M' are strongly bisimilar if their initial states are in a strong bisimulation over $M \cup M'$.

We illustrate Strong Bisimulation with an example and also show how it is different from isomorphism.

Example 3 (Strong Bisimulation)
Consider the two Markov Automata in Figure 3.3. Clearly they have the same behaviour since in both systems we will always see the trace *ab*. However, they are not isomorphic since we cannot define a one-to-one mapping between the states. However, they are strongly bisimilar with the smallest equivalence relation containing $R = \{(s_0, t_0), (s_1, t_1), (s_2, t_1), (s_3, t_2)\}$. The states in this relation have the same behaviour and their successor states are again bisimilar according to this relation.

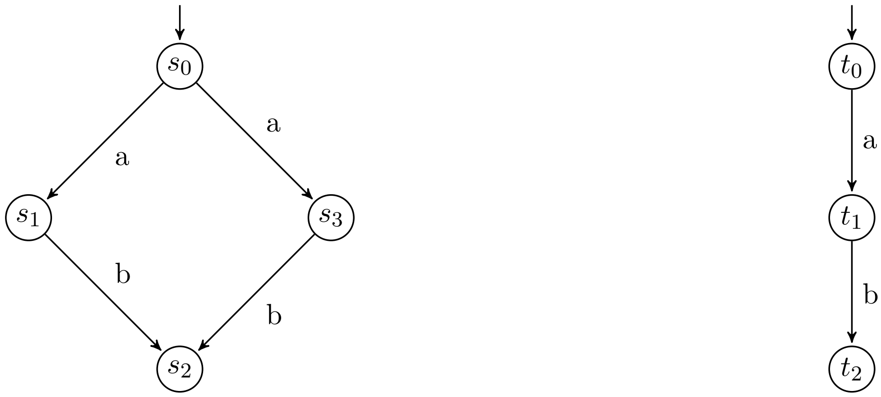


Figure 3.3.: Strong Bisimulation

As an illustration how branching simulation deals with probabilities consider the following example.

Example 4 (Strong Bisimulation 2)
Consider the union of the two systems shown in Figure 3.4. It is clear that s_0 and t_0 have the same behaviour. They can both execute an a and then with probability 0.4 reach a state that does an infinite b-selfloop. For the two systems to be strong bisimilar, we need to have the same probability distribution over possible next behaviours. So for example, after executing a in s_0 we get with probability of 0.4 the behaviour of s_1. So after executing an a in t_0 we also require this behaviour with probability 0.4 for the states to be bisimilar. s_1 and t_1 are bisimilar because they have the same behaviour. For a bisimilar state for t_2 we need a set of states with the same behaviour as t_2 with the same probability mass. The set $\{s_2, s_3\}$ is suitable for that because both s_2 and s_3 have the same behaviour as t_2. So in summary, the smallest equivalence relation containing $\{(s_0, t_0), (s_1, t_1), (s_2, t_2), (s_3, t_2)\}$ is a bisimulation relation.

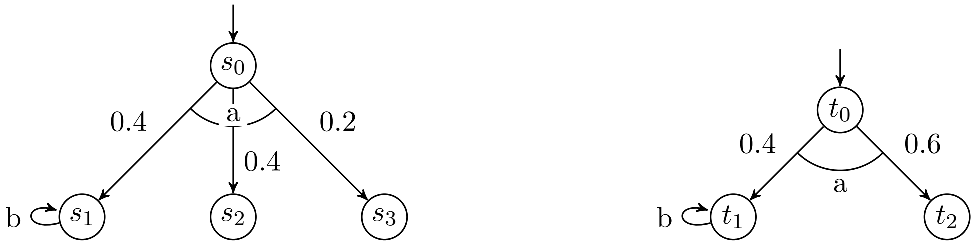


Figure 3.4.: Branching Bisimulation

The third and weakest equivalence relation that is presented here is Branching Bisimulation. Intuitively, two systems are branching bisimilar whenever they have the same observable behaviour and the probability that one external behaviour happens in one system is equal to the probability of this behaviour in the other system. The difference with strong bisimilar systems lies in what happens internally, i.e. τ-steps, which we are often not interested in. In contrast to strong bisimulation we also accept bisimilar steps to be preceded by a series of τ-steps. This is what we call a branching step, denoted by $(s \overset{a}{\Rightarrow}_R \mu)$.

In an equivalence classes a branching step is a sequence of internal transitions ended by some transition a, such that the internal transitions do not lead to another equivalence class. Or in other words, a branching step is an observable transition preceded by zero or more internal transitions that do not influence the external behaviour of the system.

Definition 3.16 (Branching Steps)
Let $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow \rangle$ be an MA, $s \in S$ and R an equivalence relation over S. Then $(s \overset{a}{\Rightarrow}_R \mu)$ is a branching step in two situations:

1. $a = \tau \wedge \mu = \mathbb{1}_s$
2. $\exists C : F_M^C(s) = \mu \wedge \forall \pi \in \text{maxpaths}_M^C : \text{last}(\pi) = a \wedge \forall i \leqslant i < n : (a_i = \tau \wedge (s, s_i) \in R)$

So a state s has a branching step to μ in two situations. The first situation is whenever it is a τ-selfloop. The second situation basically is a τ-cycle. Whenever it is possible to stay in the same equivalence class with only τ-steps and there exists a scheduler that allows this, then we also call a transition after such a cycle a branching step.

Intuitively, two states are branching bisimilar whenever they have the same branching structure. This means that every extended action by either of them can be matched by the other, such that after executing these actions they have the same probability distribution for the next state. The next states again have to be bisimilar.

Definition 3.17 (Branching Bisimulation for MAs)
Let $M = \langle S, s^0, Act, \hookrightarrow, \rightsquigarrow \rangle$ be a Markov Automaton. Then, an equivalence relation $R \subseteq S \times S$ is a branching bisimulation for M if for all $(s, t) \in R$ and every extended transition $s \xrightarrow{a} \mu$ there is a branching step $t \overset{a}{\Rightarrow} \mu'$ such that $\mu \equiv_R \mu'$. We say that two states p and q are branching bisimilar, denoted by $p \leftrightarrow_{bb} q$ if there exists a branching bisimulation R for M such that $(p, q) \in R$. Two MAs M and M' are branching bisimilar if their initial states are in a branching bisimulation over $M \cup M'$.

Example 5 (Difference between Strong and Branching bisimulation)
As an illustration of the difference between Strong and Branching Bisimulation consider Figure 3.5. Clearly they have the same branching steps. The left automaton can do an a directly and the right automaton can also do this, but delayed with one τ. Both, they reach a state with an a-selfloop with probability 0.5 and a state that can do nothing with probability 0.5. However, they are not Strongly bisimilar, since the right automaton cannot do an a directly, whereas the left automaton can.

Preserving branching bisimulation is not strong enough to preserve all desired properties. The problem is that divergences may appear in one system and not in the other system. As an example of what divergences are consider the MA in Figure 3.4. If state s_0 could do a τ-selfloop then the systems would still be branching bisimilar. However, this is not what we desire, since there might be a schedule that only schedules τ-steps for s_0. In that case we will never see an a in the left system, but we will in the right one, so the two systems do not have the same observable behaviour. To solve this problem the notion of divergence sensitivity has been introduced, which makes sure

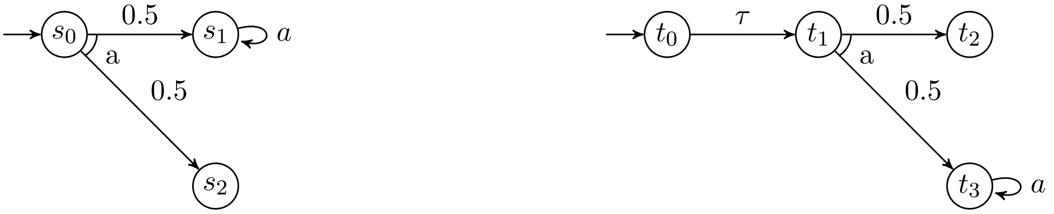


Figure 3.5.: Branching Bisimulation

that whenever two states (s, t) are in an equivalence relation then one can only do an infinite number of τ-steps (diverge) whenever the other one can.

Definition 3.18 (Divergence Sensitivity)
An equivalence relation R is divergence sensitive if for all $(s, s') \in R$ it holds that:

$$\exists C\colon \forall \pi \in \text{finpaths}_M^C(s) \;.\; \text{trace}(\pi) = \epsilon \wedge C(\pi)(\bot) = 0 \wedge \text{last}(\pi) \in [s]_R \iff$$
$$\exists C'\colon \forall \pi \in \text{finpaths}_M^{C'}(s') \;.\; \text{trace}(\pi) = \epsilon \wedge C'(\pi)(\bot) = 0 \wedge \text{last}(\pi) \in [s']_R$$

So whenever under a certain scheduler C all paths starting in s can only be extended with extra τ's, creating an infinite τ-loop, such that we remain in the same equivalence class, then this should also be possible in divergence sensitive systems. Now two MAs are divergence-sensitive bisimilar denoted $\leftrightarrow_{bb}^{div}$ whenever they are both branching bisimilar and the branching bisimulation relation is divergence sensitive.

CHAPTER 4

A PROCESS ALGEBRA FOR MARKOV AUTOMATA

Using Markov Automata we can express a lot of real world behaviour, but for an average system the state space can already blow up to millions of states and transitions. So it is not very convenient to manually draw all these states and transitions. Instead, we use an abstract way for writing down the behaviour of a Markov Automata, namely a process algebra. While the variety of existing process calculi is very large, there are a number of key features that all process algebra's have in common.

- Transitions between independent processes are represented using communication (message passing), rather than with global variables.
- Processes and systems are described using a small number of primitives. Operators are used to combine these primitives.

A process algebra consists of a set of processes, which provide communication. A process consists of a set of local variables and a process term, as defined later. Often process calculi allow for parallel composition (such that 2 or more processes can execute simultaneously), communication (such that two or more parallel processes can communicate over one channel) and hiding (such that certain actions become unobservable).

4.1. Process Terms

MAPA [10] (Markov Automata Process Algebra) is a process algebra in which the behaviour of a Markov Automaton can be described. So beside the key features mentioned before it also allows for temporal and probabilistic operators. MAPA is the input language to SCOOP [2] and extends the prCRL language described in [11] with temporal operators. In this paper we are only concerned with MAPA, so prCRL is not described. A MAPA specification consists of a number of process equations, which describe one communicating component of the system. One such component then consists of a number of process terms that describe which describe the behaviour of this component. Let A be a countable set of actions. Then syntactically a process term in MAPA is defined as follows:

Definition 4.1 (MAPA Process Terms)
A process term in MAPA is any term that can be generated by the following grammar:

$$p ::= Y(t) \mid c \Rightarrow p \mid p + p \mid \sum_{x:D} p \mid a(t) \sum\!\!\!\!\bullet_{x:D} f : p \mid \lambda \cdot p$$

Here Y is a process name, c is a conditional expression, $a \in A$ is a (parameterized) action, f is an expression yielding values in $[0, 1]$, t is a vector of parameters and x is a variable of type D.

The process terms can intuitively be describes as follows:

- $Y(t)$ is a call to some other process with its parameters initially set to t. After this call process Y starts executing.
- $c \Rightarrow p$ is a conditional expression. Whenever the condition c holds the system will continue to behave as p.
- $p + p$ represents nondeterministic choice. The system can have two possible behaviours by which it continues.
- $\sum_{x:D} p$ nondeterministically chooses a value of type D, assigns this to variable x and continues to behave as p, possibly using value x in some next action.
- $a(t) \sum\!\!\!\!\bullet_{x:D} f : p$ first executes a (parameterized) action $a(t)$. Then nondeterministically a value from D is picked and assigned to x. Now we have (paramized) probability distribution f that consists of a probability in $[0, 1]$ and then a continuation as p
- $\lambda \cdot p$ describes Markovian transitions. With a rate λ the system will move to a state in which it will behave as p.

A process equation is an equation $X(g : G) = p$ where g are the global variables of the process and G are their respective types. Unguarded recursion is not allowed. This occurs whenever it is possible to execute direct process calls in a loop. For example, a process call X=Y is allowed, but not whenever Y can again (delayed by zero or more unguarded process calls) call X. In such a loop always at least one action should occur.

A MAPA specification consists of two parts. The first is a set of process equations $X_i(g_i : G_i) = p_i$ such that every X_i has a unique name. The second is a process instantiation $X_i(t)$ for one or more process equations $X_i(g_i : G_i)$ such that the t is of type G_i. At the start of execution the system behaves as the parallel composition of the instantiated processes.

Example 6 (MAPA Specification)
Consider again the scenario described in Example 2. We extend this example with another component that handles the coin input from the user. The user can choose to put k coins, with k between 1 and 10, into the machine. The machine then sends a request to the producer to produce k cups of coffee. The coin-receiver has a cooldown time which is exponentially distributed with $\lambda = 2$. We can write process equations for the two components in MAPA. Process R is seen as the coin-accepting component and process P is the coffee-producer. The associated MAPA-specification is shown in Listing 4.1. We use the variable *broken* to determine whether the machine is broken. $< 2 >$ illustrates a Markovian transition with delay 2.

The grammar does define sequential composition, but always needs a probability distribution in between. However, we often omit a Dirac probability distribution as it makes specifications easier to read. Thus we write $a(t) \cdot p$ for sequential composition, i.e. executing two actions after each other without an intermediate probability distribution.

Listing 4.1: MAPA Specification

```
P(broken:Bool) =
  broken=F → sum k:{1..10} · receive(k) · <k> · finish · (1/3 → break · P(true)) +
      (2/3 → continue · P(false))

R() = sum k:{1..10} · coins(k) · send(k) · <2> · R()
```

4.2. A Linear Format For MAPA

To be able to efficiently reduce the state space of the MA underlying a MAPA-specification we want our process terms to all have the same structure, such that we can structurally apply simplification techniques and state space reduction techniques. This standard format is a Markovian Linear Probabilistic Process Equation (MLPPE). It allows for only one process that consists of summands in which exactly one action and one recursive process call is executed. Formally the MLPPE format conforms to the following equation:

$$\begin{aligned} X(g:G) = &\sum_{i \in I} \sum_{d_i \in D_i} c_i \Rightarrow a_i(b_i) \mathop{\sum\!\!\!\!\!\!\bullet}\limits_{e_i:E_i} f_i : X(n_i) \\ + &\sum_{j \in J} \sum_{d_j \in D_j} c_j \Rightarrow (\lambda_j) \cdot X(n_j) \end{aligned}$$

Two outer sums are sets of summands. The first $|I|$ summands are referred to as interactive summands, the last $|J|$ as Markovian summands. X is our linearised process. It has global variables g from type G. The inner sum is a nondeterministic choice between different local variables in a summand. This means that within a summand still multiple executions are possible. These depend on the values for the local variables in this summand. Furthermore, each summand has a condition c_i, an action a_i and a probability distribution over possible next state vectors n_i. The truth value of the conditional expression may depend on the local/global variables. It is followed by a parameterized action $a_i(b_i)$. After one such action there should be a self-call with a next state vector n_i. This vector may depend on e_i. So each summand first checks whether a condition is true, then executes an action (if the condition is true) and then behaves as process X with initial variables n_i. The Markovian summands are defined in a similar way, with λ_j a temporal action.

Linearisation The linearisation of a MAPA specification happens in two steps. In step 1 every right hand side of process equations becomes a summation of process terms of which each only contains one action. This results in an intermediate regular form (IRF). In step 2 these process equations are merged into one major process equation. The algorithm is rather involved and not described here. More information can be found in [11]. In the rest of this thesis we assume we are working with MLPPE's and apply our reduction techniques to them.

4.3. MAPA Simplification Techniques

Before we apply state space reduction techniques we want our MAPA-specification to be as simple as possible. The simplification techniques that already existed for the LPE format[12] have been generalized for MLPPEs [11].

Maximal Progress Reduction As we have seen before, no Markovian transitions may be taken in states that also allow for a τ-transition. This is the case because of the maximal progress property. Such Markovian transitions can thus safely be omitted. This can already been done on the MLPPE level. We can simply omit all Markovian transitions that are enabled whenever there are also τ-transitions enabled. This is the case whenever the validity of a condition for a Markovian summand implies the validity of a condition for an interactive summand. Heuristics are used to detect this.

Constant Elimination It might be the case that a specification has a parameter that never changes its value. In this case it is a constant and in the entire MLPPE it can be replaced by its initial value. A parameter is constant whenever in every summand it is unchanged or changed to its initial value. However, more precisely, we are not searching for constants, but rather for non-constants. This is done with a greatest fixed-point calculation. Initially all parameters are assumed to be constants. Now in every iteration we check for all parameters x that are still assumed to be constant whether some summand s might change it. This can be the case whenever x is bound by a probabilistic or nondeterministic sum or if its next state is not the current value of x, the initial value of x or the current value of another parameter (that is assumed to be constant) y with the same initial value as x.

Example 7 (Constant Elimination)
Assume we have a specification consisting of two process equations $X(id : 1..3) = transmit(id) \cdot Y(id)$ and $Y(k : 1..2) = process(k) \cdot X(k)$. We can show that both k and id never change their value. Whenever we call Y we always do this with id which is always one. The same holds for the calling of X which is also called with parameter 1. Thus both id and k always remain at value 1, so we change all the occurrences of id and k and replace them by 1. The variables now can be omitted.

Summation Elimination Assume we have a sum operator $\sum_{d:D}$ in a summand s with enabling condition c. Now it can be the case that the enabling condition only allows for one value of d. In that case the summation can be omitted. In practice we compute the set S of possible values for d such that c allows it to be executed (and use the empty set if we cannot establish specific values). Whenever we have a condition $d = e$ or$e = d$ where e is an expression in which d does not occur freely, we evaluate e to p and take a singleton set $S = \{p\}$ (whenever possible). Whenever c is a conjunction $e_1 \wedge e_2$ we take $S = S_1 \cap S_2$ with S_i the set of possible values e_i can have for c to be enabled. For a disjunction $e_1 \vee e_2$ we take a union $S = S_1 \cup S_2$. If it turns out that S is a singleton set $\{k\}$ the summation can be omitted and every occurrence of d can be replaced by k.

Example 8 (Summation Elimination)
Consider the following specification $X = \sum_{pc:\{1..4\}} pc = 2 \Rightarrow send(pc) \cdot X$. It is clear that the summand is only enabled whenever $pc = 2$. Thus every occurrence of pc can be replaced by the value 2 and the sum operator can be omitted. The simplified specification is $X = (2 = 2) \Rightarrow send(2) \cdot X$. This can obviously be reduced further to $X = send(2) \cdot X$ (which is done by expression simplification, as explained shortly).

Expression Simplification It can already be seen in the previous example that there occurs some condition that is always true. Such conditions can be eliminated. It can also happen that by constant elimination we have introduced functions that can be evaluated, since the only parameters they use are constants. Thus we have evaluations of functions with only constant parameters and using basic laws from logic. Summands for which the enabling condition is always false are removed. Additionally we check for each parameter whether it can have a value for conditions to become true. If not, these summands are removed as well.

Example 9 (Expression Simplification)
First consider the enabling condition $(5 = 2 + 3 \vee x > 5) \wedge 3 > 4$. All parameters of the addition are given, so we can simplify this to $(5 = 5 \vee x > 5) \wedge 3 > 4$. The equality function can be evaluated which evaluated to true. We obtain (true $\vee x > 5) \wedge 3 > 4$. We know by basic law that true absorbs all or operators. We obtain true $\wedge 3 > 4$. The $>$ operator evaluated to false, so now we obtain true $\wedge$ false which obviously evaluates to false. Since the enabling condition evaluates to false the summand can be removed since it is never taken.

All these simplification techniques preserve isomorphism, as defined in Section 3.4. Thus it does not reduce the state space, but makes generating the state space faster. The proofs for this can be found in [13] and [1].

CHAPTER 5 CONFLUENCE REDUCTION

In the previous chapter we have described how we can give a symbolic representation of a Markov Automaton. Often, the generation of the underlying Markov Automaton is not possible, since the state space is too large. In this chapter we describe how we can compute the state space and reduce its size on the fly. The technique that we use for that is what is called confluence reduction. We start by giving an intuitive idea of what confluence actually is and then give a formal definition of the concept. In Section 5.2 we show how confluence can be used to give certain transitions priority, resulting in a smaller state space. We conclude the chapter by showing how we can detect confluence on a MAPA-specification.

5.1. The notion of confluence

As an illustration of what confluence is, consider the following situations. We have the two systems depicted in Figures 5.1 and 5.2 and compute the parallel composition. This system looks as depicted in Figure 5.3. In this system only three observable traces are possible, i.e. *abc*, *acb* and *cab*. If we give one τ-transition priority this does not change the behaviour of the system. All traces that were originally possible are still possible, and after all observable traces we end up in the same state. Whenever a τ-transition has this property it is called confluent. We can use these confluent transitions to reduce the state space by skipping over them. Then the state we would encounter and the state after the τ-transition collapse into one new state. In the example, states q_0, q_1 and q_2 are eliminated from the system, reducing the state space. The reduced state space is shown in Figure 5.4.

Confluence defines whenever we can consider τ-transitions confluent. Basically, this means that they do not influence the external behaviour of the MA. This is the case if they commute with all other transitions. Assume we have a state in which both a (probabilistic) a and τ-action are possible. Now for the τ-step to be confluent it basically has to follow 2 criteria.

- After executing τ we can still execute a with the same probability.
- The possible reached states after executing an a directly and after a τ are again connected by confluent τ-steps.

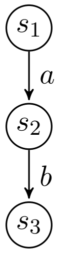


Figure 5.1.: System 1

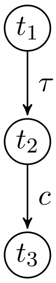


Figure 5.2.: System 2

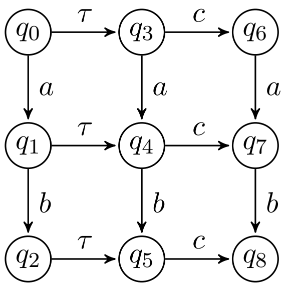


Figure 5.3.: Parallel Composition

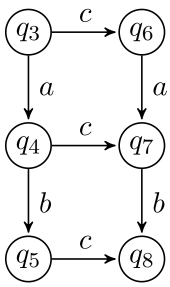
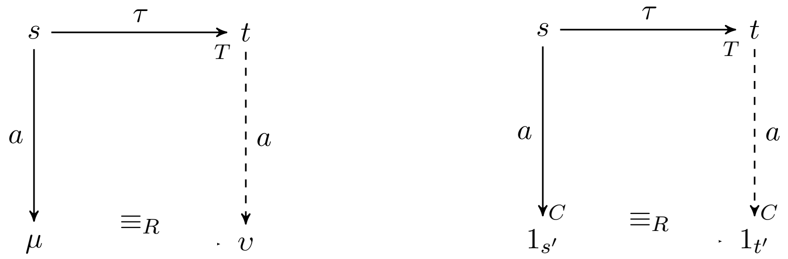


Figure 5.4.: Reduced System



(a) $s \xrightarrow{a} \mu \notin P$

(b) $s \xrightarrow{\tau} t \in P$

Figure 5.5.: Confluence

Originally confluence has been developed for Labelled Transition Systems [14]. To be able to use confluence reduction in SCOOP it has been extended to Probabilistic Automata [6] and Markov Automata [10]. We are only interested in reducing MAPA specifications, so we will only consider the latter case. We use the generalisation [6] of strong probabilistic confluence.

We divide the interactive transitions in the Markov Automata into partitions, called confluence partitions. This partitioning can then be used to define which transitions are considered confluent and which are not. It should hold that all transitions in a partition are confluent or none of them are.

Definition 5.1 (Confluence Partitioning)
Given an MA $M = \langle S, s^0, A, \hookrightarrow, \rightsquigarrow \rangle$, a confluence partitioning for M is a partial equivalence relation (PER) $P \subseteq \hookrightarrow \times \hookrightarrow$ such that if $(s, a, \mu) \in P$ it holds that $a = \tau$ and $|\text{spt}(\mu)| = 1$, i.e. executing a in s has a unique target state.

A confluence partitioning P is thus a PER on the transitions of the MA M. From P we can construct equivalence classes C over the states of M. We write $s \xrightarrow{a}_C \mu$ if the transition $s \xrightarrow{a} \mu$ is in equivalence class C. Whenever we have a subset T of all equivalence classes we write $s \xrightarrow{a}_T$ to denote that $s \xrightarrow{a}_C \mu$ for some $C \in T$. Formally confluence can be described as follows.

Definition 5.2 (Markovian Confluence)
Let $M = \langle S, s^0, A, \hookrightarrow, \rightsquigarrow \rangle$ be a Markov Automata and P a confluence partitioning for M. Then,

a set $T \subseteq S/P$ is Markovian confluent for P if for every $s \xrightarrow{\tau}_T t$ and all transitions $s \xrightarrow{a} \mu \neq s \xrightarrow{\tau} t$:

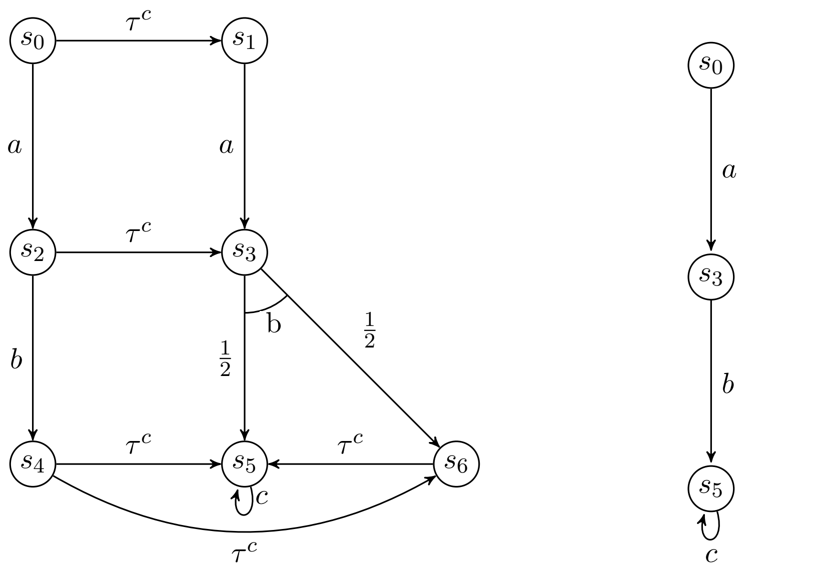
$$(s \xrightarrow{a} \mu) \notin P \longrightarrow \exists \nu \in \mathrm{Distr}(S) \cdot t \xrightarrow{a} v \wedge \mu \equiv_R \nu$$

$$(s \xrightarrow{a} \mu) \in C \text{ with } C \in S/P \longrightarrow \exists \nu \in \mathrm{Distr}(S) \cdot t \xrightarrow{a}_C v \wedge \mu \equiv_R \nu$$

with R the smallest equivalence relation such that $R \supseteq \{(s,t) \in \mathrm{spt}(\mu) \times \mathrm{spt}(\nu) \mid (s \xrightarrow{\tau} t \in T)\}$ Given a partitioning P, an internal transition $s \xrightarrow{\tau} t \in P$ is Markovian confluent if there exists a Markovian confluent set T such that $s \xrightarrow{\tau}_T t$. If an equivalence class C is contained in a Markovian confluent set T then we call this equivalence class confluent.

The definition is illustrated in Figure 5.5. If the solid transitions are present then the dashed ones should be present as well. The sign between $1_{s'}$ and $1_{t'}$ denotes that there should be a τ-transition from s' to t'. The left figure represents the first case of the definition, where $s \xrightarrow{a} \mu \notin P$. The definition now requires that there are direct transitions from the support of μ to the support of v. In the second case, there is always a unique target state after a, which is required by Definition 5.1. Thus we can reduce $\mu \equiv_R v$ to $u = v \vee u \xrightarrow{\tau}_T v$.

Example 10 (Confluence)
Consider the system depicted in Figure 5.6. In this Figure we use τ^c to mark transitions confluent. It is clear that we will always see an a, then a b and then infinitely many c's. However, if we only pick the transition $s_0 \xrightarrow{\tau} s_1$ as our candidate confluent set T this is not sufficient as s_2 and s_3 have to be shown to be confluent as well. So we make the τ transition between s_2 and s_3 an element of T. The same problem arises as s_4 needs to be equivalent with s_5 and s_6. Thus the other three τ-transitions should be in T as well for it to be confluent. Whenever we encounter one of the mentioned τ-transitions, we skip over them. This results in the state space depicted in Figure 5.6.



(a) Original System (b) Reduced System

Figure 5.6.: Confluence Example

5.2. State Space Reduction

In this section we assume that we have found a confluent transition set T and we will use this to reduce the state space. In Example 10 we have already shown how this reduction works, but we have not given a formal definition on how to do this. The reduction relies on a representation map. A representation map is a function over the state space. Every state has a representative state which can be reached by confluent τ-steps.

Definition 5.3 (Representation Map)
Let $M = \langle S, s^0, A, \hookrightarrow, \rightsquigarrow \rangle$ be an MA and T a Markovian confluent set for M. We define the function $\phi \colon S \to S$ as a function to map each state to its representative. For all $s, t \in S$ it has to fulfil the following conditions.

- $s \twoheadrightarrow_T \phi(s)$
- $s \to_T t \implies \phi(s) = \phi(t)$

with $\twoheadrightarrow_T$ a sequence of zero or more transitions in T.

The first requirement states that every state can reach its representative by zero or more confluent τ-steps. The second requirement states that two states can only be connected by confluent τ-steps whenever they have the same representative. If we have a representation map then we can define the minimal reduced state space under that representation map as M/ϕ, the quotient of M under a representation map ϕ.

Definition 5.4 (Quotient MA)
Let $M = \langle S, s^0, A, \hookrightarrow, \rightsquigarrow \rangle$ be an MA and T a Markovian confluent set for M. Let ϕ be a representation map for S. Then the quotient MA is defined as $M/\Phi = \langle \phi(S), \phi(s^0), A, \phi(\hookrightarrow), \phi(\rightsquigarrow) \rangle$ where

- $\phi(S) = \{\phi(s) \mid s \in S\}$
- $\phi(\hookrightarrow) = \{\phi(s) \overset{a}{\hookrightarrow} \phi(\mu) \mid (s \overset{a}{\hookrightarrow} \mu) \in \hookrightarrow\}$
- $\phi(\rightsquigarrow) = \{\phi(s) \overset{\lambda_\phi}{\rightsquigarrow} \phi(t) \mid \sum_{\substack{(\phi(s) \overset{\lambda}{\rightsquigarrow} t') \in \rightsquigarrow \\ \phi(t') = t}} \lambda = \lambda_\phi \wedge \lambda_\phi > 0\}$

Intuitively this definition says that the reduced state space consists of all representative states. Original interactive transitions between two states now are replaced by the same transition between their representatives. The same holds for Markovian transitions, but for this we need to sum over the rates of all Markovian transitions between representatives.

It holds that the original system and its quotient are divergence sensitive branching bisimilar, hence confluence preserves this notion. The proof for this can be found in [10]

5.3. Symbolic Detection in MAPA

It is not very useful to first generate the entire state space and then apply confluence reduction. The goal is to be able to generate state spaces that can normally not be generated. Thus we want to detect it directly on the input language, the MLPPE. Rates have no impact on whether a set of transitions is considered confluent. This is because Markovian transitions are only enabled in states where no τ-steps can happen. Therefore, we only need to consider interactive summands. Recall the structure of an interactive summand.

$$\sum_{d_i:D_i} c_i(g, d_i) \Rightarrow a_i(g, d_i) \sum^{\bullet}_{e_i:E_i} f_i(g, d_i, e_i) : X(n_i(g, d_i, e_i))$$

What we want to do is detect whether summands in the MLPPE are probabilistically confluent. Clearly, for a summand to be confluent its action should be τ, i.e. $a_i(g, d_i) = \tau$ for all possible values of g and d_i. Also, the next state should be uniquely determined, so $f_i(g, d_i, e_i) = 1$ for a single e_i and $|E_i| = 1$. Also, the confluence properties should hold. A confluent τ-summand has to commute properly with every summand j. Whenever both are enabled executing one may not influence the behaviour of the other and the order of their execution should not influence the observable behaviour of the final state. Also, a confluent transition commutes with itself (so only one transition may be generated). Formally:

$$\begin{align}
(c_i(g, d_i) \wedge c_j(g, d_j)) \rightarrow (i = j \wedge n_i(g, d_i, e_i) = n_j(g, d_j, e_j)) \vee \tag{5.1}\\
(c_j(n_i(g, d_i, e_i), d_j, e_j) \wedge c_i(n_j(g, d_j, e_j), d_i, e_i) \tag{5.2}\\
\wedge\, a_j(g, d_j, e_j) = a_j(n_i(g, d_i, e_i), d_j, e_j) \tag{5.3}\\
\wedge\, f_j(g, d_j, e_j) = f_j(n_i(g, d_i, e_i), d_j, e_j) \tag{5.4}\\
\wedge\, n_j(n_i(g, d_i, e_i), d_j, e_j) = n_i(n_j(g, d_j, e_j), d_i, e_i)) \tag{5.5}
\end{align}$$

Formula 5.1 states that whenever two summands are both enabled (their enabling condition is true) then either they should coincide (5.1) or all other formulas should hold. Two summands coincide whenever executing either results in the same next state (only one possible transition is possible in this state). Formula 5.2 states that after executing either one of the summands the enabling condition for the other is still true. Formula 5.3 states that taking the confluent τ does not change the parameters used in actions of other enabled summand(s). Formula 5.4 states that the probabilistic function in the other is not influenced by global variables that are changed in the τ-summand. Formula 5.5 requires that the order in which the actions take place does not matter for the final state. The following examples illustrate how the different formulas prevent summands from being confluent.

Example 11 (Formula 5.1)
A specification like $X(d : Int) = d \geq 0 \rightarrow \sum_{d:Int} \tau \cdot X(d)$ would not commute with itself as after the τ-transition the next state vector can have multiple values. The next state vector should be unique for this confluent τ-step to commute with itself.

Example 12 (Formula 5.2 and Formula 5.3)
Consider a specification X(div:Int) with two summands $div = 2 \rightarrow \tau \cdot X(3)$ and $div \leq 2 \rightarrow a(div) \cdot X(2)$ with X(2) our initial process. We use synactic sugar here to denote that after an a we proceed as X(2) with probability $\frac{1}{div}$ and as X(4) with probability $\frac{div-1}{div}$. Initially both summands are enabled, but after executing the τ-summand the other summand is not enabled anymore. This means that Formula 5.2 is violated and this is not a confluent summand. Also a global variable that is used in action $a(div)$ is changed, so also Formula 5.3 is violated.

Example 13 (Formula 5.4)
Consider two summands $div \geq 2 \rightarrow \tau \cdot X(3)$ and $div \geq 2 \rightarrow a \cdot \frac{1}{div} : X(2) + \frac{div-1}{div} : X(4)$ with X(2) our initial process. The τ-summand is not confluent here because the probability function of the other summand depends on div, which is changed in the τ-summand. Formula 5.4 is violated.

Example 14 (Formula 5.5)
Consider two summands $div \geq 2 \rightarrow \tau \cdot X(3)$ and $div \geq 2 \rightarrow a \cdot X(div + 1)$ with X(2) our initial process. If we first execute the a-summand we end up in state X(3), but if we first execute the τ-summand we end up in state X(4). This is not allowed by Formula 5.5 and thus this is not a confluent summand.

CHAPTER 6
DEAD VARIABLE REDUCTION

In this chapter we describe dead variable reduction. Compared to confluence, this technique makes changes to the MLPPE, where confluence is applied during state space generation. The general idea is that for each variable we determine whenever its value is relevant. A variable is called relevant in a summand whenever it might be used by its enabling/action function, or influences relevant variables in the next state. Otherwise, the parameter is called irrelevant or dead. If a parameter is dead, then its value does not matter and thus we can replace its occurrences by a constant. This reduction seems to be similar to the constant elimination as described in Chapter 4.3, but is different in a sense that dead variables are not necessarily constants. Dead variable reduction thus goes much deeper in its analysis and is able to reduce the state space whereas constant elimination only simplifies the MLPPE. We start this chapter by giving an intuitive idea behind the technique. We continue with describing how we can construct a control graph and use this to determine whenever a parameter is relevant or not. We conclude the chapter by showing how this information can be used to reduce the state space.

6.1. Background

As an illustration of when dead variable reduction works consider the following scenario. We have a system with a summand that gives a variable i a random value between 0 and 10 in the next state. However, after it reaches a value greater than 6, the system proceeds to a next stage in which i is not used anymore. However, the summand always remains enabled. Initially, this variable is relevant, but after a while it becomes irrelevant. Dead Variable Reduction can detect this situation. After it becomes irrelevant we might as well change the specification and give i a fixed value. This reduces the state space and does not change the behaviour of the system. This is the general idea behind dead variable reduction.

In order to detect whenever a variable is irrelevant we need to know as much as possible about the possible values variables can take in which summands. Often there are parameters for which we always know the value before and after we execute a summand. This can sometimes be detected whenever the parameter is used in the enabling function and/or in the next state vector of a summand. If we can determine the value for a parameter before and after *each* summand in which it is changed we call this parameter a control flow parameter (CFP). From this we can construct a control graph. This is a data structure that describes how the value of a CFP changes by executing which summands.

In order to analyse whenever a data parameter (all parameters that are not a CFP) is relevant, we use the CFPs. How this works is described later. Whenever there is no reason to mark a data parameter relevant in a summand, we consider it irrelevant. Whenever in every possible next state after executing a summand a DP is marked irrelevant, we replace the occurrences of this DP in the next state vector by its initial value.

6.2. Construction of the Control Graph

In this section we describe how we can construct a control graph for some variables. We can only construct a control graph for a parameter whenever we know its value before and after each summand in which it is changed. Otherwise there is some uncertainty in our graph and eventually in our analysis whether a variable is dead. Thus for every parameter we want to learn as much as possible about the values it can take in each summand. Therefore, we analyse the possible values of the parameters before each summand and after each summand. We also want to know which parameters are actually used or changed in each summand.

Definition 6.1 (Changed, Used, Directly Used)
In a summand i, we call a parameter d_j changed whenever its value after taking i might be different from its current value. Otherwise, we call d_j unchanged.
In a summand i, a parameter d_j is directly used whenever it occurs in its enabling function c_i or in its action function a_i.
In a summand i, a parameter d_j is used whenever it is either directly used or needed to calculate the next state.

Determining which value a parameter must have for a summand to be enabled is in general undecidable, but heuristics can be used to yield the right value. Whenever we have an enabling function consisting of logical operators, we can apply logical operators such as disjunction, conjunction and equations as heuristics and often this yields the right value for the required value. This value is what we call the source. The source of a parameter d_j in summand i is the value that d_j needs to have for the enabling function c_i to evaluate to true. For every parameter it holds that $\text{source}(i, d_j) = \bot$ is allowed, which indicates that no unique value s could be defined as the source. The following example illustrates how heuristics can be used to determine the source of a parameter.

Example 15 (Source)
Let our enabling condition $c_i(d, e_i)$ be given by $(d_j = 4 \vee d_j = 5) \wedge (d_k = 10 \vee d_k = 3) \wedge d_j = 5$. Obviously $\text{source}(i, d_j) = 5$ is valid, as $(\{4\} \cup \{5\}) \cap \{5\} = \{5\}$. As always $\text{source}(i, d_j) = \bot$ is also valid. For parameter d_k a unique source does not exist, so $\text{source}(i, d_k) = \bot$ is our only option.

We can do something similar for the (unique) value of a parameter after taking a summand. This is what we refer to as the destination of a parameter. Again, calculating this value is in general undecidable, but similar heuristics as for computing the source can be used. Again, for every parameter it holds that $\text{dest}(i, d_j) = \bot$ is allowed, which indicates that no unique value s could be defined as the destination. The destination of a parameter d_j in summand i is the value that d_j needs to have after executing i. For every parameter it holds that $\text{dest}(i, d_j) = \bot$ is allowed, which indicates that no unique value d could be defined as the destination. The following example shows how the destination of a parameter can be determined

Example 16 (Destination)
Consider a specification $X(d : \text{Int}, p : \text{Int}) = (d = 0) \to a \cdot X(d+1, p+5)$. We know that the destination of d is one higher than its source. Since its source is 0, we know its destination is $0+1 = 1$. As always $\text{dest}(i, d) = \perp$ is also valid. For parameter p we do cannot uniquely determine its source, and its destination depends on the source. Therefore, $\text{dest}(i, p) = \perp$ is our only option.

If we have a parameter d_j and a summand i for which both a non-trivial source and destination can be computed we say that d_j rules i, denoted $\text{rules}(d_j, i)$. The total set of all summands that are ruled by d_j is denoted by $R_{d_j} = \{i \in I \mid \text{rules}(d_j, i)\}$. Obviously, summands can be ruled by more than one parameter.

We call a parameter a control flow parameter (CFP) if before and after each summand we know its value. So for every summand, a CFP should either rule it or be unchanged. Note that whenever a CFP is unchanged we do not really know its value, but in the control flow graph we also do not need a transition for this.

Definition 6.2 (Control Flow Parameter)
A parameter d_j is called a control flow parameter (CFP) if it is only changed in summands which it rules. A parameter that is not a CFP is a data parameter (DP). The set of all summands that are ruled by a CFP is called its cluster. We denote the set of all CFPs by C and the set of all DPs by D.

The following example serves as an illustration for the mentioned definitions. It is taken and slightly changed from [7]

Example 17 (Communication)
Consider a system consisting of one reader and one writer. The reader reads some data and communicates it to the writer. The writer receives the data and writes it somewhere. The linearised MLPPE looks as follows:

$$\begin{aligned}
&X(a : \{1,2\}, b : \{1,2\}, x : \text{Int}, y : \text{Int}) = \\
&\quad \sum_{d:D} a = 1 \Rightarrow \text{read}(d).X(2, b, d, y) \\
&\quad + a = 2 \wedge b = 1 \Rightarrow \text{comm}(x).X(1, 2, x, x) \\
&\quad + b = 2 \Rightarrow \text{write}(y).X(a, 1, x, y)
\end{aligned}$$

For the first summand we know that $\text{source}(1, a) = 1$ because it is only enabled whenever $a = 1$. We also know $\text{dest}(1, a) = 2$ as after executing summand 1 a gets a fixed value 2. Since both the source and destination of a can be computed it holds that a rules summand 1. For the same reason it also rules summand 2. Similarly, we know that b rules summands 2 and 3. In all summands they do not rule they are unchanged, so both a and b are CFPs. In summand 1 the value of x is changed while its source and destination are unknown. The same holds for y in summand 2. So x and y are DPs.

Based on a CFP we can construct its control graph. The nodes of this graph are all the values that the CFP can take and the edges are the summands that can be taken. Specifically, an edge labelled i between nodes s and t means that if $d_j = s$ summand i may be taken resulting in $d_j = t$.

Definition 6.3 (Control Flow Graph)
Let d_j be a CFP. Then the control graph for d_j is the tuple $\langle V_{d_j}, E_{d_j} \rangle$ where

- $V_{d_j} = \{\text{source}(i, d_j) \mid i \in R_{d_j}\} \cup \{\text{dest}(i, d_j) \mid i \in R_{d_j}\} \cup \{\text{init}_j\}$
- $E_{d_j} = \{(s, i, t \mid i \in R_{d_j} \wedge \text{source}(i, d_j) = s \wedge \text{dest}(i, d_j) = t\}$

Note that we construct a control graph per CFP. Global control flow might have its uses, but this graph might grow larger than the complete state space. This defeats its main purpose completely and is thus not very useful. We use the control graphs to analyse the data flow and to detect which variables are dead and thus can be eliminated.

6.3. Data Flow Analysis

In this section we analyse the behaviour of the Data Parameters and determine whenever their value might be irrelevant in a summand. The general idea is that an irrelevant DP after taking a summand i can be reset to its initial value without changing the behaviour of the system. In this section we describe how we can determine which DPs are irrelevant in which summands and in Section 6.4 we describe how this is used in reducing the state space.

Using the CFPs we can analyse to which clusters DPs belong. Intuitively a DP belongs to a cluster of a CFP d_j whenever it is only used or changed in summands of that cluster. Formally this can be defined as follows:

Definition 6.4 (The belongs-to-relation)
Let d_k be a DP and d_j a CFP. Then d_k belongs to d_j if all summands i that change or use d_k are ruled by d_j.

We assume that each DP belongs to at least one CFP. Although this might initially not be accomplished, this can be satisfied by adding a dummy boolean variable b that remains true in every summand and adding b=true to the condition of every summand. b is a CFP since it obviously rules every summand. Its cluster is the entire system. Therefore, every data parameter belongs to this CFP. CFPs do not belong to anything. Thus whenever a DP d_k belongs to a CFP d_j we know that the entire data flow of D_k is contained in the cluster of d_j. The decision of whether we can reset its value can thus be made based on solely the summands within this cluster.

Example 18 (Belongs To Relation)
Consider again the MLPPE of the previous example. Our DPs are x and y. The value of x is only used or changed in summand 1 and 2, the cluster of a. The value of y is only used or changed in summand 2 and 3, the cluster of b. Therefore, x belongs to a and y belongs to b.

We restrict our analysis to the changes in the cluster of d_j as our DP is not used or changed in other summands. The analysis consists of defining whenever (for which values of d_j) the value of d_k is relevant. Intuitively, d_k is only relevant whenever it might still be used before it will be changed. If we are certain that d_k is not used anymore, we know it is irrelevant. d_k can still be relevant in a summand i in three situations.

1. d_k is directly used in i
2. d_k is used in i to determine the value of a DP that is relevant in another summand within the same cluster.
3. d_k is used in i to determine the value of a DP belonging to a different CFP which d_k does not belong to.

The next definition formalizes this.

Definition 6.5 (Relevance)
Let d_k be a DP and d_j be a CFP such that d_k belongs to d_j. Given some s as the value of d_j, we use $R(d_k, d_j, s)$ to denote that the value of d_k is relevant whenever $d_j = s$. R is the smallest relation such that:

1. If d_k is directly used in a summand i and it belongs to d_j, then $R(d_k, d_j, s)$ for $s = \text{source}(i, d_j)$.
2. If $R(d_l, d_j, t)$,there exists a summand i such that $(s, i, t) \in E_{d_j}$, d_k belongs to d_j and d_k is used to determine the value of d_l in the next state, then also $R(d_k, d_j, s)$.
3. If $R(d_l, d_p, t)$ and there exists a summand i and an r such that $(r, i, t) \in E_{d_j}$, then whenever a DP d_k belonging to another cluster d_j to which d_l does not belong might change d_l whenever $s = \text{source}(i, d_j)$, then $R(d_k, d_j, s)$.

So (1) states that a DP is relevant whenever a summand in the cluster of d_j uses or changes the value of the DP. (2) states that whenever a variable is relevant and in a previous summand another DP might change the value of this DP, then this other DP is also relevant in that previous summand. (3) states that whenever we have 2 different control parameters and 2 data parameters belonging to different clusters, then whenever in both clusters a summand in enabled and one of the DPs might change the other, which is relevant, then both DPs are relevant.

Example 19 (Detecting Irrelevant Variables)
Consider again the system from the previous example. We hide the comm action.

$$\begin{aligned} &X(a : \{1, 2\}, b : \{1, 2\}, x : \text{Int}, y : \text{Int}) = \\ &\quad \sum_{d:D} a = 1 \Rightarrow \text{read}(d).X(2, b, d, y) \\ &\quad + a = 2 \land b = 1 \Rightarrow \tau.X(1, 2, x, x) \\ &\quad + b = 2 \Rightarrow \text{write}(y).X(a, 1, x, y) \end{aligned}$$

Applying the first clause yields $R(y, b, 2)$ since y is directly used in summand 3. Also, in summand 2 we use x to determine the value y will get in the next state. In this next state y is relevant, so also x is relevant whenever $a = 2$ and $b = 1$. Since x belongs to a different cluster we find that, by clause 3, $R(x, a, 2)$. Then, no more clauses apply, thus we find $\neg R(x, a, 1)$ and $\neg R(y, b, 1)$.

The method presented here doesn't necessarily find all irrelevant variables in the system. However, it does never erroneously mark a variable irrelevant. This means that whenever one CFP says the DP is irrelevant, then it is irrelevant in the entire system. So a DP is only relevant whenever it is relevant for the current valuations of all CFPs it belongs to. Formally,

Definition 6.6 (Global Relevance)
A parameter d_k is relevant in a state vector v, denoted by $\text{Relevant}(d_k, v)$ whenever it is relevant

for all current valuations of the CFPs in the state vector.

$$\text{Relevant}(d_k, v) = \bigwedge_{\substack{d_j \in C \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, v_j)$$

Obviously, for CFPs this always evaluates to true, which is correct because it is always relevant.

Example 20 (Detecting Irrelevant Variables (2))
Consider again the system of the previous example. We know that x belongs to a and that it is irrelevant whenever $a = 1$. Since it does not belong to any other CFP we find the for state vector $v = (1, *, *, *)$, where * represents a don't care variable, it holds that $\neg\text{Relevant}(x, v)$. Similarly, $\neg\text{Relevant}(y, v')$ for $v' = (*, 2, *, *)$.

6.4. State Space Reduction

The most important application of the techniques presented in this chapter is that they can be used to reduce the state space of the Markov Automaton underlying an MLPPE without generating the entire state space. The basic idea behind the transformation is that if a DP is irrelevant in all summands that are enabled after executing a summand, then this data parameter might as well reset to its initial value.

Definition 6.7 (Transformed MLPPE)
Given an interactive MLPPE X of the familiar form, we define the transformed MLPPE X' as:

$$X'(d : D) = \sum_{i:I} \sum_{e_i:E_i} c_i(d, e_i) \Rightarrow a_i(d, e_i) \sum_{k_i:K_i} f_i(d, e_i, k_i) : X'(g_i'(d, e_i))$$

with

$$g'_{i,k}(d, e_i) = \begin{cases} g_{i,k}(d, e_i), & \text{if} \displaystyle\bigwedge_{\substack{d_j \in C \\ d_j \text{ rules } i \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, \text{dest}(i, d_j)) \\ init_k, & \text{otherwise} \end{cases}$$

Transformed Markovian summands are derived in a similar way.

Parameters in summands in the transformed MLPPE only deviate from the original MLPPE whenever they are irrelevant in those summands. From this it can be proved [7] that the original and its transformed MLPPE are strong bisimilar. In [7] it is also proved that the state space of the transform is as most as large as in the original system.

Example 21 (Transform)
Consider again the system of the previous example. We have shown that x is irrelevant in summand 2 and that y is irrelevant in summand 3. Thus in these summands we can replace their occurrences

by their initial values. For an initial state vector (1,1,0,0) we find the following transformed system.

$$\begin{aligned}
&X(a:\{1,2\}, b:\{1,2\}, x:\text{Int}, y:\text{Int}) = \\
&\quad \sum_{d:D} a = 1 \Rightarrow \text{read}(d).X(2,b,d,y) \\
&\quad + a = 2 \wedge b = 1 \Rightarrow \tau.X(1,2,x,0) \\
&\quad + b = 2 \Rightarrow \text{write}(y).X(a,1,0,y)
\end{aligned}$$

Originally we could reach all states $(1,1,x,y)$ where $x = y$, but in the reduced state space we have the extra restriction where $x = y = 0$. So this significantly reduces the state space.

CHAPTER 7

INTERACTIVE MARKOV CHAIN ANALYSER

7.1. Introduction

IMCA (Interactive Markov Chain Analyser) is a tool for the quantitative analysis of Interactive Markov Chains (and Markov Automata as well). It allows for the verification of IMC's against timed reachability, unbounded reachability, long run average, expected time and expected step objectives. We give a very brief explanation of what these objectives mean. For a more thorough explanation of how they are calculated we refer the reader to [3].

- Unbounded reachability simply evaluates whether a specified target state is reachable at all. The result can be either 1 or 0 meaning a reachability with probability 1 and a reachability with probability 0.
- Timed reachability extends the previous notion with a time bound. Thus it evaluates the probability that a target state will be reached within a certain time bound.
- Long Run Average measures the expected fraction of time that a certain property holds. We divide our states by goal states and non-goal states and calculate the expected fraction of time (in the long run) that we spend in a goal state.
- Expected Time measures the expected amount of time before entering a goal state for the first time.
- Expected Step properties are not relevant for this project and thus will not be explained.

Besides being able to verify these objectives, IMCA supports the minimization of IMCs with respect to strong bisimulation (as introduced in Chapter 3.4). In order to use IMCA one needs to provide an input model containing the structure of the IMC and a set of target states. It accepts for example the bcg-format, such that it can be used to validate state spaces generated by the CADP toolbox [4].

7.2. Connection with SCOOP

We want to validate properties for our case studies using IMCA. Therefore, we need to add these properties to our MAPA models. MAPA has two ways to specify requirements to be validated in

IMCA. First, you can use a reach-statement. A reach-statement consists of a Process instantiation and a certain action. An example for this is a node that can still write data to a buffer (action) while the system has crashed (process instantiation). This statement is validated in states where the process instantiation holds and the specified action is enabled directly. The second way to specify a requirement is with a reachCondition statement. In such a statement you can specify the value of one or more global variables in the MLPPE. Such a statement holds in states in which the specified global variables have the specified values. Note that variables do not need to be present in the original model, i.e. before linearizing. Rather, one should use variables from the MLPPE. Therefore, you should first create the MLPPE and analyse which variables need to gain which values for your requirement. Afterwards, you can add the reachCondition to the model.

With the reach or reachCondition statement added SCOOP can be used with the -imca option to generate an IMCA model. The final states are then the states that satisfy the specification. IMCA can then be used to verify the specification against timed reachability, unbounded reachability, long run average, expected time and expected step objectives.

CHAPTER 8

RELATED WORK

Model checking suffers from the state space explosion problem. Whenever a computer program gets even moderately large, the state space of its model grows exponentially. Research has been going on for decades already to manage this problem, and this chapter is devoted to describe some of the techniques that do this. For many classes of models, such as Labelled Transition Systems [9], Probabilistic Automata [11], Interactive Markov Chains [15], Continuous-Time Markov Chains [16], Markov Decision Processes [17] and Timed Automata [18], similar research has been going on. However, all these models, except for timed automata, are subsumed by the MA. This is both an advantage and a disadvantage. The upside is that the reduction techniques applicable to MAs are also applicable for all subsumed models. However, the downside is that the reduction techniques need to be generalized in some way and this makes them more complex and more difficult to understand.

8.1. Abstraction

SCOOP uses confluence and dead variable reduction as abstraction techniques, but similar techniques exist. In this section we attempt to show some of these .

Different forms of confluence In Chapter 5 we introduced Confluence Reduction for Markov Automata. This notion has first been introduced for Labelled Transition Systems [19] and proved to be a very efficient means of reducing the state space. It has been extended to both Probabilistic Automata [6] and Markov Decision Processes [20] before it got generalized to Markov Automata [10]. Confluence has also been adapted in order for it to be used in a statistical model checking setting [21]. For more information on confluence, see Chapter 5

Symbolic Model Checking Symbolic model checking is a technique to symbolically represent the state space in a datastructure called a Binary Decision Diagram (BDD). Advantages of BDD's are that they can represent a set of states in a compressed form and efficient in-place operations to manipulate the BDD exist. A BDD is a directed acyclic graph where each node has two outgoing edges. A node represents a boolean variable, and its outgoing edges represent true and false. Leafs in the BDD have the value 0 or 1 and have no outgoing edges. One traversal of the graph ending in a leaf with value 1 represents one reachable state of the underlying transition system. LTSmin [22] is an example of a toolset that uses symbolic model checking to generate state spaces. Symbolic Model Checking has also been developed for probabilistic systems [23]. Instead of using BDD's, probabilistic symbolic model checking uses MTBDD's. These differ from traditional BDD's by their leaf nodes, in that they can have an arbitrary value between 0 and 1. This value represents the probability that a state in the transition system will be reached. PRISM [5] is a model checking toolset that uses this technique.

Partial Order Reduction Software is usually executed asynchronously. It often consists of multiple components that are executing independent of each other. Formalisms such as process algebras model systems as interleaved sequences. This introduces multiple interleavings and often it does not matter which interleaving is taken, as the result is the same. This is the basic idea behind Partial Order Reduction (POR) and note that this idea is similar to the ideas behind confluence. Whenever multiple interleavings lead to the same result, we can as well pick a representative and do not allow other interleavings. This can significantly reduce the state space. POR and confluence are based on the same ideas and are basically only syntactically different. However, confluence is a little more generic and therefore a bit more powerful in some situations [24]

Many variants exist for both non-probabilistic [25, 26, 27] and probabilistic systems [28, 29, 30]. These variants mostly differ by the formalisms used to express the state space. Also, they differ by which logic, such as CTL*\X and LTL\X, they preserve.. Some handle fairness assumptions [28]. For example, it is quite common to assume that whenever some concurrent process is enabled then it will eventually be allowed to execute an action. Also, variants differ in whether they reduce the state space after generation or on the fly. POR has also been extended to work with symbolic model checking [31].

The main reason we use confluence in this research is because we are dealing with Markov Automata and POR has not been generalized to Markov Automata yet. Also, in theory we should be able to achieve larger state space reductions using confluence [24].

8.2. Case Studies

The main goal of the proposed research is to analyse whenever the reduction techniques implemented in SCOOP have a strong effect. Several case studies have been done already over the last few years. A leader election protocol [11] has been tested, showing state space reductions of 90-95%. Also, the state space generation time has been reduced to at most one fourth of the original time. This case study has been applied on a prCRL specification, which is a subset of MAPA. Markovian transitions are not present in prCRL.

In [10] three case studies have been done to illustrate the strength of confluence reduction by itself.

- The same leader election protocol as mentioned before
- A queuing system [1]
- A GSPN model of a 2x2 processor architecture [32]

The leader election protocol showed a reduction of 80% in number of states and around 90% in state space generation and analyis time. The other cases showed slightly weaker reductions varying from 30-70% in both number of states and generation time.

For other tools similar to SCOOP and IMCA lots of case studies have been performed already. PRISM [5] for example has been used for both MDP and CTMC models. For MDP-models, several randomised distributed algorithms have been modelled including the randomised mutual exclusion protocols of [33] and the randomised consensus protocol of [34]. Also a large number of CTMC models have been considered. Examples include a cyclic polling system [35] and a tandem queueing network [36].

In [37] an overview of case studies for the μCRL toolset is given. These include the following:

1. A cache coherence protocol
2. Common electric purse specifications (a protocol for electronic payment using a chip card as a wallet)
3. A clinical chemical analyser [38] (used for automatically analysing patient samples)
4. A digital rights management system
5. Solving instances of one-player maze puzzles of Sokoban.

For the CADP toolset [4] more than 150 case studies have been performed to this date. We refer the reader to the CADP[1] website for an overview of all case studies.

[1] Available at http://www.inrialpes.fr/vasy/cadp/case-studies

Part III.

Case Studies

CHAPTER 9
METHODOLOGY

In this section we describe three things. First, we show the different stages of the case study process, starting at an informal problem description and ending at the analysis of the results. Second, we show what results we want to obtain and how we give our results a scientific base. Last, we show how case studies can be repeated easy using automated tools.

In the subsequent chapters we describe the different case studies. For each case we give some background information, including the specification of what is modelled. We also show why these cases are relevant for our research. For a case study to be relevant it fully utilizes the possibilities of Markov Automata and MAPA. Hence, nondeterminism, rates and probabilities are present in all our models. To illustrate the strengths of MAPA the usage of data is also relevant, but if one or two cases lack this property, this is not a problem. After we have shown why a case study is relevant, we continue by describing the design we have chosen for it. We then perform our case study (of which the process is described shortly) and provide the reader with the obtained results. We analyse these results and explain notable results. We conclude this part with answering the research questions from Chapter 2.

9.1. Case Study Process

The case study process consists of a number of steps. In Figure 1.1 from Chapter 1 we already showed how the SCOOP toolchain works. In order to conveniently perform the case studies we extended this chain with some new components. This is shown in Figure 9.1. We start by informally describing the problem. This description is then formalized in a MAPA model. This model is automatically generated (see Section 9.3). Then SCOOP is run on this model, which includes linearising the MAPA model into an MLPPE, applying the simplification and reduction techniques and generating the state space. This gives us the first part of the results, such as state space generation time and memory usage by SCOOP.

As can be seen in the previously mentioned figure, the state space can be analysed by a number of tools, including CADP, PRISM and IMCA. We chose to use IMCA, which is briefly described in Chapter 7 as it can be used to verify all properties we require (unbounded reachability, expected time and long run average) and it supports Markov Automata. However, in order to verify

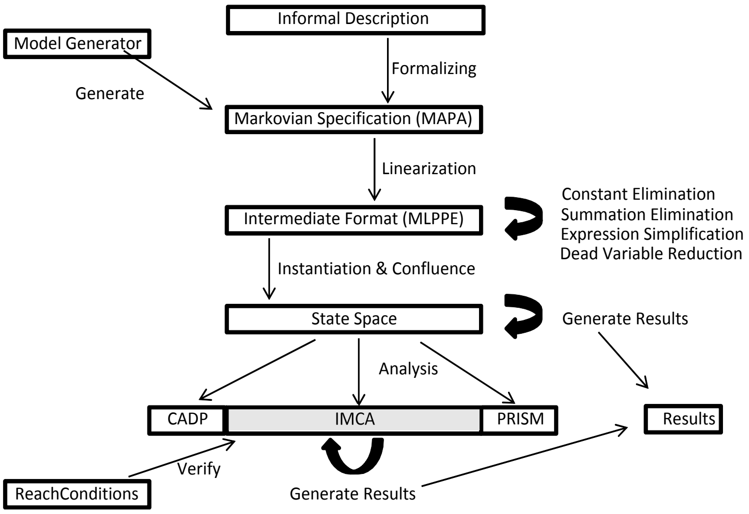


Figure 9.1.: Extended Overview of the case study process

properties, we first need to formalize these. To do that, we use the IMCA-support that is built into SCOOP. In a MAPA model we can express which states are target states. IMCA can then apply unbounded reachability, expected time and long-run average to the state space and target states to verify our properties and do calculations on the model.

9.2. Measurement Setup

In order to measure the impact of the different techniques used by SCOOP we measure both the state space generation time and state space size for the following settings:

1. No Simplification, No Dead Variable, No Confluence
2. Only Simplification Techniques
3. Simplification & Confluence
4. Simplification & Dead Variable Reduction
5. Simplification, Confluence and Dead Variable Reduction

The first setting is our base case. Performing the case studies without simplification techniques often takes a long time. Therefore, we chose to use the simplification techniques in the other 4 tests. So the impact of the simplification techniques will be compared to setting 1 and the impact of the reduction techniques will be compared to setting 2.

For each of these settings we do the following measurements and compare them to their base case:

- Model Size (number of states)
- State Space Generation Time (time used by SCOOP)
- Maximal Memory Usage
- Model Checking Time (time used by IMCA)

In order to give our results a scientific base, we used the following principles [39].

- **Repeatability and Verifiability**: The experimental setup is such that if somebody else wants to repeat the experiments, he is able to and will achieve the same results.

- **Statistical Significance**: Experiments have been repeated at least 3 times. If the results were similar 3 times, we took the average as our results. If there were big differences between experiments ($> 5\%$) we repeated them more often and removed odd results.
- **Encapsulation**: The experiments did only measure what we claimed to measure (generation time and memory usage). No other influences, such as background processes, garbage collection and network traffic, were allowed. We achieved this by using a dedicated machine (not connected to a network) and by carefully measuring the time and memory usage of SCOOP only. In this way the effect of disturbing factors is avoided.

The experiments were performed on a Ubuntu 12.04 Virtual Machine with 1 processor available and 1GB Memory. The host system is a Dell XPS15 Laptop with Intel Core i5 M560 @ 2.67GHz and 4GB memory. The operating system is Windows 7 64bit. This is supported by SCOOP.

9.3. Tool Automation

In order to make our case studies easily repeatable we automated 3 parts of the case study process. First, the generation of the MAPA-models is automated since it depends on several parameters (eg: number of nodes and gossip-items for the gossip model). Second, the collection of the results from the state space generation (SCOOP) is automated. Last, the results from the IMCA-analysis and the SCOOP results are combined into a log-file. For these 3 aspects we wrote 3 Java-programs and in combination with some basic scripting we can easily execute one or several case studies with one script. We use the script for running the consensus algorithm as a running example of how the tooling works.

Model Generation
All our MAPA-models need to have an adjustable size. This often has a large impact on the code, for example on the parallel composition when you do not know in advance how many of a certain process there are. With just MAPA we cannot determine this automatically. Therefore, we chose to generate the models in Java. We provide the necessary parameters to the program and it generates the MAPA-model (including IMCA specification) for us.

Collection of SCOOP-results
SCOOP outputs two relevant things for us. First, we have the output that serves as input for IMCA. Second, we have the output that is relevant for the log-file (such as state space size). OutputProcessor is a tool that makes two files for each of these outputs. It takes one parameter, which states whether confluence is used. Value 9 means confluence is not used and value 10 means confluence is used).

Log File Generation
For one case study to be performed, one needs to run SCOOP 5 times (for each of the different settings). The LogGenerator program combines all output from SCOOP and IMCA, extracts the relevant information and writes it into a log-file. Finally the temporary files are cleaned up and the script terminates.

Combining Tools
The script simply combines the tools described earlier with the toolchain from Section 9.1. In additional it uses the usr/bin/time program to do the time and memory measurements on both SCOOP and IMCA. This program writes its output to standard error, whereas SCOOP writes to standard output. This is very convenient, as we can easily separate the output of both tools. The first lines of the script describe where SCOOP and IMCA can be found in the system. Also,

by default some settings such as verbose mode are used. One should not modify these settings, otherwise the log-file might not be generated properly.

Expansion
One can easily expand this system with more case studies. The only thing one needs to do is write a write a generator for the model and adapt the script to suit the new model (add the right parameters and call the right generator program).

CHAPTER 10 GOSSIP PROTOCOLS

Gossip protocols are an attractive solution for searching for and distributing information in large systems or networks. The large scale behaviour of a gossip protocol is not easily predictable, increasing the demand for their analysis in a systematic way, e.g. using formal methods. Recently, Bakhshi, Gavidia, Fokkink and van Steen have been working on the formal analysis of gossip protocols [40, 41]. They proposed a modelling framework for gossip based information spread. Several case studies have been performed using this framework, including the Newscast and Shuffle protocols. Verified properties include (after each gossip round) the total number of copies of a certain item d in the network (replication property) and how many nodes in total have seen d over time (coverage property). These properties have been validates using simulations in a round-based fashion.

10.1. Background

A wise man once said that gossiping can be best described as hearing things you like about people you don't like. Gossip spreads like a fire, making it interesting to investigate its behaviour and use it in a computer network. That is exactly what a gossip protocol [41] does. A gossip protocol is a means by which computers can communicate in a way similar to how gossip spreads in social networks. These protocols can for example be used to find items in a large datacenter. This is especially useful whenever the underlying network has an inconvenient structure or is very large. The term epidemic protocol can be used as a synonym for a gossip protocol, because gossip and viruses spread throughout a community in a similar way. As an illustration of how a gossip protocol works consider the following example.

Example 22 (Gossip at the Office)
A group of office workers meets each other every break at the coffee machine. Each employee pairs with a random coworker and talks about the latest gossip. Assume that Anna has a rumor and talks about this with Bob. In the next break Anna gossips with Chris and Bob gossips with Danny. The amount of people knowing about the rumour initially roughly doubles. After a few breaks this increase gets smaller, as people are paired such that both already know about the rumor. However, very quickly everybody at the office knows about the rumor.

Listing 10.1: Active Thread

```
wait($\Delta t$ time units)
$p$ = RandomPeer();
$m$ = PrepareMsg();
send $m$ to $p$;
wait until receive($m_p$);
$c$ = Update($c$,$m_p$);
```

Listing 10.2: Passive Thread

```
wait until receive($m_p$);
$m$ = PrepareMsg();
send $m$ to sender($m_p$);
$c$ = Update($c$,$m_p$);
```

The idea behind this information spread can be implemented in a computer network [42] and this is very useful. Assume we are working in a very large network of computers and we want to search for an item that we know is somewhere in the network. We don't know the size of the network but we do know that the computers are linked to each other and that each computer is running a program that implements a gossip protocol. To start the search for the item the user simply has to ask one computer to start the search. Periodically each computer picks some other computer at random and gossips with it. This other computer then starts the search as well. In the next round both computers pick another computer and gossip again. In this way the search term spreads throughout the network very quickly. The result can then be retrieved by again gossiping about the best result so far. These results will also spread through the network and reach the local computer eventually.

The time it takes to find an item in a fully connected network using gossip protocols is logarithmic in the size of network [40]. For example, in a network with 25000 computers it takes about $^2\log(25000) \approx 15$ rounds of gossip to spread the search string and about 15 rounds of gossip to retrieve the answer. A gossip frequency of once every tenth of a second is not unrealistic, hence in this case a search in a big datacenter could be completed in about 3 seconds (assuming that the local search doesn't take very long and the network doesn't get congested).

10.2. Specification

For modelling the algorithm in MAPA we use the framework presented in [40]. This framework can be used for many protocols, including the Newscast protocol [43] that we will model. We first describe the framework and in the next section present the protocol that we model.

This framework works with large networks in which nodes interact in a peer-to-peer style. The nodes in the network have a common agreement on the frequency of gossiping. Each node has a local cache in which it stores its data. It executes two different threads, an active and a passive one. The active thread periodically initiates a contact with a neighbour and sends him a subset of the data in its cache. Such an exchange message with its content is called an *exchange buffer*. The passive thread waits until it gets contacted and then receives the data. It responds with sending some of its own data. Afterwards, both nodes update their cache based on the current and received data. We refer to the active node as the *initiator* and to the passive node as the *contacted* node. We assume we have a topology in which all nodes are connected.

The general protocol is shown in Listings 10.1 and 10.2. This is a push-pull protocol because the nodes both push the data to other nodes (active thread) and pull data from other nodes (passive thread). Other variations, such as push-only and pull-only exist and the framework supports it.

The method RandomPeer selects a random neighbour based on the structure of the underlying network graph. The method PrepareMsg() selects a random set of items from the local cache and puts them into an exchange buffer. The Update() method updates the local cache depending on the data that was already known and the newly learned data.

We can model one data exchange as a transition system. Such a system consists of four states. There is one state in which both do not have the data, two in which either node has the data and another one in which both have the data. Transitions between states model changes after the exchange. They are labelled by the respective transition probabilities. These probabilities depend on two building blocks.

- P_{select} is the probability of an item to be selected by a node from its cache to be gossiped.
- P_{drop} is the probability of an item to be replaced by another item that is received by a gossip.

These probabilities are functions depending on the number of items n, the exchange buffer size s and the cache size $|c|$. We assume that $0 < s \leq |c| < n$, i.e. not all items can be stored in the local cache and we cannot gossip more items than there are items in the cache.

The actual protocol that we will model is a simple push-pull information propagation protocol called Newscast [43]. The basic idea of this protocol is that periodically nodes in the network exchange data items. These items can be, for example, a number or the network address of a node. We should also take into account that nodes can fail, for example by a local network problem.

The protocol works in a wide area network, meaning that each node is connected to a lot of other nodes. Network latencies therefore don't have to be taken into account. Every round each node sends s random items to a random other node. The update method randomly chooses $|c|$ items and puts them into the cache.

The main goal of this case study is to model it in MAPA and apply the reduction techniques to measure their impact on the state space generation time. Initially the cache of each node should be populated by a number of items and the model should simulate how the data propagates through the system up to a maximum number of rounds. This should quickly result in a very large state space, as data can travel in a lot of different ways.

In summary, the model consists of the following components:

- All states should contain the current knowledge of each agent. With this we mean the current gossiping items it has knowledge of. We should at all times be capable of asking an individual node whether it knows about a certain gossiping item.
- Actions and communication that describe how information spreads through the network.

Relevance This is a relevant case as it contains all the main aspects of Markov Automata. Possible message loss will be modelled using probabilistic transitions. The system is also nondeterministic by nature, for example in the way the neighbour and the send data are picked. Some time will pass between gossiping rounds, which can be approximated by an exponential distribution. One of the strengths of MAPA, handling data well, can also be illustrated, as actual data is travelling throughout the network.

Verified Properties For this model we can model check several properties. In this section we propose two.

Listing 10.3: Gossip Models for 3 nodes

```
Node(data:Queue,available:Bool,emp:Bool, completed:Bool) =
!emp & available => <2> . ∑_{s:0..2} ∑_{o:1..3} o!=my_id & available = T → send(my_id,o,get(data,s)) . Node[]))
++ available => ∑_{r:0..3} ∑_{o:1..3} receive(o,my_id,r) . Node[if(r>0) then data[r-1]:=r else data:=data, if(r>0) then emp:=T else emp:=emp]))
++ available=T => <0.01> . Node[available := F]
++ available=F => failing . Node[]
++ available=T & completed=F & data=1;2;3 => finish . Node[completed:=T]

init Node(my_id=1)[1;2;3,T,F,T] || Node(my_id=2)[0;0;0,T,T,F] || Node(my_id=3)[0;0;0,T,T,F]
comm (send, receive, transfer)
encap send, receive
hide transfer, failing, complete, finish
reachCondition completed_2 & completed_3
```

The main property the model has to fulfill is that it is possible for all nodes to know about the rumor. This has nothing to do with probabilities or rates, so we can easily express this in a CTL formula

$$\Diamond \forall x, y \cdot Node(x) \wedge Rumour(y) \rightarrow Knows(x, y)$$

This states that no matter what path you take ($\forall$), eventually ($\Diamond$) all nodes and rumours are matched by a Knows relation, i.e. every node knows about every rumour eventually.

Another property we want to verify is whether the model is capable of propagating the data quickly enough. Even though all nodes may eventually know about all gossips, whenever this takes a very long time it is not very useful. So we take a maximum time t and express in CSL that after that time t there is still a node and gossip item that are not matched by the Knows-relation.

$$\Diamond^{>t} \exists x, y \cdot Node(x) \wedge Rumour(y) \wedge \neg Knows(x, y)$$

This property states that at some time after t there is a Node x and a Rumour y and x does not know about y at that time. These two properties will be model checked using IMCA.

10.3. Design

We implemented a gossip protocol in MAPA. The model and specification can be found in Listing 10.3.

Our model consists of Node processes. Each node process consists of several variables, of which *data* is the most important one. It has Queue as its type, as this is the only set-like data type available in MAPA. The goal of this model is to fill the *data* variable of all nodes with the values 1 to n (where n is the number of gossiping items in the model). Initially the data buffer of one node gets fully populated with values 1 to n and the other nodes are initially empty. An 'empty' gossip-information field is represented with value 0. Besides *data* a node process has several other variables. *available* is a boolean parameter that specifies whether the node has crashed. *emp* is a boolean variable that states whether the queue is currently empty (this is the case if it only contains 0's). This variable was necessary to prevent nodes from sending items whenever their buffer does not contain any items. Last, we have a parameter *completed* which is used for the IMCA specification and basically is true whenever the *data* variable is fully populated.

A node can do a couple of things. First, a node can send some of its information to another node (line 2). It does not matter whether this is real or 'empty' information. In case it sends empty information over the network the receiving node will simply ignore it. In case it sends real information over the network the receiving node will write it into its data queue. Nodes cannot constantly send items. The time between two send actions is exponentially distributed with $\lambda = 2$. This means that the expected time between two gossiping rounds is $\frac{1}{\lambda} = \frac{1}{2}$. In contrast to sending, a node can receive information at all times (line 3). It scans whether the information is not empty (its value is larger than 0) and writes the data to the right place in its own buffer. Furthermore, a node can crash (line 4). The time until crashing is exponentially distributed with parameter 0.01. After a node crashes it becomes unavailable and cannot send or receive items anymore, it simply keeps failing (line 5). Note that in the model we have another line (line 6). This is part of the specification that is validated using IMCA and specifies that *data* has been fully populated while the node hasn't crashed.

The specification brought up some issues. Whenever we run SCOOP on the original model we see significant state space reduction whenever we use dead variable reduction. With the specification added stating that all *data* queues should be fully populated the reduction was gone. It turns out that SCOOP adds a new summand to the MLPPE with exactly the condition from the specification and then an action IMCA can use. However, *data* now occurs in an enabling function and thus can never be dead. Originally it is dead after the node crashes as it will never be used anymore. Therefore we needed a slight change to the model. We introduced a new summand that can be executed once per process. Whenever data is fully populated it executes a complete action and sets the parameter completed to true. The reachCondition statement now specifies that all 'completed' variables should be true for the gossiping process to be completed.

10.4. Results

We ran SCOOP and IMCA using a couple different settings for the Gossip model. Gossip-x-y is a specification for a gossip model with x nodes and y gossip items. For each specification we measure the number of states, the running times for both SCOOP and IMCA and the peak memory usage (combined). The time reduction is specified for SCOOP and IMCA combined.

	Original State Space				Optimized State Space				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gossip-4-3	1280	10.93	0.09	372KB	1280	0.64	0.06	372KB	0%	94.1%	0%
gossip-4-4	24576	799.08	16.73	5352KB	24576	20.93	16.82	5352MB	0%	97.4%	0%
gossip-5-3	5120	60.45	0.78	1476KB	5120	2.97	0.79	1476KB	0%	95.1%	0%
gossip-6-3	20480	331.57	12.38	4465KB	20480	13.80	12.35	4465KB	0%	95.8%	0%
gossip-7-3	81920	1725.42	215.30	17734KB	81920	64.34	215.36	17734KB	0%	96.3%	0%

Table 10.1.: State Space generation and analysis with MLPPE simplification techniques. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (Conf)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gossip-4-3	1280	0.64	0.06	372KB	1280	0.88	0.07	372KB	0%	-37.5%	0%
gossip-4-4	24576	20.93	16.82	5352KB	24576	27.69	16.73	5352KB	0%	-32.3%	0%
gossip-5-3	5120	2.97	0.79	1476KB	5120	3.95	0.79	1476KB	0%	-33.0%	0%
gossip-6-3	20480	13.80	12.35	4465KB	20480	18.55	12.50	4465KB	0%	-34,4%	0%
gossip-7-3	81920	64.34	215.36	17734KB	81920	85.22	215.65	17734KB	0%	-32.5%	0%

Table 10.2.: State Space generation and analysis using confluence reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (DVR)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gossip-4-3	1280	0.64	0.06	372KB	800	0.53	0.02	246KB	37.5%	17.2%	33.9%
gossip-4-4	24576	20.93	16.82	5352KB	13056	14.81	3.40	3057KB	46.9%	29.2%	42.9%
gossip-5-3	5120	2.97	0.79	1476KB	3136	2.34	0.23	994KB	38.8%	21.2%	32.7%
gossip-6-3	20480	13.80	12.35	4465KB	12416	11.02	3.11	2909KB	39.4%	20.1%	34.8%
gossip-7-3	81920	64.34	215.36	17734KB	49408	51.32	50.46	10696KB	39.7%	20.2%	39.7%

Table 10.3.: State Space generation and analysis using dead variable reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (All)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gossip-4-3	1280	0.64	0.06	372KB	800	0.72	0.02	246KB	37.5%	-12.5%	33.9%
gossip-4-4	24576	20.93	16.82	5352KB	13056	20.33	3.39	3057KB	46.9%	2.2%	42.9%
gossip-5-3	5120	2.97	0.79	1476KB	3136	3.17	0.23	994KB	38.8%	-6.7%	32.7%
gossip-6-3	20480	13.80	12.35	4465KB	12416	14.97	3.19	2909KB	39.4%	-8.5%	34.8%
gossip-7-3	81920	64.34	215.36	17734KB	49408	68.98	50.30	10696KB	39.7%	-7.2%	39.7%

Table 10.4.: State Space generation and analysis using the full power of SCOOP (Simplification, Confluence and DVR). Runtimes in SCOOP and IMCA are in seconds.

10.5. Analysis

In the tables we notice a couple of interesting results.

1. Dead Variable Reduction reduces the state space significantly.
2. Confluence is not very useful for this case.
3. The simplification techniques have a huge impact on the state space instantiation time.

For each of these results we analyse their nature.

Dead Variable Reduction

We see a huge impact whenever we use dead variable reduction. It turns out this is because the value of *data* will be dead after a node crashes. Its value will never be used afterwards and thus can be replaced by the initial value. So where a node could become unavailable at first with all possible values of *data* it now can only get unavailable where $data = data_{init}$. Note that we could have obtained this 'reduction' without using Dead Variable Reduction by resetting the value of *data* manually after a crash.

Confluence

Why one would expect that confluence would reduce the state space: If you would simply look at the problem you could say that it does not matter in which order data propagates through the system. Whether first A sends data to B and then C to D or vice versa would result in the same state. That holds for all orders. So if confluence would prioritize to some of these τ-steps, this could greatly reduce the state space. What you would achieve is an abstraction of the system in which it does not matter which data each node holds. It now remembers that there is for example one node with queue 1,2 and one node with queue 0,1 instead of that these queues are strictly tied to the node number.

Why this doesn't work: There are two main reasons why the above reasoning doesn't work. The first is that nodes can send 'fake' data if they have nothing interesting to say. The receiving node would ignore this data and this results in a selfloop. For this action to be confluent it needs to commute with *all* other outgoing transitions and both interleavings should result in the same state. This last requirement is not fulfilled. If a is an outgoing transition it sends data to a node that it did not know before (otherwise the state wouldn't change). So initially sending this piece of 'fake' data resulted in a selfloop, but now that the data is real it actually changes the state. This is illustrated in Figure 10.1. We assume for a system with three nodes and one piece of data. The state (1,0,0) describes that node A has the data, but node B and C do not. We can easily see that both interleavings of the two available actions do not lead to the same target state, although by looking at the informal specification one might expect that both interleavings will lead to the same state.

The second reason why the above reasoning doesn't work is because the transfer of data is not the only thing that happens in the system. In fact, between two transfer actions at least one rate $\langle 2 \rangle$-action has occurred. This is the case because after a $\langle 2 \rangle$ action we have the choice between another $\langle 2 \rangle$ action (from another node) or a transfer action. Due to the maximal progress property the transfer action can never happen, making it impossible to make a second transfer action available before the first one was executed. Due to the nature of exponential distributions it is impossible

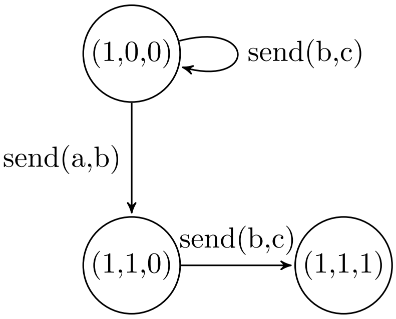


Figure 10.1.: Why Confluence Doesn't Work Here

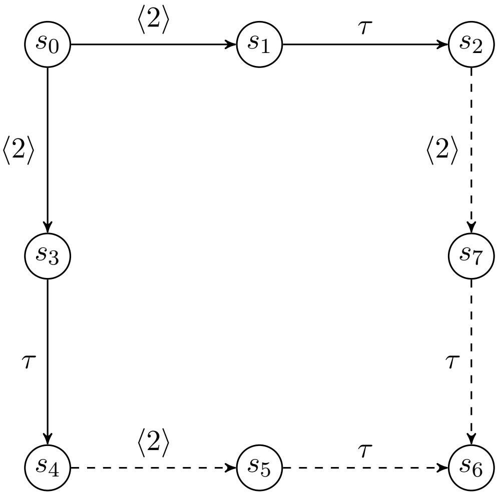


Figure 10.2.: Sequence Confluence Concept

that two rate actions occur exactly at the same time. So never two transfer actions are enabled at the same time. The result (for the time neglecting other actions such as failing and completing) is that we get an alternating sequence of rate ⟨2⟩ and transfer actions. This brings up an interesting idea. We get a situation as shown in Figure 10.2. This is a situation somewhat similar to confluence with the only difference that a sequence of actions should be marked confluent instead of just one action. This idea will be further described in Chapter 13, but implementing it into the theory remains as future work.

Simplification

We see a large decrease in state space generation time whenever we use the simplification techniques. As the impact of the simplification techniques is very different between the 3 case studies we will compare the models in Chapter 13 and explain why these techniques work very well for this model and much less for the other models.

CHAPTER 11 TRADE MARKET

Gazillionaire Deluxe is an electronic strategy game, developed by Lavasoft in 1996. We extract the trade market component from this game and turn this into a case study. We know that this is not a very common case study, as nobody has been working on it thus far (to our knowledge). However, we show that a model of this trade market contains all aspects of Markov Automata and thus is useful to show the strengths and weaknesses of SCOOP. We admit that verifying properties on these models might not be so useful, but as this is not our main research goal this is not a big issue.

11.1. Background

The game is set in a galaxy consisting of seven different planets. Eight merchants (the players) travel through the galaxy trying to be the first one to obtain a gazillion (made up term, but a lot) dollars. They do so by trading goods, transferring passengers between planets, advertising, being active on the stock market and much more.

A core component of the game is the price market. In this case study we focus on this price market. The game contains about 25 different types of goods. These goods are differently priced on different planets and these prices also fluctuate very quickly. The goal of the game is to buy and sell products and make a lot of profit on them.

The game is turn based, although each players turn occurs at the same time. Every turn the player gets the opportunity to buy/sell goods at their current location (a planet). The player can do all the trading and some other actions (which are irrelevant for this case) in any order they like and then travel to the next planet of their choice. The next turn then takes place on that planet.

During travelling some random events can occur. One of these random events is a market crash or overflow. After these events the prices are elevated drastically or lowered to a minimum. Other random events are not considered.

11.2. Specification

In this case study we present a model of Gazillionaire. However, if we would model the entire game the state space would grow practically infinite (amount of money) very quickly. Therefore, in this section we specify a greatly simplified version of the game. In order to make the size of the model customizable certain aspects of the game, such as the number of players/planets/goods/price values, have been made configurable.

We abstract away from the actual game and focus solely on the trade market in this case study. Only the distribution of products between players and planets (and their price changes) are modelled. The prices have a discrete value. We chose a 25% chance that the price increases and a 25% chance that the price decreases one level between two turns. This is completely arbitrary, but can be easily changed if desired. Prices may never increase/decrease more than one level at a time, except after a random event. There is a 1% chance for such a random event. This can either be a market overflow (lowering all prices to the minimum) or a product shortage (increasing all prices to the maximum).

The players behave according to a simple AI. On their current planet they buy a random product. Then they will travel to the next planet (they go around in a circle) and again sell a random product. From the point of view of the price market this is a valid strategy, as the price market is independent of the strategy the players use. Any product can be bought at any time (even if the price is insanely high) and the model facilitates this.

The travelling time between planets is exponentially distributed. It would be nice to let the delay parameter depend on the distance between planets, but as this increases the complexity of the model significantly we have chosen for a fixed distance between each pair of planets. The result of this exponential distribution is that the turns for players never take place at exactly the same time, but as this is also the case in the original game this is no problem.

Money doesn't have to be modelled as this can blow up the state space greatly. It may be assumed that all players are rich merchants who have more than enough money to buy everything they need. Concretely what is modelled are the following aspects:

- The price market. How do the prices for each product on each planet change?
- Players who buy and sell products and travel between planets.
- Random events (overflow or shortage).

Relevance
A model for this case contains all major aspects of Markov Automata. The travelling between planets costs time, which can be modelled by Markovian transitions. The price fluctuations on the market can be perfectly modelled by probabilistic transitions. Nondeterminism naturally occurs, as nondeterministically we choose a product to buy on each planet. The system also contains lots of data (the different products and their locations), which can be very well modelled in MAPA

Measurements
As we have greatly simplified this game, there is no real goal anymore. Therefore it is hard to specify interesting properties. However, the main goal of this research is to measure the impact of SCOOPs reduction techniques on the state space. Therefore, we should give IMCA some property to verify. We propose to measure the percentage of time that a certain product has the minimum

price. Formally we can specify this as follows, assuming that the minimum price is 1.

$$Product(x) \wedge Price(x, 1)$$

This property is verified using IMCA, using long run average analysis.

11.3. Design

Listing 11.1: Trade Market Model

```
Player(currentPlanet:{1..2}, currentProduct:{1..1}, payedPrice:{1..1}, finished:
    Bool) =
finished => send_price_request(currentProduct,currentPlanet) . sum(n:{1..1},
    receive_price_answer(n) . (
  n<=payedPrice => (sell_product(currentProduct) . ∑_{b:1..1} buy_product(b) .
      send_price_request(b,currentPlanet) . ∑_{p:1..1} receive_price_answer(p) . <2> .
      Player[currentPlanet := next(currentPlanet), currentProduct := b, payedPrice
      := p])
  ++ n>=payedPrice => (no_trade . <2> . Player[currentPlanet := next(currentPlanet
      )]))))

Planet(prices:Queue, finished:Bool) =
finished=F => sum(p:{1..1}, receive_price_request(p,id) . send_price_answer(get(
    prices,p-1)) . Planet[])
++ finished =F => <3> . ∑_{k:0..0} prices_change . ∑
  25/100 -> Planet[if(prices(k)<1) then prices(k)++]
  ++ 25/100 -> Planet[if(prices(k)<1) then prices(k)--]
  ++ 48/100 -> Planet[]
  ++ 1/100 -> Planet[prices := 1]
  ++ 1/100 -> Planet[prices := 1])

function next = (1 -> 2, 2-> 1)
init Player[1,1,1,F] || Player[2,1,1,F] || Planet(id=1)[1,F] || Planet(id=2)[1,F]
comm (send_price_request, receive_price_request, price_request), (
    send_price_answer, receive_price_answer, price_answer)
encap send_price_request, receive_price_request, send_price_answer,
    receive_price_answer
hide price_answer, price_request, buy_product, sell_product, prices_change,
    game_finished, no_trade
reachCondition get(prices_3,0)=1
```

If we look at the specification (which is shown in Listing 11.1) we see that there are two types of processes. Planets are processes that handle the price change of products on that planet. They also handle requests from players asking the price of a certain product. Player processes handle the buying and selling of products of a player and the travelling between planets.

A Planet has a Queue in which it keeps track of the current prices on that planet. Whenever it receives a question from a player about the price of a product, it looks that product up in the queue and returns it. We have chosen an exponential delay with $\lambda = 3$ for the time between price changes. After this delay 5 things can happen to the price of **one** product.

1. With probability 25% the price rises. The only requirement for this to happen is that the products are not yet at their maximum possible level.

2. Similarly with probability 25% the price drops one level whenever its price is not yet at the minimum possible level.
3. With probability 1% **all** prices rise directly to the maximum level (marked shortage).
4. With probability 1% **all** prices rise directly to the minimum level (marked overflow).
5. In the other 48% of the cases the prices remain as they are

A Player process has 3 parameters, the current planet the player is on, the current product the player has on board and the price it payed for that product. Whether a player does a trade depends on the price of the current product on the current planet. Therefore it sends a price request. Whenever this product costs at most the price that the player payed for it, it will be sold. Then it buys at random a new product and travels to the next planet (with delay 2). The process variables are updated accordingly.

In case the price of the current product is higher than the price payed for it, no trade will happen this turn and the player immediately travels to the next planet. In case the price is exactly the same as the payed price the Player has a nondeterministic choice whether it will sell or not.

11.4. Results

We ran SCOOP and IMCA using a couple different settings for the Gazillionaire model. Gaz-w-x-y-z is a specification for a Gazillionaire model with w planets, x players, y products and z price levels. For each specification we measure the number of states, the running times for both SCOOP and IMCA and the peak memory usage (combined). The time reduction is specified for SCOOP and IMCA combined.

	Original State Space				Optimized State Space				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gaz-4-1-1-1	3040	2.38	0.16	927KB	3040	1.83	0.17	927KB	0%	23.1%	0&
gaz-5-1-1-1	20768	23.88	5.70	4496KB	20768	18.14	5.72	4496KB	0%	24%	0%
gaz-6-1-1-1	140032	230.27	-	-	140032	178.05	-	-	0%	22.7%	-

Table 11.1.: State Space generation and analysis with MLPPE simplification techniques. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (Conf)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gaz-4-1-1-1	3040	1.83	0.17	927KB	2432	3.07	0.13	767KB	20.0%	-67.8%	17.3%
gaz-5-1-1-1	20768	18.14	5.72	4496KB	17968	27.95	4.81	3890KB	13.5%	-54.1%	13.5%
gaz-6-1-1-1	140032	178.05	-	-	127168	264.14	255.86	-	9.2%	-48.4%	-

Table 11.2.: State Space generation and analysis using confluence reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (DVR)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gaz-4-1-1-1	3040	1.83	0.17	927KB	3040	1.94	0.16	927KB	0%	-6.0%	0%
gaz-5-1-1-1	20768	18.14	5.72	4496KB	20768	18.39	5.72	4496KB	0%	-1.4%	0%
gaz-6-1-1-1	140032	178.05	-	-	140032	178.91	-	-	0%	-0.5%	-

Table 11.3.: State Space generation and analysis using dead variable reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (All)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
gaz-4-1-1-1	3040	1.83	0.17	927KB	2432	3.07	0.13	767KB	20.0%	67.8%	17.3%
gaz-5-1-1-1	20768	18.14	5.72	4496KB	17968	28.30	5.05	3890KB	13.5%	-56.0%	13.5%
gaz-6-1-1-1	140032	178.05	-	-	127168	263.57	254.78	-	9.2%	-48.0%	-

Table 11.4.: State Space generation and analysis using the full power of SCOOP (Simplification, Confluence and DVR). Runtimes in SCOOP and IMCA are in seconds.

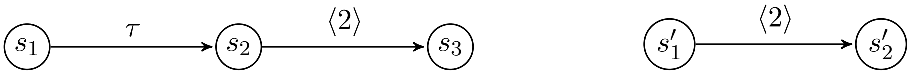
11.5. Analysis

In the tables we notice a couple of interesting results.

1. Confluence has some impact on the state space.
2. Even though confluence reduces the state space, still the generation time for SCOOP increases.
3. MLPPE Simplification reduces the state space generation time.

For each of these results we carefully analyse their nature.

Confluence
In our model there are only a few places where processes do local actions. The same holds for the sell product action in the Player process and the no trade action. Both are immediately preceded or followed by a rate action. Basically this results in the situation depicted in the left of Figure 11.1. Confluence reduces this to the situation in the right of Figure 11.1. It is a very simple case of confluence and will not reduce models in orders of magnitude. However, this simple reduction is still useful.



(a) The original system

(b) The reduced system

Figure 11.1.: Simple case of confluence

Why doesn't 'real' confluence work? Aside from the internal actions described above we only model communication between the two processes. It is never the case that two processes do independent work. The interleaving of the actions is more or less fixed and whenever we have divergence we get a similar situation as described for the gossip models. We will come back to this problem in Chapter 13.

Time increase
We see for each of the models in this section that when we use confluence, the state space is in fact reduced but still the generation time is longer. This can only mean that confluence has quite some overhead. Detecting which summands are confluent costs hardly any time, so this cannot be the problem. In the newest version of SCOOP a new flag -store has been introduced. With this flag the representant of each state is remembered. This uses more memory, but in models with lots of cycles this can save a lot of time as the representants do not need to get recalculated for every visit. The measurements done in this work do not use this flag. However, we tried whether the time increase can be resolved by using this flag. It turns out that this has some effect, but still about half of the increase remains, even with the flag. Even though the state space generation time increases, we can also state that the time IMCA needs is slightly reduced. However, the combined time increases and as the memory usage is not a problem at all, we see that using confluence actually makes generating the state space worse. This raises the question whether confluence has been inefficiently implemented or that the theory is simply very difficult. We leave the question whether we can do better as future work.

Uitleggen waar het vandaan komt en of het opgelost kan worden met -store optie in de nieuwe versie

Simplification
We see a slight decrease in state space generation time whenever we use the simplification techniques. As the impact of the simplification techniques is very different between the three case

studies we will compare the models in Chapter 13 and explain why these techniques work not so well for this (and the consensus) model and much better for the gossip model.

CHAPTER 12 CONSENSUS ALGORITHM

This case study originates from the means by which ants come to consensus over the shortest path to a food source. The way by which this happens can be implemented in a swarm of robots who need to reach consensus over the optimal option. One can think of several scenario's in which this can be applicable, for example while navigating a burning building. Krause and de Vink have started working on this scheme recently [44]. They tried validating several properties, mostly using simulation, for example using the ARGoS simulator. Properties such as dynamic adaptation to path length (explained later), efficiency (probability to succeed in reaching consensus), reset count and error rate have been measured.

12.1. Background

Populations of ants have very efficient means of selecting the shortest of two paths from the nest to a food source. Initially, ants wander around randomly, but whenever they find food they return to their colony while laying down a pheromone trail. If other ants find such a trail they are likely to follow it. Whenever the trail indeed leads to a food source the trail is reinforced. In a small amount of time the shortest path will have a very strong pheromone trail, after which all ants will follow that path.

Now consider a swarm of autonomous robots performing some activity, for example navigating through a burning building to save valuable items. It is important that the robots take the shortest paths to these valuable items, even though there might be several of those paths. We can use the way ants search for food for this. However, when we want to implement such an algorithm for a swarm of robots we need to find a physical counterpart for pheromones. In [44] the mechanism of the majority rule is proposed as a computational solution. Using this mechanism a set of autonomous robots can reach consensus on which of the two paths is the shortest. In the remainder of this section we will refer to these robots as agents.

We will first describe the mechanism for reaching consensus in a discrete setting. The process is shown in Figure 12.1. We want to reach consensus over which path is the best choice. There are 2 paths A and B with length 1 and 2 respectively. This means that it takes 1 time step to complete path A and 2 time steps to complete path B. Initially the agents have no idea which path is the

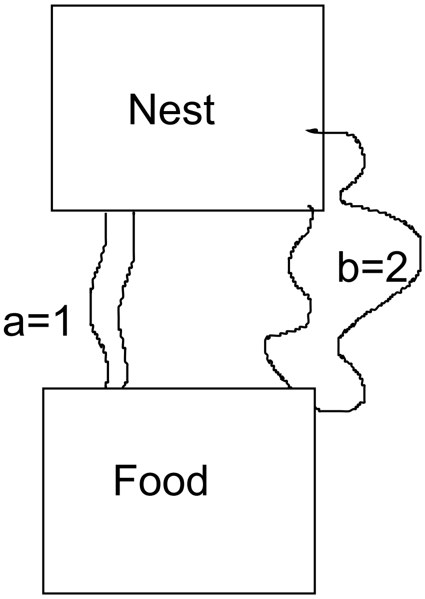


Figure 12.1.: Setting for Consensus Algorithm

best choice, so rougly 50% will have opinion A and the other 50% have opinion B. Agents can be either at the nest (sleeping) or travelling.

At each time step roughly half of the agents at the nest form groups of 3 agents. Whenever such a group has a majority of A-agents all agents om that group gain opinion A and otherwise they all gain opinion B. They then travel among that path. So in the situation described earlier the A-groups will return in the next time step and the B-groups will return in the time step after that. The reason that it is very likely that consensus is reached over opinion A is because B-agents are travelling for a longer time and thus can vote less often than A-nodes. In the next example we show how this process works.

Example 23 (Discrete Example)
Consider a nest containing 100 agents with opinions A and B. There are two paths with length $k_A = 1$ and $k_B = 2$. During each discrete time step roughly half of the agents at the nest will form groups of 3 agents. Initially there are 50 agents with opinion A and 50 with opinion B. 25 A-agents are at the nest and 25 return at $t = 0$. For the B-agents, 17 are at the nest, 17 will return at $t = 0$ and 16 return at $t = 1$.

At $t = 0$ there are 50A and 34B at the nest (some B nodes are still travelling, so they cannot form new groups). For the next trip roughly half of them form groups, so 25A and 17B agents are selected to form 14 groups. 0.6446 here. So we get 9 AAA teams (27 A-agents) to return at $t = 1$ and 5 BBB (15 B-agents) teams that return at $t = 2$. 25A and 17B stay at the nest.

At $t = 1$, there are 52A (25 left from last step and 27 return from travelling) and 33B at the nest. Again, roughly half of them form groups, so 26A and 16B are selected to form 14 groups. The expected fraction of A-groups is now 0.6794. We again get 9 AAA teams and 5 BBB teams.

This process continues until there is consensus. The first few steps are shown in table 12.1. Each row shows how many agents of each opinion there are at the beginning of the time step. It then shows how many agents stay at the nest and how many are selected to form groups. The + and - signs illustrate the change of opinion after following the groups. The column B(already) shows the ants of opinion B that are still travelling. Because $k_B = 2$ they need two time steps to return to the nest. You can easily see that it is very likely this process will reach consensus over path A, simply because there are constantly much more A-agents at the nest and thus participate in a group more often.

t	A(total)	A(stay)	A(leave)	B(total)	B(stay)	B(leave)	B(already)
0	50	25	25+2	50	17	17-2	16
1	52	26	26+1	48	17	16-1	15
2	53	26	27+3	47	17	15-3	15
3	56	28	27+3	44	17	15-3	15
4	59	31	28+2	41	15	14-2	12

Table 12.1.: Opinion change over time

Dynamic Adaptation
So far, we have focused on a static environment. However, we want the agents to be able to reconsider their preferences in a changing environment. So every once in a while, when there has been consensus for a while, we switch the path lengths. However, whenever there is consensus over opinion A for example, all groups would only contain A-agents and thus never even detect that path B is shorter now. Therefore, in order to detect that the path lengths are changed at least both opinions should stay alive. Two variants to do this are presented in [44]. In the Switch scheme there is a small probability that a unanimous team changes their its opinion after returning to the nest. We will not consider this scheme.

The scheme that we will model is called the MinPop scheme [44]. This name origins from the fact that it always keeps a minimum population of both opinions 'alive'. This makes the scheme adaptive, i.e. it can move away from consensus in reaction to a change in path length. This is done by introducing *stubborn* agents. Stubborn agents do never change their opinion, but do engage in voting. In case the path lengths change and they vote for the best alternative, they can possibly convince the other agents to also change their opinion.

Model Components In summary, the model consists of the following components:

- Group forming process
- Travelling time of groups of agents
- Stubborn agents and path length changes (optional)

Relevance A case study for this model is perfect to illustrate all the possibilities of Markov Automata and SCOOP. First of all, the agents travel which takes time. In the original MinPop model agents 'teleport' at discrete time steps, but a continuous delay would be much more realistic. Also, agents might not always travel at the same speed, so an exponential distribution could be used for this. For the forming of groups we can also make this continuous, such that whenever a group comes back to the nest, a new random group will immediately leave.

Probabilities occur at several places in the model. For example, the way the teams are formed is probabilistic. This is a perfect example of a real world scenario to illustrate the strengths and possibilities of SCOOP. Besides being able to model all aspects of the system we can also do the required measurements (next subsection) using the link with IMCA.

Verified Properties Besides the aspects we want to measure for all case studies (Section 12.3) we want to measure the following scenario specific aspects.

1. We evaluate the efficiency of the model. With this we mean the probability that a team will be sent to the destination with the shortest path.
2. We measure the adaptation time, i.e. the expected time needed to reach consensus among the agents.
3. We measure the probability that (using the MinPop model) the stubborn agents can convince the other agents after a path length change (before the path length changes again) (optional)

In order to formalize these properties we assume that we have appropriate predicates describing the state of the agents.

1. $\Diamond(\forall x.\ \text{Agent}(x) \rightarrow \text{Opinion}(x{,}A))$ (run using unbounded reachability)
2. $\Diamond(\forall x.\ \text{Agent}(x) \rightarrow \text{Opinion}(x{,}A))$ (run using expected time)
3. $(\forall x.\text{Agent}(x) \rightarrow \text{Opinion}(x{,}A) \wedge \text{Time}(t)) \rightarrow \Diamond^{\leq 2t}\forall x\ .\ \text{Agent}(x) \rightarrow \text{Opinion}(x{,}B)$ (run using unbounded reachability)

The first expression states that eventually ($\Diamond$) all agents reach opinion A. We check both the probability that this happens and the expected time it takes for it to occur. For the third property we assume that we are working with time slices of t between time slices. We express that whenever we have reached consensus over opinion A in the first time slice, then before the end of the next time slice ($\leq 2t$) we reach consensus over opinion B. We run this using unbounded reachability.

12.2. Design

In our MAPA model, which is shown in Listing 12.1 we have distinguished between two different processes, namely one for the nest and one for the ants that are currently travelling. We start by explaining the nest. This process consists of 5 parameters. *numA* and *numB* represent the number of A and B-agents at the nest. Their initial value is adjustable. We have 2 variables to count the number of A and B-agents in the next group. *currentTravel* is a variable to count how many groups are currently travelling. One needs to specify a maximum of travelling groups, otherwise each and every agent is constantly travelling and nothing changes in the model as the groups are not changing anymore.

The group forming process consists of 2 steps. First, we probabilistically select agents in the nest and add them to the group one by one. Whenever the group reaches its maximum size it is send to the travel process. The Nest process now starts forming a new group, while the other group travels with an exponential delay. After this delay the group returns and agents change their opinion to the majority opinion. The variables in the Nest process are adjusted accordingly.

The Travel process only counts the number of groups that are travelling of each opinion. The nest process already determined the majority opinion and communicated that to the travel process. The exponential delay depends on the number of groups that are currently travelling. Whenever there are more A groups travelling, the probability that one is returning soon is larger than when only one group is travelling.

We measure 2 properties. First, we measure the efficiency of the model. Path B is the worst of the two paths and the specification states that there are no more agents with opinion B. Running unbounded reachability with this specification in IMCA results in the probability that the shortest

Listing 12.1: Consensus Model for 2x50 nodes

```
Travel(numA:{0..14}, numB:{0..14}) =
numA>0 => <numA/lengthA> . send_return_a . Travel[numA--]
++ numB>0 => <numB/lengthB> . send_return_b . Travel[numB--]
++ receive_new_group(1) . Travel[numA++]
++ receive_new_group(2) . Travel[numB++]

Nest(numA:{0..100}, numB:{0..100}, groupA:{0..3}, groupB:{0..3}, currentTravel
    :{0..14}) =
groupA+groupB<groupSize & currentTravel<maxTravel => add_new_member . ∑(
  numA/numA+numB -> Nest[numA--, groupA++]
  ++ numB/numA+numB -> Nest[numB--, groupB++]
)
++ groupA+groupB=groupSize => send_new_group(if(groupA>=groupB) then 1 else 2)) .
    Nest[groupA := 0, groupB := 0, currentTravel++]
++ receive_return_a . Nest[numA := numA + groupSize, currentTravel--]
++ receive_return_b . Nest[numB := numB + groupSize, currentTravel--]

comm (send_return_a, receive_return_a, return_a), (send_return_b, receive_return_b
    , return_b), (send_new_group, receive_new_group, new_group)
encap send_return_a, receive_return_a, send_return_b, receive_return_b,
    send_new_group, receive_new_group
hide return_a, return_b, new_group, add_new_member

init Travel(lengthA=1,lengthB=2)[0,0] || Nest(groupSize=3,maxTravel=14)
    [50,50,0,0,0]
reachCondition numB_1=0 & numB_2=0
```

path is indeed the path that will be chosen. The second property states that either there are no more B-agents or there are no more A-agents, i.e. there is consensus.

12.3. Results

We ran SCOOP and IMCA using a couple different settings for the Consensus model. Consensus-a-b-c-d is a specification for a consensus model with a initial A-nodes, b initial B-nodes, a group size of c nodes, d concurrently travelling groups, and path lengths of 1 and 2 for path A and B respectively.

	Original State Space				Simplified State Space				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
consensus-10-10-3-6	757	0.36	0.08	203KB	757	0.14	0.04	203KB	0%	61.1%	0%
consensus-30-30-3-10	7011	2.15	2.52	1.84MB	7011	1.96	2.47	1.84MB	0%	8.8%	0%
consensus-50-50-3-14	19127	6.02	21.94	4.14MB	19127	6.03	22.72	4.14MB	0%	0.2%	0%
consensus-70-70-5-18	92692	36.88	17.24.81	20.07MB	92692	37.01	16.58.99	20.07MB	0%	0.4%	0%

Table 12.2.: State Space generation and analysis with MLPPE simplification techniques. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (Conf)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
consensus-10-10-3-6	757	0.14	0.04	203KB	384	0.22	0.01	106KB	49.3%	-57.1%	47.8%
consensus-30-30-3-10	7011	1.96	2.47	1.84MB	3902	2.65	0.53	1.05	44.3%	-32.5%	42.9%
consensus-50-50-3-14	19127	6.03	22.72	4.14MB	10738	8.22	6.80	2.58MB	43.9%	-36.3%	37.7%
consensus-70-70-5-18	92692	37.01	16.58.99	20.07MB	64676	52.68	8.16.22	14.00MB	30.2%	-42.3%	30.2%

Table 12.3.: State Space generation and analysis using confluence reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (DVR)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
consensus-10-10-3-6	757	0.14	0.04	203KB	757	0.15	0.03	203KB	0%	-7.1%	0%
consensus-30-30-3-10	7011	1.96	2.47	1.84MB	7011	1.90	2.46	1.84	0%	3.1%	0%
consensus-50-50-3-14	19127	6.03	22.72	4.14MB	19127	6.06%	22.04%	4.14MB	0%	0.5%	0%
consensus-70-70-5-18	92692	37.01	16.58.99	20.07MB	92692	35.17	17.31.65	20.07MB	0%	5.0%	0%

Table 12.4.: State Space generation and analysis using dead variable reduction. Runtimes in SCOOP and IMCA are in seconds.

	Simplified State Space				Reduced State Space (All)				Reduction		
Specification	States	SCOOP	IMCA	Memory	States	SCOOP	IMCA	Memory	States	Time	Memory
consensus-10-10-3-6	757	0.14	0.04	203KB	384	0.22	0.01	106KB	49.3%	-57.1%	47.8%
consensus-30-30-3-10	7011	1.96	2.47	1.84MB	3902	2.66	0.83	1.05	44.3%	-35.7%	42.9%
consensus-50-50-3-14	19127	6.03	22.72	4.14MB	10738	8.28	6.78	2.58MB	43.9%	-37.3%	37.7%
consensus-70-70-5-18	92692	37.01	16.58.99	20.07MB	64676	49.95	8.15.88	14.00MB	30.2%	-35.0%	30.2%

Table 12.5.: State Space generation and analysis using the full power of SCOOP (Simplification, Confluence and DVR). Runtimes in SCOOP and IMCA are in seconds.

Measurements results We measured two properties on this model, namely the efficiency and adaptation time. With efficiency we mean the probability that eventually we reach consensus over the shortest path, i.e. path A. With adaptation time we mean the expected time it takes to reach consensus, no matter on which opinion. The results can be found in Table 12.6.

Specification	Efficiency	Adaptation Time
consensus-10-10-3-6	75.2%	20.5
consensus-30-30-3-10	84.6%	19.5
consensus-50-50-3-14	87.5%	25.5
consensus-70-70-5-18	93.2%	11.9

Table 12.6.: Consensus Algorithm: Efficiency and Adaptation Time

12.4. Analysis

In the tables we notice a couple of interesting results.

1. Confluence reduces the state space slightly.
2. In contrast to the other case studies, simplification hardly has any effect.
3. The efficiency increases whenever the number of nodes increases.
4. The adaptation time is quite constant but decreases when the group size increases.

For each of these results we analyse their nature.

Confluence
Every point of the model where multiple continuations are possible contains either a rate or a probabilistic choice, i.e. there is no nondeterminism in the model. Therefore, we can never have 'real' confluence. However, still the model reduces slightly whenever we use confluence. The reason for this is similar to the gazillionaire model. Namely, we have first a rate (the travelling of a group) followed by a synchronized action where they return. This action is an unnecessary τ in the parallel composition. However, we cannot omit the action in our model as we work with two processes there. Only after linearisation this action can be omitted and it is nice to see that confluence can detect this.

Simplification
We see hardly any difference in state space generation time whenever we use the simplification techniques. As the impact of the simplification techniques is very different between the three case studies we will compare the models in Chapter 13 and explain why these techniques work not so well for this (and the gazillionaire) model and much better for the gossip model.

Efficiency
In the 4 models we see an upward trend in the efficiency of the model. In the model with 10 nodes, we only have an efficiency of 75.2%, but this increases to 84.6%, 87.5% and 93.2% for 30, 50 and 70 nodes respectively. These results seem very natural. In a small model there is quite a large probability that the group forming process favours the B groups at the start of the model. The B groups will immediately have a large majority and this has a stronger convincing effect then the more efficient path taken by the A nodes. For a very large model the probability that this happens is much smaller as over a large population the deviation will be much smaller and thus we end up with around the same number of A and B groups initially. This has a much larger chance to converge to a majority over path A.

We also see an efficiency increase by increasing the group size. With the change from 50 to 70 initial nodes we also increased the group size from 3 to 5. We see a very large increase in efficiency from 87.5 to 93.2%. We can clarify this in a somewhat similar way as why the efficiency increases with more nodes. It is the case that with larger groups there is less deviation in what will be the majority opinion. It is already very likely that opinion A will reach majority. So if after some time there are a few more A agents than B agents, a group of 5 nodes will more often contain a majority of A nodes than a group of 3 nodes. However, note that this is not the case with smaller groups. This can be easily seen by the fact that if we have 20 nodes and make a group of 19 nodes, it will simply be 50/50 which opinion gains majority. So, we do need a large enough number of nodes to gain an advantage by increasing the group size.

Adaptation Time

We see that even though we increase the number of nodes the adaptation time remains relatively constant. We assume this is the case because the nodes can simultaneously. As practically everything that needs to be done can be synchronised the time it takes to do this does not increase.

We also see that with an increase in the group size the adaptation time drops drastically. This is the case because with a single action (reach consensus within a group) we have now convinced 5 nodes instead of 3 and thus move ‘closer‘ to a result state. However, the time it takes to do this is equal, no matter how large the group size. Therefore, within the same time we move closer to the objective and thus also the entire model reaches a goal state faster.

Part IV.

Evaluation

CHAPTER 13

ANALYSIS

In the case studies performed here we have seen some state space reduction for each of the models. For the gossip model we got some reduction using dead variable reduction and for the other two case studies we were able to reduce the state space using confluence. However, we have not gotten as much reduction as one would hope. At most half of the state space could be eliminated, but this hardly allows us to compute larger models. In this chapter we will do a more thorough analyis of the results and compare the different case studies. We focus on two aspects, namely the large differences in impact of the simplification techniques and the reason why confluence often doesn't work very well.

13.1. Simplification

We see major differences in the impact of the simplification techniques SCOOP uses to simplify the MLPPE. For the gossip model we saw a decrease in state space instantiation time of up to 90% (thus decreasing the needed time to less than 10%)whereas for the consensus model we didn't get any decrease at all. For each of the three models we analyse which parts of the MLPPE get optimized. We then compare these results per case and answer the question when the simplification techniques work and when they will not.

Gossip
We notice a few small differences between the original and simplified MLPPE. First, a summand with an invalid condition (containing the statement True = False) has been removed. Second, we see some logic rewriting. In the original specification we see lots of statements such as $((a \wedge b) \wedge c) \wedge d$ which are rewritten to $a \wedge b \wedge c \wedge d$.

The major difference between the two MLPPEs comes from summation elimination. This is verified by isolating the different simplification techniques and measuring their individual impact on the state space generation time. The results for this are shown in Table 13.1.

	No Red	Exp Simp		Sum Elim		Const Elim	
	Time(s)	Time(s)	Red(%)	Time(s)	Red(%)	Time(s)	Red(%)
gossip-4-3	8.61	8.45	1.9%	0.69	92.0%	7.36	14.5%
gaz-5-1-1-1	19.03	19.07	-0.2%	17.82	6.4%	15.99	16.0%
consensus-70-70-5-18	37.01	36.88	0.4%	36.98	0.1%	37.05	-0.1%

Table 13.1.: Impact of the individual simplification techniques on the state space generation time

In the original model we have summands that model the communication between two nodes. The sender needs two sum operators for the receiver of the information and the information itself, both of which are randomly selected. For the receiver the same holds as it should be able to receive any data from any node. We have generated an MLPPE for three nodes, which makes six directions of communication possible. The linearisation process rewrites this to six summands in the MLPPE, however it leaves the combined four sum operators in each of the summands. This makes a total of 24 sum operators. Often it is the case that the condition allows only for one value for a parameter, so that the sum operator can be eliminated. For each of these six summands, three out of four sums can be eliminated, leaving only six in total. This makes the MLPPE significantly simpler to work with.

It seems thus that summation elimination does most of the work in reducing the time. We verified this by isolating each individual technique and measuring their effects. We saw a reduction of around 91% when solely using summation elimination, so indeed this is the major source. However, constant elimination also reduced the generation time by 15%. Expression simplification had hardly any effect.

Could we have modelled this better? It might be possible to find a more efficient model, but it would be very hard. We still need to be able to send every piece of data to every possible node. For this we simply need two sum operators.

Could we improve the linearisation? One might argue that generating an individual summand for each combination of nodes is not very efficient. The MLPPE grows large very quickly and this makes analysing hard. An improvement such that we do not get a Cartesian product of combinations might be a useful optimisation to the linearisation process.

Trade Market

In the trade market model we see a reduction of state space generation time of around 20%. This is far less than for the gossip model. The major difference between the models is that the trade market model contains far less nondeterminism. We only have a few sum operators which sometimes aren't even necessary. The most simplification comes from constant elimination elimination. This accounts for around 15% reduction whereas summation elimination accounts for 6.36%. Expression simplification again doesn't have a significant effect.

However, when we look at the MLPPE we notice a couple summands that could have been simplified further:

1. We see a $\sum_{b_1:\{1..1\}}$ that can be omitted. All occurrences of b_1 can then be replaced with the constant 1.
2. We have several parameters in our process that can only have 1 value. Therefore, these are all constants in the model.

The 'constant summation' mentioned above would not be present in a model written by hand. However, a model generator needs to be dynamic in that it can for example generate models for 1 or 4 players. In the case of one player nondeterministic choices over all possible players are not really nondeterministic. However, as our generator places a sum for this as it *can* be nondeterministic sometimes we end up with value ranges like 1..1. A simple technique should be able to detect and eliminate these sums. The same holds for ranges of values global variables can obtain.

Consensus Algorithm
The consensus model does not contain nondeterminism and we see that the simplification techniques have hardly any effect. Comparing the original and simplified MLPPE only shows some minor differences where some logic rewriting is done (things like rewriting $a\wedge$ True to a). There are no further obvious optimizations visible, so we cannot give any suggestions for improvements. The linearisation already works very well by itself.

Conclusion
Based on these results, we state that the major difference in effect of the simplification techniques comes from the density of nondeterminism in the model. Whenever there is much nondeterminism (which can explode very quickly by communication) summation elimination *might* gives a major simplification of the model. The other techniques (constant elimination, expression simplification, maximal progress) are applied but do not have a major impact. However, they make the MLPPE more understandable to read.

13.2. Complex Models

In the case studies performed before this research we saw extreme state space reductions of more than 90%. In the case studies performed in this research we do not even come close to that amount. For our analysis we use a notion called the basic sequence of actions. The basic sequence of actions in a model describes the actions necessary for the core activity of the model. For example, if we would make a model of a buffer the basic sequence of actions would be send and receive as this is the core functionality a buffer should provide. However, there might be more, less important actions in the model.

Our models differ from the older models in several ways:

1. We only model the communication. In the other models some local actions are done as well.
2. In our models, the rates are part of the basic sequence of actions. In the older models they are not (Leader Election, Polling)
3. The basic sequence of actions of the older models are often only one or two actions long. The sequences for our models are a little longer.

Communication
Our models only describe the communication between the processes. Applying confluence on such actions is much more complex than applying this on independent processes. Instead of one, two processes are involved and thus parameters of both models are influenced. Also, the data that is communicated over is real data that is relevant for the continuation of the execution. So the nondeterminism that results from what part of the data is communicated cannot be reduced with confluence. Which piece of data is communicated determines how the model continues and will never reach the same state immediately. We take our gossip model and the leader election protocol from [10] and compare the nature of their communication.

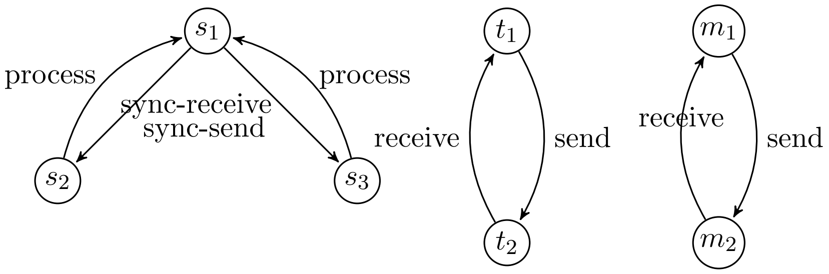


Figure 13.1.: Individual components of a 3-length basic action sequence

We see in the leader election protocol that the communicated data is immediately used again and is reset after that. The 'state' of the Channel process is the same for each and every piece of data that could have been send (after processing). This makes the Channel process irrelevant for detecting confluence. It still depends on one process. This doesn't immediately mean that confluence always works, but at least it is dependant on a smaller portion of the model, which makes it more likely that confluence will work.

In our gossip model we see that one communicating action permanently changes the state of the process that uses the information. The gossiping item is added to the local queue of the receiving node. So in our case, the value of the communicated item *is* relevant for the rest of the model. It might still be able to commute with other actions, but this is much harder. We see something similar in our other models, thus based on these results we state that the relevance of communication correlates negatively with the impact of confluence reduction.

Rates
In the analysis we have already seen that rates mess with confluence reduction. For example, if we would solely model the actions in our gossip model, we would obtain a nice reduction as the order in which two pieces of data are transferred should not matter. However, as they are interleaved by rates we cannot apply the theory of confluence reduction and thus remain with a large state space.

In the leader election protocol we also see a rate, but this rate does not intervene the main sequence of actions (send, receive). So it does not influence the core of the model. Rates are never a good thing for confluence as they can never be eliminated. Therefore, we can safely conclude that the density of rates has a negative impact on the influence of confluence reduction.

Action Sequences
What we just described with rates can also happen with other actions. In the leader election protocol we see a basic sequence of two actions which is constantly repeated. Which parameters are used is nondeterministically chosen, but many of these choices result in the same state after the sequence and afterwards still these actions can be executed. Confluence can reduce this.

A problem now arises when the main action sequence is longer than two actions. If we would for example add a processing action between the send and receive actions from the leader election protocol, we cannot apply the confluence theory anymore. This is illustrated in Figures 13.1.

In Figure 13.1 we show the individual components that result in the system in Figure 13.2 when they are put in parallel. We see that the local processing step (that is part of the long basic action sequence) messes up the confluence. We see a major expansion in the structure of this model which could have been reduced without the intermediate processing step. This is a main issue in

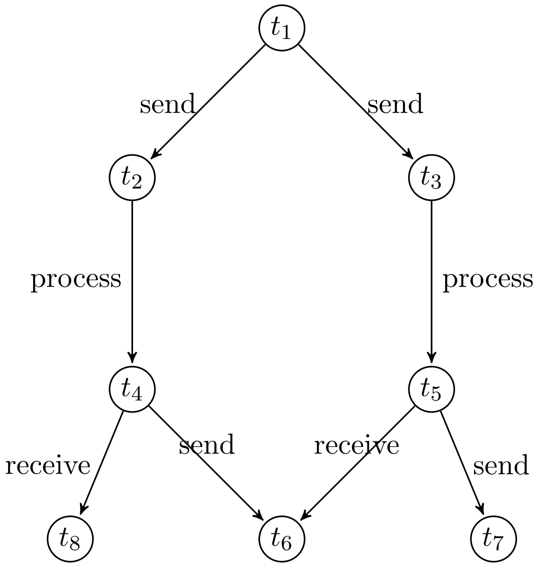


Figure 13.2.: Parallel composition of a 3-length basic action sequence

the models used for this study. The models become too complex too quickly and we see that this is another reason why confluence does not really work for our models.

So, in summary, because we used a lot of communication in our model and the events in the models became quite complicated very quickly we have not achieved the results we hoped to achieve.

CHAPTER 14

CONCLUSION

14.1. Research Questions

In Chapter 2 we gave the 4 main questions for this research. In this document we basically already answered these questions. In this Chapter however, we recite the problem statements and answer each individual question in a few sentences.

How do the reduction techniques used by SCOOP work?
SCOOP uses two state space reduction techniques, namely confluence and dead variable reduction. Confluence depends on the independence of actions. Whenever there are multiple interleavings possible and both interleavings do not exclude behaviour and lead to the same state we can give priority to one interleaving, possibly reducing the state space. Dead variable reduction analyses whether variables can become irrelevant at some point during execution. When the value of such variables can still change this results in a lot of unnecessary states. By giving dead variables a fixed value we can get rid of these unnecessary states.

What is the impact of the reduction techniques and how can we measure this?
We measured the impact of the reduction techniques by performing three case studies, namely a gossip protocol, trade market and consensus algorithm. We modelled these cases in MAPA and ran them in SCOOP. We measured the state space size and generation time for different settings and compared the results. We saw a variety in the impact of the reduction techniques, but were not able to reduce the state space by orders of magnitude.

What differences between case studies exist and how can we explain these?
We saw some differences between the different case studies. For the gossip model we only reached reduction using dead variable reduction and for the other models using confluence. The dead variable reduction for the gossip model is explained by the phased nature of the model. A node is either running or crashed and after it crashed the value of certain parameters has become irrelevant. The confluence for the other two models is explained by actions that do not change the model. They are necessary for the communication, but because they do not change the model, confluence can remove these again.

We saw greater differences with earlier performed case studies. These earlier cases showed much greater reductions of up to 90% and more. We could come nowhere near these numbers. These differences mainly result from the amount of relevant communication (that permanently changes the model), intervening rates during the main sequences of actions and the length of these sequences.

How can we improve SCOOP such that it works better in specific scenario's?
We think dead variable reduction works fine as it is. It does not always give large reductions, but the technique is pretty simple in that it doesn't require much time. Confluence detection costs a lot of time and even when there is a slight reduction in state space, instantiating it costs more time than without confluence. So it is worth looking into whether confluence can be implemented more efficiently.

In order to reach more reduction the theory needs to be extended. Right now, confluence does not go very deep in its detection whether two interleavings commute. We have seen that when the basic sequence of actions gets larger confluence might not work anymore. So going from commutation of actions to commutation of action-sequences might be worth looking into. However, this is no easy task and isn't guaranteed to work. Also, rates are often an issue in confluence reduction, so incorporating rates into the theory in a more intelligent way should be useful.

Last, we provided some small suggestions for the simplification techniques. Especially the ‘fake‘-nondeterministic choices that result from autogenerating the model can very easily be resolved. Statements such as $\sum_{n:1..1}$ can be safely removed while replacing all occurrences of n with the constant value 1. This makes the MLPPE better readable and might also reduce the state space instantiation time. The state space obviously remains unchanged.

14.2. Recommendations

	Confluence		Dead Variable		Simplification	
	Time	#States	Time	#States	Time	#States
Gossip	-	0	+	++	0	++
Trade Market	- -	+	0	0	0	+
Consensus	+	+	0	0	0	0

Table 14.1.: Overview of the impact of reduction techniques per case

In Table 14.1 we give a summary of the results from our case studies. We have seen that dead variable reduction works great on the gossip model and that confluence reduces the state space for the trade market and consensus models slightly. We have also seen big differences in the effect of the simplification techniques and that confluence has quite some overhead. Based on these results and the performed analysis from the previous chapters we answer the main question of this research and give recommendations to both potential users of SCOOP as well as to the developers.

Our main question was as follows: In which application scenarios do the state space reduction techniques performed by SCOOP work best? We subdivide our answer in two parts. First, we handle dead variable reduction. We have seen in the gossip model that the execution is phased. First the gossiping occurs and eventually the node crashes and doesn't work anymore. After such a phase change certain variables can become irrelevant and one can imagine that this is the case in lots of application scenario's. Therefore, we conclude that dead variable reduction works best in a phased type model.

For confluence reduction we have seen that independence of actions is very important. In our case studies we have only modelled the communication between processes. Such actions are very hard to be confluent as the data that has been communicated is often relevant for how the execution continues. So, whenever we have a nondeterministic choice on what is communicated these choices are not independent and thus cannot be confluent. Only when the data is processed in one step and multiple choices of data lead to the same final state after the processing step we can reduce using confluence.

Another situation where confluence works very well is when two processes in the same model (temporarily) execute independently. However, this is not something that you would model too often because in a model with multiple processes you are likely modelling the communication between them. We abstract away from internal transitions. For example in the consensus model we could have also modelled all kinds of things the agents do while travelling or in the gossip model we could have modelled the local search in each node. However, they are not relevant at all and we think this is the case in many kinds of models. There might always be cases however where it is the case that two processes behave independently.

So we conclude that confluence reduction works best either when communication is handled quickly and many communicated actions are handled the same way or when two processes in the same model (temporarily) execute independently (although this is not often the case).

What do we recommend to the developers? Currently dead variable works as desired and we do not see any directions in which this theory can be improved further. Thus we recommend to focus on the theory behind confluence. Especially finding a way for actions to commute in multiple steps is very useful.

What do we recommend to potential users? First we recommend to use dead variable reduction at all times. We have seen that this technique can significantly reduce the state space and even though it doesn't always have the desired effects, it also doesn't really have a downside. The overhead is minimal, making it worthwhile to use it. The same holds for the simplification techniques. They do not have a downside and can greatly decrease the state space generation time so they should be used at all times.

For confluence this is slightly different, as we have seen a significant increase in state space generation time whenever confluence doesn't have a great effect. However, the analysis time for IMCA is improved. Therefore, if confluence possibly reduces the state space (by the reasons described before) we recommend using it. However, if you are sure that confluence will not have the desired effect it is better to turn it off in order to save time while generating the state space (by ∼30% on average). However, if a user wants to make a large model and expects confluence reduction to not have a great effect we do not recommend to use SCOOP yet as it will likely be infeasible. We then recommend to either wait for SCOOP to be improved or to use other tools (for IMCs or CTMCs for example).

In summary, we recommend users to always use dead variable reduction and simplification techniques. Confluence should only be used if it is likely to work because of independently executing processes or quickly handled communication.

14.3. Future Work

We propose three areas for future work in this section.

First, it is useful to upgrade the bridge between SCOOP and IMCA. At the moment, the user of SCOOP needs to linearize the model first in order to find the specific names of variables that are part of the property you would like to check. Rather, a more formal language to specify properties in should be much more practical. Extending for example CSL or PCTL to incorporate both probabilistic and timing operators should solve this problem. This also should make sure that the model remains unchanged when adding in properties. Right now, extra actions are added to be used by IMCA which changes the model.

Second, we think it is important that more research is done on confluence reduction. It does what it is supposed to do, but does not go very deep in its detection. The concept of sequence confluence, as described earlier in Chapter 13, is probably worth looking into. Also, we need to find a way to better handle rates in confluence.

Last, we can extend the consensus model with stubborn agents. Right now the model reaches concensus, but is not capable to adapt to a changing environment because once there is no agent that can notice the environment change. What one can do is implement stubborn agents. These agents do engage in voting, but never change their opinion. This makes sure that at all times populations of both opinions remain alive. In a changing environment the stubborn agents might be able to convince the other agents to also change their opinion.

BIBLIOGRAPHY

[1] M. Timmer, J.-P. Katoen, J. van de Pol, and M. Stoelinga, "Efficient modelling and generation of markov automata," in *International Conference on Concurrency Theory*, Springer, 2012.

[2] M. Timmer, "SCOOP: A tool for symbolic optimisations of probabilistic processes," in *Quantitative Evaluation of SysTems*, pp. 149–150, IEEE, 2011.

[3] D. Guck, T. Han, J.-P. Katoen, and M. Neuhäußer, "Quantitative timed analysis of interactive markov chains," *NASA Formal Methods*, pp. 8–23, 2012.

[4] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2010: a toolbox for the construction and analysis of distributed processes," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 372–387, 2011.

[5] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 441–444, 2006.

[6] M. Timmer, M. Stoelinga, and J. van de Pol, "Confluence reduction for probabilistic systems," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 311–325, 2011.

[7] J. van de Pol and M. Timmer, "State space reduction of linear processes using control flow reconstruction," *Automated Technology for Verification and Analysis*, pp. 54–68, 2009.

[8] H. Hermanns, "Construction and verification of performance and reliability models," *Bulletin of the European Association for Theoretical Computer Science*, vol. 74, pp. 135–153, 2001.

[9] M. Nielsen, "Models for concurrency," *Mathematical Foundations of Computer Science*, pp. 43–46, 1991.

[10] M. Timmer, M. Stoelinga, and J. van de Pol, "Confluence reduction for markov automata," in *Quantitative Aspects of Programming Languages*, IEEE, 2013.

[11] J.-P. Katoen, J. van de Pol, M. Stoelinga, and M. Timmer, "A linear process-algebraic format for probabilistic systems with data," in *International Conference on Application of Concurrency to System Design*, pp. 213–222, IEEE, 2010.

[12] Y. Usenko, *Linearization in μCRL.* PhD thesis, Eindhoven University of Technology, 2002.

[13] J.-P. Katoen, J. van de Pol, M. Stoelinga, and M. Timmer, "A linear process-algebraic format with data for probabilistic automata," *Theoretical Computer Science*, pp. 36–57, 2012.

[14] J. Groote and M. Sellink, "Confluence for process verification," *Theoretical Computer Science*, pp. 47–81, 1996.

[15] H. Hermanns and J.-P. Katoen, "The how and why of interactive markov chains," in *International conference on Formal methods for components and objects*, pp. 311–337, 2010.

[16] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time markov chains," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 162–170, 2000.

[17] J. Filar and K. Vrieze, *Competitive Markov Decision Processes*. Springer, 1997.

[18] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[19] S. Blom and J. van de Pol, "State space reduction by proving confluence," in *Computer Aided Verification*, pp. 676–694, Springer, 2002.

[20] H. Hansen and M. Timmer, "A comparison of confluence and ample sets in probabilistic and non-probabilistic branching time," *Theoretical Computer Science*, 2010.

[21] A. Hartmanns and M. Timmer, "On-the-fly confluence detection for statistical model checking," in *NASA Formal Methods Symposium*, 2013.

[22] A. Laarman, J. van de Pol, and M. Weber, "Multi-core LTSmin: Marrying modularity and scalability," *NASA Formal Methods*, pp. 506–511, 2011.

[23] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, "Symbolic model checking for probabilistic processes," *Automata, Languages and Programming*, pp. 430–440, 1997.

[24] H. Hansen and M. Timmer, "Why confluence is more powerful than ample sets in probabilistic and non-probabilistic branching time," in *10th Workshop on Quantitative Aspects of Programming Languages*, Istituto di Scienza e Tecnologie dell'Informazione, 2012.

[25] C. Baier, M. Größer, and F. Ciesinski, "Partial order reduction for probabilistic systems," in *Quantitative Evaluation of SysTems*, pp. 230–239, IEEE Computer Society, 2004.

[26] C. Baier, P. D'Argenio, and M. Groesser, "Partial order reduction for probabilistic branching time," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 97–116, 2006.

[27] P. D'Argenio and P. Niebert, "Partial order reduction on concurrent probabilistic programs," in *Quantitative Evaluation of SysTems*, vol. 4, pp. 240–249, 2004.

[28] D. Peled, "All from one, one for all: on model checking using representatives," in *Computer Aided Verification*, pp. 409–423, Springer, 1993.

[29] A. Valmari, "Stubborn sets for reduced state space generation," *Advances in Petri Nets 1990*, pp. 491–515, 1991.

[30] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032. Springer, 1996.

[31] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani, "Partial-order reduction in symbolic state space exploration," in *Computer Aided Verification*, pp. 340–351, Springer, 1997.

[32] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte, *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994.

[33] A. Pnueli and L. Zuck, "Verification of multiprocess probabilistic protocols," *Distributed Computing*, vol. 1, no. 1, pp. 53–72, 1986.

[34] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of algorithms*, vol. 11, no. 3, pp. 441–461, 1990.

[35] O. Ibe and K. S. Trivedi, "Stochastic petri net models of polling systems," *Selected Areas in Communications, IEEE Journal on*, vol. 8, no. 9, pp. 1649–1657, 1990.

[36] H. Hermanns, J. Meyer-Kayser, and M. Siegle, "Multi terminal binary decision diagrams to represent and analyse continuous time markov chains," in *Workshop on the Numerical Solution of Markov Chains*, pp. 188–207, Citeseer, 1999.

[37] S. Blom, J. R. Calamé, B. Lisser, S. Orzan, J. Pang, J. Van De Pol, M. T. Dashti, and A. J. Wijs, "Distributed analysis with μcrl: A compendium of case studies," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 683–689, Springer, 2007.

[38] A. Wijs, J. van de Pol, and E. Bortnik, "Solving scheduling problems by untimed model checking: the clinical chemical analyser case study," in *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pp. 54–61, ACM, 2005.

[39] D. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelinga, and I. Zapreev, "How fast and fat is your probabilistic model checker? an experimental performance comparison," in *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, pp. 69–85, Springer Berlin Heidelberg, 2008.

[40] R. Bakhshi, D. Gavidia, W. Fokkink, and M. van Steen, "A modeling framework for gossip-based information spread," in *Quantitative Evaluation of SysTems*, pp. 245–254, 2011.

[41] R. Bakhshi, F. Bonnet, W. Fokkink, and B. Haverkort, "Formal analysis techniques for gossiping protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 28–36, 2007.

[42] M. Lin and K. Marzullo, "Directional gossip: Gossip in a wide area network," *Dependable Computing EDCC-3*, pp. 364–379, 1999.

[43] M. Jelasity, W. Kowalczyk, and M. Van Steen, "Newscast computing," tech. rep., Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, 2003.

[44] C. Krause, E. de Vink, and P. de Vink, "Towards dynamic adaptation of the majority rule scheme," *Quantitative Aspects of Programming Languages and Systems*, 2013.