Model Checking of Randomized Leader Election Algorithms

diploma thesis

David Faragó

supervisors: Prof. Dr. Peter H. Schmitt
Dipl.-Inf. Frank Werner

February 26, 2007
Statutory declaration

I hereby declare to have written this thesis on my own, only using the stated resources.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, 26. Februar 2007

David Faragó
Acknowledgements

First of all I would like to thank Prof. Dr. Peter H. Schmitt for offering such an interesting task, in which I could strongly influence the direction of development, and for his extensive, cooperative, and encouraging supervision.

I am grateful to Frank Werner, who was always there to helpfully and patiently answer my numerous content-related as well as technical questions.

Finally I owe a debt of gratitude to my family and friends for their support over the years.
Abstract

This thesis deals with randomized leader election algorithms for anonymous, unidirectional rings as a case study for looking into the state space explosion problem of model checking.

Six algorithms are investigated and model checked using the tools PRISM and SPIN. Various statistics and improvements of the model checking procedure are regarded.

keywords: randomized leader election algorithms for anonymous, unidirectional rings; state space explosion; probabilistic model checking; PRISM; model checking; SPIN; randomness vs. non-determinism; non-progress cycle check
# Short contents

1. Preamble
2. Introduction to model checking
3. Introduction to the leader election problem
4. Theoretical investigations
5. Model checking with PRISM
6. Model checking with SPIN
7. A non-progress cycle check for SPIN
8. Closure

A. PRISM specifications
B. SPIN specifications

Bibliography
Index
Contents

1 Preamble ............................... 1
   1.1 Motivation ................................. 1
   1.2 Conventions ............................... 1
   1.3 Overview ................................. 2

2 Introduction to model checking ............................... 3
   2.1 The models ............................... 3
      2.1.1 Models without probability .................. 4
      2.1.2 Models with probability ..................... 7
   2.2 The properties ............................. 10
      2.2.1 Basic properties ......................... 10
      2.2.2 Temporal logics ........................... 11
   2.3 PRISM .................................. 15
      2.3.1 The models .............................. 16
      2.3.2 The properties ........................... 19
   2.4 SPIN .................................. 20
      2.4.1 The models .............................. 21
      2.4.2 The properties ........................... 24
   2.5 State space explosion ....................... 26

3 Introduction to the leader election problem .................. 27
   3.1 Network models ............................. 27
### 3.2 Leader election for anonymous, unidirectional rings

#### 3.2.1 Motivation .................................................. 29

#### 3.2.2 Randomized algorithms .................................. 30

### 4 Theoretical investigations .................................... 39

#### 4.1 Reducing model checking requirements ................. 39

##### 4.1.1 Compression of the state space ..................... 39

##### 4.1.2 Reduction of the state space ....................... 40

##### 4.1.3 Combining compression and reduction ............. 43

#### 4.2 Parameters of the class $A_{\text{compare IDs}}$ ........ 44

#### 4.3 Mathematical correctness proofs ....................... 49

##### 4.3.1 General discussion about induction ............... 49

##### 4.3.2 Non-aborting $A \in A_{\text{compare IDs}}$ with roundtrips 50

##### 4.3.3 Algorithm $A_{\text{fattest no roundtrip}}$ .......... 51

##### 4.3.4 Algorithm $A_{\text{empty}}$ .......................... 52

##### 4.3.5 Algorithm $A_{\text{sorting}}$ ......................... 53

### 5 Model checking with PRISM .................................. 55

#### 5.1 Algorithm $A_{\text{in round } \leq K}$ ....................... 55

##### 5.1.1 Algorithm design and specification ............... 55

##### 5.1.2 Model properties and statistics .................. 62

#### 5.2 Algorithm $A_{\text{in round mod K}}$ ....................... 72

##### 5.2.1 Algorithm design and specification ............... 72

##### 5.2.2 Model properties and statistics .................. 73

#### 5.3 Algorithm $A_{\text{fattest roundtrip}}$ .................... 76

##### 5.3.1 Algorithm design and specification ............... 76

##### 5.3.2 Model properties and statistics .................. 79

#### 5.4 Algorithm $A_{\text{fattest no roundtrip}}$ .................. 82

##### 5.4.1 Algorithm design and specification ............... 82

##### 5.4.2 Model properties and statistics .................. 83
## Contents

5.5 Algorithm $A_{	ext{smallest}}$ ........................................... 84
  5.5.1 Algorithm design and specification ............................ 84
  5.5.2 Model properties and statistics ................................. 86
5.6 Algorithm $A_{\text{timing}}$ ........................................... 90
  5.6.1 Algorithm design and specification ............................ 90
  5.6.2 Model properties and statistics ................................. 92
5.7 Conclusion .......................................................... 95

6 Model checking with SPIN ........................................... 97
  6.1 Replacing randomness by non-determinism ..................... 97
    6.1.1 Fairness constraints ........................................ 98
    6.1.2 Temporal logic ............................................. 101
    6.1.3 Correctness of terminating paths ......................... 103
    6.1.4 Termination proofs ........................................ 103
  6.2 Model checking the promising algorithms .................... 104
    6.2.1 Algorithm $A_{\text{next round}} < k$ ......................... 104
    6.2.2 Algorithm $A_{\text{next round most}}$ ......................... 109
    6.2.3 Algorithm $A_{\text{next round random}}$ ...................... 112
    6.2.4 Algorithm $A_{\text{next round parameter}}$ ................. 113
    6.2.5 Algorithm $A_{\text{smallest}}$ ................................. 113
    6.2.6 Algorithm $A_{\text{timing}}$ ................................ 115
  6.3 Conclusion ........................................................ 118

7 A non-progress cycle check for SPIN ................................ 121
  7.1 Without partial order reduction ................................. 122
    7.1.1 Correctness ................................................. 124
    7.1.2 Complexity ................................................. 126
  7.2 With partial order reduction ................................... 129
    7.2.1 Introduction to partial order reduction .................. 129
    7.2.2 Overview of the variants .................................. 131
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example for an STS</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Three different views</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Example for an LTS</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Example for a DTMC</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Example for an MDP</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Model formalisms</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Interleaving diamond</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Message overtaking with (\text{round} \mod K) is erroneous</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>No reactivation</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Induction step: insertion of a node</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>State machine for (A_{\text{full}})</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Construction time for (A_{\text{full, round } \leq K, n=3})</td>
<td>64</td>
</tr>
<tr>
<td>5.3</td>
<td>Number of states for (A_{\text{full, round } \leq K, n=3})</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>Number of transitions for (A_{\text{full, round } \leq K, n=3})</td>
<td>65</td>
</tr>
<tr>
<td>5.5</td>
<td>Number of nodes for (A_{\text{full, round } \leq K, n=3})</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>Construction time for improved (A_{\text{full, round } \leq K, n=3})</td>
<td>66</td>
</tr>
<tr>
<td>5.7</td>
<td>Construction time for improved (A_{\text{full, round } \leq K, n=4})</td>
<td>66</td>
</tr>
<tr>
<td>5.8</td>
<td>Number of nodes for improved (A_{\text{full, round } \leq K, n=3})</td>
<td>67</td>
</tr>
<tr>
<td>5.9</td>
<td>Probability of elected leader for (A_{\text{full, round } \leq K, n=3})</td>
<td>70</td>
</tr>
<tr>
<td>5.10</td>
<td>Probability of elected leader for (A_{\text{full, round } \leq K, n=4})</td>
<td>70</td>
</tr>
<tr>
<td>5.11</td>
<td>Expected number of rounds for (A_{\text{full, round mod K}})</td>
<td>75</td>
</tr>
</tbody>
</table>
5.12 Expected number of transmissions for $A_{\text{task round mod } K}$ ............ 75
5.13 Probabilities for $A_{\text{okkink roundtrip}}$ with soft deadlines, $n=3$ ........ 81
5.14 Probabilities for $A_{\text{okkink no roundtrip}}$ with soft deadlines, $n=3$ .... 83
5.15 Expected number of transmissions for $A_{\text{okkink no roundtrip}}$ ............. 84
5.16 States and nodes for $A_{\text{sorting sync}}, n=3$ .................................. 87
5.17 Probabilities for $A_{\text{sorting sync}}$ with soft deadlines .......................... 89
5.18 Expected number of rounds for $A_{\text{timing}}$ ...................................... 89
5.19 State machine for $A_{\text{timing}}$ ......................................................... 90
5.20 States and nodes for $A_{\text{timing}}, n=2$ ............................................. 92
5.21 Average probabilities for $A_{\text{timing}}$ with soft deadlines ....................... 94
5.22 Average expected number of transmissions for $A_{\text{timing}}$ ..................... 95

6.1 A path with probability zero .............................................................. 98
6.2 Prism's fairness excludes infinite paths having probability zero ............... 99

7.1 Not traversed non-progress cycle ...................................................... 123
7.2 $s_j$ cannot be on the stack at time $t_{end}$ ........................................ 126
7.3 Constructing $\pi^2$ from $\pi^1$ .............................................................. 126
7.4 The np cycle found does not contain $s$ ............................................. 126
7.5 ample($s$) without C2 is in general erroneous ...................................... 131
7.6 ample(prune($s$)) $\subseteq$ prune(ample($s$)) = prune($s$) ......................... 132
7.7 ample(prune($s$)) $\subseteq$ prune(ample($s$)) = ample($s$) ......................... 133
7.8 Error with DFSresco and C3 ............................................................... 133
7.9 An incorrect combination of options ..................................................... 134
7.10 Failed progress ................................................................................. 135
7.11 Now the combination is correct .......................................................... 136
7.12 Progress transition with $d_{\text{step}}$ ...................................................... 136
7.13 Progress transition with $d_{\text{step}}$ and an additional state ..................... 137
7.14 prune(ample($G$)) $\neq$ prune(ample($s$)) ........................................ 139
7.15 np cycle with Caseindependent .......................................................... 141
List of figures

7.16 The simplest counterexample for reduction of \( L \) \hspace{1cm} 142
7.17 \( \text{ample(prune(s))} \neq \text{ample(prune}(G) \text{)} \) \hspace{1cm} 143
7.18 Summary of variants \hspace{1cm} 144
List of Tables

5.1 Model checking values for selected parameters .......................... 67
5.2 Average communication in bits ............................................. 96
5.1 Safety checks for Atoll round < k ........................................ 106
5.2 No cycle checks for Atoll round < k .................................... 107
5.3 Atoll round < k without saving the state space ......................... 109
5.4 Safety checks for Atoll round model ...................................... 110
5.5 No cycle checks for Atoll round model .................................. 110
5.6 Safety checks for $A_{Fokking}$ roundtrip .................................. 112
5.7 No cycle checks for $A_{Fokking}$ roundtrip ................................. 112
5.8 Safety checks for $A_{Fokking}$ no roundtrip ............................... 113
5.9 No cycle checks for $A_{Fokking}$ no roundtrip ............................ 113
5.10 Safety checks for $A_{Sarting}$ .............................................. 114
5.11 No cycle checks for $A_{Sarting}$ .......................................... 114
5.12 Safety checks for $A_{Amin}$ .................................................. 116
5.13 No cycle checks for $A_{Amin}$ ............................................. 116
5.14 Safety checks for improved $A_{Amin}$ ..................................... 118
7.1 Constraints on ample(s) .................................................. 130
7.2 Estimation for DFS using cycle ............................................ 143

Listings

3.1 Atoll ................................................................. 32
3.2 $A_{Fokking}$ roundtrip .................................................. 33
3.3 $A_{Fokking}$ no roundtrip .............................................. 34
3.4 $A_{Sarting}$ ........................................................... 35
3.5 $A_{Amin}$ ............................................................ 37
7.1 Basic depth-first search ................................................. 122
7.2 Incremental depth-first search with starting over ....................... 123
7.3 Generic DFS with pruning and no cycle check .......................... 124
7.4 Incremental depth-first search with a FIFO ............................ 127
7.5 Incremental depth-first search with modified stack ................. 128
7.6 Progress process ..................................................... 137
Chapter 1

Preamble

1.1 Motivation

In the field of formal deductive verification, a question often to be heard is: "Why do all that mathematical work and not just model check?" Is this question justified? As model checking is already being used in industry, there are definitely successful application areas. But these are often specific, e.g. certain hardware verification problems. So model checking might only be feasible for small or special problem instances. In this case the rhetorical question stated above is ultimately incorrect.

Since the abstract viewpoint is often too far away from practice, we will stay tangible by investigating a use case: leader election algorithms for anonymous, unidirectional rings. Leader election is a fundamental problem of distributive computing, which is a strength of model checking. We will model check correctness and other properties for several such algorithms, e.g. for derived variants of $A_{init}$, one of the first of these algorithms, introduced in [Rai and Rodeh, 1981].

Analysing statistics about the model checking procedure with the tool PRISM, we will conclude how practicable model checking is and what improvements are possible. One will be to use the tool SPIN instead, another to replace its weakest technique.

1.2 Conventions

As it is convenient to write the equivalence in one word, we use the abbreviation "iff" for "if and only if".

Because of the high amount of definitions in this thesis, most of them are not put into extra definition environments, but printed bold and explained in the regular text flow. Since the locations of the definitions are given in the index as bold page numbers, references with definition numbers can be saved.
The necessary bibliographical references for a section are usually listed together at the beginning of the section, for giving an overview of the needed literature and to facilitate look-ups.

The elements of the bibliography contain backreferences to the pages where they were mentioned in the thesis.

Since the definition whether $0 \in \mathbb{N}$ varies, we write $\mathbb{N}_{\geq 0}$ for $\mathbb{N} \cup \{0\}$ and $\mathbb{N}_{> 0}$ for $\mathbb{N} \setminus \{0\}$ whenever it is relevant.

1.3 Overview

The following two chapters are mainly introductory. Chapter 2 defines model checking, the formalisms describing its models, and temporal logics in general. Then the tools PRISM and SPIN are introduced, each with its own specification language and checkable properties. At the end of the second chapter, the major weakness of model checking, its state space explosion, is explained. Chapter 3 introduces various network models and presents our randomized leader election algorithms for anonymous, unidirectional rings with asynchronous transmissions.

Before model checking is put into action, chapter 4 theoretically investigates possibilities to reduce its requirements. Thereafter several variations of the derivatives of $A_{\text{Inst}}$ are analysed. Finally some mathematical correctness proofs are provided, as opposed to the correctness verifications in the following two chapters.

The model checking procedures for our algorithms are performed in the next two chapters. Chapter 5 uses PRISM, and checks besides correctness several other properties. This gives insight to both the leader election algorithms and the model checking procedure itself. Because of the limited problem instances feasible with PRISM, chapter 6 tries SPIN as an alternative for the correctness checks of our algorithms. Before performing the verifications, the required transfer from probability to solely non-determinism is explained. After model checking, the achievements are inspected and compared to PRISM.

Since SPIN's non-progress cycle check breaks the efficiency of the complete correctness verifications, improvements of the check are proposed in chapter 7.

The closing chapter gives a summary of this thesis and final conclusions.

All PRISM and SPIN specifications are listed in appendix A and appendix B respectively.
Chapter 2

Introduction to model checking

(Clarke Jr. et al., 1999) is the definitive book on model checking. But it does not deal with probabilistic model checking, which is discussed in (Rutten et al., 2004). (Merz, 2000) is a tutorial overview of model checking with reference to SPIN, which is introduced in section 2.4. (Palsberg, 2004) is a very short and practical introduction. (Thomas and L"{o}ding, 2004) shows a lecture on video.

A model checker is a tool for complete, formal verification that takes as input a specification of a model and a property to be checked, and outputs whether the property holds for the model. If it does not, a counterexample may be given. The model checker solves this task through explicit or implicit exhaustive enumeration of the reachable states of the model.

Model checking (mc) is the verification whether the model satisfies the properties of interest by using a model checker.

Section 2.1 describes various model formalisms we use for model checking. The properties are explained in section 2.2. Section 2.3 introduces the model checking tool PRISM, section 2.4 the tool SPIN. The final section 2.5 defines state space explosion.

2.1 The models

A model is a formal description that semantically embodies the miniworld, i.e. the relevant parts of the application domain. The notion "miniworld" originates from database theory, but since it is apt, we use it as well.

The miniworld can for instance be a hardware description, a protocol, or an algorithm. Then the design can be checked against the desired properties at an early stage of development, before implementation, thus reducing the cost of
an error. But currently there are also attempts to take a given source code as miniworld and automatically generate the corresponding model. Then model checking can verify the source code.

Since the model checker exhaustively enumerates the states of the miniworld, it must in general be finite. As we want to fully verify properties of the miniworld, it must also be closed, i.e. completely specified and may not depend on any hidden assumptions, components or input.

Many notions occurring in the literature for describing the model are often used synonymously, although they have different meaning. The rest of this section will rigorously define them.

2.1.1 Models without probability

A model contains a non-empty set of states, denoted as state space $S$. Since the miniworld must generally be finite, we have a finite state space. Thus we only need a finite amount of memory, the state vector, to uniquely describe a state.

A transition $t$ in the model is an ordered pair of states $(s \rightarrow s') \in S^2$ and corresponds to an action in the miniworld. $s$ is called the source of $t$, $s'$ its destination. $T \subseteq S^2$ is the set of all transitions of the model. We call $(S, T)$ the (unlabelled) state transition system (STS) or the Kripke frame of the miniworld.

Figure 2.1 shows an example for an STS. The example was generated by SPIN, explained in section 2.4.

Since we want to analyse the miniworld, i.e. look at its properties, these must also be comprised in the model: We formalize a property with a propositional variable $p: S \rightarrow \{tt, ff\}$ which maps a state $s$ to $tt$ iff $s$ has the property. With $P$ being the set of all relevant atomic propositional variables of the model, we can extend a Kripke frame $(S, T)$ to a Kripke structure $(S, T, P)$.

The initial states $I \subseteq S$ of a model correspond to the starting points or default states of the miniworld. Sometimes $I$ is an additional annotation to the model, i.e. to the formalisms STS, Kripke structure and those introduced later. Often $|I| = 1$, in which case we call the unique initial state init.

A model is generally described in a specification language. Hence we have three different views, as depicted in figure 2.2. The miniworld and the model are semantically equivalent. Each specification defines exactly one model, but there can be different specifications for the same model (e.g. on page 30).

Data. The data of the miniworld is specified by variables. The set of all variables in the specification forms the state vector, and the state space is the set of all possible values for the state vector. A variable $x$ of the specification can produce atomic propositional variables, e.g. $(x \neq 0)$ or $(x$ is even).
2.1. The models

![Diagram](image)

**Figure 2.1: Example for an STS**

![Diagram](image)

**Figure 2.2: Three different views**

**Behaviour.** The atomic instructions of the specification that describe the miniworld’s actions, i.e. its dynamic aspects, are called statements or actions. A state $s$ is **enabled** in a state $s'$ iff $s'$ can be executed in $s$, otherwise $s'$ is **blocked.** enabled($s$) is the set of all statements enabled in $s$. Executing a statement $\alpha \in$ enabled($s$) in $s$ performs the side effects of the corresponding action of the miniworld: The value of the state vector for $s$ is **updated**, i.e. changed to some new value by variable assignments. Let $s'$ be the state corresponding to the new value. The execution of $\alpha$ in $s$ contributes the transition $s \rightarrow s'$ to $T$. If we want to state that $\alpha$ generated $s \rightarrow s'$, we can label the transition accordingly: $s \xrightarrow{\alpha} s'$.

Let $\mathcal{S}$ be the set of all statements in the specification of the model. By labelling each transition $\alpha \in T$ of the STS $(S, T)$ with the statements that generate $\alpha$, we get the **labelled transition system** (LTS) $(S, \mathcal{S}, \sigma)$ with $\sigma \subseteq S \times \mathcal{S} \times S$. So a transition can be labelled by more than one statement. If we identify each $\alpha \in \mathcal{S}$ with the subset $\{(s, s') \in T | (s, \alpha, s') \in \sigma\}$ of transitions generated by $\alpha$,.
we can omit $\sigma$ and only need $(S, \mathcal{G})$ to describe the LTS.
We often solely consider deterministic statements, i.e. each $\alpha \in \mathcal{G}$ is right-
unique. Then we can write $\alpha(s) = s'$ instead of $s \xrightarrow{\alpha} s'$.
Figure 2.3 shows an example for an LTS, again generated by SPIN.

Figure 2.3: Example for an LTS

An STS $(S, T)$ is a special case of an LTS with $\mathcal{G} = \{\lambda\}$. If we ignore the
labelling of an LTS $(S, \mathcal{G})$, we get the STS $(S, \cup_{\alpha \in \mathcal{G}} \alpha)$.
Using the notion graph does not determine whether the transitions are labelled,
since both $(S, T)$ and $(S, \mathcal{G})$ are graphs.

A path $\pi$ is a finite or infinite sequence of (labelled) transitions, $\langle t_1, t_2, \ldots \rangle$, such that the destination of $t_i$ is the source of $t_{i+1}$. The length $|\pi|$ of $\pi$ is the
number of transitions in $\pi$. $\pi$ starts in the source of $t_1$. If $\pi$ has finite length $n$, $\pi$ ends in the destination of $t_n$.
If $\pi$ is a path of an STS, i.e. if the statements generating $\pi$ are irrelevant, $\pi$ is
uniquely determined by the sequence of states $\pi$ visits. So the set of all paths
is bijective to $S^*.$
If $\pi$ is a path of an LTS and starts in $s$, its sequence of statements and $s$ uniquely
determine $\pi.$
2.1. The models

A state \( s' \) is **reachable** from a state \( s \) if there exists a path in the graph from \( s \) to \( s' \). The **reachable state space** is the set of all states that are reachable from an initial state.

Unwinding an STS \((S, T)\) starting from \( s \in S \) yields a tree containing all paths starting in \( s \), denoted the computation tree of \((S, T)\) starting from \( s \). It is uniquely defined as the tree whose root has the value \( s \) and for each vertex \( u \) of the tree, with \( u \) being the value of \( v \) and \( \{s_1, \ldots, s_n\} = \{u \in S| (s \xrightarrow{T} s') \in T\} \), \( v \) has exactly \( n \) children with the values \( s_1, \ldots, s_n \).

Likewise, we can unwind an LTS \((S, \mathcal{G})\) starting from \( s \in S \) to get the computation tree of \((S, \mathcal{G})\) starting from \( s \): The root has the value \( s \) and, with \( s' \) being the value of some vertex \( u \) of the tree, \( u \) has for every \( s' \xrightarrow{\alpha} s'' \) with \( s'' \in S, \alpha \in \mathcal{G} \), and \( (s' \xrightarrow{T} s'') \in \alpha \) an outgoing transition labelled with \( \alpha \) to a vertex with the value \( s'' \).

If the graph has only one initial state, the computation tree of the graph is the one starting from \( \text{init} \). The paths of the tree, or the corresponding sequences of values for the relevant atomic propositional variables, are called **behaviours**.

### 2.1.2 Models with probability

If the application domain exhibits probabilistic behaviour we want to capture, the model must in addition be probabilistic. For this reason each transition has a probability. A model checker that uses probabilistic models is called a **probabilistic model checker**. The model formalisms in this section are equivalent to the ones from [Rutten et al., 2004], but not identical, as they are derived from the ones we already have, i.e. by extending STSs and LTSs.

Annotating an STS \((S, T)\) with transition probabilities results in a discrete-time Markov chain (DTMC) \((S, M)\), with \( M \) being a right-stochastic \( S \times S \) matrix, i.e. the transition probability between \( s \in S \) and \( s' \in S \) is \( 0 \leq M_{s,s'} \leq 1 \) and \( \sum_{s' \in S} M_{s,s'} = 1 \). Imposing a probability density function over \( \{s \rightarrow s'| s' \in S\} \), i.e. over the choices of leaving a state \( s \), changes the non-determinism of the STS into probability. Since \( M \) is right-stochastic, each state has outgoing transitions. If we also want to use models having end states or terminals, i.e. states with no outgoing transitions, we either allow \( M \) to have rows with solely zeros, or we add artificial self-loops to these states so they do have an outgoing transition, but no possibility of leaving.

Instead of \((S, M)\) we can also use \((S, T, P)\) with \( P[s \rightarrow s'] = M_{s,s'} \) for all \((s \rightarrow s') \in T\).

Figure 2.4 shows an example for a DTMC. The example was generated by PRISM, explained in section 2.3 and plotted by DOT, which is supported by PRISM.

DTMCs are the basic formalism to describe probabilistic models. Two additional annotations are possible: If we use a Kripke structure instead of an STS, we have on top of \((S, T)\) the set \( P \) of atomic propositional variables. We can also annotate the model with a **cost structure** \( C : S \times S \rightarrow \mathbb{R}_{\geq 0} \), which indicates the costs of the transitions. Both additions are sometimes referred to
as labelled DTMCs, as we label parts of the original model with supplementary information. This notion does not mean we are using a labelled transition system.

![Diagram of a DTMC](image)

**Figure 2.4: Example for a DTMC**

If the miniworld also exhibits non-determinism, we do use an LTS \((S, \mathcal{G}, \sigma)\) and annotate it with transition probabilities, i.e. each \((s \xrightarrow{\alpha} s') \in \sigma\) has a probability. The result is a Markov decision process (MDP) \((S, \mathcal{G}, \sigma')\) with \(\sigma' : S \times \mathcal{G} \times S \rightarrow [0, 1]\), such that for each \(s \in S\) and \(\alpha \in \mathcal{G}\) the function \(\lambda s'.\sigma'(s, \alpha, s')\) is a probability density function over \(S\) if \(\alpha\) is enabled in \(s\), and constantly zero otherwise.

If we identify each \(\alpha \in \mathcal{G}\) with the function \(\lambda s \lambda s'.\sigma'(s, \alpha, s')\), we can omit \(\sigma'\) and only need \((S, \mathcal{G})\) to describe the MDP.

Figure 2.4 shows an example of an MDP, generated by PRISM.

Just like DTMCs, we can also label MDPs with propositional variables and costs, sometimes denoted as labelled MDPs.

DTMCs and MDPs inherit their interrelationship from STSs and LTSs, since the former are extensions of the latter: An MDP is a special case of a DTMC with \(\mathcal{G} = \{\lambda\}\). Ignoring the labelling of an MDP \((S, \mathcal{G})\) results in \((S, T, P)\) with
2.1. The models

Figure 2.5: Example for an MDP

\[ P = \sum_{\alpha \in \mathcal{A}} \alpha \text{ and } T = \{ (s \to s') \in S^2 \mid P [s \to s'] = \sum_{\alpha \in \mathcal{A}} \alpha (s, s') > 0 \}. \]

Since then \( P \) is a general measure, and usually not a probability measure, we additionally have to normalize \( P \) for \((S, T, P)\) to be a well-defined DTMC:

\[ P : S^2 \to [0, 1], (s, s') \mapsto \frac{\sum_{\alpha \in \mathcal{A}} \alpha (s, s')}{\sum_{\alpha \in \mathcal{A}} \sum_{s' \in S} \alpha (s, s')} = \frac{\sum_{\alpha \in \mathcal{A}} \alpha (s, s')}{\left| \{ \alpha \in \mathcal{A} \mid \alpha \text{ is enabled in } s \} \right|} \]

Only a few of the facts and notions described for models without probability change when adding probability:

- Statements are no longer deterministic, but probabilistic.
- A graph is still the general notion that does not specify whether transitions are labelled with statements. So for probabilistic models a graph can either refer to a DTMC or an MDP.
- Transitions must have probabilities greater than zero.
Then the definition and properties of a path can be adopted from before without change. To a path \( \pi = \langle t_1, t_2, \ldots \rangle \) starting in \( s \) can now be assigned the probability

\[
P[\pi] := \prod_{i \in \{1, \ldots, |\pi|\}} P[t_i]
\]

to be chosen from amongst all paths starting in \( s \).

Let \( \text{Path}_{s}^{\text{fin}} \) be the set of all paths with finite length starting in \( s \), and \( \text{Path}_{s}^{\text{max}} \) be the set of all paths with maximal length starting in \( s \), i.e. they either have infinite length or stop in an end state. Identifying each \( \pi^{\text{fin}} \in \text{Path}_{s}^{\text{fin}} \) with the subset of \( \text{Path}_{s}^{\text{max}} \) of all possible extensions of \( \pi^{\text{fin}} \), we can extend \( P \) to a probability measure on \( \sigma(\text{Path}_{s}^{\text{fin}}) \), the \( \sigma \)-algebra generated by \( \text{Path}_{s}^{\text{fin}} \).

- A computation tree is now built from a DTMC or MDP.

Figure 2.6 shows all formalisms describing miniworlds, and depicts their relationships.

Figure 2.6: Model formalisms

2.2 The properties

2.2.1 Basic properties

The properties that can be checked depend on the type of model and on the model checker. Some basic properties are:

- **unreachable code existence**, i.e. there exists a statement that does not generate any reachable transitions
2.2. The properties

- **end state validity**, which holds if all end states are labelled as valid. Otherwise we have a **deadlock**

- **assertions**, which check whether propositional formulae hold in specific states

- **correct **xr** and **xs** assertions,  i.e. if a channel is stated as exclusive read (xr) or exclusive send (xs) for a process, it holds

- **non-progress cycle existence**, i.e. there exists a reachable cycle with no progress states

- **acceptance cycle existence**, i.e. there exists a reachable cycle with an acceptance state

- **never claims**, which check whether some path in the computation tree has a given undesired or illegal behaviour.

2.2.2 Temporal logics

The most important and very powerful properties are propositional temporal logic formulae, which subsume most of the basic properties.

**Propositional temporal logics** extend propositional logic with **temporal operators** to be able to make statements about time. Therefore their algebras must contain different points in time and relationships between them. So we can use STSs and LTSs labelled with P. The paths of their computation tree starting from a given state s are the possible linear sequences of time starting at s.

Propositional temporal logics are often classified into the types **linear-time temporal logics**, which describe events along single computation paths, and **branching-time temporal logics**, which state properties of the complete computation tree. Their main representatives are the **linear-time logic (LTL)** and the **computation tree logic (CTL)**, respectively. **CTL** is a superset of LTL and CTL. **PCTL** is a probabilistic extension of CTL.

([Clarke Jr. et al., 1999]) first explains CTL and later CTL and LTL; so does ([Clarke and Draghicescu, 1988]), one of the first articles comparing CTL and LTL via CTL*. ([Clarke et al., 1986a]) takes the reversed order, so do the German lecture notes ([Schmitt, 2004]). ([Merz, 2000]) contains a short introduction to CTL and LTL, ([Menzel and Schmitt, 2001]) also, but in German. PCTL is explained in ([Rutten et al., 2004]) and shortly introduced in ([Parker et al., 2006]).

We also start with CTL and afterwards explain their subsets CTL and LTL. Then we introduce PCTL.

**CTL**. A **state formula** describes a property of a state s, i.e. it is interpreted in s. We can existentially quantify over all possible linear sequences of time
starting at $s$ with the path quantifier $E$, sometimes also denoted by $\exists$. We also use the universal path quantifier $A$, or $\forall$, as abbreviation for $\neg E \neg$. To state a property about the paths, we use a path formula, which is therefore interpreted over the given paths. The formula can exceed propositional logic by additionally using the linear-time operators:

- the **nexttime operator** $X g_1$
- the **until operator** $g_1 \mathbin{U} g_2$
- the operator $F g_1$, also $\diamondsuit g_1$, called **eventually** or **sometimes**, is an abbreviation for $\text{true} \mathbin{U} g_1$
- $G g_1$, alternatively $\Box g_1$, called **globally** or **always**, is an abbreviation for $\neg F \neg g_1$
- the **release operator** $g_1 \mathbin{R} g_2$ is an abbreviation for $\neg (\neg g_1 \mathbin{U} \neg g_2)$.

So the chosen operator basis is $\{ \lor, \neg, E, U, X \}$ since all other binary Boolean operators, the universal path quantifier, and the linear time operators $F, G$, and $R$ are abbreviations thereof. Hence the following syntax description is sufficient:

\[
\begin{align*}
\langle \text{state-formula} \rangle &::= \langle \text{atomic proposition} \rangle \mid \langle \text{state-formula} \rangle \lor \langle \text{state-formula} \rangle \mid \\
&\quad \neg \langle \text{state-formula} \rangle \mid E (\langle \text{path-formula} \rangle) \\
\langle \text{path-formula} \rangle &::= \langle \text{state-formula} \rangle \mid \langle \text{path-formula} \rangle \lor \langle \text{path-formula} \rangle \mid \\
&\quad \neg \langle \text{path-formula} \rangle \mid \langle \text{path-formula} \rangle \mathbin{U} \langle \text{path-formula} \rangle \mid \\
&\quad X \langle \text{path-formula} \rangle
\end{align*}
\]

Parentheses can be inserted for disambiguation.

CTL* is the set of all state-formulae, since a path-formula is only sensible within a state-formula.

Let $G$ be a graph labelled with $P$, $f_i$ state formulae, and $g_i$ path formulae. We write $s \models f_1$ iff $f_1$ holds in state $s$ of $G$, and $\pi \models g_1$ iff $g_1$ holds for path $\pi = \langle t_1, t_2, \ldots \rangle$ of $G$. We say $G$ is a **model for** $f_1$ iff $(G, s) \models f_1$ for all states $s$. This is the origin of the terminology "model checker", since the tool checks whether $G$ is a model for $f_1$. 
2.2. The properties

To define the semantics of CTL* we inductively define $\models$:

\[
\begin{align*}
s & \models p \quad \text{iff} \quad p(s) = tt \\
s & \models \neg f_1 \quad \text{iff} \quad s \not\models f_1. \\
s & \models f_1 \lor f_2 \quad \text{iff} \quad s \models f_1 \text{ or } s \models f_2. \\
s & \models E(f_1) \quad \text{iff} \quad \text{there exists a path } \pi \text{ starting in } s \text{ such that } \pi \models g_1. \\
\pi & \models f_1 \quad \text{iff} \quad \pi \text{ starts in } s \text{ and } s \models f_1. \\
\pi & \models \neg g_1 \quad \text{iff} \quad \pi \not\models g_1. \\
\pi & \models g_1 \lor g_2 \quad \text{iff} \quad \pi \models g_1 \text{ or } \pi \models g_2. \\
\pi & \models Xg_1 \quad \text{iff} \quad \langle t_2, t_3, \ldots \rangle \models g_1. \\
\pi & \models g_1 \lor g_2 \quad \text{iff} \quad \text{there exists a } k > 0 \text{ such that } \langle t_k, t_{k+1}, \ldots \rangle \models g_2 \text{ and for all } 0 < j < k \text{ we have } \langle t_j, t_{j+1}, \ldots \rangle \models g_2.
\end{align*}
\]

\(\text{CTL}\) is the subset of CTL* that restricts the path-formulae to:

\[
\langle \text{path-formula} \rangle \ ::= \ X \langle \text{state-formula} \rangle \ | \ \langle \text{state-formula} \rangle \ U \langle \text{state-formula} \rangle \ | \ \neg (X \langle \text{state-formula} \rangle) \ | \ \neg (\langle \text{state-formula} \rangle) \ U \langle \text{state-formula} \rangle
\]

So CTL only contains the following temporal operators, denoted as branching-time operators:

\[
\begin{align*}
E(X f_1), \ E(\neg X f_1), \ E(f_1 \ U f_2), \ E(\neg (f_1 \ U f_2)).
\end{align*}
\]

Just as in CTL*, we also allow the abbreviations \(E(F f_1), E(\neg F f_1), E(G f_1), E(\neg G f_1), E(f_1 R f_2), E(\neg (f_1 R f_2))\), and all of these branching-time operators with the universal path quantifier instead of the existential one.

\(\text{LTL}\) is the subset of CTL* that only allows formulae with a single path quantifier at the beginning. So LTL is the set \(A(\langle \text{path-formula} \rangle) \cup E(\langle \text{path-formula} \rangle)\) with

\[
\langle \text{path-formula} \rangle \ ::= \ \langle \text{atomic proposition} \rangle \ | \ \langle \text{path-formula} \rangle \ U \langle \text{path-formula} \rangle \ | \ \neg (\langle \text{path-formula} \rangle) \ | \ \langle \text{path-formula} \rangle \ U \langle \text{path-formula} \rangle \ | \ X \langle \text{path-formula} \rangle
\]

Therefore we can again use the abbreviations for the alternative linear-time operators, as in CTL*.

\(\text{PCTL}\). PCTL is described in [Rutten et al., 2004]. We state equivalent definitions and some consequences to better relate PCTL to CTL.

If the models exhibit probability, we usually also want to formulate properties about it. Since CTL is insufficient, we extend it to the probabilistic computation tree logic (PCTL) by including the \textbf{probabilistic operators}:

\[
\begin{align*}
P_{\text{top}}(g_1), \text{ which is true in a given state } s \text{ iff the probability for taking a path from } s \text{ that satisfies the path-formula } g_1 \text{ is } \geq p \text{ for } p \in [0, 1].
\end{align*}
\]
• $E_{ac}(f_1)$, which is true in a given state $s$ iff the expected cost to reach from $s$ a state that satisfies the state-formula $f_1$ is $\leq c$ ($c \in \mathbb{R}_{\geq 0}$).

PCTL also uses the nexttime operator and the until operator. The latter is additionally available in the modified form $f_1 U \leq^k f_2$ ($k \in \mathbb{N}_{\geq 0}$). It is called the **bounded until operator**, since $f_2$ must be satisfied within $k$ time steps.

So the syntax is:

(state-formula) ::= (atomic proposition) | (state-formula) V (state-formula) |
                  ¬(state-formula) | $P_{ap}(\text{path-formula})$ | $E_{ac}(\text{state-formula})$

(path-formula) ::= X (state-formula) | (state-formula) U (state-formula) |
                  $E_{ac}(\text{state-formula}) U \leq^k (\text{state-formula})$

Let $G$ be a DTMC or an MDP labelled with $P$, $f_1$ state formulae, $g_i$ path formulae, $s$ a state of $G$, and $\pi = \langle t_1, t_2, \ldots \rangle$ a path of $G$. Again we describe the semantics by inductively defining $\models$:

- $s \models p$ iff $p(s) = \text{tt}$ ($p \in P$).
- $s \models \neg f_1$ iff $s \not\models f_1$.
- $s \models f_1 \lor f_2$ iff $s \models f_1$ or $s \models f_2$.
- $s \models P_{ap}(g_i)$ iff $\mathbb{P}_{\sigma(\text{Path}^s)}\left[ \{ \pi \in \text{Path}^s_{\text{max}} \mid \pi \models g_i \} \right] \propto p$
  (for all non-deterministic choices, if $G$ is an MDP).
- $s \models E_{ac}(f_1)$ iff $\mathbb{E}_{\sigma(\text{Path}^s)}\left[ \sum_{t_i = 1}^{t_i = n}^{j = \text{(source of } t_j) \neq f_1} C(t_i) \right] \propto c$
  with the sum being $\infty$ for paths not reaching $f_1$
  (for all non-deterministic choices, if $G$ is an MDP).
- $\pi \models X f_1$ iff (destination of $t_1$) $\models f_1$.
- $\pi \models f_1 U \leq^k f_2$ iff $\exists i \in \mathbb{N} : 0 < i \leq k$ and (source of $t_i$) $\models f_2$ and
  $\forall j < i :$ (source of $t_j$) $\models f_1$.
- $\pi \models f_1 U f_2$ iff there exists a $k > 0$ such that $\pi \models f_1 U \leq^k f_2$.

$\models$ is well-defined because all $\{ \pi \in \text{Path}^s \mid \pi \models g_i \}$ are measurable sets, as proved in [Moshe, 1985] via Büchi automata.

Negation for path-formulae as in CTL is not needed anymore, since

\[
\begin{align*}
P_{ap}(-g_1) & \Leftrightarrow \mathbb{P}_{\sigma(\text{Path}^s)}\left[ \{ \pi \in \text{Path}^s \mid \pi \not\models g_i \} \right] \propto p \\
& \Leftrightarrow 1 - \mathbb{P}_{\sigma(\text{Path}^s)}\left[ \{ \pi \in \text{Path}^s \mid \pi \models g_i \} \right] \propto p \\
& \Leftrightarrow P_{\exists[1-p]}(g_i)
\end{align*}
\]

(for all non-deterministic choices, if $G$ is an MDP)

where $\leq_1 := \geq$, $\geq_1 := \leq$, $\leq := \leq$, $\geq := \geq$. $P_{ap}(-g_1)$ is therefore used as abbreviation for $P_{\exists[1-p]}(g_i)$. Having the negation, we can again use the abbreviations for the alternative linear-time operators, as in $CTL^*$. Additionally, we have the **bounded linear-time operators**:
2.3. PRISM

- \( F^{\leq k} f_1 \), also \( Q^{\leq k} f_1 \), which stand for \( \text{true} U^{\leq k} f_1 \)
- \( G^{\leq k} f_1 \), alternatively \( \square^{\leq k} f_1 \), is an abbreviation for \( \neg F^{\leq k} \neg f_1 \).

PCTL does not contain the CTL* path quantifiers. In some cases we can use \( \mathcal{P}_{\mathsf{sep}} \) instead:

\[
E(g_1) \iff \mathcal{P}_{>0}(g_1) \text{ for } g_1 \in \{ G f_1, f_1 R f_2 \}.
\]

\[
A(g_1) \iff \mathcal{P}_{\geq 1}(g_1) \text{ for } g_1 \in \{ F f_1, f_1 U f_2 \}.
\]

Counterexamples to the implications not given can be constructed through paths with probability zero, e.g. with the path \( \text{init} \to \text{init} \) of figure \( \ref{fig:counterex} \).

Thus the following branching-time operators are allowed as abbreviations:

- The ones from CTL except \( E(G f_1) \), \( E(f_1 R f_2) \), \( A(F f_1) \), \( A(f_1 U f_2) \).
- Additionally the new \textbf{bounded branching-time operators} \( E(F^{\leq k} f_1) \), \( E(\neg F^{\leq k} f_1) \), \( E(G^{\leq k} f_1) \), \( E(\neg G^{\leq k} f_1) \), \( E(f_1 U^{\leq k} f_2) \), \( E(\neg (f_1 U^{\leq k} f_2)) \), and all of these branching-time operators with the universal path quantifier instead of the existential one.

2.3 PRISM

\cite{kwiatkowska2004model} is a short overview of the tool PRISM. \cite{kwiatkowska2003introduction} introduces PRISM and its models and properties, the latter are thoroughly investigated in \cite{rutten2004introduction}. \cite{parker2006prism} is the user’s guide for PRISM. Most of the literature about PRISM can be found on its homepage, \cite{prism}.

\textbf{PRISM} is an acronym for \textbf{p}robabilistic \textbf{S}ymbolic \textbf{M}odel \textbf{C}hecker. \textbf{Symbolic} means that the model checking algorithm is based on manipulations of Boolean formulae, as opposed to \textbf{explicit} model checking algorithms, which use direct representations of the state space. Using symbolic structures to represent the model can lead to significantly smaller memory requirements. PRISM uses multi-terminal binary decision diagrams (MTBDDs), which extend binary decision diagrams (BDDs) to be able to represent arbitrary function ranges, not just \{0,1\} (cf. \cite{rutten2004introduction} for a short description or \cite{pajtai1997probabilistic} for a thorough analysis).

Being a probabilistic model checker, PRISM does not give a counterexample if a property does not hold.
2.3.1 The models

PRISM can check MDPs, and therefore also DTMCs. PRISM also makes use of continuous time Markov chains (CTMCs), in which transitions can occur in real time, in contrast to the discrete time-steps we had up to now. As we are only using MDPs and DTMCs, we do not investigate model checking CTMCs.

The language for specifying the models is a simplification of Reactive Modules, described in [Alur and Henzinger, 1996]. Hence it is not an imperative language, but declarative.

Data. Data is specified by variables of type Boolean or finite ranged integer (e.g. in appendix [A.1] line 9 and 10). Each variable definition can contain an initial value, which is assigned to the variable in the init state. If an initial value is not given, the minimum is taken. Variables defined within a module (see below) are local to that module, variables defined outside of all modules are global.

Behaviour. In PRISM a statement is called a command and has the structure [action] guard => p1: update1 + ... + pn: update_n (e.g. in appendix [A.1] line 20 to 26), with (p1, ..., pn) being expressions evaluating to a probability vector. action is an action label for synchronization and can be the empty string. guard is a propositional formula over all variables describing in which states the command is enabled, i.e. in which states the updates update1,...,update_n generate transitions. These transitions are annotated with the probabilities p1, ..., pn, respectively. update_i describes local variable assignments with a propositional formula using variable decoration like the specification language Z. Local variables of the module (and global variables) that do not get assigned new values are implicitly defined as constant.

For modularization and concurrency, commands can be grouped together into modules, usually corresponding to components in the miniworld. To ease the multiple use of one component, its module can be duplicated by module renaming (e.g. in appendix [A.1] line 89 to 95). The module's local variables must be renamed, its other variables and its action labels may be renamed. The renaming is done by textual substitution.

Parallelism of the modules' behaviour is simulated by scheduling the modules, i.e. by interleaving their transitions (cf. figure [2.7]). This is put into effect for Module1 and Module2 by parallel composition:

- It generates a new module $M$ that subsumes all commands of Module1 and Module2.
- If the action labels from a command $c = \text{[action]} \ \text{guard} \Rightarrow p_1: \text{update}_1 + \ldots + p_n: \text{update}_n$ of Module1 and from a command $c' = \text{[action']} \ \text{guard'} \Rightarrow p'_1: \text{update}'_1 + \ldots + p'_{n'}: \text{update}'_{n'}$ of Module2 are the same, i.e. action=action', and Module1 and Module2 synchronize on action, not the commands $c$ and $c'$ are propagated to $M$, but the combined command

\[
\text{[action]} \ \text{guard} \& \text{guard'} \Rightarrow \\
\begin{align*}
p_1 \cdot p'_1: \text{update}_1 & \& \text{update}'_1 + \ldots + p_n \cdot p'_{n'}: \text{update}_n & \& \text{update}'_{n'} \\
+ p_1 \cdot p'_1: \text{update}_1 & \& \text{update}'_1 + \ldots + p_n \cdot p'_{n'}: \text{update}_n & \& \text{update}'_{n'} \\
\end{align*}
\]
So synchronization of \( c \) and \( c' \) is really accomplished, since their updates are executed atomically.

Parallel compositions and their degree of synchronization are regulated by the following process algebraic composition operators:

- \(\text{Module}_1 || \text{Module}_2\) for full parallel composition, causing synchronization on all actions which appear in both \(\text{Module}_1\) and \(\text{Module}_2\). This is PRISM's standard operator.
- \(\text{Module}_1 ||| \text{Module}_2\) for asynchronous parallel composition, i.e. fully interleaved, with no synchronization.
- \(\text{Module}_1[[\text{action}_1, \ldots, \text{action}_n]] \text{Module}_2\) for restricted parallel composition, i.e. synchronizing on the actions from \(\{\text{action}_1, \ldots, \text{action}_n\}\) which also appear in both \(\text{Module}_1\) and \(\text{Module}_2\).
- \(\text{Module}_1/\{\text{action}_1, \ldots, \text{action}_n\}\) for hiding of actions \(\text{action}_1, \ldots, \text{action}_n\) in \(\text{Module}_1\), i.e. substituting the empty string for them.
- \(\text{Module}_1\{\text{action}_1 \leftarrow \text{action}_2, \ldots, \text{action}_{2n-1} \leftarrow \text{action}_{2n}\}\) for renaming of actions \(\text{action}_1, \ldots, \text{action}_{2n-1}\) to actions \(\text{action}_2, \ldots, \text{action}_{2n}\), respectively, in \(\text{Module}_1\).

Figure\(\ref{fig:parallel-composition}\) shows an example for a parallel composition of \(\text{Module}_1\) and \(\text{Module}_2\) without any synchronization.

The parallel composition of all modules of the model specification results in the system module, which describes the complete miniworld. By using the keyword \texttt{mdp} at the start of the model specification, the system module is interpreted as MDP. So the statements of the MDP are the system module's commands.

Using the keyword \texttt{dtmc} results in the corresponding DTMC by ignoring the labelling of the MDP and by normalization, as described on page\(\ref{page:dtmc}\).

Both models are additionally annotated with propositional variables, optionally also with a cost structure. This is done in an extra section of the model specification starting with \texttt{rewards} and ending with \texttt{endrewards} to annotating certain states or transitions with real-valued costs. These costs for transitions and states can then be transformed into a cost structure \(C\).

\textbf{Note.} The strict independence depicted in figure\(\ref{fig:parallel-composition}\) is weakened if a command \(c\) of a module \(M\) does not solely contain local variables: If \(c\) is executed and updates of \(c\) contain non-local variables, the new local state of \(M\) does not only depend on the old local state of \(M\), but also on global variables and local states of other modules. If the guard of \(c\) contains non-local variables, \(c\) might only sometimes be executable in a fixed local state of \(M\), dependent on global variables and local states of other modules.

When two commands synchronize, the combined command is generated by conjugating guards, multiplying probabilities and inserting updates into the set of all updates of the command. Since all these operations are associative and
commutative, full parallel composition and asynchronous parallel composition inherit associativity and commutativity. Restricted parallel composition also inherits commutativity, but not associativity, as

\[
\left( \text{Module}_{\text{actionset} = \{a\}} || \text{Module}_{\text{actionset} = \{b\}} \right) || \text{Module}_{\text{actionset} = \{a, b\}}
\]

\[
\neq \left( \text{Module}_{\text{actionset} = \{a\}} || \text{Module}_{\text{actionset} = \{b\}} \right) || \text{Module}_{\text{actionset} = \{a, b\}}
\]

The left-associative case synchronizes on action \(a\), whereas the right-associative case does not synchronize at all.

Besides the described essential features, PRISM additionally offers:

- **constants** (e.g. in appendix A.1 line 3)
- a more flexible **initial states definition**
- **ranges and functions** (e.g. in appendix A.2 line 58) for expressions
- **formulae** for defining and reusing expressions (e.g. in appendix A.6 line 6)
2.3.2 The properties

Basic properties

PRISM

- always checks for end states and optionally adds self-loops
- does not check for unreachable code existence, i.e. commands whose guards are not satisfiable for any reachable state are not reported.

Temporal logics

PRISM can check PCTL-formulae using the following syntax for our operator basis from page 14:

- ! for negation
- | for disjunction
- \( P \bowtie \langle p \rangle \) for \( \mathcal{P}_{\text{prop}} \), so \( \bowtie \) is the relation \(<, =, >\) or \(>,\) and \(\langle p \rangle\) an expression evaluating to a double within \([0,1]\)
- \( R \bowtie \langle c \rangle F \) for \( \mathcal{E}_{\text{net}} \), so \( \bowtie \) is the relation \(<, =, >\) or \(>,\) and \(\langle c \rangle\) an expression evaluating to a non-negative double
- \( X \) for the nexttime operator \( X \)
- \( U \) for the until operator \( U \)
- \( U \leq \langle k \rangle \) for the bounded until operator \( U^{\leq k} \), so \(\langle k \rangle\) is an expression evaluating to a non-negative integer.

Hence we have the following syntax for the formulae:

\[
\begin{align*}
\langle \text{atomic prop} \rangle & ::= \text{true} \mid \text{false} \mid \langle \text{expression evaluating to a Boolean} \rangle \\
\langle \text{state-formula} \rangle & ::= \langle \text{atomic prop} \rangle \mid \langle \text{state-formula} \rangle \mid \langle \text{state-formula} \rangle \mid \langle \text{state-formula} \rangle \\
& \quad \mid !\langle \text{state-formula} \rangle \mid P \bowtie \langle p \rangle \mid \langle \text{path-formula} \rangle \\
& \quad \mid R \bowtie \langle c \rangle F \langle \text{state-formula} \rangle \\
\langle \text{path-formula} \rangle & ::= X \langle \text{state-formula} \rangle \mid \langle \text{state-formula} \rangle U \langle \text{state-formula} \rangle \mid \langle \text{state-formula} \rangle U \leq \langle k \rangle \langle \text{state-formula} \rangle
\end{align*}
\]

Additionally, we can use:

- \( \langle \text{state-formula} \rangle \land \langle \text{state-formula} \rangle \) for conjunction
- \( \langle \text{state-formula} \rangle \Rightarrow \langle \text{state-formula} \rangle \) for implication
• $P_\Rightarrow\Phi$ ([path-formula]) (without $\models\Phi(p)$) as outermost operator for DTMCs, which returns the actual probability, not a Boolean value.

• $P_{\text{max}}\Rightarrow\Phi$ ([path-formula]) and $P_{\text{min}}\Rightarrow\Phi$ ([path-formula]) as outermost operators for MDPs. They return the maximum and minimum probabilities, respectively, over all non-deterministic choices.

• $R\Rightarrow\Phi$ (state-formula)] as outermost operator for DTMCs, which returns the actual expected reward.

• $R_{\text{max}}\Rightarrow\Phi$ (state-formula]) and $R_{\text{min}}\Rightarrow\Phi$ (state-formula]) as outermost operators for MDPs. They return the maximum and minimum expected rewards, respectively, over all non-deterministic choices.

In PRISM a model $M$ satisfies a PCTL property $F$ iff $(M, s) \models F$ for all reachable states $s$.

To ease defining properties, PRISM additionally offers:

• **constants**

• **labels** for defining and reusing atomic propositions. The predefined label "init" is true in initial states, "deadlock" is true in states where deadlocks were found (and fixed by adding self-loops).

• **filters**, which are also atomic propositions and define states for the P operators. They are delimited by braces and put just before the closing square bracket.

As we do not use CTMCs, we do not consider the continuous stochastic logic (CSL) needed for CTMCs.

### 2.4 SPIN

(Holzmann, 2004b), the successor of (Holzmann, 1992), is the definitive book on SPIN. (Holzmann, 1997) is a short introduction. There is a wide range of literature available online, many of them on SPIN’s homepage (Spin, 2007). (Holzmann, 2004a) is the manual for SPIN; a practical and very short introduction can be found in (Holzmann, 2001). The slides (Ruys, 2002) and (Ruys, 2004) have extensive graphical explanations. Lecture notes can be found in (Ireland, 2002). (Dwyer and Hatcliff, 2001) explains the algorithms SPIN uses.

SPIN is an acronym for simple PROMELA interpreter (cf. section 2.1), but it has evolved into a full scale explicit on-the-fly model checker. **On-the-fly** means that the states of the model are generated only when needed, therefore possibly saving space and time by ignoring states irrelevant to the current check. If the property being checked does not hold, SPIN gives a counterexample. SPIN was awarded the Software System Award by the Association for Computing Machinery (ACM) in the year 2001.
2.4. SPIN

2.4.1 The models

SPIN checks LTSs that are specified in PROMELA, the process meta language, which is concisely explained in [Gerth, 1997], extensively in [Holzmann, 2004b].

Model specifications in PROMELA consist of declarations of variables and of process types. The latter contain variable declarations and commands in imperative style. Thus we have a program counter (pc) as additional variable for each process. The syntax of PROMELA is C-like.

**Data.** Data is specified by finite-ranged variables of integer types, data type channel, or composed data types declared by array or typedef (cf. appendix B.1 line 10 to 13). Each variable definition can contain an initializer, i.e. an expression that is evaluated when the variable is initialized. If not present, the variable is initialized with the value zero. Variables defined within a process declaration or being one of its formal parameters are local to that process, variables defined outside of all process declarations are global.

Channels are used for sending and receiving data, called messages. These must be of the (composed) type specified for the channel, which is done either by declaring the channel to be of a specific type (cf. appendix B.1 line 13), or by assigning an initialized channel to it if it is uninitialized. Each channel has a capacity, which defines the number of messages the channel can store in a first in, first out (FIFO) buffer:

Channels with capacity greater zero are called buffered channel and transmit asynchronously, i.e. temporally decoupled.

Channels with capacity zero do not have a buffer and therefore provide synchronous transmission, i.e. sending and receiving is executed in parallel in one atomic step.

**Behaviour.** The statements of the LTS are PROMELA's basic commands (called basic statements in [Holzmann, 2004b]):

- a **variable assignment** with := (e.g. in appendix B.1 line 25)
- an **expression**, which blocks if it evaluates to false, i.e. to zero. Otherwise it is enabled - [Holzmann, 2004b] uses the notion executable instead. Special expressions are:
  - **skip** (e.g. in B.1 line 27), a synonym for the constant value one
  - **else** (cf. in B.1 line 56), which can be the first expression (called guard) in at most one sequence (see below) of a control flow construct. It evaluates to true iff all alternative sequences within the control flow construct block. A **sequence** blocks if its guard blocks
  - **timeout** (e.g. in B.1 line 81), which evaluates to true iff all alternative commands within the global context of the system block
  - **run** (e.g. in B.1 line 77), which has the side effect of creating a new instance of the given process type
an assertion with assert() (e.g. in [B.1] line 81), which reports a violation if a Boolean expression evaluates to false.

- an output with println() (e.g. in [B.1] line 82)

- a send or receive operation on a channel c:
The standard send operation ! (cf. [B.1] line 37) inserts the new message at the end of the FIFO buffer of c. If c is a buffered channel, a sorted send operation inserts the new message according to a numerical, lexicographical ordering. Both send operations are executable if c is initialized and its buffer is not full.
The standard receive operation ? (cf. [B.1] line 40) fetches the oldest message in the FIFO buffer of c. If c is a buffered channel, variations of the operation can fetch an arbitrary element of the buffer, which is called random receive operation, or read a message without removing it from the buffer. All receive operations are executable if c is initialized and its buffer is not empty.

Compound commands (named compound statements in [Holzmann, 2004b]) are composed of the basic commands and the following control flow constructs:

- concatenation with ; or −> (e.g. in appendix [B.1] line 81) of basic and compound commands to sequences of the form command [: command ]

- goto label_name (e.g. in [B.1] line 62) to jump to a label set by label_name: (e.g. in [B.1] line 24). Labels starting with the string "end" (e.g. in [B.1] line 39), "progress" (e.g. in [B.0] line 16), or "accept" are special, as they mark valid end states, progress states, or acceptance states, respectively.

- selection of the form if :: sequence [: sequence ] fi (cf. [B.1] line 60 to 68), which selects non-deterministically among all executable sequences

- repetition of the form do :: sequence [: sequence ] od (cf. [B.1] line 76 to 79), which repeatedly selects non-deterministically among all executable sequences

- break (e.g. in [B.1] line 78) within a repetition to terminate it (the innermost repetition)

- exception handling of the form sequence unless sequence, which successively executes each command from the first sequence, but beforehand always checks whether the second sequence is executable. If it is, the first sequence is no longer executed and control transfers to the second sequence. Otherwise execution of the first sequence continues. If it terminates, the second sequence is ignored.

The concurrent components of the miniworld are specified by processes, which are declared (e.g. in appendix [B.1] line 15 to 70) by
2.4. SPIN

```plaintext
proctype proc_name (formal_parameters)
{
    sequence
}
```

An instance of the process declaration `proc_name`, i.e. a process \( p \), is dynamically constructed with the expression `run proc_name (actual_parameters)` (e.g. in [5.1] line 77): \( p \)'s local variables are instantiated and initialized, the program counter points to the first command of `proc_name`. The flow of control in \( p \)'s execution is guided by the control flow constructs in `proc_name`. \( p \) terminates when it reaches the end of the body of `proc_name`. Only processes that have already terminated can be deconstructed, which takes place in reversed order of their creation.

If the declaration `init { sequence }` is present (e.g. in [5.1] line 72 to 83), the `special process init` is instantiated once at the beginning, i.e. in the state `init` of the LTS.

The `system` is the set of all global variables and all running processes, therefore describing the LTS.

As PROMELA is imperative, the program counter is increased when executing a basic command. Consequently the latter is implicitly parameterized with the program counter of the associated process. For example, if in a state \( s \) of the LTS two processes \( P_1 \) and \( P_2 \) can execute the basic statement `printn("foo")`, the result is not the same since the program counters being increased differ. Therefore, using solely the basic commands as statements for the LTS can result in non-deterministic statements. If we want deterministic statements, we need to make the program counter an explicit argument, i.e. to index the basic commands accordingly, e.g. `printn("foo")_{pc of \( P_1 \)}` and `printn("foo")_{pc of \( P_2 \)}`.

Asynchronous execution of the processes, i.e. parallelism of the components' behaviour, is simulated by non-deterministically scheduling the running processes. In the LTS this corresponds to interleaving the transitions generated by the enabled statements in all possible orders (cf. figure 2.7). `d_step` or `atomic` (e.g. on page [5.7] restrict the degree of simulated parallelism, i.e. the degree of interleavability, by constraining execution elsewhere in the system: A sequence within `d_step{ ... }` is executed indivisibly, i.e. without any interleaving. A sequence within `atomic{ ... }` is executed with exclusive privilege whenever possible.

If the declaration `never { sequence }` is present in the model specification, a special process called `never claim` is instantiated once at the beginning. `sequence` can contain the special functions `enabled()`, `pc_value()`, `_last`, `remoterefs`, and `n_p`, as well as commands that should not have side-effects. `sequence` is interpreted as a series of conditions on the system state, which must all become true for a path in the computation tree to be reported as undesired behaviour.

Besides the described essential features, PROMELA additionally offers:

- **initial process instance definitions** to specify processes being active immediately in `init`
- **type casting**
- **blocking receive operations** if the message does not satisfy a given `pattern`
• *xr* or *xs* assertions (e.g. in appendix A.1 line 21 and 22), which state that a channel is read exclusive or send exclusive, respectively, for a process

• braces for disambiguation, e.g. `{command1; command2} unless command3 versus command1; {command2 unless command3}`

• preprocessor macros (e.g. in A.1 line 6 for a constant) and inline

• symbolic constants with `mtype`

• conditional expressions with `(case_expr -> true_expr : false_expr)`, which have the value of `false_expr` if `case_expr` evaluates to zero, otherwise the value of `true_expr`

• hidden variables, which are not part of the system state

• some special variables.

2.4.2 The properties

Basic properties

SPIN's basic properties comprise the state properties

• end state validity

• assertions

called safety checks, the path properties

• non-progress cycle existence

• acceptance cycle existence

called liveness checks, and the properties

• unreachable code existence

• correct *xr* and *xs* assertions

• never claims.

So assertions and never claims are properties about the miniworld, not part of the miniworld. SPIN treats a never claim this way, too, as it is not part of the system. To monitor every execution step of the system, the latter and the never claim are executed in lockstep. As a result, a blocking never claim aborts the current execution of the system, i.e. prunes its computation tree, and therefore reduces the LTS. But SPIN does integrate assertions into the system, which therefore become part of the LTS.

Most of these properties are used in section 6.2
2.4. SPIN

Temporal logics

SoLely checking for never claims can detect undesired finite behaviour.
To check for a certain infinite behaviour, the basic properties of acceptance
ycles and never claims need to be combined. The infinite paths of the com-
pilation tree can be matched by an appropriate never claim with repetition
 structs and acceptance cycle checks. This kind of never claim implements a
B"uchi automaton, described in [Clarke Jr. et al., 1999 chapter 9], [Schmitt,
2007], and [Mukund, 1997]: an automaton which accepts certain infinite paths—a
so called \( \omega \)-automaton. Its set of accepted infinite paths is the corresponding
\( \omega \)-regular property. For every LTL formula \( F \) there exists a \( B"uchi \) automaton
that accepts exactly the paths satisfying \( F \), but not vice versa. Hence SPIN
can verify LTL formulae, but the combinations of SPIN’s basic checks are more
powerful than LTL.

SPIN can mechanically translate LTL formulae into never claims.
Our operator basis from page\textsuperscript{13} needs to have the following syntax for this:

- ! for negation
- \( \| \) for disjunction
- U for the until operator U
- X for the nexttime operator X

Thus LTL is the set \( A(\text{path-formula}) \cup E(\text{path-formula}) \) with

\[
\begin{align*}
(\text{atomic prop}) & := \text{true} | \text{false} | \text{symbolic name for a Boolean expression} \\
(\text{path-formula}) & := (\text{atomic proposition}) | (\text{path-formula}) \| (\text{path-formula}) \\
& | ! (\text{path-formula}) | (\text{path-formula}) \text{ U } (\text{path-formula}) | \\
& X (\text{path-formula})
\end{align*}
\]

So Boolean expressions need to be defined by symbolic names, which then can
be used as atomic propositions. Parentheses can be inserted for disambiguation.

In SPIN a model \( M \) satisfies an LTL property \( F \) iff \( (M, \text{init}) \models F \).

LTL formulae without the next operator are stutter invariant or stutter-
closed: They are insensitive to stuttering of the states (or their corresponding
subset of \( F \)) on a path (cf. [Clarke Jr. et al., 1999 chapter 10] or [Holzmann,
2004b, chapter 6]). This is a necessary property for partial order reduction,
which therefore has to be disabled for LTL formulae containing the next oper-
or. But this is not a severe problem: As the LTS contains all possible inter-
leavings, very little is known about the next state. Hence the next operator is
rarely used.

Additionally, we can use the following operators:

- \( \langle \text{state-formula} \rangle \&\& \langle \text{state-formula} \rangle \) for conjunction
• \((\text{state-formula}) \rightarrow (\text{state-formula})\) for implication
• \((\text{state-formula}) \leftarrow (\text{state-formula})\) for equivalence
• \(\square(\text{path-formula})\) for the operator \(\Box\)
• \(\langle \rangle (\text{path-formula})\) for the operator \(\Diamond\)
• \((\text{state-formula}) \lor (\text{state-formula})\) for the release operator \(\text{R}\)

### 2.5 State space explosion

A major domain of model checking is the analysis of concurrent behaviour. As the system’s performance increases with the extent of parallelization, a high degree thereof is sought after. Hence the systems under consideration often have very little synchronization between their modules. Therefore model checking often constructs the systems using almost full parallel composition of their modules (cf. section 2.3.1). This causes the state space to exponentially blow up with the number of modules, as almost all combinations of the modules’ local states are inspected. E.g. if we use a module with \(c\) states \(n\) times, the state space will have the size \(c^n\) (cf. figure 2.7). If the module’s size grows with \(n\), even a size of \(n^n\) for the state space is possible. This exponential or super-exponential blow-up is referred to as state space explosion or state explosion.
Chapter 3

Introduction to the leader election problem

Good introductions to the general concepts of distributed algorithms are (Lynch, 1997) and [Tel, 1994], but they do not investigate anonymous networks. These and appropriate leader election algorithms are subject of [Itai and Rodeh, 1981], its revised version [Itai and Rodeh, 1990], of [Fokkink and Pang, 2001], and its revised and condensed version [Fokkink and Pang, 2005].

Leader election solves the problem of distributively selecting a unique node, called the leader, in a network of nodes. Distributively implies that all nodes execute the same local algorithm, terminate, and thereupon know whether they are the leader.

The following section 3.1 introduces various network models with their parameters.
Section 3.2 investigates the network model we choose for this thesis and introduces corresponding leader election algorithms.

3.1 Network models

A network can be described by a graph with nodes and edges:
The nodes, also called processes or processors, are the computing elements. The edges, also called connections or links, do not need to be physical. They determine the possibilities of communication, i.e. between which nodes messages of the specified type can be sent or received. The edges can either be unidirectional, also called directed, to only allow simplex transmissions, i.e. transmissions in the given direction, or bidirectional, also designated as undirected, to allow duplex transmissions, i.e. transmissions in both directions.
Connections can be implemented by channels, which can have multiple sources and sinks. Channels with buffers transmit asynchronously, i.e. temporally decoupled. If the message transmissions for a channel e are first come, first
served (FCFS), i.e. if the order of the messages received from \( c \) is the same as the order of the messages sent to \( c \), the buffer of \( c \) is a queue or first in, first out (FIFO) buffer. Unbuffered channels transmit synchronously, i.e. perform rendezvous message passing.

A major aspect for the leader election problem is the topology of the network, i.e. the structure of the connections of the nodes, as most leader election algorithms are designed for specific topologies. If, for instance, the nodes are arranged on a bidirectional line, the two nodes at the ends can both induce a message transmission along the line, and the one or two nodes where the two messages meet are the only leader candidates. This procedure takes advantage of the facts that there are exactly two end nodes and that the messages must at some point cross each other.

A topology is symmetric if all nodes in the network have the same relative location. Examples are rings, hypercubes, and complete graphs. In these cases no asymmetry of the topology can be exploited, as is done in the example just mentioned.

Another important aspect is whether the nodes are anonymous, i.e. indistinguishable. If they are, we talk about anonymous networks.

In concurrent systems, anonymity can be an important aspect, for example in ubiquitous computing, i.e. if computation is present everywhere by integrating it into the environment. Anonymity for the users can only be guaranteed definitely if the nodes are anonymous, too.

Non-anonymous networks often use nodes with unique identification numbers (IDs), which entail a further problem: If the size of the IDs is small, they can be transmitted without causing much traffic, but using many nodes, the identities may collide. On the other hand, if their size is large, they are unique but cause a lot of traffic being transmitted, furthermore administration becomes costly.

For non-anonymous networks with each node having a unique ID, a stronger definition of leader election additionally requires that upon termination each node knows who the leader is. In some literature this condition is also used for anonymous networks, which is misleading since the nodes do not have a name or reference to the leader at the moment of election. Of course once a leader is elected, it can initiate a de-anonymization algorithm, which gives each node a unique ID. For example the leader gives itself the ID 1, its \( n \) direct, unnamed neighbours the IDs \( 1 \_1 \) to \( 1 \_n \), and makes each of them recursively do the same thing, using their ID instead of 1. Thereafter each node \( 1 \_x_1 \ldots x_m \) knows that the leader is node 1 and reachable through the path \( 1 \_x_1 \ldots x_m \rightarrow 1 \_x_1 \ldots x_{m-1} \rightarrow \ldots \rightarrow 1 \_x_1 \rightarrow 1 \).

So requiring the stronger definition from above for anonymous networks is reverse to causality, as the leader election determines the IDs.

A symmetric network has a symmetric topology and anonymous nodes, since otherwise the distinguishable nodes can impose asymmetry on the network. That is why leader election in symmetric networks is also known as symmetry breaking.
3.2 Leader election for anonymous, unidirectional rings

3.2.1 Motivation

Our miniworld will always be a leader election algorithm for anonymous rings with directed edges and asynchronous transmissions.

We fix this special application to have a bounded problem domain as manageable use case for checking the severity of state space explosion, i.e. for checking the usability of model checking.

We choose an algorithm since state space explosion is too severe when taking some source code as miniworld and automatically generating the corresponding model. Because the models are able to describe concurrency, distributive algorithms are the strength of model checking. Leader election is a suitable choice as it is a fundamental distributive algorithm. For example, its symmetry breaking is used to enable the execution of a centralized protocol in a decentralized environment like an anonymous network. An advantage of decentralized environments are their fault tolerance. If, for instance, a node in a token ring breaks, it can simply be taken off the ring. A leader election then recovers from the token loss.

Having a symmetric network, we only need to design one algorithm, which is executed by every node. We have no asymmetry to exploit, but need to deal with symmetry breaking: [Anghin, 1980] theorem 4.2) proved that no deterministic algorithm can elect a leader in an anonymous ring. Having a ring with four nodes, for instance, there is always a schedule such that antipodal nodes are in the same local state for all times, thus preventing leader election.

Hence we have to use randomness to solve the problem. As random algorithms are becoming more and more popular, model checking them is an interesting task. But as probability is an additional demand on model checking, negative results for our algorithms might not apply to non-probabilistic algorithms.

A ring is one of the simplest symmetric topologies. It has a simple messaging technique, i.e. we do not need to be concerned about some complicated routing protocol, such as flooding. Hence our models will always be the n-fold composition of a component containing one node and its connection to the next node. As bidirectional rings can be simulated by unidirectional ones, it is sufficient to investigate unidirectional rings.

Having very little synchronization yields a higher performance of the system, and state space explosion is more severe (cf. section 2.3). Hence using asynchronous transmissions is another suitable choice.

Let \( n \) be the number of nodes in the ring, i.e. the size of the ring, also denoted the size of the problem instance. [Itai and Rodeh, 1981] theorem 3.1) shows that no Las-Vegas algorithm exists for anonymous rings that computes \( n \): The argumentation is similar to that of [Anghin, 1980] mentioned before. If a ring of size \( n \) correctly computes its size, the ring that is twice as large can falsely also return \( n \) since antipodal nodes can be in the same local state for all times and therefore behave the same as in the ring of size \( n \).

Hence the nodes need to know \( n \) in advanced. This is necessary so that the
nodes are able to detect their own messages. This can be done by incorporating into each message a counter *hop* that counts the number of nodes the message has travelled so far.

**Note.** As message hops have to count to $n$, each message requires $o(\log(n))$ bits. So the size of the state vector is $o(n \cdot \log(n))$ bits, as each node needs to be able to initiate a message at the beginning. If no restrictions on the messages are made, like requiring FCFS transmissions, the messages are independent, and the size of the state space is $o(2^{n \cdot \log(n)}) = o(n^n)$.

### 3.2.2 Randomized algorithms

This section gives an overview of the algorithms we will model check. They are presented as pseudocode together with their main ideas. Details are given in chapter 5 and 8 where the statistical results of model checking one algorithm imply the design decisions for the next.

As it is often the case in algorithm design, we have a conflict between the memory requirements and the runtimes of algorithms. For example, if the nodes use large random numbers and save every information they receive, runtime – and also traffic – improves, but memory requirements increase. These contrary positions imply a conflict between the runtime of a leader election algorithm and the size of the state space for model checking it.

We start with algorithms that compare IDs to turn nodes passive. $A_{\text{compare IDs}}$ denotes the class of these algorithms.

Then we describe two alternatives: $A_{\text{sorting}}$ is generous with memory and greedy on runtime, $A_{\text{timing}}$ conversely.

The first algorithm, $A_{\text{Itai, round} \leq K}$, can use arbitrary transmissions and is a **Monte-Carlo algorithm**, i.e. its runtime is bound, but its result may be erroneous. All other algorithms require FCFS transmissions and are **Las-Vegas algorithms**, i.e. their runtime is not bound, but finite on average, and their result is always correct. In some literature, e.g. in (Motwani and Raghavan, 1995), the additional requirement of a finite expected runtime is not given.

$A_{\text{Itai, round} \leq K}$

$A_{\text{Itai, round} \leq K}$ is a modification of the algorithm $A_{\text{Itai}}$, described in (Itai and Rodeh, 1981).

$A_{\text{Itai}}$ is a probabilistic variant of the deterministic algorithm from (Chang and Roberts, 1979), which is designed for a ring with nodes having unique IDs. The IDs are sent around the ring, with each node forwarding only messages with monotonically non-decreasing IDs, and purging all other messages. The node with the largest ID becomes the leader. The algorithm's message complexity is $n^2$ in the worst case and $n \log(n)$ on average.
3.2. Leader election for anonymous, unidirectional rings

In $A_{flat}$ a node, lacking a unique ID, generates a random ID from the finite domain $\{0, \ldots, M - 1\}$ and saves it in the local variable $id$. As the IDs of the nodes need not be unique, the largest ID can collide, i.e. be present more than once, forcing another election trial. To exploit the work already done, each node has a local flag $active$ which is set to false when the node encounters a message with a larger ID than its own. In future election trials only active nodes create messages and are leader candidates. Since FCFS transmissions are not required, there can be outdated nodes and messages from previous rounds. To detect them, we use another variable, $round$, counting the election trials.

Each message contains a flag $unique_m$ showing whether the message crossed another node with the same ID. If the largest ID in a round is unique, its node detects this after receiving its own message. The node is elected as leader by setting its flag $leader$. All other nodes have been turned passive, i.e. their flag $active$ is set to false. Hence the algorithm terminates, i.e. all nodes either terminate or halt in a valid end state and message transmission ceases.

So a message consists of the quadruple $(round_m, id_m, hop_m, unique_m)$. The algorithm executed by each node is shown in listing 5.1. The receive command waits until a new message arrives. Depending on the type of buffer, send can also block until the network is ready for transmission. Almost all selections in listing 5.1 depend on message variables, i.e. the message transmissions terminate the events and the behaviour of the nodes. Therefore the algorithm is message driven. This is often so for distributed algorithms and will be the case for all of our algorithms.

As the domain of $round$ is infinite in $A_{flat}$, its state space is infinite, too. If we want to verify the algorithm with a model checker, we need to reduce its state space to a finite set since model checking exhaustively enumerates the state space. The easiest way is to bound the domain of $round$ and halt the algorithm when the limit $K$ is reached. $A_{flat}.round \leq K$ uses this method and hence can terminate erroneously with no leader elected, i.e. it is a Monte-Carlo algorithm. But at least it only makes the error of not electing any leader, electing more than one leader never happens. Also, the algorithm detects the error and can give an error message.

$A_{flat}.round \mod K$

Only for FCFS transmissions (cf. section 4.2) $A_{flat}.round \mod K$ is correct, i.e. for every fair schedule it terminates with probability one and upon termination exactly one leader exists. So it does not halt when $round$ reaches the limit $K$, but wraps around the variable, i.e. uses $round:=(round+1) \mod K$ instead of $round++$ in line 22 of listing 5.1. Using a factor ring, we can no longer use $round_m > round$ in line 26 of the listing. Instead we can use $round_m == round + 1 \mod K$ (cf. section 4.2).
proc node
bool leader=false, active=true;
int id, round=0;
/* m indicates message variables: */
int round_m, id_m, hop_m;
bool unique_m;

RESTART:
id:=rand(0,M-1);
id_m:=id; round_m:=round; hop_m:=1; unique_m:=true;
SENDANDRECEIVE:
send (round_m, id_m, hop_m, unique_m) to buffer for node next;

RECEIVE:
receive (round_m, id_m, hop_m, unique_m) from node preceding;
if (hop_m==n)
then if (active)
then if (unique_m)
then leader=true; printf("I am the leader\n")
else /* start a new round: */
round++; goto RESTART
fi
else printf("This should never be the case\n")
fi
else if (round_m>round || (round_m==round && id_m>id))
then /* become passive: */
round:=round_m; id:=id_m; active:=false; hop_m++; goto SENDANDRECEIVE
else if (round_m==round && id_m==id)
then /* collision detected: */
hop_m++; unique_m:=false;
goto SENDANDRECEIVE
else /* purge message: */
goto RECEIVE
fi
fi
end;

Listing 3.1: A太湖

A太湖,round mod K requires FCFS transmissions. But in this case we can even dispense with round numbers completely if we modify A太湖 such that a passive node merely forwards messages and increases their hop_m counters. An active node is either turned passive or starts its new round before receiving any messages from the new round. So a passive node neither updates its variables id and round, nor purges messages, nor alters their unique_m flag (cf. section 3.2.1). This results in A太湖,roundtrip, which was introduced in [Fokkink and Fang, 2004] and in [Fokkink and Fang, 2005].
3.2. Leader election for anonymous, unidirectional rings

Thus a message consists of the triple \((id_m, hop_m, unique_m)\), and each node executes the algorithm shown in listing 3.2.

```
proc node
  bool leader:=false, active:=true;
  int id;
  /* m indicates message variables : */
  int id_m, hop_m;
  bool unique_m;

  RESTART:
  id:=rand(0,M-1);
  id_m:=id; hop_m:=1; unique_m:=true;

  SEND_AND_RECEIVE:
  send (id_m,hop_m,unique_m) to buffer for node_next;

  RECEIVE:
  receive (id_m,hop_m,unique_m) from node_preceeding;

  if (!active)
    then hop_m++;
    goto SEND_AND_RECEIVE
  else if (hop_m==n)
    then if (unique_m)
      then leader:=true; printf("I am the leader\n")
      else goto RESTART /* start a new round */
    fi
  else if (id_m>id)
    then active:=false; hop_m++; /* become passive */
    goto SEND_AND_RECEIVE
  else if (id_m==id)
    then /* collision detected : */
      hop_m++; unique_m:=false;
    goto SEND_AND_RECEIVE
  else goto RECEIVE /* purge message : */
  fi
fi;
end;
```

Listing 3.2: \( A_{Fokkink,roundtrip} \)

\( A_{Fokkink,norroundtrip} \)

If transmissions are FCFS, each node with the largest ID detects the collision before its message completed the roundtrip, as the other messages containing the largest ID passed through it. In \( A_{Fokkink,roundtrip} \) each of these nodes has to wait for its own message before it can take action against the collision by randomly choosing a new ID. \( A_{Fokkink,norroundtrip} \) and \( (Fokkink and Pang, 2004) \) show that this is not necessary: Each node can select a new random ID the
instance it detects this, i.e. the moment it receives a message $m$ with the same ID as its own. Instead of forwarding $m$, it sends the newly created message. This variation of $A_{Fokkink, roundtrip}$ results in $A_{Fokkink, no roundtrip}$.

As detected collisions are resolved immediately, messages need not contain $unique_m$ anymore. Thus a message consists of the pair $(id_m, hop_m)$. The algorithm executed by each node is shown in listing 3.3

```
proc node
    bool leader:=false, active:=true;
    int id;
    /* $m$ indicates message variables: */
    int id_m, hop_m;

    RESTART:
    id:=rand(0, M - 1);
    id_m:=id; hop_m:=1;

    SEND_AND_RECEIVE:
    send (id_m, hop_m) to buffer for node_next;

    RECEIVE:
    receive (id_m, hop_m) from node_previous;

    if (!active)
        then hop_m++;
    else if (hop_m==n)
        then leader:=true; printf("I am the leader\n")
    else if (id_m>id)
        then /* become passive: */
            active:=false; hop_m++;
            goto SEND_AND_RECEIVE
    else if (id_m==id)
        then /* coll. detected $\Rightarrow$ start new round: */
            goto RESTART
    else /* purge message: */
        goto RECEIVE

    fi

    fi;
end;
```

Listing 3.3: $A_{Fokkink, no roundtrip}$

$A_{sorting}$

$A_{sorting}$ does not compare IDs to turn nodes passive. Instead each node saves all the information it receives, i.e. all messages passing through. Utilizing this information, $A_{sorting}$ can significantly reduce the runtime.
3.2. Leader election for anonymous, unidirectional rings

So this algorithm generously uses memory and is greedy on runtime, which is opposite to the concept of the next algorithm, $A_{runint}$.

Each node flips a coin and sends the result around the ring. As both values 0 and 1 are transmitted using FCFS, each node can deduce the origin of each message by counting the number of received messages. Hence we can also omit the counter hop, as the $n$-th bit received is the message a node created itself. Thereafter a round is finished. As all nodes save all the transmitted information in bit arrays, they know the coin tosses of all other nodes at the end of the round and can agree on a leader: the node with the largest bit array, if there is a unique one. Otherwise all nodes start a new round.

One implementation of this decision procedure is generating all possible circular shifts of the bit array, i.e. rotating it bitwise. By checking whether the largest rotated bit array is unique, a node can decide whether a leader exists; by checking whether the largest rotation is its own bit array, a node can detect whether it is the leader. Of course a node does not need to store all rotations, their offsets to the node's own bit array is sufficient.

We do not use a concrete decision procedure, but abstract from leader election algorithms to an algorithm that only shows that a unique leader would exist if the algorithm implemented a correct decision procedure. Therefore we can spare the sorting routine and its memory. We only need to check whether the bit array is periodic to be able to decide whether a new round has to be started.

So a message solely consists of one bit.

The abstract algorithm executed by each node is shown in listing 3.4.

```c
proc node
    int index;
    bit bitvector[n];

    Restart:
        index := 0;
        bitvector[index] := rand(0, 1);
        send bitvector[index] to buffer for node_next;
        index++;
        for (index := 1; index < n - 1; index++) do
            receive bitvector[index] from node_next;
            send bitvector[index] to buffer for node_next;
        od;
        received bitvector[n - 1] from node_next;
        if (bitvector is periodic)
            /* start a new round: */
            goto Restart
        else
            printf("There exists a leader\n")
        fi;
    end;

Listing 3.4: A_{sorting}
```
**Note.** As each node uses a bitvector of size $n$, the state vector requires $o(n^2)$ bits. If the contents of each bitvector was independent, the state space would have the size $o(2^n) = o((2^n)^n)$. Luckily at the end of each round all bitvectors are bitshifts of each other. So the combinations of the bitvectors in the state space only result from scheduling, i.e. from the order in which the bits are transmitted.

If $A_{\text{sorting}}$ uses a sorting routine, each node can deduce the distance to the leader after the last round. Computing this distance as $ID$ is a de-anonymization algorithm (cf. page 343) which does not require transmitting any further message.

It turns out that [Itai and Rodeh, 1981] used a similar algorithm for synchronous rings.

$A_{\text{timing}}$

Just like $A_{\text{sorting}}$, $A_{\text{timing}}$ also avoids comparing IDs to turn nodes passive. Asymmetry is imposed onto the ring by each node waiting for a random period of time before sending a message. This asymmetry breaking requires less memory and traffic than transmitting random numbers. Of course the waiting can entail a very high runtime. Therefore $A_{\text{timing}}$ and $A_{\text{sorting}}$ really have contrary requirements.

So a node $m$ waits for a random time before initiating a message that requests $m$ being the leader. If no other node wants to become the leader, the message returns to $m$ with the `unique` flag still set, signaling $m$ that it can become the leader. Therefore the message has to contain a `hop` counter, so that $m$ can detect its own message. If the `unique` flag is not set, another node also wants to become the leader. Consequently the competing nodes have to start a new leader election round.

If $m$ receives a message before initiating its own one, i.e. while still waiting in its own random time period, it becomes passive, only forwarding messages and increasing their `hop` counters.

Thus a message consists of the pair $(hop_m, unique_m)$.

The algorithm executed by each node is shown in listing 3.3.
3.2. Leader election for anonymous, unidirectional rings

```
proc node
  bool leader:=false;
  /* m indicates message variables: */
  int hop_m;
  bool unique_m;

RESTART:
  if (message (hop_m, unique_m) is coming from node_preceding) 
    /* become passive: */
    goto PASSIVE_SEND_AND_RECEIVE
  else randomly either goto RESTART
      or hop_m:=1; unique_m:=true;
      goto ACTIVE_SEND_AND_RECEIVE
fi;

PASSIVE_SEND_AND_RECEIVE:
  forward message from node_preceding to
  buffer for node_next with hop_m++; 
  goto PASSIVE_SEND_AND_RECEIVE

ACTIVE_SEND_AND_RECEIVE:
  send (hop_m, unique_m) to buffer for node_next;
  receive (hop_m, unique_m) from node_preceding;
  if (hop_m!=n)
    then /* coll. detected: >1 node wants to become leader: */
       hop_m++; unique_m:=false;
    goto ACTIVE_SEND_AND_RECEIVE
  else if (!unique_m)
    then /* start a new round: */
       goto RESTART
    else leader:=true; printf("I am the leader\n")
      fi;
  fi;
end;
```

Listing 3.5: Aiming
Chapter 4

Theoretical investigations

The following section investigates the possibilities of reducing the memory requirements of model checking to increase the maximal problem size that can be verified. Section 4.2 introduces the parameters of the class $A_{\text{compared IDs}}$ and deduces several dependencies. The last section provides some mathematical correctness proofs, as opposed to the correctness proofs in the chapters 6 and 7 performed by model checking.

4.1 Reducing model checking requirements

Decreasing the memory requirements of the model checking algorithm can be accomplished by compressing the state space or by reducing it, described in the following two subsections. A third possibility is using approximation. But these methods degrade model checking to a non-rigorous tool. Thereby model checking loses its main advantage compared to other non-rigorous tools like testing. Thus we do not consider approximations.

4.1.1 Compression of the state space

The state space can be compressed using two variants:

Compressing each state individually

An example is SPIN’s collapse compression, which uses a hierarchical indexing method for compression (cf. Holzmann, 2004). As each state needs to be compressed before comparison within the explored state space or insertion into it, the runtime of model checking is increased.
Compressing the complete state space at once

As compressing and uncompressing the complete state space is very time-consuming, checks and updates on the state space are more efficient if they do not require uncompressing it first. In this case we have a model checking algorithm that uses a more compact model representation. So the runtime does not have to be higher. For example, PRISM represents the state space using MTBDDs and manipulates them directly. In some cases this can exponentially reduce the memory requirements. SPIN can optionally also represent the state space more compactly by using minimized automata (cf. [Holzmann, 2004] page 20). But this can yield a significant runtime penalty.

4.1.2 Reduction of the state space

The following reduction methods are either manual or static, except for partial order reduction and symmetry reduction, which are dynamic, but with just little computation requirements. So the time saved from having a reduced state space outweighs the additional computation overhead. Hence the result is a model checking algorithm with smaller memory and smaller time requirements. Therefore we will investigate this kind of reductions thoroughly.

Several of the following reduction methods are based on abstraction techniques, which map the original state space to a smaller, abstract one, trying to preserve the relevant behaviours of the system. Often the techniques do not operate on the state space directly, but achieve reduction with an abstract interpretation of the model specification by subsuming values of variables or by eliminating data objects. These techniques are described in (Holzmann et al., 1996; chapter 10), with a broader view in [Dams, 2002]. An abstraction method tries to preserve all of the relevant behaviours, i.e. all behaviours relevant to the property being verified. Hence abstraction techniques are often parameterized with the property being checked. If the abstraction does not achieve an exact approximation of the relevant behaviours, the procedure

- either under-approximates, i.e. the abstract system contains less relevant behaviours than the original. In this case model checking the abstract system can entail false positives, so the abstraction is logically unsound,
- or over-approximates, i.e. the abstract system contains more relevant behaviours. Then model checking the abstract system can yield false negatives, i.e. spurious errors, so the abstraction is logically incomplete.

False positives are very hard to deal with. False negatives can sometimes be analysed and eliminated by counterexample-guided abstraction refinement (CEGAR), which establishes logical completeness (cf. [Clarke et al., 2004] and [Sinha, 2005]).
4.1. Reducing model checking requirements

As our miniworld will always consist of repeated parallel compositions of at most two different modules (cf. section 3.2.1), we investigate the possibilities for reducing the size of the state space when using \( n \) parallel compositions of a module \( M \) having \( c \) states. The reduction can tackle within \( M \), the number of modules, or globally the number of combinations of local states.

**Reducing the number of local states in \( M \).** Let \( f : \mathbb{N} \to \mathbb{N}, c \mapsto f(c) \) map the number of states of \( M \) to the number of states of the reduced module \( M' \). So the complete state space decreases from \( c^n \) to \( f(c)^n \), i.e. by the factor \( (c/f(c))^n \). Often \( f \) is independent of \( n \). Then the reduction factor for the state space is exponentially larger than that of a single module, but the reduced state space is still exponential in \( n \). Hence the complexity class of model checking does not change by this reduction. Therefore we will have to look closer into the function \( f \) to determine the exact degree of reduction, which we will point out by the factor \( F \) by which the maximal possible \( n \) increases: Let \( n_{\text{max}} \) be the maximal number of parallel compositions of \( M \) before memory overflow occurs, \( n'_{\text{max}} \) analogously for \( M' \), and \( F := n'_{\text{max}}/n_{\text{max}} \). As the amount of available memory is about \( c^{n_{\text{max}}} = f(c)^n_{\text{max}} \), we have \( n'_{\text{max}} \approx n_{\text{max}} \cdot \log f(c)(c) \), i.e. \( F \approx \log f(c)(c) \).

Usually one of the following reduction methods is applied for reducing the number of local states in \( M \):

- **explicitly excluding some states of \( M \),** e.g. by strengthening conditions. Then we have \( f(c) = c - k \) with the subbrand \( k \in \mathbb{N} \) being the amount of erased states, and \( F \approx \log_{c-k}(c) \). Since usually \( c \gg k, F \approx 1 \) and the reduction has almost no effect.

- **data type abstraction,** which reduces the value range of a variable in \( M \) to get \( M' \), selective restriction, e.g. reducing the size of a channel, or completely eliminating a variable (and the associated statements) by selective data hiding (e.g. by SPIN’s static slicing algorithm, cf. (Millet, 1998)). This yields a constant divisor \( k \in \mathbb{R}_{>0} \) with \( f(c) = c/k \) and \( F \approx \log_{c/k}(c) = \log_{c/k}(c) = 1 + \log_{c/k}(k) \). As usually \( c/k \gg k \), we have \( F \approx 1 \), i.e. still only very little reduction.

- **To increase \( F \) with the help of the technique from above, the divisor \( k \) needs to increase, such that \( c/k \gg k \).** Therefore more and more variable range reductions or variable eliminations are needed. For \( F \) to reach \( 2 \), i.e. to be able to solve problems of twice the original size, we already have to halve the range of each variable or eliminate every other variable, since then the parallel composition of two new modules has the same number of states as one old module. Then \( \log_{c/k}(k) = 1 \), i.e. \( c/k = k \) and \( f(c) = \sqrt{c} \). An example is predicate abstraction (e.g. (Clarke et al., 1986b)), which does so by only keeping track of certain relevant predicates over the original variables, which are eliminated. So the original data is mapped to a set of Boolean variables.

Selective data hiding with SPIN’s slicing algorithm only eliminates irrelevant data objects, and is thus an exact approximation. Data type abstraction is
often logically incomplete, selective restricting even logically incomplete and unsound. Since we do not want false positives and false negatives, we will not consider these techniques. Abstraction methods yield good results if either the original model has high redundancy, or there are several properties to be checked which have different relevant parts in the model. In our case of checking the correctness of a leader election algorithm, the lack of redundancy probably entails unsatisfactory results for these reduction methods.

**Reducing the number of modules.** Reducing the amount of modules without increasing their number of states usually results in a model with reduced functionality. An example is selective restriction by limiting the number of active processes. In the optimal case, the relevant functionality of several modules can be inserted into a single one without increasing its number of states. This procedure is similar to the previous case, since combining two modules into one in this manner corresponds to reducing the number of states of both modules to \( \sqrt{k} \), each, and parallelly composing them. Although this process facilitates the reduction as it looks at all modules instead of only an isolated one, the requirements cannot often be met. We try this method in the next chapter by integrating the relevant functionality of the channel modules into the node modules without increasing their number of local states significantly. Usually the requirements are still too strong to be able to put this strategy into effect.

**Reducing the number of combinations.** There are three major methods that regard the whole system instead of isolated modules and therefore reduce the number of combinations of local states or transitions: abortion of paths, symmetry reduction and partial order reduction:

If the current execution, i.e. the current path in the computation tree, is of no interest for the property being checked, the path can be aborted, saving runtime. States and transitions that are only reachable through such irrelevant paths are never visited, causing reduction of the state space. Section 2.4.2 describes how SPIN's **never claims** implement this kind of reduction.

**Symmetry reduction** (cf. [Bosnacki et al., 2001]) reduces the number of combinations of the local states by only regarding equivalence classes defined by symmetric properties. So the reduction maps sets of states of the original state space to a signal state in the reduced state space. Such a mapping generally reduces the number of states, but increases the number of behaviours, therefore potentially over-approximates.

An example for our algorithms is symmetry reduction by rotation: The state space is partitioned, each class containing all states equivalent modulo rotation, as the result of electing exactly one leader does not change by rotation. As we can rotate the ring with \( n \) nodes \( n \) times, every class contains \( n \) elements. So the complete state space is reduced by the factor \( n \). This can be described by the above case of a divisor, but now depending on \( n \): If \( f(c) = f_n(c) = c/k(n) \), i.e. the factor \( k \) increases with \( n \), we have \( F \approx \log_{c/k(n)^{\infty}}(c) = 1 + \log_{c/k(n)}(k(n)) \). So the division of the complete size of the state space by \( n \) corresponds to reducing \( M \) by the factor \( \sqrt{n} \). As usually \( c/k(n) \gg k(n) \), we have as before \( F \approx 1 \), i.e. only little reduction.
4.1. Reducing model checking requirements

**Partial order reduction** reduces the number of interleavings of transitions by only selecting few of the interleavings having the same result. To be an exact approximation, several conditions must hold. These are described in subsection 7.2.1.

**Statement merging** is a special case of partial order reduction, explained in [Holzmann, 1999]. A sequence of invisible and deterministic statements within one process can be executed atomically using partial order reduction, i.e. the sequence results in a branchless path in the partial order reduced LTS.

This special case of partial order reduction can be computed at compile time, i.e. statically. Therefore, besides reducing the degree of interleaving, we can also eliminate the intermediate states to save computation time and memory. This method is implemented by wrapping such sequences up within a step. So statement merging compresses certain paths, which would be branchless in the partial order reduced graph, into a single transition.

Symmetry reduction and partial order reduction are often also considered as abstraction techniques. General partial order reduction and statement merging frequently achieve strong reductions, as they often reduce the large amount of interleavings that cause the blow up of the state space. So these two reductions are the most promising for our leader election problem. We will exploit them in chapter 6 especially in subsection 5.2.6.

Of course all considerations are still valid with more than one refinement, i.e. if $M$ is not a basic module or if the regarded parallel composition is not the system module.

### 4.1.3 Combining compression and reduction

Since there are various methods for decreasing memory requirements, an important question is to which degree they are combinable. Most of SPIN’s methods reducing memory can work in parallel, e.g. one can first reduce the state space by selective data hiding, partial order reduction, and statement merging and thereafter compress it using both collapse compression and minimized automata storage.

But not all methods are orthogonal, i.e. their factors of improvement do not multiply. As the efficiency of the combinations often depends on the model being checked, many statistics thereof can be found in literature, e.g. about the effectiveness of symmetry reduction in parallel to partial order reduction in [Kosnacki et al., 2003]. Hence if memory overflow occurs and runtime is still acceptable, adding further available reduction methods is a promising approach. We will do this in chapter 6 using minimized automata storage.
4.2 Parameters of the class $A_{\text{compare IDs}}$

Comparing our algorithms in $A_{\text{compare IDs}}$, i.e. $A_{\text{Itai,round } \leq K}$, $A_{\text{Itai,round mod } K}$, $A_{\text{Fokkink,roundtrips}}$ and $A_{\text{Fokkink,no roundtrip}}$ (cf. section 5.2.2), yields the following parameters:

- **Message overaking**, which means that the order of the messages in the ring need not be fixed. This results from using arbitrary transmissions, whereas FCFS transmissions retain the order. FCFS transmissions can be implemented by channels using FIFO buffers, for instance. Algorithm $A_{\text{Itai,round } \leq K}$ can deal with message overaking.

- **Message overwriting**, which denotes that messages not yet transmitted can be overwritten by newer ones. Message overwriting can be implemented by using a local buffer in each node. The processed messages are not directly transmitted, but pushed into the buffer for later transmission. So the buffer's location is between the node and the outgoing connection. If the buffer is full, new messages overwrite older ones. Hence nodes need not block to wait for a channel to become ready, and the number of messages, i.e. traffic, is reduced. Both Itai algorithms can perform message overwriting if a local buffer for $\text{node}_{\text{next}}$ (cf. listing 5.1) is used, for instance of size 1.

- **Round numbers**, i.e. nodes and messages contain the variable round, as in algorithm $A_{\text{Itai,round } \leq K}$. Besides omitting round completely, as both Fokkink algorithms do, round numbers can also be used modulo $K$, as in $A_{\text{Itai,round mod } K}$.

- **Passive nodes filter**, i.e. nodes with their flag active not set also update unique flags and purge messages, depending on their ID. Therefore passive nodes also need to update their variable id. If round is present, not id and id$_m$ are compared, but (round, id) and (round$_m$, id$_m$) lexicographically (cf. listing 5.3 and 5.2). Both Itai algorithms filter, whereas both Fokkink algorithms do not.

- **Roundtrips**, denoting the requirement that messages neither purged nor overwritten travel once completely around the ring. So using algorithms with no roundtrips, messages do not necessarily perform this complete roundtrip. If a node's and a message's ID collide, the node deletes the message and creates a new one. $A_{\text{Fokkink,roundtrip}}$ and both Itai algorithms use roundtrips, whereas $A_{\text{Fokkink,no roundtrip}}$ does not.

For easier descriptions the nodes are named consecutively $p_0, \ldots, p_{n-1}$ (with $n$ being the size of the ring), and the messages that $p_i$ creates consecutively $m_i, m'_i, m''_i, \ldots$. Hence the indices are from the factor ring $\mathbb{Z}/n\mathbb{Z}$.

**Lemma 4.1.** If a correct algorithm $A \in A_{\text{compare IDs}}$ uses round numbers, roundtrips, and no message overaking, $A$ is also correct using round numbers modulo $K$ with $K \geq 2$. 

4.2. Parameters of the class $\mathcal{A}_{\text{compare IDs}}$

Proof. If a node $p_l$ increases its variable $\text{round}$ from $l$ to $l+1$, its former message with $\text{round} = l$ made a complete roundtrip, updating the round numbers of all outdated nodes. As no message overtaking is allowed, all other messages with $\text{round} = l$ already passed through $p_l$. Thus round numbers in the ring differ by maximal one and can therefore be regarded modulo $K$. Since no outdated messages arrive at a node, a different round number always indicates a newer message. Hence $K$ can be set to two. \qed

Lemma 4.2. If a correct algorithm $A \in \mathcal{A}_{\text{compare IDs}}$ either uses round numbers modulo $K$ or no round numbers at all, $A$ cannot allow message overtaking.

Proof. Figure 4.7 shows that the use of round numbers modulo $K$ combined with message overtaking is erroneous. The counterexample uses $n = K = 3$, and depicts that for certain random IDs all nodes turn passive in a specific scheduling. Thus the leader election algorithm terminates with no leader elected.

In the first network state depicted in the figure, all nodes are active (red). Two of them choose the ID $u$, the other chooses $v < u$. The messages containing the ID $u$ make complete roundtrips, while the message with the ID $v$ is still in its first transmission.

Thus in the second shown network state, one node is passive (yellow) and the other two choose new IDs, again $u$. Then these two new messages make complete roundtrips, while the old message is still stuck.

In the third network state depicted in the figure, the two active nodes, now in $\text{round} = 2$, again choose the ID $u$, hence create the messages $(2, u, 1, l)$. Now all three messages in the network are transmitted to their next nodes, i.e. no message is stuck anymore.

Thus in the fourth network state, another node is passive. Of course its round number and ID are updated. Therefore the node purges its next incoming message. The other two messages are transmitted to their next nodes again.

This results in the fifth state depicted, with all nodes being passive. Both remaining messages are purged by their next nodes. The message containing ID $v$ causes the output "This should never be the case" if a variant of $A_{\text{alt round mod K}}$ is used (cf. listing 4.1).

Hence the final network state does not contain any messages and only has passive nodes.

Other values for $n$ and $K$ work exactly the same. The case $K = 1$ corresponds to not using any round numbers at all.

The message with ID $v$, causing the error by being stuck in a transmission for two rounds, makes a complete roundtrip and returns to a passive node. If messages do not overtake one another, this situation does not occur, since either the message was purged or its origin is still active. Thus a message returning to its passive origin implies message overtaking. Therefore reactivating that node might be sufficient to avoid erroneous termination. The next figure, 4.2, shows that this is not the case.

Again all nodes are active (red) at the beginning. Two adjacent nodes choose the ID $u$, the other two choose $v < u$. The messages containing the ID $u$ make complete roundtrips; one of the other messages gets filtered, the remaining is stuck.

Consequently the two nodes that chose $v$ have become passive (yellow) in the
second depicted network state. The other two choose again \( u \) as new ID. The
two new messages make complete roundtrips, while the old message is still stuck.
Thereafter the two active nodes are in round 2, again choose \( u \), and create the
messages \((2, u, 1, t)\). Now the message \((0, v, 1, t)\), being stuck so far, travels the
distance two.
Consequently both active nodes are turned passive.
Then the message is purged, as is one of the two messages \((2, u, 1, t)\).
The last existing message can make two hops before being filtered, resulting in
the last network state without any messages and only passive nodes.
4.2. Parameters of the class $A_{\text{compare IDs}}$

In this example no messages return to their origins turned passive. Consequently no reactivation is possible, i.e. using reactivation is not sufficient to avoid erroneous termination.

Counterexamples for other values of $K$ and larger $n$ can be constructed analogously. The case without any round numbers is shown with $K = 1$. $\Box$

Hence $A_{\text{Hai, round } \leq K}$ is our only algorithm allowing message overtaking, and there is no finite space Las-Vegas algorithm in $A_{\text{compare IDs}}$ for networks with message overtaking. [Hai and Rodeh, 1981] allows arbitrary transmissions, i.e. message overtaking, and therefore erroneously suggests using round numbers modulo 3.

**Lemma 4.3.** If a correct algorithm $A \in A_{\text{compare IDs}}$ uses message overwriting and roundtrips, $A$ must use round numbers or round numbers modulo $K$.

**Proof.** Let $p_i$ and $p_j$ be two distinct nodes that choose the largest ID $M - 1$. It can happen that $m_i$ overwrites $m_j$ in some node. Hence without round numbers (modulo $K$), $p_j$ cannot recognize messages from the next round, since $m_j$ did not finish its roundtrip. Consequently $p_j$ stays active and purges messages. If no node chooses the ID $M - 1$ in the new round, $p_j$ purges all messages and the algorithm terminates without a leader. $\Box$

Thus $A_{\text{Fokkink, roundtrip}}$ cannot be optimized to allow message overwriting.

**Lemma 4.4.** If a correct algorithm $A \in A_{\text{compare IDs}}$ uses message overwriting and no roundtrips, $A$ must use round numbers or round numbers modulo $K$.

**Proof.** We assume $A$ is an algorithm using message overwriting, no roundtrips and no round numbers (modulo $K$). The following situation shows that $A$ is not correct:

Let $p_i$ and $p_j$ be distinct and the only two nodes that choose the largest ID $M - 1$. No node chooses the ID $M - 2$. $m_j$ is stuck in $p_j$ until $m_i$ arrives. Hence all nodes between $p_i$ and $p_j$ have been turned passive by $m_i$. $p_j$ chooses for its new message $m'_j$ the ID $M - 2$. $m'_j$ overwrites $m_j$. When $m'_j$ is sent around the ring, it turns all nodes between $p_j$ and $p_i$ passive. Finally $p_i$ purges $m'_j$. Thereafter $p_i$ and $p_j$ are the only active nodes, but no message is left in the ring. So the algorithm terminates without a leader elected. $\Box$

Thus $A_{\text{Fokkink, no roundtrip}}$ cannot be optimized to allow message overwriting.

The following corollary is a consequence of lemma 4.4 and 4.3

**Corollary 4.5.** If a correct algorithm $A \in A_{\text{compare IDs}}$ uses message overwriting, $A$ must use round numbers or round numbers modulo $K$.

**Lemma 4.6.** If passive nodes filter in a correct algorithm $A \in A_{\text{compare IDs}}, A$ must use round numbers or round numbers modulo $K$. 
Proof. We assume $A$ is an algorithm with filtering passive nodes and no round numbers (modulo $K$). Then we can construct the following counterexample:
Let $n = 4$, $p_1$ and $p_2$ choose the ID $M - 1$, $p_2$ and $p_4$ the ID 0. Then $p_2$ and $p_4$ are turned passive by $m_1$ and $m_3$ and update their ID to $M - 1$. If in the next round $p_1$ and $p_2$ choose IDs smaller than $M - 1$, $p_2$ and $p_4$ will purge them, as without round numbers (modulo $K$) they are not able to distinguish between the newer IDs in the messages and their outdated own IDs. Thus all messages in the network are removed before a leader is elected.

Thus both our Itai algorithms cannot be optimized to omit round numbers (modulo $K$). By contraposition, both Fokkink algorithms cannot be optimized so that passive nodes filter.

So in short, message overwriting as well as filtering by passive nodes imply using round (modulo $K$). Message overtaking implies using round numbers.

Lemma 4.7. Without message overtaking, passive nodes need not filter.

Proof. Each active node only sends monotonically non-decreasing messages, i.e. the pairs $(round, id)$ are monotonically non-decreasing. Therefore the unique flag only needs to be switched from true to false in a node $p$ if the message was generated by $p$'s predecessor and the round number of the message is equal to the one from $p$. Hence $p$ is still active. So without message overtaking passive nodes never get the chance to filter.

Lemma 4.8. With both message overtaking and message overwriting, passive nodes must filter in a correct algorithm $A \in A_{\text{compare IDs}}$.

Proof. Because of message overtaking, a passive node $p$ can first receive a larger message $m_1$, afterwards a smaller one $m_2$. Assuming $p$ does not filter, it can overwrite $m_1$ by $m_2$. If $m_1$ had a unique largest ID, its originator will purge $m_2$ and forever wait for $m_1$.

So far we have no restrictions on the use of no roundtrips, so maybe these can be combined with other optimizations. Because of lemma 4.4, we forbid message overtaking. Lemma 4.4 implies that we cannot omit round numbers (modulo $K$). As $A_{\text{Fokkink, no roundtrips}}$ uses neither round numbers nor roundtrips, a better algorithm needs additional optimization besides round numbers modulo $K$ and no roundtrips.

Lemma 4.9. If a correct algorithm $A \in A_{\text{compare IDs}}$ uses no roundtrips, $A$ cannot use round numbers modulo 2.

Proof. A counterexample is $p_0$ selecting ID $M - 1$, $p_1$ and $p_2$ ID $M - 2$. $m_1$ makes $p_2$ increase its round number and create a new message $m'_2$. $m_0$ turns $p_1$ and $p_2$ passive, while $m'_2$ turns $p_0$ passive.

Lemma 4.10. If a correct algorithm $A \in A_{\text{compare IDs}}$ uses round numbers modulo $K \geq 3$ and no roundtrips, $A$ cannot use message overwriting.
4.3 Mathematical correctness proofs

Proof. For a fixed \( K \), we choose \( n = K \), and all nodes always select ID 0. Older messages are always transmitted first, except in \( p_{n-1} \), which does not transmit any messages until it creates the message with round number \( K - 1 \). Hence all other messages \( p_{n-1} \) created are overwritten, and the message with round number \( K - 1 \) is the first arriving at \( p_0 \), hence being purged. Therefore no more messages are in the network, so no leader gets elected.

So the only possible additional optimization besides using modulo \( K \geq 3 \) and no roundtrips are filtering passive nodes. Because of lemma \[4.3\] passive nodes need not filter, i.e. \( A_{Fokkink, no roundtrip} \) without any round numbers is the best possible choice when using no roundtrips.

4.3 Mathematical correctness proofs

Since one can never guarantee that a proof is correct we also show the correctness of some algorithms manually, as contrasted with the verifications in the following two chapters. Not requiring a finite state space for a mathematical proof, we can show the correctness of the algorithms for all \( n \in \mathbb{N}_{>0} \).

4.3.1 General discussion about induction

As we want to prove that a leader election algorithm \( A \) is correct for every size \( n \in \mathbb{N}_{>0} \), we try induction on the size of the ring. So the induction hypothesis says that \( A \) is correct for all rings of size \( \leq n \), and the induction step \( n \mapsto n+1 \) inserts a new node into the ring. Hence we get a new ring with \( n+1 \) nodes, one of which is newly inserted, as depicted in figure \[4.3\]

![Figure 4.3: Induction step: insertion of a node](image)

Since this is a manual proof, we have the liberty to insert the new node at any location. But to be able to perform a reduction in the induction step from \( n+1 \) to at most \( n \) so that the induction hypothesis for \( n \) can be used, the right part of figure \[4.3\] and the leader election algorithm have to be brought to coincidence, e.g. by turning the one new node passive. So the reduction from \( n+1 \) to \( n \) in the proof induces some symmetry breaking of the ring, which has to be performed by the algorithm as well. This looks absurd at first glance, since breaking symmetry is the goal of the algorithm. But leader election is a very specific and strong symmetry breaking: All nodes are involved because exactly one leader has to be elected and all nodes need to know whether they

\[1\ldots, the best one can say is: "I have not discovered any mistakes" \[Dijkstra, 1965\]
are the leader. On the other hand, to be able to superimpose a reduction step, it is sufficient for the algorithm to perform some **weak symmetry breaking** so that we can distinguish between some nodes:

If this distinction is not possible, then each of the \( n + 1 \) nodes could be the new node in the induction step because of the coincidence from above. Thus the ring stays symmetric in the proof for all times, hence preventing leader election.

If it is possible to distinguish between some nodes by weak symmetry breaking, we can choose the new node amongst a proper subset of all nodes, so we do not have the completely symmetric case from before.

As in the previous section, the nodes are named consecutively \( p_0, \ldots, p_{n-1} \) to ease referencing. \( p_i \) creates consecutively the messages \( m_i, m_i', m_i'', \ldots \) with the IDs \( ID_i, ID'_i, ID''_i, \ldots \). So all indices are from the factor ring \( \mathbb{Z}/n\mathbb{Z} \).

### 4.3.2 Non-aborting \( \mathcal{A} \in \mathcal{A}_{\text{compare IDs}} \) with roundtrips

Inspecting the class \( \mathcal{A}_{\text{compare IDs}} \), i.e. the algorithms that compare IDs to turn nodes passive, the weak symmetry breaking is guaranteed if at some point at least one, but not all nodes become passive. Because of lemma 4.2 there is no finite space Las-Vegas algorithm in \( \mathcal{A}_{\text{compare IDs}} \) for networks allowing message over taking. Thus we exclude this network feature for the following proofs, i.e. the order of the messages is fixed.

**Lemma 4.11.** Let \( \mathcal{A} \) be a non-aborting algorithm from \( \mathcal{A}_{\text{compare IDs}} \) that uses roundtrips and satisfies all necessary conditions from section 4.2. Then \( \mathcal{A} \) is correct for all ring sizes \( n \in \mathbb{N}_{>0} \).

**Proof.**

**Induction claim:**

\( \mathcal{A} \) is correct for rings of size \( \leq n \).

**Induction start with \( n = 1 \):**

In this trivial case, the only node immediately detects a message with a unique ID finishing its roundtrip. Thus the node becomes the leader.

**Induction step \( (n \land n + 1) \):**

**Case \text{ident.IDs}:** If all IDs are identical and \( \mathcal{A} \) does not use message overwriting, all messages make a complete roundtrip. As their order is fixed, each node makes a restart and is then in a state as if its first round, i.e. the phase between its first and second random choice, never had happened. The probability that eventually not all IDs of a round are identical equals one.

If message overwriting is enabled and happens, then at least one, but not all nodes do not make a restart. Because of lemma 4.3 round numbers (modulo \( K \)) are used. Thus the nodes without a restart become passive the moment they receive the first message from the next round. Then the ring behaves just like the one of size \( \leq n \), except for the following differences:

- Messages pass through additional passive nodes. Because of lemma 4.7 the passive nodes always forward these messages, with only their hop...
The transmissions of the messages through the passive nodes requires additional time. Since $A$ uses asynchronous transmissions, there are no constraints on the speed of message transmissions, so the delay is irrelevant.

Since $A$ selects a unique leader with probability one for the rings of size $\leq n$ by the induction claim, so does the algorithm for the ring of size $n + 1$, i.e. $A$ works correctly for the ring of size $n + 1$ for this case.

**Case not id:ent IDs:** If not all IDs are identical and there is a unique smallest one, then let $p_i$ be the node which generated that ID. $p_i$ deletes $m_i$ since $ID_{i+1}$ is larger than $ID_i$. Without message overwriting, $m_i$ turns passive because $ID_{i+1} > ID_i$. If $A$ uses message overwriting and round numbers, $p_i$ turns passive because the (round, ID) pair of the first message that $p_i$ receives is lexicographically larger than $p_i$'s. Henceforth the ring behaves just like the one of size $n_i$ except for the two differences mentioned above. So with the help of the induction claim, $A$ also works correctly for the ring of size $n + 1$ for this case.

If there are different IDs, but the smallest ID is not unique, then let $p_i$ be one of the nodes whose ID is minimal and $p_i$'s ID is not. $p_i$ deletes $m_i$, just as in the special case from above. As long as $p_i$ is receiving messages with minimal identification, it behaves just like a passive node, except for setting the unique flags of the messages to false before passing them on. But $p_i$ will purge all these messages anyway. At some point, $p_i$ receives a message with a bigger ID, which turns $p_i$ passive. Thereafter the ring behaves again like the one of size $n_i$ besides the differences from above. Using again the induction claim, $A$ also works correctly in this case for $n + 1$, and hence for all cases.  

As $A_{\text{Hal, round mod } K}$ and $A_{\text{Fokkin, round Trip}}$ are non-aborting algorithms from $A_{\text{compare IDs}}$ that use roundtrips and satisfy all necessary conditions from section 4.2, they are correct for all ring sizes $n \in \mathbb{N}_{>0}$.

### 4.3.3 Algorithm $A_{\text{Fokkin, noroundtrip}}$

Since $A_{\text{Fokkin, noroundtrip}}$ is the best amongst $A_{\text{compare IDs}}$ when using no roundtrips (cf. end of section 4.2), we only consider this specific algorithm.

**Lemma 4.12.** $A_{\text{Fokkin, noroundtrip}}$ is correct for all ring sizes $n \in \mathbb{N}_{>0}$.

**Proof.** The proof of lemma 4.11 reasons that the differences caused by additional passive nodes in the ring are irrelevant. This argumentation can be avoided by using a modified induction on $a \in \mathbb{N}_{>0}$ with the induction claim:

For arbitrary big, but known, $n \geq a$, $A_{\text{Fokkin, noroundtrip}}$ works correctly for rings of size $n$ with $\leq a$ active nodes.
Induction start with \( a = 1 \):
The only active node creates a message, which is forwarded once around the
ring. The node detects the roundtrip and the uniqueness of its ID, and therefore
becomes the leader.

Induction step \((a \land a + 1)\):
**Case \(\text{\textit{ident}}\text{ID}_k\):** If all randomly chosen IDs are identical, each active node restarts
upon receiving a message. The probability that eventually not all IDs of a round
are identical is one.

**Case \(\text{\textit{not ident}}\text{ID}_k\):** If not all IDs are identical and there is a unique smallest one,
then let \(p_k\) be the node which generated that ID. Let the following active node
be \(p_j\) and the previous active node be \(p_k\). \(p_j\) deletes \(m_i\) since \(ID_j > ID_i\).
\(m_k\) turns \(p_k\) passive because \(ID_k = ID_i\). Henceforth the ring behaves just
as if \(p_k\) had been passive all along. Since \(A_{Fokkink, no roundtrip}\) selects a unique
leader with probability one for the rings of size \(n\) with \(a\) active nodes by the
induction claim, \(A_{Fokkink, no roundtrip}\) is also correct for the ring of size \(n\) with
\(a + 1\) active nodes for this case.

If there are different IDs, but the smallest ID is not unique, then let \(p_k\) be one of
the nodes whose ID is minimal and the ID of the preceding active node \(p_k\) is
not. Let \(p_k\) be the first active node following \(p_j\) whose ID is not minimal. As
in the case from above, \(p_k\) is turned passive by \(m_k\). For \(a \in \{i, \ldots, k - 2\}\) the
message \(m_a\) causes a restart in \(p_{k+1}\), which creates a new message \(m_{a+1}'\) with
\(ID_{a+1}'\). The message \(m_{a-1}'\) is purged by \(p_k\) since \(ID_{a-1} > ID_k\). Thereafter
the ring is the same as if \(p_k\) had been passive all along and \(p_{k+1}, \ldots, p_k\) had
immediately created the messages \(m_{k+1}', \ldots, m_{k-1}'\). Using again the induction
claim, \(A_{Fokkink, no roundtrip}\) works correctly also in this case for the ring of size
\(n\) with \(a + 1\) active nodes.

### 4.3.4 Algorithm \(A_{\text{timing}}\)

\(A_{\text{timing}}\) also turns some, but not all nodes passive, not with the help of random
IDs but by randomly waiting. Hence weak symmetry breaking is guaranteed.

**Lemma 4.13.** \(A_{\text{timing}}\) is correct for all ring sizes \(n \in \mathbb{N}_{>0}\).

**Proof.** The proof is similar to that of lemma 4.11 as nodes are turned passive
and messages make complete roundtrips. But we use induction on the number
of active nodes \(a \in \mathbb{N}_{>0}\), as in lemma 4.12.

Induction claim:
For arbitrary big, but known, \(n \geq a\) \(A_{\text{timing}}\) works correctly for rings of size \(n\)
with \(a\) active nodes.

Induction start with \(a = 1\):
With probability one, the only active node eventually initiates a leader request,
which then makes a complete roundtrip. Thereafter the active node becomes
the leader.
4.3. Mathematical correctness proofs

Induction step \((a \rightarrow a + 1)\):
With probability one eventually some active node initiates a leader request.
Case all request: If all active nodes initiate leader requests before receiving one, then all requests make complete roundtrips and all active nodes reset into their initial state. The probability that eventually not all active nodes initiate leader requests is one.

Case not all request: If some but not all active nodes initiate leader requests, then there is an active node \(p_i\) which receives a request before initiating an own one. Upon receipt the node turns passive and hence the ring is the same as if \(p_i\) had been passive all along. With the help of the induction claim, the ring elects a unique leader with probability one, i.e. \(A_{\text{timing}}\) also works correctly for rings of size \(n\) with \(a + 1\) active nodes.

4.3.5 Algorithm \(A_{\text{sorting}}\)

In \(A_{\text{sorting}}\) each node is equally involved in the process of finding a leader, as each node contributes a bit to each bit array. Hence just until a leader is elected, \(A_{\text{sorting}}\) does not perform any weak symmetry breaking, and no reduction step is possible. But \(A_{\text{sorting}}\) is an abstract algorithm, i.e. with no concrete decision procedure. If we assume that transmissions are without errors and that the decision procedure either invokes a restart or selects a unique leader, then there is nothing to prove, anyway.
Chapter 5

Model checking with PRISM

This chapter is model checking the leader election algorithms from chapter 4 using PRISM (version 3.0.31). For each algorithm, we first explain the design of the algorithm, then analyse its properties and give statistics about the model checking procedure. The chapter finishes with a conclusion.

5.1 Algorithm $A_{Itai, round \leq K}$

5.1.1 Algorithm design and specification

$A_{Itai, round \leq K}$ is based on one of the first leader election algorithms for unidirectional, anonymous rings: $A_{Itai}$ (cf. listing 4.1).

Since the model specification language of PRISM is not imperative, but declarative using sets of guarded commands, we introduce an additional variable state to describe in which kind of state a node is in. The design of $A_{Itai, round \leq K}$ is facilitated by translating the imperative description of $A_{Itai}$ from listing 4.1 into a state machine, given in figure 5.1.

The asynchronous send and receive operations are put into effect by channels. Since these are not offered by PRISM, they must be implemented, too. This gives us the chance to specify the precise channel behaviour we want: We will first describe channel simple with a buffer of size one which forbids message overtaking but allows message overwriting.

Later in this section we also consider the extension channel realistic of the previous channel with a buffer of size three, which can therefore non-deterministically transmit the messages, hence modeling a more realistic channel behaviour.

The last kind of channel we specify, channel overtaking, has a buffer of size two and allows message overwriting as well as message overtaking.
At the beginning of the model specification the type of model and the constants in use are given. Since we want to model the scheduling of concurrent events by non-determinism, we use an MDP:

**mdp**

```
const int N=3;         //number of nodes n
const int K=3;         //number of rounds
const int M=3;         //number of IDs
const double Prob = 1/M;
```

Thereafter channel `simple` is specified:

**module channel1**

```
Buffer1_round : [0..K];
Buffer1_id : [0..M];      //0 = empty buffer
Buffer1_hop : [1..N];
Buffer1_unique : bool;

[p1_to_c1] true  -> (Buffer1_hop=mes1_hop) &
             (Buffer1_round=mes1_round) & (Buffer1_id=mes1_id) &
             (Buffer1_unique=mes1_unique);

[c1_to_p2] (Buffer1_id > 0)  -> (Buffer1_round=0) &
            (Buffer1_id=0) & (Buffer1_hop=1) &
            (Buffer1_unique=false);
```

**endmodule**

The local variables define the buffer of size one, `Buffer1_id=0` stands for an empty buffer. The first command synchronizes with the first node (see below) on the
5.1. Algorithm $A_{\text{Alt}_i, \text{round} \leq K}$

action label $p_{1\to c1}$. The channel reads the values of the message from the first node and saves them in its buffer. Since the command’s guard is always true, the channel also reads a message in its buffer when it already holds a message, hence allowing message overwriting. The second command synchronizes with the second node on the action label $c1\to p2$. As the second node reads the message from the buffer, the channel can delete its buffer. Setting Buffer1id back to zero would be sufficient for a correct algorithm, but cleaning up all scratch data by resetting all local variables reduces the state space.

The specification of the module for the first node begins with:

```
module node1
  active1 : bool init true;
  round1 : [0..K];
  id1 : [1..M];
  state1 : [0..4];
  mes1_round : [0..K];
  mes1_id : [1..M];
  mes1_hop : [1..N];
  mes1_unique : bool;
```

The variables with the prefix "mes1" hold incoming and outgoing messages. The other variables define the node's behaviour, which is similar to the state machine of figure 5.1. The states "before creating", "before sending", "before receiving", "before deciding", and the final state of the state machine correspond to the values 0, 1, 2, 3, and 4 for state1, respectively. The following commands successively define the node's behaviour for the different values of state1.

In state1=0 the node randomly chooses an ID, creates the appropriate message, and changes into state1=1. The code is for $M = 3$ and has to be changed accordingly for other values:

```
[] (state1 = 0) ->
  Prob: (state1 =1) & (id1' =1) & (mes1_id' =1) & (mes1_hop' =1) & (mes1_unique' =true) & (mes1_round' =round1) +
  Prob: (state1 =1) & (id1' =2) & (mes1_id' =2) & (mes1_hop' =1) & (mes1_unique' =true) & (mes1_round' =round1) +
  Prob: (state1 =1) & (id1' =3) & (mes1_id' =3) & (mes1_hop' =1) & (mes1_unique' =true) & (mes1_round' =round1);
```

In state1=1 the node sends the message to the first channel by synchronization, cleans up its scratch data to reduce the state space, and changes to state1=2:

```
[p1_to_c1] (state1 = 1) -> (state1 =2) & (mes1_unique' =false) & (mes1_id' =1) & (mes1_hop' =1) & (mes1_round' =0);
```

The node waits in state1=2 until it can receive a message from the last channel by synchronization, sets its message variables accordingly, and moves into state1=3. The code is designed for $n = 3$ and has to be changed accordingly for other values:

```
c3_to_p1] (state1 = 2) & (Buffer3_id > 0) ->
  (state1 =3) & (mes1_round' =Buffer3_round) &
```
(mes1\_\text{id}='Buffer3\_\text{id}) & (mes1\_\text{hop}'='Buffer3\_\text{hop}) &
(mes1\_\text{unique}'='Buffer3\_\text{unique});

In state 1=3 the node checks the message to decide what to do:

If the message has a larger pair \((\text{mes1\_\text{round}}, \text{mes1\_\text{id}})\) than the node, the node becomes passive, updates \text{round1} and \text{id1}, increases the message's \text{mes1\_\text{hop}} counter, and returns to state 1=1. \text{mes1\_\text{hop}} < \text{N} is a consequence of the other propositions in the guard. Stating it nevertheless prevents error messages by PRISM (cf. page 52):

\[
[\text{state1} = 3) \land (\text{mes1\_\text{hop}} < \text{N}) \land (\text{mes1\_\text{round} > round1}) \land 
(\text{mes1\_\text{round} = round1}) \land (\text{mes1\_\text{id} > id1}) \land 
(\text{state1'}=1) \land (\text{round1'}=\text{mes1\_\text{round}}) \land (\text{id1'}=\text{mes1\_\text{id}}) \land 
(\text{active1'}='false) \land (\text{mes1\_\text{hop}'}=\text{mes1\_\text{hop}+1});
\]

In case \((\text{mes1\_\text{round}}, \text{mes1\_\text{id}})\) is equal to \((\text{round1}, \text{id1})\), but the message has not yet finished its round trip, its \text{mes1\_\text{unique}} flag is set to false, its \text{mes1\_\text{hop}} counter is increased, and the node moves to state 1=1:

\[
[\text{state1} = 3) \land (\text{mes1\_\text{hop}} < \text{N}) \land (\text{mes1\_\text{round} = round1}) \land 
(\text{mes1\_\text{id} = id1}) \land 
(\text{state1'}=1) \land (\text{mes1\_\text{unique}'}='false) \land 
(\text{mes1\_\text{hop}'}=\text{mes1\_\text{hop}+1});
\]

If \((\text{mes1\_\text{round}}, \text{mes1\_\text{id}})\) is smaller than \((\text{round1}, \text{id1})\), the message is purged by directly moving to the receiving state state1=2:

\[
[\text{state1} = 3) \land (\text{mes1\_\text{round} < round1}) \land 
(\text{mes1\_\text{id} < id1}) \land 
(\text{mes1\_\text{round} = round1}) \land 
(\text{mes1\_\text{unique}'}='false) \land 
(\text{mes1\_\text{hop}'}=\text{mes1\_\text{hop}+1});
\]

In case the message has completed its round trip, but its \text{mes1\_\text{unique}} flag is false, the node's behaviour differs from figure 5.1 to enforce a finite state space (cf. section 4.2.2). The node's action depends on the number of rounds it has already performed: If it has not reached its limit \text{K}, it increases its variable \text{round1} and restarts by changing state 1 to 0. Otherwise the node terminates, modelled in PRISM by a self-loop. To avoid error messages, we again strengthen the guard by demanding the node to be active (cf. section 5.7):

\[
[\text{state1} = 3) \land (\text{round1} < \text{K}) \land (\text{mes1\_\text{hop} = N}) \land (\text{active1}) \land 
(\neg \text{mes1\_\text{unique}) \rightarrow (\text{state1'}=0) \land (\text{round1'}=\text{round1}+1));
\]

\[
[\text{state1} = 3) \land (\text{round1} = \text{K}) \land (\text{mes1\_\text{hop} = N}) \land (\text{active1}) \land 
(\neg \text{mes1\_\text{unique}) \rightarrow \text{true};
\]

In the event of a round trip with the message being unique and the node still being active, the node becomes the leader by moving to state1=4:

\[
[\text{state1} = 3) \land (\text{mes1\_\text{hop} = N}) \land (\text{active1}) \land 
(\text{mes1\_\text{unique}) \rightarrow (\text{state1'}=4));
\]
Finally, in state1 = 4 the leader terminates by making a self-loop:

```
[] (state1 = 4) -> true;
endmodule
```

The two further nodes and channels in the ring of size \( n = 3 \) are specified by module renaming the first ones. Although the full parallel composition operator is associative and commutative (cf. subsection 2.3.1), the order of the modules in the specification does matter: it determines the variable ordering in the MTBDD, which influences its size. Grouping together modules which are related to each other often yield better results. Hence we will not group the modules for the channels separate from those for the nodes, but interleave them:

```
module channel2=channel1 [Buffer1_round=Buffer2_round,
  Buffer1_id=Buffer2_id, Buffer1_hop=Buffer2_hop,
  Buffer1_unique=Buffer2_unique, p1_to_c1=p2_to_c2,
  c1_to_p2=c2_to_p3, mes1_round=mes2_round, mes1_id=mes2_id,
  mes1_hop=mes2_hop, mes1_unique=mes2_unique]
endmodule
```

```
module node2-model [active1=active2, round1=round2,
  id1=id2, state1=state2, mes1_round=mes2_round,
  mes1_hop=mes2_hop, mes1_unique=mes2_unique,
  mes1_id=mes2_id, p1_to_c1=p2_to_c2, c3_to_p1=c1_to_p2,
  Buffer3_round=Buffer1_round, Buffer3_id=Buffer1_id,
  Buffer3_hop=Buffer1_hop, Buffer3_unique=Buffer1_unique]
endmodule
```

And likewise for the third channel and node.

This completes the specification of \( \mathcal{Alba}, \text{round} \leq K \). Every node and channel has been defined explicitly, as in PRISM the initialization of the models needs to be hard-coded in the specification. Hence we need a model specification for each ring size \( n \). Also, the command for the random choice of the ID depends on \( M \). If \( M \) is bounded by \( x \in \mathbb{N} \), the modification of the command can be omitted by only using the command with \( x \) updates and setting the probability \( \text{Prob} \) of each choice to \( 1/M \). If \( M < x \), PRISM ignores all but the first \( M \) updates and gives the warning "Update i of command i of module node j doesn’t do anything". Alternatively, by hard-coding the commands for all choices of \( M \) and selecting the appropriate one with the help of the guards, the warning can be avoided. To perform a PRISM experiment over a range of values for \( M \) (e.g. in figure 5.11), the modification of the command must be avoided by one of the two variants.

Channels\text{simple} does not precisely model transmissions in the network, since receiving from such a channel can happen immediately after sending to it. By setting the buffer size to three, channel\text{realistic} can non-deterministically move its messages from the first buffer entry to the last, where it is readable by the subsequent node. This models the behaviour of the network:

```
module channel1

  pre Buffer1_round : [0..K];
```
Again, this channel synchronizes with the first node to receive a message into preBuffer1. Since the guard is always valid, this channel allows message overwriting, too:

$$[p1\rightarrow c1] \text{true} \rightarrow (\text{preBuffer1}._\text{unique} = \text{mes1}._\text{unique}) \& (\text{preBuffer1}._\text{id} = \text{mes1}._\text{id}) \& (\text{preBuffer1}._\text{round} = \text{mes1}._\text{round}) \& (\text{preBuffer1}._\text{hop} = \text{mes1}._\text{hop});$$

Channel realistic can then non-deterministically move the message from preBuffer1 to cable1 if the latter is empty:

$$[] (\text{preBuffer1}._\text{id} > 0) \& (\text{cable1}._\text{id} = 0) \rightarrow$$

$$(\text{cable1}._\text{round} = \text{preBuffer1}._\text{round}) \& (\text{cable1}._\text{id} = \text{preBuffer1}._\text{id}) \& (\text{cable1}._\text{hop} = \text{preBuffer1}._\text{hop})$$

$$\& (\text{cable1}._\text{unique} = \text{preBuffer1}._\text{unique}) \& (\text{preBuffer1}._\text{id} = 0) \& (\text{preBuffer1}._\text{round} = 0) \& (\text{preBuffer1}._\text{hop} = 1) \& (\text{preBuffer1}._\text{unique} = \text{false});$$

The channel afterwards non-deterministically moves the message from cable1 to postBuffer1, which does not have to be empty, i.e., messages can be overwritten in postBuffer1 as well.

$$[] (\text{cable1}._\text{id} > 0) \rightarrow$$

$$(\text{postBuffer1}._\text{round} = \text{cable1}._\text{round}) \& (\text{postBuffer1}._\text{id} = \text{cable1}._\text{id}) \& (\text{postBuffer1}._\text{hop} = \text{cable1}._\text{hop})$$

$$\& (\text{postBuffer1}._\text{unique} = \text{cable1}._\text{unique}) \& (\text{cable1}._\text{round} = 0) \& (\text{cable1}._\text{hop} = 1) \& (\text{cable1}._\text{unique} = \text{false});$$

Now that there is a message in postBuffer1, the second node can receive it by synchronization. Thus the channel can clear its postBuffer1 by resetting postBuffer1._id. All other scratch data is removed, too.

$$[c1\rightarrow p2] (\text{postBuffer1}._\text{id} > 0) \rightarrow (\text{postBuffer1}._\text{round} = 0) \& (\text{postBuffer1}._\text{id} = 0) \& (\text{postBuffer1}._\text{hop} = 1) \& (\text{postBuffer1}._\text{unique} = \text{false});$$

endmodule

As the buffer names of the channels have changed, the specifications of the nodes have to be modified accordingly.
5.1. Algorithm $A_{\text{alt.round}} \leq K$

Channel overtaking can model message overtaking. The channel puts this into effect by non-deterministically swapping its two buffer entries. To save the cable variables, transmissions are simplified so they require no time at all, just as for channel\_simple.

```plaintext
module channel

Buffer1\_round : [0..K];
Buffer1\_id : [0..M]; //0 = empty buffer
Buffer1\_hop : [1..N];
Buffer1\_unique : bool;

Buffer1b\_round : [0..K];
Buffer1b\_id : [0..M]; //0 = empty buffer
Buffer1b\_hop : [1..N];
Buffer1b\_unique : bool;
```

Since we want to use some of our algorithms for both cases of message overtaking enabled and disabled, we encapsulate a channel's behavior into its module. So the nodes do not know whether the channels in use allow message overwriting, and always synchronize using the same action labels and transfer messages over the same variables. Hence we spare modifying the modules for the nodes when changing between channel\_simple and channel\_overwriting.

Since the guard is always valid, channel\_overwriting allows message overwriting, too:

```plaintext
[pl\_to\_c1] (Buffer1\_id=0) \rightarrow (Buffer1\_round'=mes1\_round) \&
(Buffer1\_id'=mes1\_id) \& (Buffer1\_hop'=mes1\_hop) \&
(Buffer1\_unique'=mes1\_unique);
```

Because of the encapsulation, the second node always reads the message from the first Buffer. Consequently the possibility of message overwriting is modelled by non-deterministically swapping the buffers it may be both filled:

```plaintext
[pl\_to\_c1] (Buffer1\_id > 0) \rightarrow (Buffer1b\_id'=mes1\_id) \&
(Buffer1b\_round'=mes1\_round) \& (Buffer1b\_hop'=mes1\_hop) \&
(Buffer1b\_unique'=mes1\_unique);

[pl\_to\_c1] (Buffer1\_id > 0) \rightarrow (Buffer1b\_id'=Buffer1\_id) \&
(Buffer1b\_round'=Buffer1\_round) \&
(Buffer1b\_unique'=Buffer1\_unique) \&
(Buffer1b\_hop'=Buffer1\_hop) \& (Buffer1\_round'=mes1\_round) \&
(Buffer1\_id'=mes1\_id) \& (Buffer1\_hop'=mes1\_hop) \&
(Buffer1\_unique'=mes1\_unique);
```

If Buffer1b contains a message, it is immediately moved to the first buffer when the second node fetches the message from the first Buffer1:

```plaintext
[c1\_to\_p2] (Buffer1\_id > 0) \& (Buffer1b\_id = 0) \rightarrow
(Buffer1\_round'=0) \& (Buffer1\_id'=0) \& (Buffer1\_hop'=1) \&
(Buffer1\_unique'=false);

[c1\_to\_p2] (Buffer1\_id > 0) \& (Buffer1b\_id > 0) \rightarrow
(Buffer1\_round'=Buffer1b\_round) \&
```
(Buffer1.id = Buffer1b.id) & (Buffer1hop = Buffer1b.hop) &
(Buffer1_unique = Buffer1b_unique) & (Buffer1b_round = 0) &
(Buffer1.id = 0) & (Buffer1b.hop = 1) &
(Buffer1b_unique = false);
endmodule

5.1.2 Model properties and statistics

All verifications are performed on the following hardware: An Intel(R) Pentium(R) M processor with 1.86GHz and 1 GByte RAM.
PRISM’s options are always set the following way:

- **Engine:**
  As engine performing numerical computations we select hybrid, as it often yields about the same high compression rate as the pure MTBDD engine with about the same fastness as the sparse (matrices) engine.

- **Use fairness:**
  Fairness (cf. section 5.6 and section 5.1.1) is required for certain property checks of MDPs. Therefore we enable fairness.

- **Use precomputation:**
  Precomputation performs an initial, graph-based detection of cases where probabilities are trivially zero or one. Thus we activate it.

- **Do prob checks:**
  We enable this check for probability vectors, which gives an error if a command’s probabilities do not sum to one for some state. Unfortunately PRISM always checks for out-of-range values, even for unreachable states, and an update with an out-of-range value for some (maybe unreachable) state s is ignored. Hence the command’s probabilities do not sum to one for s. Consequently the guards sometimes unnecessarily have to be strengthened, which can obfuscate the code. An alternative to unnecessarily strengthening the guard so that it excludes s is to deactivate the correctness check for probability vectors. But then real errors in probability vectors are not reported, either.

- **Iterative method:**
  For solving the linear equation systems to compute probabilities, we choose the numerical iterative method Jacobi.

- **Termination criteria:**
  The computation of the difference of successive iterations can be set to relative or absolute. If the difference drops below the threshold set in the following option, iteration stops. To avoid premature termination, we choose relative differences.

- **Termination epsilon:**
  As mentioned in the former option, this is the threshold for the difference between successive iterations. We set it to 1.0e-6.
5.1. Algorithm $A_{\text{Hai}, \text{round} \leq K}$

- **Termination max iterations:**
  If the computed value does not converge fast enough for termination by differences moving below the threshold (cf. the two former options), the computation stops after a maximum number of iterations. We choose 10000 iterations.

- **Cudd max memory:**
  We set the maximal memory for CUDD, the Colorado University Decision Diagram library used for the MTBDDs, to 1000000KB.

- **Cudd epsilon:**
  If two double precision floating point terminals of the MTBDD differ by less than this threshold, they are considered to be identical. We set cudd epsilon to 1.0e-15, which is almost as precise as double floating points. Nevertheless this and the Jacobi method are approximations, which are of course necessary since the set of probability values is continuous. So strictly our verifications are not 100% stringent.

- **Hybrid num. levels:**
  This parameter influences the heuristics for the hybrid engine. In the absence of a detailed description we leave the default value -1 unchanged.

- **Hybrid max memory:**
  We set the maximal memory for the hybrid engine to 1024.

We will first thoroughly investigate the ring using channel$_{\text{simple}}$.

Before any property can be checked, the model must be constructed. For this PRISM parses the model specification, converts it into an MTBDD, and performs reachability tests to remove the non-reachable states from the MTBDD. Figure 5.2 plots the time for constructing the corresponding models for $n=3$.

Since PRISM reuses the MTBDD structure built for the previous model, the runtime can significantly change depending on whether a similar model has been constructed before. E.g. for $n=3, M=3$ and $K=5$, the time for model construction is 437 seconds without a similar model already in memory, and 53 seconds with the model for $n=3, M=3$ and $K=4$ in memory. Hence we measure the runtimes using experiments, which perform verifications over a range of values for the parameters not constant. E.g. in figure 7.2 each function corresponds to an experiment which successively increases $K$ while $n$ and $M$ are fix.

We exclude investigating the trivial case of $K=0$ since it does not contain different round numbers and consequently does not use the complete functionality of the algorithm.

The values of $K$ not plotted exceed the construction time of one hour. For $M$ equal to 7 or 8 and $K=4$ PRISM fails to build the model by receiving an abortion signal: Either signal 11, which is caused by a segmentation fault, or signal 22, which is caused by an erroneous instruction. Both error signals are probably results of memory overflow in PRISM’s native code.
In the plot the time increases exponentially with both $M$ and $K$.

Besides the runtime for model construction, there are several values concerning the size of the model and its representation:

Figure 5.2 displays the number of states of the MDPs for the parameters in figure 5.2. Figure 5.3 the number of transitions, and figure 5.5 the number of nodes of the MTBDDs that represent the MDPs. The number of terminals, i.e. terminal nodes, of the MTBDDs is always three.

The figures 5.3 to 5.5 show that the number of states, transitions and nodes are all about linear in $K$ and quadratic in $M$.

So the rise in both space and time requirements is very severe. Investigating the fact that the variable ordering in the MTBDD influences its size might entail an improvement: [Rutten et al., 2004 page 194] shows that a diminution of the distance between dependent variables can significantly increase the compression ratio of the MTBDD. Inspecting our source code in this regard yields a simple
5.1. Algorithm $A_{Hai, round \leq K}$

![Graph](image1.png)

Figure 5.4: Number of transitions for $A_{Hai, round \leq K}$, $n=3$

![Graph](image2.png)

Figure 5.5: Number of nodes for $A_{Hai, round \leq K}$, $n=3$

variation which probably improves the verification: Swapping the $n$th channel with the $n$th node results in listing the modules in the same order as their location on the ring. The complete PRISM specification for $n=M=3$ can be found in appendix A.1.

The construction times of the improved model specification for the same parameters as in figure 5.2 are plotted in figure 5.6.

Time requirements for model construction is between 6% and 60% of the former. This confirms the heuristic that grouping dependent variables together decreases the MTBDD. Interestingly, for $M \in \{4, 5, 7, 8, 9\}$ the plotted function is almost identical to the function for $M=1$ in figure 5.2.

We also plot the construction times for $n=4$ in figure 5.7 to display their rise with increasing size of the ring.
Figure 5.6: Construction time for improved $A_{Hai, round} \leq K$, $n=3$

Figure 5.7: Construction time for improved $A_{Hai, round} \leq K$, $n=4$

In figure 5.4 the reachability computations succeed for $M \in \{4,5,6\}$ and the corresponding last value of $K$, but an error signal appears before the model construction completes. Now $M=2$ looks very similar to $M=4$ for $n=3$. Besides $M=2$ with $K=1$, all values are at least 10 times bigger in $n=4$ than in $n=3$, sometimes the factor is almost 100.

To be able to compare the sizes of the MTBDDs for the former model specification and the improved one, we again show their number of nodes for $n=3$ in figure 5.3. There are still only three terminals in the MTBDD. The model, i.e. the number of states and transitions, does not change, since the full parallel composition operator is associative and commutative (cf. subsection 2.3.1).

The number of nodes for the improved model specification are between 77% and 86% of the old version’s nodes, with smaller percentage values for larger MTBDDs. Since the state space is identical in both cases, the MTBDD of the improved version has a higher compression ratio.
For $n=5$ with $M \geq 4$, for $n=6$ with $M \geq 3$, and for $n=7$ no model construction is possible. Some parameters which do not entail memory overflow are given in Table 5.1 which lists:

- the parameters chosen for the constants of the model specification
- the number of states and transitions of the MDP
- the number of nodes and terminals for the corresponding MTBDD
- the construction time, which is given throughout this thesis in an appropriate unit of either seconds ("s"), minutes ("m"), or hours ("h").

<table>
<thead>
<tr>
<th>constants: $n$</th>
<th>model: $M$ $K$</th>
<th>states</th>
<th>transitions</th>
<th>MTBDD: nodes</th>
<th>term.</th>
<th>construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n=3$</td>
<td>2 1</td>
<td>4185</td>
<td>8952</td>
<td>45432</td>
<td>3</td>
<td>2&quot;s</td>
</tr>
<tr>
<td>$n=4$</td>
<td>2 1</td>
<td>61109</td>
<td>164264</td>
<td>219230</td>
<td>3</td>
<td>14&quot;m</td>
</tr>
<tr>
<td>$n=5$</td>
<td>2 2 1</td>
<td>860128</td>
<td>2799655</td>
<td>1334080</td>
<td>3</td>
<td>752&quot;s</td>
</tr>
<tr>
<td>$n=5$</td>
<td>2 3</td>
<td>2447532</td>
<td>7897855</td>
<td>2586080</td>
<td>3</td>
<td>3666&quot;s</td>
</tr>
<tr>
<td>$n=5$</td>
<td>3 1</td>
<td>3529399</td>
<td>12282825</td>
<td>2422887</td>
<td>3</td>
<td>1691&quot;h</td>
</tr>
<tr>
<td>$n=6$</td>
<td>2 1</td>
<td>11884362</td>
<td>45541800</td>
<td>3086247</td>
<td>3</td>
<td>3031&quot;h</td>
</tr>
</tbody>
</table>

Table 5.1: Model checking values for selected parameters

The table shows that the number of states, transitions, nodes, and the construction time are all exponential in $n$.

The first property we are interested in is whether the algorithm and its model specification are correct. So $\mathcal{A}_{\text{Hai, round $\leq K$}}$ must either elect exactly one leader or reach the limit $K$. This is described with the PCTL \textbf{oneLeader property}

$$P >= 1 \ [\text{true} \ U \ ("\text{oneLeader" | "limitReached"}) ]$$
In the case \( n = 3 \), the label "**oneLeader**" is defined as

\[
\text{(state1 = 4 ? 1 : 0) + (state2 = 4 ? 1 : 0) + (state3 = 4 ? 1 : 0) = 1,}
\]

i.e. exactly one node is in state number four, and the label "**limitReached**" is defined as

\[
\text{round1 = K & round2 = K & round3 = K.}
\]

The property was successfully checked for all relevant instances that allow model construction, except for \( n=4, M=3, K=4 \), which causes an abortion by an error signal. Irrelevant instances are the medium ones enframed by instances with successful checks, e.g. \( n=3 \) and \( M=8 \) with \( K=2 \) enframed by \( K=1 \) and \( K=3 \).

The time requirements for model checking the property are at most 70% of the improved construction time, usually much smaller. So we do not list the time requirements, as the model construction time is the bottleneck.

**Note.** Section 5.6 shows that for oneLeader property checks that only consider fair schedules, the MDP can be transformed into the corresponding DTMC. This change can be an improvement since sometimes the verifications are more efficient for DTMCs.

The oneLeader property does not imply that once a leader is elected the algorithm terminates with exactly that leader. Therefore (Parker and Norman, 2001) checks the **leader preservation invariance property**, which states that never more than one leader is elected, i.e.

\[
\text{(state1=4?1:0)+(state2=4?1:0)+...+(state n=4?1:0) <= 1.}
\]

Only after manually checking in the model specification that a node in state 4 never leaves it, this property says that once a leader is elected exactly that one node always stays the leader. To avoid this manual check, we can include into the property that once a node \( i \) is in state 4, it always stays there. This results in the CTL formulae

\[
\forall i \in \{1, \ldots, n\} : AG (state i = 4 \implies AG (state i = 4 \land \forall j \neq i \text{ (state } j \neq 4)).
\]

The formula with \( i=1 \) is sufficient because the network is symmetric. As PRISM checks a property for all reachable states, this translates into the **second leader preservation property**

\[
\text{(state1 = 4) => P <= 0 [ true U (state1=4) \& (state2=4) \& \ldots \& (state n!=4) ]].}
\]

Since states that terminate are modelled by self-loops, the termination cannot be checked directly. But if we remove all self-loops, i.e. all commands with the update **true**, PRISM automatically adds self-loops to all terminals and labels them with "**deadlock**". Then we can use the **termination property**

\[
P >= 1 [ \text{true U } "\text{deadlock}" ].
\]
Besides leader election, the following property of **empty buffers** can be of interest.

\[ \text{("oneLeader") } \Rightarrow \text{(Buffer1.id + Buffer2.id + Buffer3.id = 0).} \]

Assuming that the algorithm is really message driven, the correctness of the oneLeader property and the empty buffer property implies the correctness of the leader preservation properties and of the termination property. Among the instances that allow model construction, \((n, M, K) = (5, 3, 1)\) and \((n, M, K) = (4, 6, 1)\) entail an error signal when checking the termination property. So does \((n, M, K) = (4, 3, 4)\), which also causes an error signal when checking the leader preservation property. For \((n, M, K) = (6, 2, 1)\) the model construction without explicit self-loops, needed for the termination property, already causes an abortion signal. The properties hold for all relevant verifications that do not abort with an error signal.

The time requirements for the leader preservation invariance property is at most one second, for the empty buffer property at most seven seconds. The second leader preservation property requires at most 10% of the improved model construction time, the termination property at most 80%.

As \(A_{\text{Hai,round} \leq K}\) is a Monte-Carlo algorithm, we compute the probabilities of termination with exactly one leader, for various values of \(n\), \(M\), and \(K\). A **worst case** probability, i.e. minimal probability taken over all non-deterministic choices, is computed with the formula

\[
P_{\text{min}} = ? \left[ \text{true U ("oneLeader")} \right].
\]

The maximal probabilities are always equal to one if message overwriting is enabled: In a scheduling with one node not transmitting until it received all remaining messages, all but the last message are deleted by overwriting, and a unique leader is elected in the next round, independent of which IDs were randomly chosen.

The ring with size \(n=2\) is a special case, since for each round a leader is elected iff the other node turns passive. Thus

\[
P_{\text{["oneLeader" within K rounds]}} = P_{\text{["oneLeader" in first round]}} + (1 - P_{\text{["oneLeader" in first round]}}) P_{\text{["oneLeader" within K-1 rounds]}}.
\]

Since \(P_{\text{["oneLeader" in first round]}} = (M - 1)/M\), we get

\[
P_{\text{min}} = ? \left[ \text{true U ("oneLeader")} \right] = \sum_{i=0}^{K} \frac{M - 1}{M^{i+1}}.
\]

In figure 5.11 \(n\) is set to 3 while \(M\) and \(K\) vary. To display how the probabilities change when \(n\) is increased, figure 5.10 shows them for \(n=4\). The parameters not plotted cause an error signal.
As expected the probability of an elected leader increases with both $M$ and $K$, most of the time slightly stronger with $K$ than with $M$.

There is a high correlation between the probability of an elected leader and the time for model construction. Of course the probabilities for $n=4$ are smaller than for $n=3$, but approach faster the value one for increasing $K$.

Although the MTBDD is smaller for the improved order of the modules, the time for computing the probabilities increases for this optimization. So the computation time not only depends on the size of the MTBDD, but also on its compression ratio.

The time required for computing the probabilities is either smaller than 1000 seconds or less than 50% of the construction time, except for the instances with $n=4$ and $M > 3$, which require less than 200% of the construction time to compute the probabilities.

Since $\mathcal{A}_{\text{ltai,round } \leq K}$ is a Monte-Carlo algorithm, we will not compute mathematical expectations like the average traffic or the average number of rounds.
5.1. Algorithm $A_{lat, round \leq K}$

until a leader is elected: Because of the algorithm’s premature erroneous terminations, the values are misleading, and a comparison to the same expectations for the other algorithms, which are Las-Vegas algorithms, is unfair.

When changing from channel$_{simple}$ to channel$_{realistic}$, the construction time and the size of the model and MTBDD increases. Therefore the following instances also entail error signals while constructing the model: $n=3$ and $M=9$ with $K \geq 2$, $n=4$ with $M \geq 5$, $n=5$ with $M \geq 3$ and $n \geq 6$.

Among all parameters with successful property checks from above, the following cannot be applied because of error signals: of course all parameters that do not allow model construction, and for $n \geq 5$ the oneLeader property and the termination property.

The properties hold for all relevant verifications that do not abort with an error signal.

The values of ($P_{min} = ? [true U ("oneLeader")]) are identical with the ones for channel$_{simple}$. So the imposed delay by realistic message transmissions does not influence the worst case probability values. The reason is that the nondeterministic scheduling of channel$_{simple}$ already comprises all possible orders of concurrent events, i.e. all race conditions of message transmissions. We will therefore no longer consider channel$_{realistic}$.

Using channel$_{overtaking}$ increases the construction time and the size of the model and its MTBDD even more because message overtaking opens up many further courses of the leader election procedure (figure 4.2 depicts an example). Consequently for $n=3$ with $M > 6$ no model can be constructed. For $n=3$ with $M = 6$ only $K = 1$ is possible, for $n=3$ and $M=2$ the value for $K$ may reach $5$. For $n \geq 4$ no model construction is possible.

The empty buffer property does not hold, since an old message can stay in a channel arbitrarily long. All other properties can successfully be checked for all instances allowing model construction.

The values for ($P_{min} = ? [true U ("oneLeader")]) are the same as without message overtaking, so the worst case scheduling does not change: If an overtaken message is transmitted within the same round, the result in the following node after the swapped messages are received is the same as without changing the order. An outdated message from a previous round is being purged, since round numbers always increase.

Because of the increased requirements, we use channel$_{simple}$ as regular channel and only consider special cases with channel$_{overtaking}$.

The proof of lemma 4.1 reasons that round numbers in the complete ring maximally differ by one for algorithms in $A_{compare,1Ds}$ with round numbers, roundtrips, and no message overtaking. The linearity in $K$ of the number of states in figure 5.3 is an indication of this fact for $A_{lat, round \leq K}$ using channel$_{simple}$, which is such an algorithm. But the fact can also be verified
with the following PCTL formulae:
\[ \forall i,j \in \{1,\ldots,n\} : \]
\[ P > I[ (\text{round}_i = \text{round}_j) | (\text{round}_i + 1 = \text{round}_j) | (\text{round}_i = \text{round}_j + 1) ] \]
\[ U ("\text{oneLeader}\" | "\text{limitReached}\") \]

Because we have a symmetric network, the formulae with \( i = 1 \) and \( j \in \{1,\ldots,[(n + 1)/2]\} \) are sufficient.

PRISM successfully checks these formulae for the parameters \( n=3, M=3, K=5 \)
and for \( n=4, M=3, K=3 \). For larger \( n \) with \( K > 2 \) PRISM gives error signals.
The time for model checking one of the PCTL formulae requires about one third
of the corresponding construction time.

So for all instances verifiable without memory overflow, round numbers really
maximally differ by one. We will take advantage of this property in the following
section.

5.2 Algorithm \( A_{Hai, \text{round} \mod K} \)

5.2.1 Algorithm design and specification

A major disadvantage of algorithm \( A_{Hai, \text{round} \leq K} \) is its use of round numbers.
Aborting the algorithm when the round numbers reach a limit \( K \) results in
a Monte-Carlo algorithm, with the error probability getting the bigger, the
smaller \( K \) is chosen. But since the state space grows linearly with \( K \), the time
requirements even exponentially, only small increments of \( K \) are possible.

When \( K \) is reached the alternative to abortion is restarting the algorithm. This
eliminates the error and thus leads to a Las-Vegas algorithm, without the need
to increase \( K \). As the nodes do not know whether there are still messages in
the network, there is no clean global restart. So each node starts over when
its round numbers reach \( K \), even if there might still be messages and nodes
in previous rounds. This corresponds to regarding round numbers modulo \( K \),
which is correct by lemma 5.2 if message overtaking is forbidden. This leads to
the algorithm \( A_{Hai, \text{round} \mod K} \).

So the specification of the first node is almost the same as in \( A_{Hai, \text{round} \leq K} \).
The only differences are in state 1 = 3:

- \( \text{mes1} \text{round} > \text{round}1 \) is replaced by \( \text{mes1} \text{round} = \text{func}(\text{mod}, \text{round}1+1, K) \),
  and \( \text{mes1} \text{round} < \text{round}1 \) by \( \text{func}(\text{mod}, \text{mes1} \text{round}+1, K) = \text{round}1 \).
- The command that defines a self-loop when the limit \( K \) is reached is
  omitted.
- The increment of round numbers is computed modulo \( K \), i.e. we use
  \( \text{round}1' = \text{func}(\text{mod}, \text{round}1+1, K) \).
5.2. Algorithm $A_{\text{Hai}}$, round mod K

Hence using round numbers modulo $K$ corresponds to round numbers in $\{0, \ldots, K - 1\}$ for $A_{\text{Hai}}$, round $\leq K$.

For $K = 2$ the specification has to be modified since $\text{mes1 round} = \text{func} \text{mod, round1+1, K}$ iff $\text{func} \text{mod, mes1 round1+1, K} = \text{round1}$. The proof of lemma 1.1 explains that differing round numbers always indicate a newer message. So instead of $\text{mes1 round} = \text{func} \text{mod, round1+1, K}$ we can use $\text{mes1 round} = \text{round1}$, and the use of $\text{func} \text{mod, mes1 round1+1, K} = \text{round1}$ can be omitted.

The statistics for $A_{\text{Hai, round}} \leq K$ have shown that the model size can grow linearly with $K$, the construction time even exponentially. As $A_{\text{Hai, round mod K}}$ is similar to $A_{\text{Hai, round} \leq K}$, we will use the smallest value 2 for $K$.

Since $A_{\text{Hai, round mod K}}$ forbids message overtaking, we use channel simple. As the channel does not check whether the buffer is full, message overwriting is allowed.

We only consider the optimized order of the modules, as they are arranged on the ring. The complete PRISM model specification for $A_{\text{Hai, round mod 2}}$ is given in appendix A.2

5.2.2 Model properties and statistics

For $n = 3$ all $M \in \{2, \ldots, 9\}$ can be constructed, for $n = 4$ only $M \leq 5$. For $n = 5$ $M$ must be two, for $n = 6$ no model can be constructed.

Construction of the model for $A_{\text{Hai, round mod 2}}$ requires about twice the construction time of $A_{\text{Hai, round} \leq 1}$. The number of transitions doubles at most, whereas the amount of states only increases by maximally 50%. The number of nodes in the MTBDDs increase by 20% at most.

Since $A_{\text{Hai, round mod K}}$ does not abort, the oneLeader property changes to

$$P >= 1 \left[ \text{true U ("oneLeader" )} \right].$$

Both leader preservation properties, the empty buffer property, and the termination property are still valid.

For $n = 4$ with $M = 5$ the oneLeader property and the termination property entail error signals. All other instances which allow model construction also successfully check all correctness properties, i.e. the properties oneLeader, empty buffer, termination, the leader preservation invariance, and the second leader preservation.

For $n = 3$ and $n = 4$, all correctness properties require less time than the model construction. For $n = 5$ the correctness properties require at most 3% more time than the construction.
Since
\[ P \left[ A_{\text{Las}, \text{round} \mod K} \text{ terminates within } K \text{ rounds} \right] = \]
\[ P \left[ A_{\text{Las}, \text{round} \leq K} \text{ terminates correctly} \right], \]
the probabilities of correct termination within \( K \) rounds are presented in figure 5.7 and figure 5.10.

As \( A_{\text{Las}, \text{round} \mod K} \) is a Las-Vegas algorithm, expectations are now sensible values. We first compute \( E[\text{restarts}] \), the expected number of restarts before a leader is elected. For this we label the command of node 1 that starts a new round:

\[
[r1] \ (\text{state} \equiv 3) \ & (\text{round} < K) \ & (\text{mes} \equiv \text{hop} = N) \ & (\text{active1}) \\
\ & (!\text{mes} \equiv \text{unique}) \ \rightarrow (\text{state} \equiv 0) \ & (\text{round} \equiv \text{func} (\text{mod}, \ \text{round} + 1, \ K));
\]

Since action labels are used non-orthogonally for both synchronization and rewards, the label \( r1 \) needs to be renamed to \( r2, \ldots r_n \) in the other nodes. Additionally, we define the following reward section:

\[
\text{rewards} \\
[r1] \ (\text{round} = \text{round} + 2) \ & \ldots \ & (\text{round} = \text{round} + n) : 1; \\
\vdots \\
[\text{rn}] \ (\text{round} = \text{round} + 2) \ & \ldots \ & (\text{round} = \text{round} + n) : 1; \\
\text{end rewards}
\]

This gives each transition that increases a round number the first time in a round a reward of one. Both lemma 5.11 and the previous section guarantee that round numbers maximally differ by one, and hence before the first time in a round some node increases its round number, all round numbers must be identical. So in each round exactly one transition has a reward of one.

The maximal expected reward taken over all non-deterministic choices is computed with the formula

\[ R_{\text{max}} = E[\text{"oneLeader"}]. \]

So the value corresponds to the worst case expectation of restarts. We follow the convention of using the worst case when giving probability values and expectations. But also considering the best case, \( R_{\text{min}} = E[\text{"oneLeader"}] \), is helpful to inspect how much better an algorithm can perform for less adverse schedules. E.g. for \( n = 3 \) and \( M = 2 \) the best case expectation of restarts is about 0.6 whereas the worst case is about 1.4. The low value for the best case is not astonishing, as the previous section has shown that the maximal probability, taken over all non-deterministic choices, for leader election within two rounds is always equal to one for enabled message overwriting.

The worst case expected number of rounds is \( E[\text{rounds}] = 1 + E[\text{restarts}] \). The expectations are plotted in figure 5.11 for \( n = 3 \) and \( n = 4 \) over a range of values for \( M \). For \( n = 4 \) and \( M > 4 \) the computations entail error signals.
5.2. Algorithm $A_{Hai, \text{round mod } K}$

For $M \to \infty$ the expected number of rounds approaches one, since the probability of electing a leader within the first round also approaches one. Since each node only generates one message per round, and each message maximally travels once around the ring, the number of steps in each round is bounded. Hence $E[\text{rounds}] < \infty$ implies that the expected runtime of $A_{Hai, \text{round mod } K}$ is also finite.

We get the expected number of message transmissions, $E[\text{transmissions}]$, if we give each transition that sends or receives a message a reward of 0.5. The plot is depicted in figure 5.12 and again similar.

Since there are various schedules of transmissions, some with message overwriting, the precise values cannot be deduced easily as above. For example for $n=3$ and $M=2$, the worst case expected number of transmissions, also called transmission complexity is 14.830, the best case only 5.875. By strengthening the guards of the first commands of the channels to only receive messages when
their buffer is empty, message overwriting is disabled. Then the expected number for transmissions for the worst case as well as the best case is 14.750. For \( n = 4, M = 2 \) without message overwriting the expected number of transmissions is 1% less than that of enabled message overwriting. For all other parameters the expectation is at most 0.5% less. Thus with enabled message overwriting, the worst and best case schedules both actually overwrite messages. Hence although message overwriting usually reduces the number of transmissions as it deletes messages, there are situations where the number is increased. E.g. if a message \( m_1 \) is overwritten by another message \( m_2 \) just before \( m_1 \) finishes its roundtrip in node \( p \), then \( m_2 \) is not purged in \( p \) which would have been the case if \( p \) had received \( m_1 \) and increased its round number. But even though the transmission complexity slightly increases, the expected number of transmissions can be much smaller for less adverse schedules.

If we use channel\_overtaking (cf. page \( \Box \)), the oneLeader property does not hold anymore, for \( K \) set to two or above. Thus \( A_{\text{lati,round mod } K} \) with message overwriting is erroneous and can be used as counterexample for the proof of lemma \( \Box.2 \)

### 5.3 Algorithm \( A_{\text{Fokkink,roundtrip}} \)

#### 5.3.1 Algorithm design and specification

\( A_{\text{lati,round mod } K} \) forbids message overtaking. But then round numbers can be omitted completely, as stated in subsection 5.2.3 Corollary [46] shows that with omitted round numbers messages must not be overwritten. Lemma [46] and lemma [47] show that then passive nodes do not filter. This results in \( A_{\text{Fokkink,roundtrip}} \), given in (Fokkink and Pang, 2004) and in (Fokkink and Pang, 2005). The specification below is from (Fokkink and Pang, 2004), besides renaming of identifiers.

The state machine for \( A_{\text{Fokkink,roundtrip}} \) is almost the same as figure 5.1. It has the two additional states “passive, before sending” and “passive, before receiving” for passive nodes. Hence instead of the transition that changes active to false there is a new transition from state “before deciding” to ”passive, before sending”. Two new transitions between the two new states send and receive messages without filtering, only increasing their hop counters.

Like the state machine, the model specification is also similar to \( A_{\text{lati,round } \leq K} \). The same type of model and constants is defined, except for \( K \), which is only needed when using round numbers:

```markdown
mdp

const int N=3;  //number of nodes n
const int M=3;  //number of IDs
const double Prob = 1/M;
```
5.3. Algorithm $A_{Fokkink,roundtrip}$

Thereafter the first channel is specified. As it does not allow message overwriting, a FIFO buffer is required.

```plaintext
module channel1

Buffer1a_id: [0..M]; Buffer1b_id: [0..M]; Buffer1c_id: [0..M];
Buffer1a_hop: [0..N]; Buffer1b_hop: [0..N]; Buffer1c_hop: [0..N];
Buffer1a_unique: [0..1]; Buffer1b_unique: [0..1];
Buffer1c_unique: [0..1];

[pl_to_c1] (Buffer1a_id=0) -> (Buffer1a_id'=mes1_id) & (Buffer1a_hop'=mes1_hop) & (Buffer1a_unique'=mes1_unique);
[pl_to_c1] (Buffer1a_id>0) & (Buffer1b_id=0) -> (Buffer1b_id'=mes1_id) & (Buffer1b_hop'=mes1_hop) & (Buffer1b_unique'=mes1_unique);
[pl_to_c1] (Buffer1b_id>0) & (Buffer1c_id=0) -> (Buffer1c_id'=mes1_id) & (Buffer1c_hop'=mes1_hop) & (Buffer1c_unique'=mes1_unique);

c1_to_p2] (Buffer1a_id>0) -> (Buffer1a_id'=Buffer1b_id) & (Buffer1a_hop'=Buffer1b_hop) & (Buffer1a_unique'=Buffer1b_unique) & (Buffer1b_id'=Buffer1c_id) & (Buffer1b_hop'=Buffer1c_hop) & (Buffer1b_unique'=Buffer1c_unique) & (Buffer1c_id'=0) & (Buffer1c_hop'=0) & (Buffer1c_unique'=0);
endmodule
```

The local variables define a FIFO buffer of size three. Again an id of zero stands for an empty buffer. Usually the size of the buffer is chosen to be of size $n$. The first set of commands synchronizes with the first node on the action label $pl_{to_c1}$. The commands’ guards guarantee that the channel saves the values of the message from the first node in the first empty slot of the buffer. As long as the buffer is full, the send operation of the first node blocks. The last command synchronizes with the second node on $c1_{to_p2}$. Since the second node always reads the message from the buffer’s first slot, the channel deletes that slot and moves each message one slot backwards, so that messages which first come are first served.

A node has almost the same variables as in $A_{Fokkink,roundtrip}$:

```plaintext
module node1

leader1 : [0..1];
idi1 : [0..M];
state1 : [0..5];
mes1_id : [0..M];
mes1_hop : [0..N];
mes1_unique : [0..1];
```

The variables for round numbers and active1 are no longer in use. leader1 is a new flag. Again the variables with the prefix "mes1" hold incoming and outgoing messages. mes1_hop, mes1_id, and idi1 additionally have the value zero, state1 the value five.
The following commands successively define the node’s behaviour for the different states in the state machine.

Again in state1=0 the node randomly chooses an ID, creates a message, and changes into state1=1:

\[ \text{[] (state1 = 0) \rightarrow} \]
\[ \text{Prob : (state1' = 1) & (id1' = 1) & (mes1_id' = 1) & (mes1_hop' = 1)} \]
\[ \text{& (mes1_unique' = 1) & (leader1' = 0) +} \]
\[ \text{Prob : (state1' = 2) & (id1' = 2) & (mes1_id' = 2) & (mes1_hop' = 1)} \]
\[ \text{& (mes1_unique' = 1) & (leader1' = 0) +} \]
\[ \text{Prob : (state1' = 3) & (id1' = 3) & (mes1_id' = 3) & (mes1_hop' = 1)} \]
\[ \text{& (mes1_unique' = 1) & (leader1' = 0);} \]

In state1=1 the node waits until it can send its message to the first channel by synchronization, then sets the message variables to zero and changes to state1=2:

\[ \text{[p1 to p1] (state1 = 1) \rightarrow (state1' = 2) & (mes1_unique' = 0) &} \]
\[ \text{(mes1_id' = 0) & (mes1_hop' = 0);} \]

The node waits in state1=2 until it can receive a message from the last channel by synchronization. Then the node sets its message variables accordingly and moves into state1=3:

\[ \text{[c3 to p1] (state1 = 2) \rightarrow} \]
\[ \text{(state1' = 3) & (mes1_unique' = Buffer3a_unique) &} \]
\[ \text{(mes1_id' = Buffer3a_id) & (mes1_hop' = Buffer3a_hop);} \]

Depending on the values of the message an active node decides what to do in state1=3:

If mes1_id < id1, the message is purged and the node changes to state1=2:

\[ \text{[\_ (state1 = 3) & (mes1_id < id1) \rightarrow (state1' = 2) &} \]
\[ \text{(mes1_id' = 0) & (mes1_hop' = 0) & (mes1_unique' = 0);} \]

In case mes1_id > id1, the node increases the message’s mes1_hop counter and becomes passive by changing to state1=4:

\[ \text{[\_ (state1 = 3) & (mes1_hop < N) & (mes1_id > id1) \rightarrow} \]
\[ \text{(state1' = 4) & (id1' = 0) & (mes1_hop' = mes1_hop + 1);} \]

If mes1_id = id1, but the message has not yet finished its roundtrip, its mes1_hop counter is increased, its mes1_unique flag is set to false, and the node moves to state1=5:

\[ \text{[\_ (state1 = 3) & (mes1_hop < N) & (mes1_id = id1) \rightarrow} \]
\[ \text{(state1' = 5) & (mes1_unique' = 0) &} \]
\[ \text{(mes1_hop' = mes1_hop + 1);} \]

The state machine shows that state1=1 and state1=5 behave identical, i.e. the command with the guard (state1=5) is the same as the command with the guard (state1=1) from above. So the value five for state1 is not necessary. Nevertheless we use the value to stay loyal to [Pokhinki and Pang, 2004], for the present.
In case the message has completed its roundtrip but is not unique, the node restarts by changing state1 to 0.

\[
[ ] \text{(state1 = 3)} \& \text{(mes1\_hop = N)} \& \text{(mes1\_unique = 0)} \& \\
(\text{mes1\_id = id1}) \rightarrow \text{(state1' = 0)} \& \text{(id1' = 0)} \& \\
(\text{mes1\_id' = 0}) \& \text{(mes1\_hop' = 0)};
\]

If the message finished its roundtrip and is unique, the node becomes the leader by moving to state1=4 and setting the flag leader1:

\[
[ ] \text{(state1 = 3)} \& \text{(mes1\_hop = N)} \& \text{(mes1\_unique = 0)} \& \\
(\text{mes1\_id = id1}) \rightarrow \text{(state1' = 4)} \& \text{(leader1' = 1)} \& \\
(\text{id1' = 0}) \& \text{(mes1\_id' = 0)} \& \text{(mes1\_hop' = 0)};
\]

So in the additional states "passive, before sending" and "passive, before receiving" state1 is set to four and leader1 to zero. If a message is present in the node, it is sent, otherwise a message is received:

\[
[p1\_to\_p1] \text{(state1 = 4)} \& \text{(mes1\_id > 0)} \& \text{(mes1\_hop > 0)} \& \\
(\text{leader1 = 0}) \rightarrow \text{(mes1\_id' = 0)} \& \text{(mes1\_hop' = 0)} \& \\
(\text{mes1\_unique' = 0});
\]

\[
c3\_to\_p1] \text{(state1 = 4)} \& \text{(mes1\_id = 0)} \& \text{(mes1\_hop = 0)} \& \\
(\text{leader1 = 0}) \& \text{(Buffer3a\_hop < N)} \rightarrow \\
(\text{mes1\_id' = Buffer3a\_id}) \& \text{(mes1\_hop' = Buffer3a\_hop + 1)} \& \\
(\text{mes1\_unique' = Buffer3a\_unique});
\]

The leader terminates by making a self-loop:

\[
[\text{done}] \text{(state1 = 4)} \rightarrow \text{true};
\]

endmodule

In the ring of size \(n = 3\) channel2, node2, channel3, and node3 are specified by module renaming the first ones.

### 5.3.2 Model properties and statistics

Model construction is possible for \(n = 3\) with all regarded \(M \in \{2, \ldots, 9\}\), for \(n = 4\) with \(M \leq 6\), for \(n = 5\) only for \(M = 2\). For \(n > 5\) no model can be constructed.

The model specification from [Pookink and Pang, 2004] has the following minor flaws:

- The range of 1, \ldots, \(n\) is sufficient for most message variables, e.g. for Buffer1\_hop, i.e. the value zero is unnecessary.
- The range of state1 can also be reduced, as the value five is redundant.
- Instead of an extra Boolean variable leader1, "passive, before sending" and "passive, before receiving" can be modelled by two new value for the variable state. Then also the value zero becomes redundant for id1 and mes1\_id.
• As message overtaking is not allowed, the flag `unique` can as well be located in the nodes instead of the messages. This decreases traffic, but might increases the number of states, since then all active nodes update their unique flag. In the original variant, there can be much less unique flags if many messages have already been purged.

Some selected experiments have shown that these small improvements only slightly decrease the time and space requirements. This is not astonishing regarding section 4.1. Hence we do not investigate these minor changes. But we will improve the model specification from [Fokkink and Pang, 2001] by listing the modules in the order of their appearance on the ring since this was a strong improvement in section 5.1.2. The complete improved model specification can be found in appendix A.3.

Using the improved model specification, the number of nodes in the MTBDD is between 55% and 85% of the nodes of the specification from (Fokkink and Pang, 2004). Therefore model construction is additionally possible for n=5 with M=3. Time requirements for model construction is either less than 100 seconds or between 20% and 95% of the former time.

Comparing the improved versions of \( A_{\text{Hatip,round \ mod \ 2}} \) and \( A_{\text{Fokkink,roundtrip}} \) show that the latter yields better values: The number of states of the model for \( A_{\text{Fokkink,roundtrip}} \) is about 30% of the corresponding number for \( A_{\text{Hatip,round \ mod \ 2}} \), the number of transitions about 20% to 30%. The size of the MTBDDs for \( A_{\text{Fokkink,roundtrip}} \) is between 60% and 90% of the size for \( A_{\text{Hatip,round \ mod \ 2}} \). There are still 3 terminals in every MTBDD. The construction time is slightly shorter in most cases, but can vary from at least 15% to at most 150% of \( A_{\text{Hatip,round \ mod \ 2}} \)'s construction time.

For checking the correctness properties, the label "oneLeader" is now defined as

\[
\text{state1} = 4 \land \text{state2} = 4 \land \text{state3} = 4 \land \text{leader1 + leader2 + leader3} = 1,
\]

in the case of n=3. The names of the buffer variables in the empty buffer property must be renamed, too.

Fokkink states the property

\[
P > 1 \ [ \text{true U ("oneLeader" \ & \ Buffer1a_id+Buffer2a_id+Buffer3a_id=0) }],
\]

which checks that finally one leader is elected and no more messages are in the channels. But this property allows arbitrary many and changing leaders until all buffers are empty. Hence we prefer the oneLeader property and the empty buffers property.

The oneLeader property and the termination property entail error signals for n=4 with M=6 and for n=5 with M=3. For other instances that allow model construction all correctness properties can successfully be checked.

Checking the oneLeader property requires 30% to 110% for the improved model construction time, except for n=5 with M=2 needing 200%. The leader preservation invariant requires at most 1 second, the empty buffer invariant at most
7 seconds. The second leader preservation property can be verified using at most 10% of the construction time. Termination checks generally need about the same time as model construction, but can vary between 70% and 250%.

As $\mathcal{A}_{\text{Fokkin roundtrip}}$ does not use message overwriting, we compute new probability values for electing a leader. Lacking round numbers, we use the bounded until operator to limit the number of transitions traversable. This results in the following PCTL formula using soft deadlines:

$$\text{Pmin} =? \left[ \text{true} \land U < l \left( \text{"oneLeader"} \right) \right].$$

Again the minimal probabilities over all non-deterministic choices are taken. Using the best case probability instead, i.e. $\text{Pmax}$, returns the same values, so the probabilities are identical for all non-deterministic choices. Figure 5.13 plots the probabilities for $n=3$ and $M \in \{2, \ldots, 9\}$.

![Figure 5.13: Probabilities for $\mathcal{A}_{\text{Fokkin roundtrip}}$ with soft deadlines, $n=3$](image)

The minimal number of steps until a leader might be elected is 18 for $M > 2$. This is the case if three different IDs are chosen in the first round, and the messages not containing the biggest IDs are purged after their first hop. As this situation cannot happen for $M=2$, its probability of leader election is zero for $l \leq 20$.

The horizontal lines between 21 and 35 steps indicate that these sections correspond to finished first rounds and unfinished second rounds. So their values are the probabilities of electing a leader within the first round. The values are identical with the ones in figure 5.21 for $K=0$. This shows that the probability for the worst schedule for the Ital algorithms is the same as the probability of every schedule for $\mathcal{A}_{\text{Fokkin roundtrip}}$, whereas the best schedule for the Ital algorithms always causes a leader election.

Comparing the probabilities of the two figures suggests that the second round for $\mathcal{A}_{\text{Fokkin roundtrip}}$ is finished after 72 steps, the third after 92, and that the probabilities of electing a leader after a certain number of rounds are identical for both algorithms. But the values might be misleading for large $l$, as after a
certain number of steps \( A_{Fokking, roundtrip} \) can be in different rounds depending on which random numbers were chosen.

Like the probabilities, the values of \( E[\text{transmissions}] \) are identical for all schedules. They are also identical, except for a deviation of at most 1e-04%, with the expected number of transmissions of \( A_{Itai, round mod K} \) without message overwriting and without message overtaking. This is obvious because of lemma \(^{[7]}\). Hence the expected transmissions for \( A_{Itai, round mod K} \) with message overwriting are about 0.5% (cf. page \(^{[70]}\)) worse than the expectations for the worst schedule for \( A_{Fokking, roundtrip} \). But \( A_{Itai, round mod K} \) with message overwriting has much better expected transmissions for less adverse schedules. Hence, unlike stated in \(^{[8]}\), the algorithms behave differently, for the worst case, for the best case, and on the average. Since the worst case values only differ slightly, the plot of \( E[\text{transmissions}] \) for \( A_{Fokking, roundtrip} \) looks very similar to figure \(^{[5.12]}\).

If we use chanell\textunderscore simple to allow message overwriting, as in the Itai algorithms, model checking the oneLeader property shows that it does not hold. Hence \( A_{Fokking, roundtrip} \) with message overwriting confirms lemma \(^{[4.3]}\).

### 5.4 Algorithm \( A_{Fokking, no roundtrip} \)

#### 5.4.1 Algorithm design and specification

In \( A_{Fokking, roundtrip} \) a node detecting a collision of IDs does not immediately take action against the collision, but has to wait for its message to finish the roundtrip. In \( A_{Fokking, no roundtrip} \) this disadvantage is removed by immediately exchanging a message causing a collision by a newly created one.

The model specification for \( A_{Fokking, no roundtrip} \) is very similar to the one for \( A_{Fokking, roundtrip} \).

Both node1 and channell no longer use unique flags. When node1 detects in its state "before deciding" that mes1\_id = id1 and that the message has not yet finished its roundtrip, it behaves differently from \( A_{Fokking, roundtrip} \). Instead of editing the message and changing state1 to five, the node removes the message and moves into state1 = 0 to generate a new message.

\[
[] \text{(state1 = 3) \& (mes1_id = id1) \& (mes1_hop < N) } \Rightarrow \text{(state1' = 0) \& (mes1_id' = 0) \& (mes1_hop' = 0) \& (id1' = 0)};
\]

So the value five for state1 is unnecessary and the command with the guard (state1 = 5) is removed.

Since collisions are resolved immediately, it can no longer happen that a message completes its roundtrip but is not unique. So the corresponding command is also deleted.
We immediately use the improved order of the modules, as they are arranged on the ring. The complete PRISM specification for $n=3$ is given in appendix A.4.

5.4.2 Model properties and statistics

For $n=3$ all regarded $M \in \{2, \ldots, 9\}$ can be constructed. For $n=5$ $M \leq 3$ is possible, as for $A_{Fokkink, roundtrip}$. But now for $n=4$ only $M \leq 5$ is possible, and for $n=6$ with $M=2$ the model can be constructed. For $n \geq 7$ no model construction is possible.

Compared to $A_{Fokkink, roundtrip}$, the construction time of $A_{Fokkink, no roundtrip}$ is at least 20% and at most the same time. The number of states and transitions both vary between 90% and 310%. The amount of nodes in the MTBDD is usually about the same as for $A_{Fokkink, roundtrip}$, but can be as little as 60% and as much as 150%.

Model checking of the termination property and the oneLeader property entail error signals for $n=6$ (with $M=2$) and for $n=5$ with $M=3$, the latter property also for $n=4$ with $M=5$. All other instances that allow model construction can successfully be verified.

The leader preservation invariant is checked within one second, the empty buffer property within seven seconds. The second leader preservation property needs at most 15% of the construction time. The time requirements for checking the property of termination or oneLeader are at most four times the model construction time.

Figure 5.14 shows the probabilities for electing a leader within soft deadlines for $A_{Fokkink, no roundtrip}$ with $n=3$ and $M \in \{2, \ldots, 9\}$. Again the values are identical for all schedules.

![Figure 5.14: Probabilities for $A_{Fokkink, no roundtrip}$ with soft deadlines, $n=3$](image)
Again the minimal number of steps until a leader might be elected is 18 for \( M > 2 \), since both algorithms do not differ in the optimal case when all IDs are unique. For \( M=2 \) this situation cannot happen, and the probability of leader election is zero for \( l \leq 21 \), which is one step longer than for \( A_{Fokkink, roundtrip} \). The extra step is for creating a new message. Because of additional steps, the probabilities are smaller for \( A_{Fokkink, no roundtrip} \) until \( l \approx 30 - 2 \cdot M \) steps. Thereafter the overhead of creating new messages to avoid redundant transmissions amortizes, i.e. then \( A_{Fokkink, no roundtrip} \) approaches faster the probability one than \( A_{Fokkink, roundtrip} \).

The expected number of transmissions for \( n \in \{3, 4\} \) are plotted in figure 5.15. As \( A_{Fokkink, no roundtrip} \) avoids redundant transmissions, \( E[\text{transmissions}] \) is not as big as for \( A_{Fokkink, roundtrip} \) or \( A_{lat, roundmod K} \). For \( M \to \infty \) the expectations approach the same value as for \( A_{Fokkink, roundtrip} \), since the probability of colliding IDs approaches zero, and both algorithms only differ when IDs collide.

![Figure 5.15: Expected number of transmissions for \( A_{Fokkink, no roundtrip} \)](image)

If we allow message overwriting, i.e. use channel simple, the oneLeader property does not hold. Thus \( A_{Fokkink, no roundtrip} \) with message overwriting can be used as counterexample for the proof of lemma 4.4.

### 5.5 Algorithm \( A_{\text{sorting}} \)

#### 5.5.1 Algorithm design and specification

As mentioned in subsection 5.2.2, \( A_{\text{sorting}} \) does not compare IDs to turn nodes passive. Instead each node saves all information of a round and then uses a decision procedure.

Besides improving leader election by utilizing all information crossing a node, this section also tries to improve the performance of model checking. Therefore
only IDs of one bit are used, i.e. $M=2$, to reduce the size of the state space. As $n$ is implicitly given by the number of local variables, no constant needs to be defined.

\textbf{module} node1

\begin{verbatim}
p1_var1 : [0..2]; //=p1_id, 0 = empty buffer
p1_var2 : [0..2]; //=p3_id, because of clockwise transmis.
p1_var3 : [0..2]; //=p2_id

state1 : [0..2];

[] (state1=0) -> 0.5: (p1_var1'=1) & (state1'=1) +
    0.5: (p1_var1'=2) & (state1'=1);

[] (state1=1) & (p1_var1'=0) & (p1_var2=0) & (p3_var1'=0) ->
    (p1_var2'=p3_var1);
[] (state1=1) & (p1_var2'=0) & (p1_var3=0) & (p3_var2'=0) ->
    (p1_var3'=p3_var2);
\end{verbatim}

As additional improvement of model checking, the channels are further abstracted and integrated into the nodes (cf. subsubsection \ref{sec:abstract}). Thus the nodes read from each others buffers, and the size of a message is solely one bit. Consequently no send operation and no state "before sending" is necessary, and reading is done in state1=1, which corresponds to "before receiving". Of course only values the preceding node has already received can be read:

\begin{verbatim}
[] (state1=1) & (p1_var1'=0) & (p1_var2'=0) & (p3_var1'=0) & (p3_var2'=0) ->
    (p1_var1'=p1_var2) & (p1_var1'=p1_var3) & (state1'=0) & (p1_var1'=0) & (p1_var2'=0) & (p1_var3'=0);
\end{verbatim}

Another abstraction, already mentioned in subsubsection \ref{sec:abstract}, is from a concrete decision procedure, e.g. bitshifting and sorting, to an abstract algorithm that solely checks whether the missing decision procedure would compute a unique leader. So in state1=1 with $p1\_var3\neq0$, which corresponds to the state "before deciding", a node checks if its bit array is periodic. If so, no unique largest bit array exists, and no leader is elected, but a new round is started. Otherwise node1 moves into state1=2, indicating that there exists a unique leader:

\begin{verbatim}
[] (state1=1) & (p1_var3'=0) & (state1'=1) & (p1_var3'=0) & (p1_var1'=p1_var2) & (p1_var1'=p1_var3) ->
    (state1'=0) & (p1_var1'=0) & (p1_var2'=0) & (p1_var3'=0);
\end{verbatim}

Deducing periodicity by checking whether all IDs are the same is only correct for prime $n$. For other values alternative criteria must be chosen. So the model specification is very dependent on $n$.

As usual, termination is modelled by a self-loop:

\begin{verbatim}
[] (state1=2) -> true;
\end{verbatim}

\textbf{endmodule}
For the complete PRISM specification, which can be found in appendix A.3 the second and third node must be specified by renaming the first module.

The label restart is not renamed in the second and third node. Thus all nodes synchronize when starting a new round by resetting their local variables. As a result, no node reads an old value from a previous round. But consequently the degree of parallelization is reduced. A global synchronization in a distributed algorithm is unpracticable, too. As an alternative to this globally synchronizing algorithm $\mathcal{A}_{\text{sorting, sync}}$, the algorithm $\mathcal{A}_{\text{sorting, a sync}}$ is globally asynchronous and only synchronizes locally. To avoid global synchronization, the label restart is removed. node1 has an extra local flag $p_{1\text{read.all}}$, which is set if all values from node3, the node preceding node1, of a round are read. Only then node3 can start a new round and reset its local variables. This reset synchronizes with node1 on the action label $p_{3\text{made.reset}}$, so that node1 recognizes node3’s new round and sets $p_{1\text{read.all}}$ to false:

$$p_{1\text{read.all}} : \textbf{bool} \textbf{ init} \textbf{ true};$$

$$[p_{1\text{made.reset}] \ (\text{state1}=0) \rightarrow \ 0.5:(p_{1\text{var1}}'=1) \ & \ (\text{state1}'=1) \ + \ 0.5:(p_{1\text{var1}}'=2) \ & \ (\text{state1}'=1);$$

$$[p_{3\text{made.reset}] \ \textbf{true} \rightarrow p_{1\text{read.all}}'='\text{false};$$

$$[\] \ (\text{state1}=1) \ & \ (p_{1\text{var3}}'=0) \ & \ (p_{2\text{read.all}}) \ & \ (p_{1\text{var1}=p_{1\text{var2}}}) \ & \ (p_{1\text{var1}=p_{1\text{var3}}}) \ \rightarrow \ (\text{state1}'=0) \ & \ (p_{1\text{var1}}'=0) \ & \ (p_{1\text{var2}}'=0) \ & \ (p_{1\text{var3}}'=0);$$

Only if the flag $p_{1\text{read.all}}$ is not set, node1 can read a new value from node3. After reading the last value, node1 sets the flag:

$$[\] \ (\text{state1}=1) \ & \ (p_{1\text{var1}}'=0) \ & \ (p_{1\text{var2}}'=0) \ & \ (p_{3\text{var1}}'=0) \ & \ (!p_{1\text{read.all}}) \ \rightarrow \ (p_{1\text{var2}}'=p_{3\text{var1}});$$

$$[\] \ (\text{state1}=1) \ & \ (p_{1\text{var2}}'=0) \ & \ (p_{1\text{var3}}'=0) \ & \ (p_{3\text{var2}}'=0) \ & \ (p_{3\text{var1}}'=0) \ \rightarrow \ (p_{1\text{var3}}'=p_{3\text{var2}}) \ & \ (p_{1\text{read.all}}'='\text{true});$$

The rest of the model specification is not changed, except for additionally renaming the flag and the action label for the other nodes. The complete model specification for $\mathcal{A}_{\text{sorting, a sync}}$ can also be found in appendix A.3

Note. The Boolean flags $p_{1\text{read.all}}$ have similarities to round numbers modulo two, since both enable node1 to differ between new and outdated values, and in both cases round numbers maximally differ by one.

5.5.2 Model properties and statistics

The models for $\mathcal{A}_{\text{sorting}}$ can successfully be built for $n$ up to seven. For $\mathcal{A}_{\text{sorting, sync}}$ the number of states and nodes of the models are plotted in figure 5.11. As expected, these values are exponential in $n$. The number of transitions is between two and four times the number of nodes. Model construction always requires less than 300 seconds.
5.5. Algorithm $A_{\text{sorting}}$

For $A_{\text{sorting,async}}$, the number of states, transition, and nodes is between 120% and 240% of $A_{\text{sorting,sync}}$. The time for construction the model is between 130% and 200%, but no more than 440 seconds.

Since the bit arrays that the nodes save in their local variables are circular shifts of one another, the variables which mostly depend on each other are spread over all modules. So the compression ratio of the MTBDD might be increased by reordering the variables. Since this facility is not yet implemented in PRISM, the only possibility is altering the variables from local to global. Sadly this counterbalances the improvements made by reordering.

We cannot check the correctness of a concrete leader election algorithm because $A_{\text{sorting}}$ is abstracted from a specific decision procedure. Thus the verification can solely prove that there exists a correct leader election algorithm for the given protocol, i.e. that $A_{\text{sorting}}$ could correctly elect a leader if it contained a correct decision procedure. Consequently we can no longer use the oneLeader property, since it states that exactly one leader is elected. We have to use the weaker property of leader existence, which only formulates that all nodes have decided that there exists a leader:

$$P \geq 1 \left\{ \text{true U "allState2"} \right\}.$$  

In the case $n = 3$, the label "allState2" is defined as

$$(\text{state1} = 2 \& \text{state2} = 2 \& \text{state3} = 2).$$

Additionally, a necessary condition for a correct leader election can be checked: the circular shifts property, stating that the bit arrays of two adjacent nodes are circularly shifted by one bit. As we have a symmetric ring, the formula

$$\text{"allState2"} \rightarrow (p_1 \text{var1} = p_2 \text{var2} \& p_1 \text{var1} = p_3 \text{var3})$$

is sufficient.
If a correct decision procedure is used, then the algorithm is correct iff the leader existence property and the circular shifts property hold.

A necessary condition for a correctly working decision procedure is the aperiodicity of the bit arrays the moment a leader exists. So the aperiodicity property

\[ (\text{allState2}) \Rightarrow (p1_{\text{var1}} = p1_{\text{var2}} \mid p1_{\text{var1}} = p1_{\text{var3}}), \]

at least detects some cases of incorrect decision procedures. The propositional formula for aperiodicity of the bit array has to be modified for other values of n than three.

Lacking an elected leader, the leader preservation property has to be reduced to the allState2 preservation property, which only checks whether a node in state two stays there. Because of the ring's symmetry, the formula

\[ (\text{state1 = 2}) \Rightarrow P \leq 0 \text{ true U (state1 = 2)} \]

is sufficient.

The termination property can still be used, but it only checks whether the abstract algorithm works correctly up to the point where all nodes are in state two. Having no channels with buffers, the empty buffer property is no longer needed.

For n=7 the termination property yields an error signal for A\text{sorting, sync}, and the leader existence property for A\text{sorting, async}. Apart from those verifications, all properties can successfully be checked for n \in \{3, \ldots , 7\}.

The maximal times required to check the leader existence property are 250 seconds for A\text{sorting, sync} and 100 seconds for A\text{sorting, async}. The termination property requires 150 seconds for A\text{sorting, sync} and 720 seconds for A\text{sorting, async}. The times are for the respective largest possible instance. Circular shifts and allState2 preservation always need at most 30 seconds, aperiodicity at most 2 seconds.

As the required number of steps of a round and its outcome are independent of the schedule for A\text{sorting}, the worst case and best case probabilities for electing exactly one leader with soft deadlines are identical. Figure 5.17 displays the probability values for A\text{sorting, sync} with n \in \{3, \ldots , 6\}.

Since the plot for n is zero up to n^2 + n - 1, this is the required number of steps for a successful round. As the next stair appears after 2 \cdot n^2 + n steps and corresponds to one unsuccessful and one successful round, the number of steps for an unsuccessful round is n^2 + 1. The plot for A\text{sorting, async} looks very similar, except that an unsuccessful round requires n^2 + n steps, i.e. n - 1 steps more than A\text{sorting, sync}. The reason is A\text{sorting, sync}'s label restart, which causes the restart of all nodes in a single step, instead of n steps as for A\text{sorting, async}. Both A\text{sorting, sync} and A\text{sorting, async} have better probability values than the inspected algorithms in A\text{compare I_Ds} with M = 2. This is still the case when increasing M up to 9, except for the ends of the rounds in the A\text{sorting} algorithms, at which points A\text{Fokkink no roundtrip} performs slightly better.
5.5. Algorithm $A_{sorting}$

Since all nodes in $A_{sorting, sync}$ restart synchronously, the expected number of restarts can be computed by specifying

```python
rewards
[restart] true : 1;
endrewards
```

As the outcome of a round is independent of the schedule, the best case and worst case values are identical.

The expected number of rounds is $\mathbb{E}[\text{rounds}] = 1 + \mathbb{E}[\text{restarts}]$ and plotted in figure 5.18. Besides requiring a few steps more in each round and possibly slightly blending successive unsuccessful rounds, $A_{sorting, async}$ behaves just like $A_{sorting, sync}$. Thus the expectations are identical.

![Figure 5.17: Probabilities for $A_{sorting, sync}$ with soft deadlines](image)

![Figure 5.18: Expected number of rounds for $A_{sorting}$](image)

The figure depicts that the best expectations are achieved for $n$ prime.

The expected number of transmissions is $\mathbb{E}[\text{transmissions}] = \mathbb{E}[\text{rounds}] \cdot n^2$ messages of one bit each.
**Note.** A possibility to reduce traffic is that only the random ID one generates messages, whereas the random ID zero is discarded. Then each message needs a hop counter, requiring $O(\log(n))$ bits. The ID needs not be contained in the message anymore. By reducing the probability of choosing an ID one to $1/n$, the expected number of transmitted bits in each round can be reduced to $O(n \log(n))$. Of course this increases the expected number of rounds a little, but does not counterbalance the savings.

### 5.6 Algorithm $A_{\text{timing}}$

#### 5.6.1 Algorithm design and specification

Although $A_{\text{sorting}}$ uses several abstractions, the performance and limits of model checking the algorithm are rather disappointing. Therefore we now investigate algorithm $A_{\text{timing}}$ with the contrary concept of being greedy with memory instead of saving all information. To save memory, a node of $A_{\text{timing}}$ imposes asymmetry onto the ring by waiting for a random period of time. The exact behaviour is shown in the state machine given in figure 5.19. As in $A_{\text{sorting}}$, no channels are used in $A_{\text{timing}}$. But since the processed messages have to be discarded, the nodes must explicitly send the messages to their successive nodes.

![State machine for $A_{\text{timing}}$](image)

Figure 5.19: State machine for $A_{\text{timing}}$

Before specifying the modules, the type of model, the constant $n$ and some PRISM formulae (cf. subsection 2.3.1) are defined:

```plaintext
mdp
const int N=3; //number of nodes n
cost double Prob = 1/N;
formula preceding_hop_is_0 = mes3_hop=0;
```
5.6. Algorithm \( A_{\text{timing}} \)

\[
\begin{align*}
\text{formula} \quad \text{preceding-hop-is-N} & = \text{mes3-hop=N}; \\
\text{formula} \quad \text{preceding-hop-plus-1} & = \text{mes3-hop+1}; \\
\text{formula} \quad \text{preceding-unique} & = \text{mes3-unique};
\end{align*}
\]

Since \( A_{\text{timing}} \) is verifiable up to \( n = 10 \), as shown in the next subsection, the need to explicitly hard-code the network structure for each \( n \) becomes more annoying. Thus these PRISM formulae are introduced to eliminate the need to modify the specification of the first node when changing \( n \). So only the formulae and the module renamings have to be modified with \( n \).

As before we use the variable state1 to specify the first node’s behaviour: The states ”before creating”, ”passive, before sending”, ”passive, before receiving”, ”active, before sending”, ”active, before receiving and deciding”, and the final state of the state machine correspond to the values 0, 1, 2, 3, 4, and 5, respectively.

As message consists of a hop counter and of a unique bit, the variables of the first node are:

\[
\begin{align*}
\text{module node1} \\
\text{state1} : [0..5]; \\
\text{mes1-hop} : [0..N]; & \quad //0=empty\ buffer \\
\text{mes1-unique} : \text{bool};
\end{align*}
\]

The following commands successively define the node’s behaviour for the different values of state1.

In state1=0 there might be an incoming message. Then the node receives it and turns passive. Otherwise the node waits with probability 1-Prob, and with probability Prob it creates a message as leader request and moves to state1=3.

To make the code independent of \( n \) the action label toP1 is used instead of P3toP1:

\[
\begin{align*}
\lbrack \text{toP1} \rbrack & (\text{state1}=0) \land (\neg \text{preceding-hop-is-N}) \rightarrow \\
& (\text{state1}=1) \land (\text{mes1-hop}=\text{preceding-hop-plus-1}) \land \\
& (\text{mes1-unique}=\text{preceding-unique}); \\
\lbrack \square \rbrack & (\text{state1}=0) \land (\text{preceding-hop-is-0}) \rightarrow \\
& 1-\text{Prob} : \text{true} + \\
& \text{Prob} : (\text{mes1-hop}=1) \land (\text{mes1-unique}=\text{true}) \land (\text{state1}=3);
\end{align*}
\]

If the node is active and waiting for its leader request, it forwards requests from other nodes, increasing their hop counters and unsetting their unique flag. When the node receives its own request, it either returns to state1=0 if the message’s unique flag is false, or becomes the leader if the request is unique:

\[
\begin{align*}
\lbrack \text{toP2} \rbrack & (\text{state1}=3) \rightarrow (\text{state1}=4) \land (\text{mes1-hop}=0); \\
\lbrack \text{toP1} \rbrack & (\text{state1}=4) \land (\neg \text{preceding-hop-is-N}) \rightarrow (\text{state1}=3) \land \\
& (\text{mes1-hop}=\text{preceding-hop-plus-1}) \land (\text{mes1-unique}=\text{false}); \\
\lbrack \text{toP1} \rbrack & (\text{state1}=4) \land (\text{preceding-hop-is-N}) \land \\
& (\neg \text{preceding-unique}) \rightarrow (\text{state1}=0);
\end{align*}
\]
\[ \text{[toP1]} \ (\text{state1}=4) \land (\text{preceding\_hop}\_is\_N) \land \\
(\text{preceding\_unique}) \rightarrow (\text{state1}^\prime=5); \]

If the node is passive, it solely forwards messages and increases their hop counters:
\[ \text{[toP2]} \ (\text{state1}=1) \rightarrow (\text{state1}^\prime=2) \land (\text{mes1\_hop}^\prime=0); \]
\[ \text{[toP1]} \ (\text{state1}=2) \land (!\text{preceding\_hop}\_is\_N) \rightarrow (\text{state1}^\prime=1) \land \\
(\text{mes1\_hop}^\prime=\text{preceding\_hop}\_plus\_1) \land \\
(\text{mes1\_unique}^\prime=\text{preceding\_unique}); \]

The leader terminates by making a self-loop:
\[ [] \ (\text{state1}=5) \rightarrow \text{true}; \]
endmodule

The modules node2 and node3 are specified by module renaming the first ones, as usual. Hence this part of the specification is still dependent on \( n \). The complete PRISM model specification for \( A_{\text{Timing}} \) with \( n=3 \) can be found in appendix A.6.

The corresponding MDP is depicted in figure 5.25.

### 5.6.2 Model properties and statistics

The maximal constructable instance for \( A_{\text{Timing}} \) is \( n=10 \), which is three more than the previous highest limit, for \( A_{\text{Sorting}} \). Constructing the largest instance requires 350 seconds, for \( n <10 \) less than 100 seconds. Figure 5.20 shows the number of states and nodes of the constructable models. As expected the values are exponential in \( n \). The number of transitions varies between two and four times the number of states.

![Figure 5.20: States and nodes for \( A_{\text{Timing}}, n=3 \)](image)

We have modelled breaking symmetry by randomly waiting for some time, so that another node has the possibility to meanwhile request being the leader.
Of course this request corresponds to making some transitions in the MDP, so we have an entanglement of randomness and non-determinism, i.e. scheduling. If a node randomly waits long enough, another node has to make progress in requesting to be the leader. This corresponds to demanding fairness. So if fairness is not enabled, a node can randomly wait arbitrarily long without any other node doing anything. Consequently the oneLeader property does not hold without enabled fairness. Figure 2-4 which depicts the MDP for A_timing with n=3, contains three schedules for which the oneLeader property and the termination property is not satisfied. An example is the leftmost cycle with the unfair scheduling that always chooses the transitions labelled with zero. Hence in the states 18 and 25 one node can wait arbitrarily long without any other node taking any action. Hence A_timing traverses the cycle infinitely often without any leader being elected.

With fairness enabled, these unfair behaviours are avoided and the oneLeader property holds for all instances. The verification time is at most 500 seconds. The termination property also holds and requires no more than 530 seconds.

Weakening the oneLeader property by using the precondition of termination we get the property of correctly terminating paths,

"deadlock" => "oneLeader",

which checks whether the algorithm elected a unique leader upon termination, i.e. whether we have a partially correct randomized algorithm. The property holds for all verifications, even with disabled fairness. The check needs at most one second.

All other correctness checks are also successful without fairness, and require no more than 16 seconds.

**Note.** An alternative to enabling fairness is switching from an MDP to a DTMC, if precise probability values p ∈ (0, 1) are not needed, as in the correctness checks (cf. section 5.1). Because of the normalization when changing from an MDP to a DTMC, the unfair infinite paths have probability zero, as they make the same choice with probability p < 1 again and again in some state. All correctness properties can be checked successfully for n ≤ 10, and the verification time can be reduced to at most 450 seconds. So every unfair path has probability zero. This yields a further variant of verification when precise probability values p ∈ (0, 1) are irrelevant: using solely non-determinism, and no probability, and activating fairness for properties with P >= 1[...]. If the verification run is successful, then the properties are really true, since only paths with probability zero are excluded by the fairness constraint. This kind of verification is also successful for A_timing, though unnecessary when using the model checker PRISM. The next chapter investigates this kind of verification thoroughly.

For the exemplary unfair schedule from above, which always selects a transition labelled with zero, the probability of electing a leader is even zero. Hence P[min=? [ true U ("oneLeader") ]] is also zero for disabled fairness. Since finite
paths are always fair, \( P_{\text{min}} = 0 \) [true \( U \leq l \) ("oneLeader")], is always zero even if fairness is activated. Thus we switch from an MDP to a DTMC and compute \( P_e = ? \) [true \( U \leq l \) ("oneLeader")]. This imposes a Laplace distribution on the set of all schedules, and computes the average probability of electing exactly one leader with soft deadlines. These values are plotted in figure 5.21 for \( n \in \{3, \ldots, 10\} \).

![Figure 5.21: Average probabilities for \( A_{\text{timing}} \) with soft deadlines](image)

The figures 5.13, 5.14, and 5.17 have the same probabilities for all schedules. Hence for these figures the worst case probabilities and the average case probabilities are identical, and can be compared to figure 5.21 \( A_{\text{timing}} \) usually performs even better than \( A_{\text{sorting}} \) for average probability values. For \( n \in \{3, 4, 6\} \) the exceptions are the points when \( A_{\text{sorting, sync}} \) starts a new round. For \( n = 5 \) the exceptions are all points but the end of rounds of \( A_{\text{sorting, sync}} \).

But this superiority is only valid on average, whereas \( A_{\text{sorting}} \) guarantees having the given probability values. On the other hand, as some unfair schedules with probabilities constantly zero pull down the average probabilities for \( A_{\text{timing}} \), there are also schedules for which \( A_{\text{timing}} \) performs significantly better than the average. The best case probabilities for \( n \leq 4 \) have higher probabilities than the plot for \( n=3 \) in figure 5.21 the best case probabilities for \( n \leq 9 \) higher values than the plot for \( n=4 \) after 15 steps.

Since the exemplary unfair schedule mentioned above never terminates and sends messages again and again, the worst case expected number of transmissions is infinite. Thus we use the average expectation of transmissions, shown in figure 5.22.

The expected number of transmissions is almost linear. As the number of steps is on average proportional to the expected number of transmissions, the average runtime of \( A_{\text{timing}} \) is also almost linear.

**Note.** Waiting for a random period of time has to be modeled in PRISM by traversing a self-loop a random number of times whenever the node is scheduled.
This causes an entanglement of randomness and non-determinism. Time can semantically precisely only be modelled using timed automata. With big help from Frank Werner, \( A_{\text{timing}} \) has been specified and verified in UPPAAL, a model checker for timed automata (cf. \cite{Uppaal,2007}). As UPPAAL only offers a very restricted subset of CTL, only few correctness properties can be checked, e.g. the leader preservation invariance. The maximal instance that can be constructed without memory overflow is \( n=10 \), for various values of time periods a node waits. The time required to construct the model is about 300 seconds.

5.7 Conclusion

Among the investigated algorithms from \( A_{\text{compare \textit{1D}s}} \), \( A_{\text{Fokkink, no \textit{roundtrip}}} \) has the highest worst case probability values for electing a leader within a given number of steps. Since the algorithm cannot use message overwriting, its best case probability is identical with the worst case.

For \( A_{\text{fai, round \mod K}} \), message overwriting can be enabled, which only slightly decreases its worst case, but significantly improves the best case. Consequently \( A_{\text{fai, round \mod K}} \) with message overwriting performs better than all other algorithms for the best case.

If the network allows message overtaking, \( A_{\text{fai, round \leq K}} \) must be taken. The worst case probability values for \( A_{\text{sorting}} \) are better than those for algorithms in \( A_{\text{compare \textit{1D}s}} \) and works especially well for prime \( n \).

\( A_{\text{timing}} \) has the best average probability, and very high best case probabilities, but for some unfair schedules the probability is zero.

Comparing the expected number of transmissions for all algorithms is misleading, since the size of a message can vary between algorithms, and \( A_{\text{timing}} \) only has informative average expectations. Hence table 5.2 shows the average communication in bits, for the instances constructable for all algorithms, i.e. \( n \in \{3, 4, 5\} \), and \( M=2 \).
So $A_{\text{sorting}}$ has the lowest average communication for the regarded instances, followed by $A_{\text{timing}}$. But the asymptotical behaviour can be different, e.g. $o(n^2)$ bits for the variant of $A_{\text{sorting}}$ which transmits both values zero and one.

The model checking statistics show that the better algorithms, $A_{\text{sorting}}$ and $A_{\text{timing}}$, also have reduced space and time requirements. Whereas some instances of the algorithms from $A_{\text{compare IDs}}$ had impracticable high time requirements for model checking, the improved algorithms only failed because of memory overflow, which is favourable because of the faster notification.

Model checking with PRISM is also very handy to explore properties of the model: Several plots and experiments with probability values and expectations give insight of the algorithms. Also some lemmata from chapter 11 can be verified. So PRISM can be used successfully, except for the disappointingly strong limitation of constructable instances, usually $n < 8$. Consequently our results may not be valid in general, i.e. for arbitrarily big $n$. For example the average communication of $A_{\text{sorting}}$, as mentioned above.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$A_{\text{fast round med}}$</th>
<th>$A_{\text{Fok roundtrip}}$</th>
<th>$A_{\text{Fok no roundtrip}}$</th>
<th>$A_{\text{sorting}}$</th>
<th>$A_{\text{timing}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>54</td>
<td>60</td>
<td>35</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>100</td>
<td>58</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>160</td>
<td>189</td>
<td>112</td>
<td>27</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 5.2: Average communication in bits
Chapter 6

Model checking with SPIN

The limited constructability of problem instances in the last chapter motivate testing an alternative to PRISM, model checking the promising algorithms with it, and comparing the obtained statistics with the ones from PRISM. SPIN (version 4.2.6) seems to be a good choice, because it is the most popular, and one of the most powerful model checker. Unlike PRISM, SPIN is not a probabilistic model checker, so we need to use non-determinism instead of randomness, as mentioned in section 5.9 while looking into the fairness constraint of PRISM.

The difficulty which arises from only using non-determinism and omitting randomness, and how to cope with it will be explained in section 6.1. Section 6.2 will then model check the various algorithms. The chapter finishes with a conclusion.

6.1 Replacing randomness by non-determinism

In this chapter we do not want to calculate any probabilistic values for a randomized leader election algorithm, like the expected number of rounds. We only want to prove the algorithm’s correctness, i.e. that for every fair schedule it terminates with probability one and upon termination there is exactly one leader. So we do not care about the precise probability of a path, i.e. the probability of the algorithm choosing this path, since the correctness property needs to be true in any case. Hence we can use non-determinism instead of randomness in the SPIN model specification, and check the LTL formula $\text{Leaders} = 1$ on all paths. As we will see in section 6.2 this simple approach does not work. The reason is that infinite paths with probability zero are not excluded from the verification when using non-determinism instead of randomness. So the error lies in the above statement that probabilities do not matter. Of course the correct version says that precise probabilities $0 \leq p \leq 1$ are irrelevant, but we need to distinguish events with the probability $p = 0$ from those with probability $p \neq 0$, even if the probability 0 emerged from taking a limit, as for the path $(\text{init} \rightarrow \text{init})^\omega$ in figure 6.1. This distinction turns out to
be rather difficult without the ability to use explicit randomness. So, in short, we are not ignoring probabilities, but rather are abstracting from probabilities to possibilities, i.e. from quantitative to qualitative inspection.

![Figure 6.1: A path with probability zero](image)

In section 6.1.1, we will try to use fairness constraints to exclude infinite paths having probability zero.

Section 6.1.2 considers temporal logic to prove the correctness of an algorithm. In section 6.1.3, we are weakening the correctness proof of an algorithm by regarding it as a Las Vegas algorithm. This means we only have to look at finite paths in the computation tree. Section 6.1.4 shows different ways of proving that an algorithm terminates with probability one.

### 6.1.1 Fairness constraints

In section 6.6, we are able to use the fairness constraint provided by PRISM to verify the correctness of \( A_{\text{timing}} \) using solely non-determinism and no probability. In this subsection, we will investigate the possibility to use SPIN’s weak fairness constraint to exclude infinite paths having probability zero.

PRISM’s fairness is defined in (Rutten et al., 2004, page 130) as follows:

**Definition 6.1 (PRISM’s fairness).** A path \( \pi \) in a computation tree is fair if for all states occurring infinitely often on \( \pi \), each non-deterministic choice is taken infinitely often.

Using PRISM’s fairness results in the **fair computation tree**, which restricts the original computation tree to only fair paths.

Let \( A \) be a correct algorithm with solely non-determinism.

**Proposition 6.2.** PRISM’s fairness excludes infinite paths of \( A \) having probability zero from the computation tree.

**Proof.** For every reachable state the probability is one that \( A \) terminates with exactly one leader. Let us assume there exists an infinite path \( \pi \) in the fair computation tree of \( A \). As the number of states in the model is finite, there must be a state \( s_0 \) which appears infinitely often on \( \pi \). Because of the correctness of \( A \), there exists a path \( \pi' = (s_0, s_1, \ldots, s_n) \) from \( s_0 \) to an end state \( s_n \) with \( \text{nr Leaders} = 1 \), as depicted in figure 6.2. Since \( \pi \) cycles through \( s_0 \) infinitely often, \( \pi \) must also infinitely often take the same transition out of \( s_0 \) as \( \pi' \) does because \( \pi \) is fair. So \( \pi \) cycles through \( s_1 \) infinitely often. By induction on \( i \in \{0, \ldots, n\} \) every state \( s_i \) occurs infinitely often on \( \pi \), which contradicts the fact that \( s_n \) is an end state.

\( \square \)
So if an algorithm $A$ terminates with probability one, its fair computation tree is finite. Otherwise there is an infinite path $\pi$ in the fair computation tree and a state $s_0$ infinitely often on $\pi$. Applying the induction from proposition 6.2 on every path $\pi' := \langle s_0, s_1, \ldots, s_n \rangle$ in the graph of $A$ starting from $s_0$ (and having no repeating states) shows that $\pi$ traverses the complete subgraph that is reachable from $s_0$, $S$, infinitely many times. Thus $S$ is a strongly connected component without any end states, i.e. $\pi$ is trapped in $S$.

![Diagram of computation tree and automaton](image)

Figure 6.2: Prism’s fairness excludes infinite paths having probability zero

SPIN supports a different constraint called weak fairness, cf. [Holzmann, 2004b] or [Gerth, 1997]:

**Definition 6.3.** A process is **starving** iff it makes infinitely many requests but never gets scheduled.

Weak fairness forbids process starvation if the process remains continuously enabled. Strong fairness, which is not provided by SPIN, forbids any process starvation.

So both of these versions of fairness only impose restrictions on process scheduling decisions, not on general non-determinism decisions. Consequently they do not exclude all paths with probability zero, for example if all nodes always choose zero as their ID.

[Holzmann, 2004b] page 141) suggests using the LTL formula $\phi \land G$ to enforce a fairness constraint on local non-determinism, where $G := (\Box \Diamond \text{Process}_i \Diamond \text{Label}_k)$ → $(\Box \Diamond \text{Process}_i \Diamond \text{Label}_k)$ and $\phi$ is the original LTL correctness claim. $G$ is true iff Process $i$ visiting a state labelled Label $k$ infinitely often implies that it also visits a state labelled Label $k$ infinitely often. So $\phi \land G$ is true on all paths only if the model already fulfills the fairness claim $G$. To add $G$ as a fairness constraint on the computation tree, we should rather check the LTL formula $\phi \lor \neg G$: Paths which do not meet the fairness $G$ will already make $\neg G$ true, so they do not have to be considered for $\phi$ anymore.

But this is still not enough to solve our problem of excluding infinite paths, because using $\phi \lor \neg G$, we get a fairness that solely limits the non-deterministic choices made within one process. Hence this constraint is only local to processes,
i.e. it does not regard the whole system state. For example, if all nodes choose ID zero in even rounds and ID one in odd rounds, we get an infinite path with probability zero which is still considered.

To get a constraint strong enough to exclude all paths with probability zero, we would have to impose this kind of fairness constraint globally by postulating that an infinitely often visited system state is left through every possible transition infinitely often. This constraint is exactly PRISM’s fairness and stated in the following formula:

\[
G_{\text{PRISM}} := \bigwedge_{s \in \text{states}} \left( \square \diamond s \Rightarrow \bigwedge_{t \in \text{trans.}} (s \land X t) \right)
\]

For our purpose, we can use a weaker LTL formula without the next operator, therefore guaranteed to be stutter invariant:

\[
G_{\text{st.closed}} := \bigwedge_{s \in \text{states}} \left( \square \diamond s \Rightarrow \bigwedge_{t \in \text{trans.}} (s \land \diamond t) \right)
\]

This formula does not demand that every infinitely often visited system state \( s \) is left through every possible transition infinitely often. It is enough if every child of \( s \) in the computation tree is also visited infinitely often. This is equivalent to:

\[
G_{\text{simplified}} := \bigwedge_{s \in \text{states}} \left( \square \diamond s \Rightarrow \bigwedge_{t \in \text{trans.}} \diamond t \right)
\]

But using \( G_{\text{simplified}} \) yields two severe problems:

- The formula is unfeasibly big. Weakening it further so that the first conjunction iterates only over some states may still be enough to exclude infinite paths.
  For example, if we have an algorithm based on guessing IDs, we could select those states for which every node but one already chose an ID in one specific round. If an infinite path visits such a state infinitely often, we can demand that after these visits the last node chooses each identification infinitely often. In this case the states \( t \) in \( G_{\text{simplified}} \) might not be the children of \( s \) in the computation tree, but rather some descendants. This constraint is sufficient to guarantee that the number of active nodes is eventually being reduced with probability one. Hence infinite paths having probability zero are being excluded.
6.1. Replacing randomness by non-determinism

Formulating these kind of LTL formulae is awkward, and there is probably no generic procedure in doing so that works for all algorithms. Maybe there are algorithms for which this kind of reduction is impossible. For a complete verification, we also have to manually prove that the reduction from $G_{\text{simplified}}$ to the smaller LTL formula is correct.

- Naming a global state in an LTL formula requires fixing the values of all variables. To do this for the process counter variables, we need to use labels, as suggested by Holzmann in the example above. We have to use only global variables in the model specification, since local ones cannot be addressed in LTL formulae. This results in an obscure specification and reduces the power of partial order reduction dramatically. We can use the prefix 'local' for the global variables which were originally local ones, i.e. for the global variables which are only used by a single process instance. This restores the power of partial order reduction. But if a variable prefixed with 'local' is used by more than one process instance, the verification result might get erroneous, without SPIN detecting such violations.

Because of these deficiencies, it does not seem promising to dig deeper into the attempt of using fairness to reach our goal of excluding infinite paths having probability zero. Hence we look into other attempts in the following sections.

6.1.2 Temporal logic

The question whether the correctness of an algorithm $A$ can be defined with a temporal logic formula is first clarified for CTL, thereafter for LTL.

**Lemma 6.4.** The CTL formula $C := \text{AG }\text{EF (nr\_Leaders =1)}$ characterizes the correct algorithms.

**Proof.** Let $A$ be correct. Then for every reachable state $s$ there is a path in the computation tree of $A$ from $s$ to an end state with $\text{nr\_Leaders =1}$. So the CTL formula $C := \text{AG }\text{EF (nr\_Leaders =1)}$ is true in the initial state.

Let $C$ be true in the initial state. Then upon termination exactly one leader exists. Otherwise there would be an end state in which the CTL formula $\text{EF (nr\_Leaders =1)}$ were false. All of our regarded algorithms terminate correctly if at some point $\text{nr\_Leaders =1}$: For $A_{\text{timings}}$ and the class of algorithms $A_{\text{compare 1Ds}}$ this is due to the roundtrips made before a leader is elected. They guarantee that no other node wants to become a leader, too. In $A_{\text{sorting}}$, on the other hand, the nodes check if the transmitted string is unique and only in that case a unique leader is selected.

We can use $C' := \text{AG }\text{EF (nr\_Leaders =1 }\land \land \text{EX true)}$ instead to prove this fact if end states are leaves in the computation tree. If end states are modelled as self-loops, we could have used

$$C'' := \text{AG }\text{EF (nr\_Leaders =1 }\land \land \text{ vart} \in \text{ state vector } \land \land \text{ const} \in \text{ range(v) } \land \land \text{ ((v = t) }\leftrightarrow \text{ AX(v = t)})$$,
if it was not unfeasibly big. Another possibility is the weaker condition $C'' := AG EF AG (\text{nr\_Leaders} = 1)$, which does not guarantee end states, only that from some point on there will always be exactly one leader. But the leader node could vary.

Using the technique from the next subsection does not require this argumentation.

Let $s$ be a reachable state. Since $s \models EF (\text{nr\_Leaders} = 1)$, there exists a, w.l.o.g. cycle free, path $n_{s}^{\text{leader}}$ starting in $s$ and ending in a terminal with $\text{nr\_Leaders} = 1$. Since $n_{s}^{\text{leader}}$ is finite, the probability to choose it in state $s$ is $P [n_{s}^{\text{leader}}] > 0$. We assume that $A$ is not correct. As $C$ is true, upon termination exactly one leader exists, so there must be an infinite path $\pi$ with probability $P [\pi] > 0$. Since for every state $s$ of $\pi$ there is a path $n_{s}^{\text{leader}}$ with probability greater zero, $\pi$ contains infinitely many transitions with probability at most $\max_{0 \leq \ell \leq |\pi|} P [|\pi| < 1] < 1$. Consequently $P [\pi] = 0$, contradicting $P [\pi] > 0$. Thus $A$ is correct.

PRISM is a model checker which uses the probabilistic extension PCTL of CTL for specifying properties. Section 2.2.2 explains PCTL and shows that the branching-time operators $A$, $G$, and $F$ are expressible by PCTL. So $C$ is within PCTL, but because of PRISM’s fairness or the support of probabilities, we do not need to use $C$ in chapter 5 to prove the correctness of algorithms. Unfortunately, SPIN does not support verification of branching-time temporal logics, but verification of linear-time temporal logics using LTL. So we also have to inspect whether correctness of algorithms can be defined with an LTL formula $L$, too.

Theorem 6.5 from [Clarke and Draghici, 1988] is used to prove theorem 6.6.

**Theorem 6.5.** Let $\phi$ be a $\text{CTL}^*$ state formula. Then $\phi$ is expressible in LTL iff $\phi$ is equivalent to $A\phi^d$.

$\phi^d$ is the formula obtained from $\phi$ by removing all path quantifiers.

**Theorem 6.6.** There exists no LTL formula that characterizes the correct algorithms.

**Proof.** Using lemma 6.4 and theorem 6.5 we get the equivalence chain:

- There exists an LTL formula that characterizes the correct algorithms
- $A C^d$ is equivalent to $C$.

To apply theorem 6.5 we do not need to transform $C$ into an equivalent $\text{CTL}^*$ formula that only contains basic operations, since deleting the operator $A$ from the formula $A (f)$ and deleting $E$ from the formula $\neg E (\neg f)$ both result in the formula $f$. All other operators do not interfere with the path quantifiers. So we have $A C^d = A G F (\text{nr\_Leaders} = 1)$. Figure 6.1 is a counterexample for the equivalence of $A C^d$ and $C$, as $(\text{init} \rightarrow \text{init})^0 \not\models G EF (\text{nr\_Leaders} = 1)$ but $(\text{init} \rightarrow \text{init})^0 \not\models G F (\text{nr\_Leaders} = 1)$.

But this result does not imply that LTL verification is worthless for our purpose. It just means that it is not enough to use solely an LTL formula to characterize
the correct algorithms. Using additional techniques offered by SPIN and looking into the semantics of the algorithms, we can still prove the correctness of our algorithms by LTL verification, as mentioned at the end of section 6.1.4. But the result shows that we do need to look at other options, which we will do in the following two sections.

6.1.3 Correctness of terminating paths

In this subsection we do not care about infinite paths, which we will consider in the next subsection, 6.1.4. Here we look into proving the correctness of the finite paths. So we want to ignore infinite paths, for example by using the LTL formula

\[ A \left( \Diamond \Box (\text{nr\_Leaders} = 1) \lor \Box X \text{ true} \right). \]

But SPIN offers a more efficient verification method for this. By using assertions, we can show that upon termination exactly one leader is elected. There are several ways to formulate in PROMELA that the assertion \text{nr\_Leaders} \equiv 1 should be checked whenever the traversal of the state space reaches a valid end state. An efficient procedure is using the variable \text{timeout} at the end of the init process: \text{timeout} \rightarrow \text{assert(leader==1)}. This only causes a tiny increase in memory and time requirement. It also tests the assertion in invalid end states, i.e. deadlock states, but we want a correct algorithm anyway and do a valid end state test beforehand (cf. section 6.2.1). So instead of checking the correctness of an algorithm in a single complex verification, we check for valid end states, correctness of terminating paths and properties from the next subsection. This is an example for the technique of splitting up a complex property we want to prove into a sequence of subproperties. Other examples are the sets of correctness properties used in chapter 5. Proving the subproperties individually is more manual work, but the subproperties often have a smaller complexity than the main property.

Verification of this correctly terminating paths property already yields a partial correctness proof, i.e. \( A \) is correct in the cases of termination.

6.1.4 Termination proofs

The property of correctly terminating paths in the previous subsection only proves that an algorithm \( A \) is a partially correct randomized algorithm. To show that \( A \) is correct, we still need to prove the termination property, i.e. that \( A \) terminates with probability one.

One possibility is to prove this manually, like in [Pökkink and Pang, 2004] page 7:

For \( A \in \mathcal{A}_{\text{compare IDs}} \) with \( l \geq 2 \) active nodes, the probability that all nodes choose the same ID in a round is \((1/M)^l < 1\). Only in this case the number of active nodes is not reduced. So the probability that the number of active nodes never decreases is identical to the probability that all active nodes choose...
identical IDs over and over again, which approaches zero. Thus the probability of active nodes turning passive is one. If only one active node is left, it will become the leader.

For \( A_{\text{wiring}} \), the probability of a periodic bit array in a round is smaller than one. Hence the probability of termination approaches one for rising round numbers.

For \( A_{\text{wiring}} \), the probability that \( l \geq 2 \) active nodes initiate a leader request within a given, sufficiently large number of steps is smaller than one, too. Like before, if a single active node is left, it will become the leader.

But these proofs are manual and not stringent. E.g., for the algorithms in \( A_{\text{compare IDs}} \), the proof does not exclude the case that all nodes are turned passive and keep on forwarding left over messages in the ring.

For an automated verification, we need to show that the algorithm repeatedly visits states which have the probability to terminate. Hence for increasing runtime the probability that \( A \) does not terminate approaches zero.

For this we set a progress label in each process at the point where the process performs the random choice. If we successfully check that no np cycles exist, then all cycles contain random choices. For each node at least one random choice results in a node becoming the leader or becoming passive. Hence eventually a leader is successfully elected or all nodes are passive, in which case a np cycle is found.

Having introduced all SPIN methods that are necessary for our verifications, checking the correctness of the algorithms is now also possible using the stutter invariant LTL formula:

\[
L := \Diamond (\text{nr}_n, \text{Leaders} = 1) \lor \Box \Diamond \neg \text{np}_{-n}
\]

For terminating paths \( \Diamond \neg \text{np}_{-n} \) does not hold in the end state, so \( L \) is true iff \( \Diamond (\text{nr}_n, \text{Leaders} = 1) \) is true. For an infinite path \( \pi \) the assumption from the proof of lemma 6.3 is required. Then \( \pi \not\models \Diamond (\text{nr}_n, \text{Leaders} = 1) \), hence \( L \) is true iff \( \pi \) has probability zero.

But avoiding LTL verification and using the mentioned subproperties (cf. page 103) instead is more efficient and does not require the assumption.

6.2 Model checking the promising algorithms

6.2.1 Algorithm \( A_{\text{Hai,round} \leq K} \)

As PROMELA is an imperative language, most of the pseudocode in chapter 4 can be translated directly. Thus only few sections of the complete model specification, given in appendix B.3 need to be explained in detail.

Besides the required constants, two variables to assist verification are defined, as well as the global channels using an array of the composed data type \text{Msg}. The channels are FIFO buffers, so this model specification does not allow message overwriting, and send and receive operations block if the channel is full or empty, respectively. Hence they can only attain high asynchrony by using
large FIFO buffers. Subsection 6.2.2 considers variants of message overwriting for $A_{tk\text{round mod } K}$.

```c
byte nr_leaders=0;
bool limitReached=false;
typedef Msg { byte round; byte id; byte hop; bool unique };
chan q[N] = [L] of {Msg};
```

The nodes behave just as described in chapter 4. Additionally, a node states $x_r$ and $x_s$ assertions to show that only this node uses the corresponding channel to read or send messages. These assertions significantly improve the partial order reduction. To guarantee that this reduction is correct, we additionally check for correct $x_r$ and $x_s$ assertions during verification.

The nodes create their random IDs by non-deterministically incrementing their variables ID until the maximal value is reached:

```c
  id=0;
  do
    : skip -> break
    : id < M-1 -> id++
  od;
```

The decision process additionally prints out error messages for cases that should never happen. Although the messages are only printed for actual runs in simulation mode, SPIN can check for unreachable code existence while verifying. Therefore these print commands for error messages should be reported as unreachable. This check can be avoided by using `assert(0)`.

The init process creates the ring by instantiating $n$ nodes and placing the channels between the nodes. So the initialization of the ring does not have to be hard-coded for each size $n$, solely changing the constant instead is sufficient. This is a big advantage over PRISM.

As described in subsection 6.1.3 the property of correctly terminating paths is put into effect by using the init process, the timeout variable, and an appropriate assertion:

```c
timeout -> assert(nr_leaders==1 || limitReached);
printf("Asserted: exactly one leader or limit reached\n")
```

The variable `limitReached` is set to true if a node reaches the limit $K$. The variable $nr\_leaders$ is increased whenever a node decides to become a leader.

As the correctly terminating paths property uses the variable `timeout`, it checks that exactly one leader exists when all nodes have finished computing. Though exactly one leader is elected, this could be in an erroneous deadlock mode, e.g. with some nodes still being active or with messages still being sent. Thus we also do a valid end state test, which additionally checks that if all nodes have finished computing, then the termination is valid. For this we set the valid end state label `endRECEIVE` just before a node tries to receive a message. Then a terminal is valid if each node is either located at `endRECEIVE` or has terminated. This simple version, used in appendix B.1 can be improved to only set the label for passive nodes.
if
    ; !nodeActive -> end: in?curMsg
    ; else -> in?curMsg
fi;

This check almost includes the **empty buffer property**, since each passive
node no longer receives any messages, i.e. their channels are empty. To check
that all channels are empty, the additional run-time option --q must be set.
Then SPIN additionally requires empty buffers for valid end states.

The correctly terminating paths property only checks that exactly one leader
is elected upon termination. But before termination, there could be more leaders.
Hence we also use the **leader preservation invariance property** (cf.
subsection 5.1.2), which checks that never more than one leader is elected. It is
sufficient to check this right after a leader is chosen and nr_leaders increased:

leader=true; nr_leaders++; assert (nr_leaders==1)

Table 6.1 shows the statistics of selected problem instances for all safety checks.
To avoid thrashing, the memory limit in SPIN is set to 1005MB, although the
physical memory available is 1024MB. As partial order reduction, collapse compression
and statement merging only induce some overhead, but significantly reduce the memory requirements, they are always enabled. Since compression with minimized automata (MA) requires a lot of runtime, it will only be activated for instances that originally entail memory overflow, indicated in the column MA. The column states gives the number of states in the hash table, i.e. the number of different states visited. Matched shows how often a state is revisited, i.e. matched in the hash table, which causes abortion of traversal. Depth gives the maximal reached depth, i.e. the maximal depth of the DMA without the aborted paths. Mem are the memory requirements in MB for model checking the instances, t the time requirements. The memory requirements are influenced by SPIN's parameters of maximal depth and estimated number of states.

<table>
<thead>
<tr>
<th>n</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td>5818</td>
<td>443</td>
<td>191</td>
<td>1.3</td>
<td>5&quot;</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>99328</td>
<td>8877</td>
<td>394</td>
<td>6.5</td>
<td>5&quot;</td>
</tr>
<tr>
<td>4</td>
<td>700</td>
<td>4</td>
<td>4</td>
<td>x</td>
<td>1.7e+07</td>
<td>1.5e+06</td>
<td>50506</td>
<td>1021</td>
<td>43&quot;</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td>1.4e+06</td>
<td>141821</td>
<td>707</td>
<td>123</td>
<td>11&quot;</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td>1.9e+07</td>
<td>2.0e+06</td>
<td>1154</td>
<td>897</td>
<td>135&quot;</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>8.8e+06</td>
<td>574761</td>
<td>1108</td>
<td>400</td>
<td>70&quot;</td>
</tr>
</tbody>
</table>

Table 6.1: Safety checks for $A_{\text{max round}} \leq K$

As expected, the number of states increases exponentially with n. For $n=M=L=4$ with $K=750$ and for $n=K=L=M=7$ memory overflow is inevitable. All safety checks hold for all instances not entailing memory overflow. So compared to PRISM, a few additional instances can be verified.
6.2. Model checking the promising algorithms

Since $A_{Itai, round} \leq K$ is a Monte-Carlo algorithm, it must always terminate. This can be verified with a check for np cycles without any progress labels in the specification, as there is an infinite run iff there is a cycle in the MDP. Since SPIN executes the np cycle check only if a progress label is present, we do set a label at a position that is definitely outside of any cycle, e.g. at the beginning of the init process.

Table 6.2 shows the statistics for the successful np cycle checks for $A_{Itai, round} \leq K$.

<table>
<thead>
<tr>
<th>n</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>24</td>
<td>24543</td>
<td>24570</td>
<td>375</td>
<td>2.1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>752741</td>
<td>719877</td>
<td>779</td>
<td>30.5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1403</td>
<td>2.8e+07</td>
<td>1.9e+07</td>
<td>989</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>2297</td>
<td>5.7e+07</td>
<td>4.6e+07</td>
<td>616</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Np cycle checks for $A_{Itai, round} \leq K$

For $n=K=L=M=7$ no verification is possible because of memory overflow. The np cycle check requires much more time and space than the safety checks, even for this trivial case of setting the progress label at the start of the init process.

So for a full correctness check, the limits on the verifiable problem instances are about the same as in PRISM. Hence we try to raise the limit in the following subsection by avoiding the large hash table.

Note. SPIN’s np cycle check is implemented by the nested DFS algorithm described in chapter [Holzmann, 2004b, page 189] states that an np cycle check with the nested DFS requires at most twice the state space of the safety check. But for $A_{Itai, round} \leq K$ with $n=K=L=M=6$, the liveness check has a state space about three times as large as the safety check, even though the trivially set progress label cannot decrease the redundancy significantly.

Model checking without saving the state space

The results for verifying $A_{Itai, round} \leq K$ in SPIN were not overwhelming. Therefore we will try the following alternative, which exploits the fact that round numbers are being successively increased until termination.

SPIN saves the explored state space to be able to omit the redundant work of exploring states more than once.

An exhaustive enumeration of the state space without saving the part already explored is also possible. This saves memory, but can induce a lot of redundant work. The model checking algorithm also needs to avoid visiting states that are already on the stack, as otherwise cycles in the LTS are traversed infinitely often. Besides memory requirements, the advantages of this trivial DFS technique are easy parallelization and the possibility to detect non-progress cycles directly (cf. page 122). SPIN does not support this technique.
But for $A_{ltai, \text{round}} \leq K$ the LTS does not have any cycles because round numbers are being successively increased until the algorithm terminates. Hence checking whether states are already on the stack is unnecessary. Therefore we can use a method mentioned in [Ruys, 2004] slide 80 to prevent SPIN from saving the explored state space. By setting the parameter $-\text{DBITSTATE}$ SPIN switches into the supertrace approximation mode, also called bitstate hashing, which only uses one bit per hash table entry, assuming that no collisions occur. In this mode the percentage of visited states being saved in the bitstate hash table can be set using the parameter $-\text{DRANDSTOR} =$ (percentage) (aspin erroneously sets the parameter $\text{DRANDSTORE}$, which is being ignored). By using $-\text{DRANDSTOR}=-1$ no states at all are being stored in the hash table, i.e. the hash table passes out of use. Thus we have an exact verification, not an approximation. Its memory requirements are linear in the depth of the LTS, as the stack is still required.

This technique is very useful if the LTS is a tree, since then no states are being revisited, i.e. this method does not perform any additional redundant work. If lots of states in the LTS are reachable through many different paths, it is very likely that trading the unwanted memory requirements for additional time will result in an algorithm with impractically high runtime.

Table 5.3 shows the results of the safety checks for $A_{ltai, \text{round}} \leq K$ using this technique. The values from the regular safety verifications are also mentioned, to be able to compare the results. For both verification techniques we have enabled partial order reduction, statement merging and compression. All regular verifications except $n=K=L=M=7$ work without minimized automaton storage. Since $n=K=L=M=7$ causes a memory overflow, we present in table 5.3 the values for enable minimized automaton storage. As can be seen, memory is still not sufficient. All verifications requiring more than 24 hours were executed on a larger machine with the help of Frank Werner. But the time requirements were converted to the hardware used throughout this thesis.

As the depth only slightly increases for the trivial DFS, there are really no cycles. Sadly the runtime turns out to be much larger without the use of a hash table, so the corresponding LTS does have many revisited states.

Having no cycles in the LTS, there is no need to perform the non-progress cycle check. Trying to run the check nevertheless results in the SPIN error "pan: cannot happen - unstack_put (at depth 760)".

Increasing the percentage of stored states from 0 to 1 yields significantly faster verifications, e.g. for $n=6$, $K=L=M=4$ only 15 seconds instead of 552 minutes (and a depth of 830, a state vector of 212 byte, and 4.8e+06 states). Of course, since now a bitstate hash table is used, we have an approximation. If collisions of different states in the hash table occur seldom, using the trivial DFS technique with an additional, small hash table which randomly stores 1 percent of the explored states would be similarly fast and would only need $4.8 \cdot 10^6 \cdot 0.01 \cdot 212$ byte $< 10$ MByte. Thus the following technique might be a fast rigorous verification with small memory requirements for our algorithm:

- Using the trivial DFS technique. Checking whether a newly visited state is already on the stack to avoid endless cycling can be implemented by a
### 6.2. Model checking the promising algorithms

<table>
<thead>
<tr>
<th>Prob. size $n$</th>
<th>Limit $K$ size $L$</th>
<th>FIFO max. ID $M$</th>
<th>hash table</th>
<th>visited states depth</th>
<th>memory</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>yes</td>
<td>5.8e+03</td>
<td>191</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>9.9e+04</td>
<td>394</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>yes</td>
<td>5.0e+05</td>
<td>592</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>2.2e+06</td>
<td>830</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>8.8e+06</td>
<td>1108</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>yes</td>
<td>7.1e+07</td>
<td>1759</td>
</tr>
</tbody>
</table>

Table 6.3: $A_{\text{safe, round}} \leq K$ without saving the state space

- second hash table of the size of the maximum search depth. Newly visited states are always inserted into the hash table, whereas backtracked states are removed from it.

- Using a hash table of fixed memory size or saving a given small percentage of the explored state space.

- Not using bitstate approximation, but saving complete states.

#### 6.2.2 Algorithm $A_{\text{safe, round mod 2}}$

The model specification for algorithm $A_{\text{safe, round mod 2}}$ is almost the same as for $A_{\text{safe, round}} \leq K$.

The variable limitReach is not required anymore because we now have a Las-Vegas algorithm.

As we use round numbers modulo two, differing round numbers always imply that the incoming message is newer (cf. page 14). Checking whether round numbers reach a limit is no longer required. The complete model specification can be found in appendix B.2

The correctly terminating paths property now uses `assert(nr leaders==1);`

Table 6.4 shows the statistics for the safety checks of selected problem instances for $A_{\text{safe, round mod 2}}$.

For $n=L=M=7$ memory overflow is inevitable. So compared to PRISM, the problem instances allowing safety checks increase from $n=5$ to $n=7$. 

As stated in subsection 6.1.4, it is sufficient for the termination property to set a progress label just before a node randomly chooses its ID. So we can simply exchange the label \texttt{START} with \texttt{progressSTART}. The np cycle checks are successful for \( n \in \{3, 4, 5\} \), for \( n=6 \) memory overflow occurs. So the termination property holds for \( A_{\text{Itai,round mod} \hspace{1pt} n} \) with \( n \in \{3, 4, 5\} \). Hence together with the correctly terminating paths property, the correctness of the algorithm is shown.

The termination property used the fact that for each node at least one random choice results in a node becoming the leader or becoming passive. This is true for algorithms in \( A_{\text{compare ID} \hspace{1pt} n} \) since at least one node becomes passive or the leader if not all nodes choose the same ID. This can also be verified automatically by setting the progress labels more restrictively only when all IDs passing through an active node are equal to its own ID. For this matter, a new local Boolean variable \texttt{allSameID} is defined, set to true whenever the node chooses a new ID, and modified appropriately when new messages arrive. When a roundtrip occurs, the progress label is only set for the appropriate case, just before the node decides whether to restart or become the leader:

\[
\begin{align*}
\text{:: } \text{curMsg. counter} = \text{N} & \rightarrow \\
\quad \text{if} & \\
\quad \text{:: allSameID} & \rightarrow \text{progress: skip} \\
\quad \text{:: ! allSameID} & \rightarrow \text{skip}
\end{align*}
\]

Hence when not all IDs are identical, a node must really become passive or the leader, as otherwise there would be a cycle without a progress label. The complete model specification is given in appendix \[3.2\].

Table 6.4: Safety checks for \( A_{\text{Itai,round mod} \hspace{1pt} 2} \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( L )</th>
<th>( M )</th>
<th>( \text{MA} )</th>
<th>( \text{states} )</th>
<th>( \text{matched} )</th>
<th>( \text{depth} )</th>
<th>( \text{mem} )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>MA</td>
<td>5623</td>
<td>955</td>
<td>894</td>
<td>3.1</td>
<td>5&quot;</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>86058</td>
<td>18233</td>
<td>6239</td>
<td>20</td>
<td>5&quot;</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td>1.0E+06</td>
<td>218325</td>
<td>45841</td>
<td>202</td>
<td>1&quot;</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td>1.0E+07</td>
<td>2.0E+06</td>
<td>182448</td>
<td>411</td>
<td>5&quot;</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td></td>
<td>2.3E+07</td>
<td>5.7E+06</td>
<td>235515</td>
<td>420</td>
<td>90&quot;</td>
</tr>
</tbody>
</table>

Table 6.5: Np cycle checks for \( A_{\text{Itai,round mod} \hspace{1pt} 2} \)

For \( n=L=M=6 \) no verification is possible because of memory overflow. Therefore compared to PRISM, the verifiable problem instances only increase significantly for the safety checks. For the np cycle checks, only few instances are additionally verifiable.
Since SPIN only offers the option to either block or lose new messages when a buffer is full, message overwriting has to be implemented. The first variant replaces the channels by global variables:

\[
\text{Msg buffer [N];}
\]

buffer[i] is defined to be empty if buffer[i].id==M. An initialization of a node no longer has channels as parameters, only an index for the array buffer. Additionally, sending and receiving changes:

\[
\text{SEND AND RECEIVE: skip; /* send message */}
\]

\[
\text{d_step}
\]

\[
\text{buffer[(index+1)modN].id=curMsg.id;}
\]

\[
\text{buffer[(index+1)modN].round=curMsg.round;}
\]

\[
\text{buffer[(index+1)modN].hop=curMsg.hop;}
\]

\[
\text{buffer[(index+1)modN].unique=curMsg.unique;}
\]

\[
\text{endRECEIVE: /* get message */}
\]

\[
\text{buffer[index].id<M -->}
\]

\[
\text{d_step}
\]

\[
\text{curMsg.id=buffer[index].id;}
\]

\[
\text{curMsg.round=buffer[index].round;}
\]

\[
\text{curMsg.hop=buffer[index].hop;}
\]

\[
\text{curMsg.unique=buffer[index].unique;}
\]

\[
\text{buffer[index].id=M;}
\]

For \( n \leq 4 \) both safety and liveness checks are successful. For \( n \geq 5 \) verification entails memory overflow. This might be due to the reduced po reduction because of the global variables.

Hence the second variant does not use global variables, but splits up a channel into a preBuffer and postBuffer, with a new process bufferproc in between. This process non-deterministically either forwards a message to the next node or overwrites its buffer by receiving a new message from the previous node. Hence buffers of size one are sufficient:

\[
\text{chan preBuffer [N] = [1] of {Msg};}
\]

\[
\text{chan postBuffer [N] = [1] of {Msg};}
\]

The nodes behave identical as before, i.e. they do not know whether the channels in use allow message overwriting. The bufferprocs are defined as follows:

\[
\text{proc type bufferproc (chan in, out; pid next)}
\]

\[
\{\text{Msg buffer; bool bufferEmpty = true; xr in; xs out; endvalid: do}
\]

\[
:\text{in?buffer --> bufferEmpty = false;}
\]

\[
:\text{bufferEmpty=false & node[next]@endRECEIVELABEL -->}
\]
By checking whether the next node waits for a message, the messages are forwarded lazily. Therefore as many messages as possible are overwritten. This also avoids checking whether the channel out is empty, which would violate the \( \sigma \)s assertion. Of course initialization has to be modified according to these changes. This model specification can also be found in appendix 6.2.

Sadly this variant also causes memory overflow for \( n \geq 5 \). This is probably caused by reduced po reduction because of the referencing in bufferproc. For \( n \leq 4 \) the verifications are successful.

### 6.2.3 Algorithm \( \mathcal{A}_{Fokkink,roundtrip} \)

There are only few changes between \( \mathcal{A}_{Itai,roundmod K} \) and \( \mathcal{A}_{Fokkink,roundtrip} \): Variables and checks for round numbers are omitted. The decision of what to do for the incoming message is just as described in listing 6.2. The complete model specification can be found in appendix 6.3.

Table 6.6 shows the statistics for the safety checks.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( L )</th>
<th>( M )</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3204</td>
<td>326</td>
<td>269</td>
<td>2.7</td>
<td>5''</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>105229</td>
<td>15879</td>
<td>2117</td>
<td>5.9</td>
<td>5''</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3.6e+06</td>
<td>562904</td>
<td>8906</td>
<td>119</td>
<td>25''</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>x</td>
<td>1.3e+08</td>
<td>2.0e+07</td>
<td>31039</td>
<td>491</td>
<td>151''</td>
</tr>
</tbody>
</table>

Table 6.6: Safety checks for \( \mathcal{A}_{Fokkink,roundtrip} \)

For \( n=L=M=7 \) memory overflow is inevitable. As expected, the depth is much smaller than for \( \mathcal{A}_{Itai,roundmod 2} \). Curiously, the state space increases. Compared to PRISM, the maximal problem instance increases from \( n=4 \) to \( n=6 \) for the safety checks.

Table 6.7 shows the statistics for the np cycle check for \( \mathcal{A}_{Fokkink,roundtrip} \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( L )</th>
<th>( M )</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>12494</td>
<td>12939</td>
<td>745</td>
<td>35</td>
<td>5''</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>54589</td>
<td>579508</td>
<td>8277</td>
<td>618</td>
<td>11''</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>x</td>
<td>3.6e+07</td>
<td>2.8e+07</td>
<td>144667</td>
<td>782</td>
<td>36''</td>
</tr>
</tbody>
</table>

Table 6.7: Np cycle checks for \( \mathcal{A}_{Fokkink,roundtrip} \)

For \( n=L=M=6 \) memory overflow occurs. Compared to \( \mathcal{A}_{Itai,roundmod 2} \) the liveness checks have improved. So the complete verification with SPIN works for \( n \) up to five, which is one more than in PRISM.
6.2. Model checking the promising algorithms

6.2.4 Algorithm $A_{Fokkink, no roundtrip}$

The model specification for $A_{Fokkink, no roundtrip}$ can be obtained from the one for $A_{Fokkink, roundtrip}$ by removing the unique flags and modifying the decisions for incoming messages as in listing 4.3. This results in the model specification in appendix 4.4.

The statistics of the safety checks are presented in table 6.8:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$L$</th>
<th>$M$</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3755</td>
<td>607</td>
<td>310</td>
<td>5</td>
<td>5&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>202119</td>
<td>34599</td>
<td>2843</td>
<td>12</td>
<td>5&quot;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1.3e+07</td>
<td>2.2e+06</td>
<td>32806</td>
<td>428</td>
<td>65&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8: Safety checks for $A_{Fokkink, no roundtrip}$

So the safety checks for $A_{Fokkink, no roundtrip}$ are worse than for $A_{Fokkink, roundtrip}$, and memory overflow already happens for $n=L=M=6$.

Table 6.9 shows $A_{Fokkink, no roundtrip}$'s statistics for the np cycle check:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$L$</th>
<th>$M$</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>23988</td>
<td>30363</td>
<td>3669</td>
<td>35</td>
<td>5&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2.4e+06</td>
<td>3.2e+06</td>
<td>139437</td>
<td>168</td>
<td>25&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.9: Np cycle checks for $A_{Fokkink, no roundtrip}$

Again the statistics are worse than for $A_{Fokkink, roundtrip}$, now $n=L=M=5$ yields memory overflow already. So the complete verification with SPIN works for $n$ up to four, which is not better than in PRISM.

6.2.5 Algorithm $A_{sorting}$

Most of the specification for $A_{sorting}$, given in appendix 4.4 for prime $n$, directly corresponds to the pseudocode presented in listing 4.4. So we again abstract from a concrete decision procedure and only check whether the bit array is periodic.

The local variables differ from those of $A_{compare IDs}$: The flag nodeActive is no longer of use, leader is not, either, because we have an abstract algorithm. Instead of the variables id and curMsg, $A_{sorting}$ uses the bit array vector of size $n$ to store all IDs, and a variable index indicating which elements of the bit array have already been received. The variable sum is used for prime $n$ to ease the periodicity check: If all IDs have been received, the bit vector is periodic iff the sum of all its bits is either zero or $n$:

:: index $\in \mathbb{N} \rightarrow$

\begin{verbatim}
do
\end{verbatim}
:: index>0 -> index--; sum=sum+vector[index]
:: index==0 -> break
od;
if
:: sum==0 || sum==N -> goto progressRESTART /* periodic */
:: else -> leader_exists = true; goto FINISH /* aperiodic */
fi

Because $A_{\text{sorting}}$ solely uses an abstract decision procedure, we can only check the leader existence property, i.e., whether all nodes have decided that there exists some leader. So the variable n_leader is replaced by the Boolean variable leader_exists, and the correctly terminating paths property now uses the assertion

\[
\text{timeout} \rightarrow \text{assert(leader_exists)}; \\
\text{printf("Asserted: some leader exists\n")}
\]

Section 5.3 has shown that $A_{\text{sorting}}$ works best for prime $n$. Hence we prefer these values for the safety checks, listed in Table 6.10

<table>
<thead>
<tr>
<th>$n$</th>
<th>$L$</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>2014</td>
<td>325</td>
<td>395</td>
<td>7</td>
<td>5''</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>120022</td>
<td>19290</td>
<td>3102</td>
<td>19.2</td>
<td>5''</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>x</td>
<td>7.0e+06</td>
<td>1.0e+06</td>
<td>29432</td>
<td>460</td>
<td>1''</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>x</td>
<td>1.6e+07</td>
<td>7.3e+06</td>
<td>206105</td>
<td>249</td>
<td>9''</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>x</td>
<td>1.1e+08</td>
<td>5.1e+07</td>
<td>2.1e+06</td>
<td>929</td>
<td>78''</td>
</tr>
</tbody>
</table>

Table 6.10: Safety checks for $A_{\text{sorting}}$

For $n=L=10$ memory overflow occurs, as for $n=L=11$.

The termination property is performed just as for the previous algorithms. For $A_{\text{sorting}}$, all nodes terminate if the bit array is not periodic. To show this, the progress label can be set more restrictively, only for the cases when the bit array is periodic. An easy implementation is setting the progress label right after the guard for periodicity, before jumping back to the START label. But since this is the only location of the command goto START, using the label progressSTART is equivalent. So for all aperiodic bit arrays, $A_{\text{sorting}}$ really terminates, as there would be a np cycle otherwise.

Table 6.11 shows the statistics for the np cycle checks of $A_{\text{sorting}}$, again with prime numbers preferred.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$L$</th>
<th>MA</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>7127</td>
<td>9660</td>
<td>939</td>
<td>18</td>
<td>5''</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>757220</td>
<td>1.1e+06</td>
<td>12966</td>
<td>61</td>
<td>15''</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>x</td>
<td>1.1e+08</td>
<td>1.1e+08</td>
<td>248678</td>
<td>652</td>
<td>77''</td>
</tr>
</tbody>
</table>

Table 6.11: Np cycle checks for $A_{\text{sorting}}$
6.2. Model checking the promising algorithms

For \( n=L=8 \), as for \( n=L=11 \), memory overflow is inevitable. Hence the limits on the verifiable problem instances are slightly higher for SPIN than for PRISM.

Just like in PRISM, the termination property and the correctly terminating paths property do not imply the correctness of \( A_{\text{sorting}} \), only that there exists a correct version of the algorithm, which correctly implements the decision procedure.

6.2.6 Algorithm \( A_{\text{timing}} \)

The pseudocode from listing 5.3 translates directly into the PROMELA specification in appendix 5.5 too. The two variables \( \text{hop}_m \) and \( \text{unique}_m \) are comprised in \( \text{curMsg} \) of composed data type. Randomly waiting for a message or initiating one by itself could be implemented in PROMELA using the race condition

if
   :: in?\text{curMsg} \rightarrow \ldots
   :: \text{else} \rightarrow \ldots
fi;

But SPIN does not allow this race condition. Consequently it has to be implemented by non-deterministically deciding over and over again within a repetition:

\[
\text{progressSTART:}
\]
\[
\text{do}
\]
\[
:: \text{atomic}\{\text{empty}(\text{in}) \rightarrow \text{in?}\text{curMsg}; \text{curMsg}.\text{hop}++\};
\]
\[
:: \text{else} \rightarrow \text{if}
\]
\[
:: \text{skip} \rightarrow \text{progressWait: skip}
\]
\[
:: \text{skip} \rightarrow \text{curMsg}.\text{hop}=1; \text{curMsg}.\text{unique}=\text{true};
\]
\[
\text{goto A\text{ACTIVESENDANDRECEIVE}}
\]
\[
\text{fi}
\]
\[
\text{od};
\]

The STS and LTS of a single node are presented in figure 6.11 and 6.13. They are similar to the state machine shown in figure 5.19.

The safety checks are performed just like before. But for \( n \geq 16 \) the state vector exceeds the original limit of 1024 byte, so the limit must be raised by the compile-time option \( \text{-DVECTORSZ} \). Table 6.12 shows the statistics for selected problem instances. The column vector gives the size of the corresponding state vector in byte.

For \( n=L=24 \) memory overflow is inevitable. The table shows that both state space and depth double when increasing \( n \) and \( L \) by one. So the state space is still exponential in \( n \).

\( A_{\text{timing}} \) can cycle endlessly either if all nodes keep waiting all the time, or some nodes keep initiating leader requests parallelly over and over again. Thus the repetition mentioned above contains two progress labels: progressWait for the
Table 6.12: Safety checks for $A_{timing}$

<table>
<thead>
<tr>
<th>n</th>
<th>L</th>
<th>MA</th>
<th>vector</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>100</td>
<td>105</td>
<td>33</td>
<td>54</td>
<td>2.6</td>
<td>5'</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>488</td>
<td>7095</td>
<td>6903</td>
<td>1614</td>
<td>2.6</td>
<td>5'</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>684</td>
<td>29965</td>
<td>33781</td>
<td>6238</td>
<td>27</td>
<td>25'</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>908</td>
<td>127347</td>
<td>159731</td>
<td>24686</td>
<td>274</td>
<td>1'</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>1308</td>
<td>1.1e+06</td>
<td>1.6e+06</td>
<td>196742</td>
<td>275</td>
<td>1'</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>1616</td>
<td>4.7e+06</td>
<td>7.1e+06</td>
<td>786582</td>
<td>297</td>
<td>6'</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>x</td>
<td>1952</td>
<td>2.0e+07</td>
<td>3.1e+07</td>
<td>3.1e+06</td>
<td>247</td>
<td>21'</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>x</td>
<td>2324</td>
<td>8.4e+07</td>
<td>1.4e+08</td>
<td>1.3e+07</td>
<td>489</td>
<td>95'</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>x</td>
<td>2520</td>
<td>1.7e+08</td>
<td>2.9e+08</td>
<td>2.5e+07</td>
<td>976</td>
<td>206'</td>
</tr>
</tbody>
</table>

first case, progressSTART for the second. Therefore each progress cycle contains a random choice.

Table 6.13 shows the statistics of the np cycle checks for $A_{timing}$.

<table>
<thead>
<tr>
<th>n</th>
<th>L</th>
<th>MA</th>
<th>vector</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>104</td>
<td>1400</td>
<td>1542</td>
<td>387</td>
<td>3.1</td>
<td>5'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>148</td>
<td>18716</td>
<td>22921</td>
<td>2185</td>
<td>3.1</td>
<td>5'</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>200</td>
<td>276779</td>
<td>382421</td>
<td>30615</td>
<td>17</td>
<td>6'</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>264</td>
<td>4.3e+06</td>
<td>6.7e+06</td>
<td>335635</td>
<td>251</td>
<td>70'</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13: Np cycle checks for $A_{timing}$

Unfortunately for $n=L=7$ memory overflow is inevitable.

For a stringent proof of the termination property, we have to show that there is a path from each progress label to a terminal. This is the case if one node initiates a leader request and the others do not. But for the completely automated proof with SPIN, we again prefer the equivalent condition that for the right random decisions a node can become passive or the leader. As a node in $A_{timing}$ can require more than one random decision for this, the proof is more complicated than for $A_{compare 1Dx}$. After a progress label a node $p$ can non-deterministically switch into another mode for testing whether there is the possibility to become passive or the leader. For this $p$ sends a leader request and waits to receive the next incoming message. If the message’s originator is $p$, it can become the leader. If another node created the message, then $p$ could have become passive if it had not created a leader request but had simply waited. So in short, testing in all situations whether $p$ can receive a message some time after creating one is sufficient to show the termination property. Since the ring is symmetric, only one node needs to be replaced with the test node $p$. The test is performed by the following code in $p$, with nodeWaits being a global Boolean variable initialized with zero:

```c
nodeSwitchesIntoTestingMode:
    curMsg . counter = 1; curMsg . unique = true;
    out ! CurMsg; nodeWaits = 1;
in ? curMsg; nodeWaits = 0;
```
6.2. Model checking the promising algorithms

The init process now checks

```c
timeout -> assert (nodeWaits==0);
printf("Node can definitely become passive or leader\n");
```

The complete model specification can be found in appendix B.6. The assertion can successfully be checked for all \( n \in \{3, \ldots, 23\} \).

The depth of the models for the safety checks are at least 1/7th of the corresponding state space, which is much larger than for the other algorithms. The reason is the strong partial order reduction, which increases the limit to \( n=23 \) and produces long paths with only few branches. So if the models can be improved to have shorter depths, the state spaces will probably decrease likewise.

As statement merging eliminates intermediate states of branchless paths, trying to optimize the model specification for this reduction looks the most promising. Experiments with several variants show that decreasing the number of progress labels to one and positioning it within the repetition just after the \textbf{else} guard is a simple yet efficient improvement:

```
START:
do
  : atomic [nempty(in) -> in?curMsg; curMsg.hop++];
goto PASSIVE_SEND_AND_RECEIVE
  : else -> progress: if
    : skip ->
    : skip → curMsg.hop=1; curMsg.unique=true;
goto ACTIVE_SEND_AND_RECEIVE
  fi
od;
```

The np cycle check is still valid, as the new progress label covers both previous cases. The improved statistics for the safety check are given in table 6.14. The minimized automaton storage never needs to be enabled. The np cycle check still yields memory overflow for \( n=7 \).

So the safety checks are successful up to \( n=254 \), i.e. the safety checks are no longer only proved for small problem instances. \( n \geq 255 \) results in the error "too many processes", as SPIN limits the number of active processes to 255.

The reduction strength for this optimized specification almost compensates the state space explosion, so that the resource requirements for the safety checks are about quadratic in \( n \): The size of the state space, i.e. of the hash table, is about \( 4 \cdot n^2 \), the depth, i.e. the size of the stack, is \( 7 \cdot n + 12 \). The time requirements are about quadratic too, but hard to estimate because of the few steps caused by overhead.

The number of matched states is exactly \( n \). This shows how strong the partial order reduction is: The LTS of a single node of \texttt{Atiming}, given in figure 6.4, shows that each node can independently switch between the states line 24 and line 26 for randomly waiting. So without partial order reduction there are already \( 2^n \) global states for random waiting, which can all but one be matched.
6.3 Conclusion

SPIN’s efficiency and number of parameters show that it is a mature model checker. Consequently we only have to deal with minor errors:

- A minor error in the verifier without a hash table is the output of the verification time, which is much too small. E.g. the verification of $\mathcal{A}_{\text{tim}, \text{round} \leq K}$ with $n=6$, $K=L=M=4$, given in table 6.13 requires 552 minutes, SPIN claims 53 minutes.

- Trying to check for non-progress cycles without a hash table results in the error "pan: cannot happen - unstack_put (at depth 760)". This check is unnecessary (cf. section 6.2.1), but its error shows that something is malfunctioning.

- In section 6.2.3 we use various sequences containing progress label within atomic { ... }. SPIN gives the warning "progress label inside atomic - is invisible" only inconsequently.

Except for the experiments in subsection 6.2.1 SPIN only fails because of memory overflow when the state space explodes, whereas the time requirements only increase within a practicable range.

SPIN can prove the correctness of all investigated algorithms for the instances not entailing memory overflow. As in PRISM, the better algorithms also yield better model checking statistics. For the best algorithm, $\mathcal{A}_{\text{tim,prog}}$, the optimized specification has overcome the state space explosion problem for the safety check, guaranteeing that the properties are not only valid for small problem instances. Chapter 7 will investigate the possibilities to transfer this success to non-progress cycle checks and therefore to the complete correctness checks for our algorithms.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$L$</th>
<th>vector</th>
<th>states</th>
<th>matched</th>
<th>depth</th>
<th>mem</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>100</td>
<td>66</td>
<td>3</td>
<td>33</td>
<td>2.3</td>
<td>5^*</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>144</td>
<td>103</td>
<td>4</td>
<td>40</td>
<td>2.3</td>
<td>5^*</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>196</td>
<td>148</td>
<td>5</td>
<td>47</td>
<td>2.3</td>
<td>5^*</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>260</td>
<td>201</td>
<td>6</td>
<td>54</td>
<td>2.3</td>
<td>5^*</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>328</td>
<td>262</td>
<td>7</td>
<td>61</td>
<td>2.3</td>
<td>5^*</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>2936</td>
<td>2728</td>
<td>25</td>
<td>187</td>
<td>38</td>
<td>5^*</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>10864</td>
<td>10453</td>
<td>50</td>
<td>362</td>
<td>38</td>
<td>5^*</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>41716</td>
<td>40903</td>
<td>100</td>
<td>712</td>
<td>120</td>
<td>5^*</td>
</tr>
<tr>
<td>150</td>
<td>150</td>
<td>92564</td>
<td>91353</td>
<td>150</td>
<td>1062</td>
<td>230</td>
<td>30^*</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>163416</td>
<td>161803</td>
<td>200</td>
<td>1412</td>
<td>341</td>
<td>45^*</td>
</tr>
<tr>
<td>250</td>
<td>250</td>
<td>254264</td>
<td>252253</td>
<td>250</td>
<td>1762</td>
<td>341</td>
<td>45^*</td>
</tr>
<tr>
<td>253</td>
<td>253</td>
<td>260352</td>
<td>258316</td>
<td>253</td>
<td>1783</td>
<td>341</td>
<td>98^*</td>
</tr>
<tr>
<td>254</td>
<td>254</td>
<td>262396</td>
<td>260353</td>
<td>254</td>
<td>1790</td>
<td>341</td>
<td>100^*</td>
</tr>
</tbody>
</table>

Table 6.14: Safety checks for improved $\mathcal{A}_{\text{tim}}$
Subsection 6.2.1 has shown that small changes in the model specification can cause drastically different model checking requirements. But the improvements in the code can only be made with many experiments and the knowledge of the model checking algorithms and reduction methods underlying SPIN. Thus the successful model checking of our algorithms is far from a push-button technology.
Chapter 7

A non-progress cycle check for SPIN

Progress of the system is modelled in PROMELA by using labels: "statement; progress: statement;" in a process sets the local state after statement, and before statement, to a local progress state. A global progress state (also simply called progress state) is a global system state in which at least one of the processes is in a local progress state. SPIN marks global progress states by setting the global variable np_ to false.

A non-progress cycle check detects (and returns a path to) a reachable non-progress cycle (np cycle), i.e. a reachable cycle with no progress states (no progress), iff there exists one in the LTS.

SPIN puts the check into effect with the LTL formula $\Diamond \Box np_\Box$ and the standard acceptance cycle check via B"uchi automata, which is implemented by the nested depth-first search algorithm: Before the basic depth-first search (DFS) backtracks from an accepting state $s$ and removes it from the stack (i.e. closes $s$), a second DFS is started to check whether $s$ can reach itself, thus resulting in an acceptance cycle. Pseudo-code for the basic depth-first search (DFS) is given in listing [11] for the nested DFS in [Holmann et al., 1996]. For the reason of convention and briefness we use the term statespace in our listings synonymously for the explored state space. init is the initial state of the LTS.

Section [6] used np cycle checks to verify that our algorithms terminate with probability one. Unfortunately, these checks perform much worse than the safety checks. E.g. for $A_{time}$ the np cycle checks yield memory overflows for $n > 6$ nodes, whereas the safety checks are quadratic in $n$ and work up to $n = 254$. The reason for the bad performance lies within the nested depth-first search algorithm with activated partial order reduction (cf. Holmann et al., 1996 section 5.2)). Hence this chapter searches for an alternative to SPIN's np cycle check.

The next section, [7], explains the basic algorithm, proves its correctness, and estimates its complexity. Section [8] introduces SPIN's partial order reduc-
proc DFS(s)  
add s to statespace;  
push s onto stack;  
for each successor t of s do  
  if (t \notin statespace) then DFS(t) fi  
end;

proc main()  
  DFS(init);  
end;

Listing 7.1: Basic depth-first search

The basic DFS generates the complete LTS. Since the DFS aborts recursion in states which have already been visited, not all cycles are traversed. Figure 7.1 shows an example of a non-progress cycle that is not traversed and therefore not found by the DFS. From s, the DFS first traverses and backtracks the states in the triangle, then the DFS uses the path right of the triangle and aborts it before closing the np cycle.

Hence, if a non-progress cycle has states that have already been visited, the cycle will not be found by the DFS. The idea for our alternative non-progress cycle check is to enforce traversing np cycles first, achieved by the incremental DFS, given in the following definition. Therefore a second, nested DFS is not needed anymore to detect np cycles.

**Definition 7.1 (incremental DFS).** Let \( G \) be the complete LTS, \( L \in \mathbb{N}_{\geq 0} \), and \( G_L \) be the maximal subgraph of \( G \) such that all paths in \( G_L \) without cycles have at most \( L \) progress states. We call \( L \) complexity limit. The incremental DFS algorithm, described in listing 7.3 and in listing 7.2 (or 7.4) or 7.5, repeatedly builds \( G_L \) for \( L = 0, 1 \ldots \) with the basic DFS algorithm and
7.1. Without partial order reduction

![Diagram of a graph with nodes and edges labeled init, progress counter, non-progress cycle, and aborted traversal, with already closed states indicated.]

Figure 7.1: Not traversed non-progress cycle

terminates either with a (shortest w.r.t. \( L \)) error-path when a non-progress cycle is found or without an error when \( L \) becomes big enough for the incremental DFS to build the complete graph \( G \), i.e. \( G_L = G \).

```plaintext
proc DFSstarting_over(s)
    L := 0;
    repeat
        DFS_pruned := false;
        DFS_is_nps_check(s);
        L++;
    until (!DFS_pruned);
end;

proc main()
    DFSstarting_over(init);
    printf("LTS does not contain np cycles");
end;
```

Listing 7.2: Incremental depth-first search with starting over

So in each state \( s \) we might prune some of the outgoing transitions by omitting those which exceed the current complexity limit \( L \), and only consider the remaining transitions, denoted by

\[ \text{prune}_L(s) \] (or \( \text{prune}(s) \) if \( L \) is not important or implicitly given). Since pruning in a state \( s \) depends on the number of progresses made so far, the current path \( \pi \) to \( s \) is an implicit argument to \( \text{prune}(s) \), indicated by \( \text{prune}_\pi(s) \) or \( \text{prune}_* = \text{prune}_\pi(\text{end of } \pi) \).

Since we label transitions with statements, each outgoing transition for a given state \( s \) is generated by only one enabled statement, denoted \( \text{statement}_s(t) \). Because of the partial order reduction considered in section 7.2.1 we will only use deterministic statements, cf. page 244. Hence \( \text{statement}_s \) is a bijection from the outgoing transitions of \( s \) to \( \text{enabled}(s) \).
proc DFSprune np_check (s)
add s to statespace;
push s onto stack;
for each successor t of s do
if (t \notin statespace)
then if (!pruned(s,t))
then DFSprune np_check (t)
else pruning_action(t)
fi
else if (t \in stack \&\& np_cycle (t))
then halt with error_message
fi
fi
od;
pop s from stack;
end;

Listing 7.3: Generic DFS with pruning and np cycle check

We extend prune_L(s) to sets of statements \( E \) and to complete graphs \( G \):

\[
\text{prune}(E) = \text{prune}_L(E) = \text{prune}_{L\pi}(E) = \text{statement}_{end\ of\ \pi} \cap E
\]

\[
\text{prune}(G) = \text{prune}_L(G) = G_L
\]

One way to implement DFS_clear is by using a progress_counter, which counts the number of progress states on the path to the current state. This counter is saved for every state on the stack, but is ignored when states are being compared. With this concept, we can firstly update the progress_counter when backtracking, secondly abort traversal when the progress_counter exceeds the complexity limit \( L \), and thirdly check for progress whenever a cycle is found. To complete this implementation, we still have to determine the unsettled functions of DFSprune np_check, underlined in listing 7.3

- \text{pruned}(s, t) \text{ returns true iff } \text{progress}\_\text{counter} = L \text{ and } t \text{ is in a progress state.}
- \text{pruning}\_\text{action}(t) \text{ sets DFS}\_\text{pruned to true.}
- \text{np}\_\text{cycle}(t) \text{ returns true iff } \text{progress}\_\text{counter} = (\text{counter on stack for } t).
- The error_message can print out the stack, which corresponds to the path from init to the np cycle, inclusively.

### 7.1.1 Correctness

Since our incremental DFS only omits transitions (when pruning the LTS) but does not generate new ones, it does not output false negatives.
Theorem 7.2. If non-progress cycles exist, the incremental DFS will find a smallest one with respect to the number of progresses that have to be made from the initial state until the cycle is reached. With \( L_0 \in \mathbb{N}_{\geq 0} \) being the smallest such number, our algorithm finds this cycle after \( L_0 \) increments of \( L \).

Proof. If \( L < L_0 \), all paths leading to a non-progress cycle are pruned before the cycle is reached. So the incremental DFS cannot find a non-progress cycle. Let \( L = L_0 \), \( s \) be the first state reached in the DFS which is on a non-progress cycle, \( t_{\text{begin}} \) be the time the DFS reaches \( s \) (the first time), and \( \pi^1 = \langle s, s_1, s_2, \ldots, s_{n_1} = s \rangle \) be a np cycle containing \( s \). Since the progress counter equals \( L_0 \) for \( s \), the DFS will stop traversing transitions that increase the progress counter on any path after it contains \( s \).

We assume no non-progress cycle is found for \( L = L_0 \). Hence the traversal of \( \pi^1 \) must be aborted because a state \( s_{h_1} \neq s \) for \( h_1 \in \{2, \ldots, n_1 - 1\} \) is revisited (i.e. visited when already in the hash table) before \( \pi^1 \) is closed, i.e. before \( s \) could be reached the second time. Let \( t_{\text{middle}_1} \) be the time when \( s_{h_1} \) is visited the first time and \( t_{\text{end}} \) be the time when \( s_{h_1} \) is revisited and \( \pi^1 \) is aborted. \( s_{h_1} \) cannot be on the stack at time \( t_{\text{end}} \) not below \( s \) (i.e. pushed onto the stack earlier than \( s \)), because \( s \) is the first visited state of \( \pi^1 \), and not above \( s \) (i.e. pushed onto the stack after \( s \)), since then a np cycle \( \langle s_{h_1}^1, \ldots, s_{h_1}^1 \rangle \) would be found, cf. figure 7.2. So our algorithm first visits \( s \) in \( t_{\text{begin}} \), then visits \( s_{h_1}^1 \) at \( t_{\text{middle}_1} \), then backtracks from \( s_{h_1}^1 \) and finally revisits \( s_{h_1}^1 \) at \( t_{\text{end}} \) while traversing \( \pi^1 \).

Let \( \pi^2 = \langle s_1^2 = s, s_2^2, \ldots, s_{n_2}^2 = s \rangle \) be the path from \( s \) at time \( t_{\text{begin}} \) to \( s_{h_1}^1 \) at time \( t_{\text{middle}_1} \), concatenated with \( \langle s_{h_1+1}^1, s_{h_1+2}^1, \ldots, s_{n_1}^1 = s \rangle \), i.e. \( \pi^2 \) is also a np cycle containing \( s \), cf. figure 7.3.

Therefore we can apply the argumentation from above to \( \pi^2 \) instead of \( \pi^1 \) to obtain a state \( s_{h_2}^2 \) on \( \pi^2 \) and a time \( t_{\text{middle}_2} \) with the properties described above. As on \( \pi^2 \) the state \( s_{h_1}^1 \) is visited the first time at \( t_{\text{middle}_1} \), the DFS also reaches \( s_{h_1+1}^1 \) on \( \pi^2 \) (at some time after \( t_{\text{middle}_2} \) and not after \( t_{\text{middle}_1} \)). So \( s_{h_1}^2 = s_{h_2}^2 \) for \( h_2 = \{h_1 + 1, \ldots, n_1 - 1\} \). Let \( \pi^3 = \langle s_1^3 = s, s_2^3, \ldots, s_{n_2}^3 = s \rangle \) be the np cycle from \( s \) at time \( t_{\text{begin}} \) to \( s_{h_2}^2 \) at time \( t_{\text{middle}_2} \), concatenated with \( \langle s_{h_2+1}^2, s_{h_2+2}^2, \ldots, s_{n_2}^2 = s \rangle \).

Applying this argumentation to the last element of the sequence \( \langle \pi^1, \ldots, \pi^d \rangle \) we get the sequence \( \langle \pi^1, \ldots, \pi^{j+1} \rangle \) with strictly monotonically increasing \( h_i < \cdots < h_{j+1} \) and all \( h_i \in \{2, \ldots, n_1 - 1\} \). This is a contradiction since \( j \) can get arbitrarily big. Thus the assumption that no non-progress cycle is found for \( L = L_0 \) is wrong.

So our algorithm always terminates and finds a shortest non-progress cycle iff one exists. If all cycles in the LTS make progress, \( L \) will eventually be big enough to contain the complete graph. After traversing it the algorithm terminates without an error message and we can be sure that no non-progress cycle exists. Thus our algorithm is correct.

Note. The proof shows that the incremental DFS finds a np cycle before backtracking from \( s \). The np cycle does not have to contain \( s \), though: Figure
Figure 7.2: $s^1_{hi}$ cannot be on the stack at time $t_{end}$

Figure 7.3: Constructing $\pi^2$ from $\pi^1$

Figure 7.4: The np cycle found does not contain $s$

7.1.2 Complexity

A comparison to the nested DFS is difficult because of its detour over LTL verification and the used heuristics. Hence we will relate the resource requirements of the incremental DFS to those of SPIN's safety checks, which is more expressive for our correctness verifications, anyway.
7.1. Without partial order reduction

Since $G_L < G_{L+1}$ for all $L \in \mathbb{N}_{\geq 0}$, we only need a neglible amount of additional memory for DFS\textunderscore starting\_over compared to a safety check (additional memory for the progress\_counter on the stack). If a non-progress cycle can be found for an $L$ s.t. $G_L \leq G$, then the memory requirement will be much smaller than for a complete safety check.

If there are only progress cycles, the partial LTSs will in most cases increase exponentially with the progress\_counter, so in the worst case the complete execution of DFS\textunderscore starting\_over should take about twice the time of the safety check.

This complexity analysis was for the case of starting all over with building a partial LTS after increasing $L$. But this procedure induces about as much redundancy as the nested DFS, namely traversing the LTS twice in the worst case. This redundancy can be avoided by reusing the part of the graph already built in previous runs. But then we have to use extra information to know which transitions have been pruned. The additional memory has approximately the size $|G_{L+1}| - |G_L|$, so the total memory is about the same as if we had had to search until $L + 1$ instead of $L$, i.e. the worst case memory does not change. But the time complexity is about the same as for safety checks and not doubled anymore.

One way to track the pruned transitions is by using a FIFO. Listing 7.4 shows that we do not construct the graph from scratch, but rather use the graph already built, gradually pick the states out of the FIFO, and expand the graph further by continuing the basic DFS. When a new progress state is reached, traversal is postponed by putting the state into the FIFO. When the basic DFS is finished and the FIFO is empty, the complete graph $G$ is built. This algorithm does not know which $G_L$ is currently constructed. If we want to clearly separate the different runs as before, we can use two FIFOs, one for reading and one for writing. When the FIFO that is read from is empty, the current run is finished and we swap the FIFOs for the next run.

\begin{verbatim}
proc DFS_FIFO(s)
    put s in FIFO;
    repeat
        pick first s out of FIFO;
        DFS према ap(check)(s)
    until (FIFO is empty);
end;

proc main()
    DFS_FIFO(init);
    printf("LTS does not contain np cycles");
end;
\end{verbatim}

Listing 7.4: Incremental depth-first search with a FIFO

The original stack gets lost by this algorithm. Therefore, some cycles are not detected anymore. But these undetected cycles go back to states from previous runs, so they are progress cycles anyway. The detectable cycles are exactly the non-progress cycles.
As an additional advantage, we do not need the \texttt{progress} counter anymore. As a disadvantage, we lost the error-path (the original stack) and must content ourselves with the non-progress cycle (the stack from the current run). If we want an error-path, an additional DFS as reachability-test is needed.

The unsettled functions of \texttt{DFS\_prune\_np\_check} have the following specification for \texttt{DFS\_FIFO}:

- \texttt{pruned(s, t)} returns true iff \(t\) is in a progress state.
- \texttt{pruning\_action(t)} puts \(t\) into \texttt{FIFO}.
- \texttt{np\_cycle(t)} always returns true, since checking whether \(t\) is on the stack is sufficient.
- The \texttt{error\_message} can print out the stack, which now corresponds to the path of the \texttt{np} cycle found, but does not contain the path from \texttt{init} to the cycle anymore.

Instead of introducing a FIFO, a simpler version of \texttt{DFS\_FIFO} just put omitted states underneath the stack. The basic DFS will pop them back from the stack when it is time. This results in a simpler algorithm, described in listing 7.5:

```haskell
proc DFS\_Stack (s)
  DFS\_prune\_np\_check (s)
end;

proc main()
  DFS\_Stack (init);
  printf("LTS does not contain np cycles");
end;
```

Listing 7.5: Incremental depth-first search with modified stack

Compared to the general \texttt{DFS\_FIFO}, the unsettled function that have to be changed for \texttt{DFS\_Stack} are \texttt{pruning\_action(t)}, which now puts \(t\) underneath the stack, and the \texttt{error\_message}, which should only print out the part of the stack corresponding to the \texttt{np} cycle found.

\textbf{Note.} We can also compare our incremental DFS with the former \texttt{np} check from SPIN. The old check uses the nested DFS directly and can be found in (Holzmann, 1992) (Holzmann et al., 1996) explains that this algorithm is not compatible with partial order reduction because of condition C3 (cf. subsection 7.2.1). The authors "do not know how to modify the algorithm for compatibility with partial order reduction" and suggest the alternative SPIN is now using. But the algorithms \texttt{DFS\_FIFO} can be regarded as such modification of the old SPIN \texttt{np} check by changing the order of traversal:

Instead of separating the state space creation from the search for a \texttt{np} cycle, the algorithm does both at the same time. To be able to omit the nested DFS it needs to postpone traversing progress transitions, i.e. transitions causing
progress, for the \((L+1)\)-th time until the creation of \(G_L\) is finished and thus also the np cycle check for \(G_L\). Then \(\text{DFS}_{\text{RF}}\) retrieves the postponed transitions and does the combined state space creation and np cycle check for \(G_{L+1} \setminus G_L\). So we can regard the algorithm doing the nested DFS in advance, therefore not needing the outer DFS anymore, since the corresponding part of the state space is already created. Hence \(\text{DFS}_{\text{RF}}\) omits the redundancy, saving space and time, and might avoid the incompatibility with partial order reduction because of C3. As tradeoff, the algorithm does not have an error path from the initial state to the np cycle anymore, only the np cycle itself. For a complete error-path, an additional DFS is required.

### 7.2 With partial order reduction

After introducing SPIN's partial order reduction, we will look into different variants of combining the incremental DFS with the reduction.

#### 7.2.1 Introduction to partial order reduction

One of the main reasons for state space explosion is the interleaving technique to cover all possible executions of the asynchronous product of the system's component automata. These combined executions usually cause an exponential blow-up of the number of transitions and intermediate states.

But often statements of concurrent processes are independent:

\[
\begin{align*}
\alpha, \beta \in S & \text{ are independent if } \forall s \in S : \alpha, \beta \in \text{enabled}(s) \implies \\
& \alpha \in \text{enabled}(\beta(s)) \text{ and } \beta \in \text{enabled}(\alpha(s)) \text{ (Enabledness)} \\
& \text{ and } \alpha(\beta(s)) = \beta(\alpha(s)) \text{ (Commutativity)}
\end{align*}
\]

\[
\alpha, \beta \in S \text{ are dependent if } \alpha, \beta \text{ are not independent}
\]

So the different combinations of their interleaving have the same effect. Partial order reduction (po reduction) tries to select only few of the interleavings having the same result. [Peled, 1994] is the main paper on the type of partial order reduction SPIN utilizes. Some of its results and notations are used by the proof of lemma [7.3]. A shorter and less thorough introduction is (Clarke et al., 1999).

Selecting as few interleavings as possible is done by choosing in each state \(s\) a subset \(\text{ample}(s) \subseteq \text{enabled}(s)\), called the ample set of \(s\). If \(\text{ample}(s) = \text{enabled}(s)\) we say \(s\) is fully expanded. The choice of \(\text{ample}(s)\) must meet the conditions C1 to C3 listed in table [7.1] C3' is a sufficient condition for C3 and can be checked locally in the current state. In order to check conditions C1 and C3', the original graph and the current path are implicit arguments to \(\text{ample}(s)\), sometimes indicated by subscripts: \(\text{ample}_e(s)\), \(\text{ample}_G(s)\), or \(\text{ample}_{e,G}(s)\).
We extend this function to sets of statements $E$ and to complete graphs $G$:

$$ample(E) = ample_*(E) = ample_*(\text{end of } \pi) \cap E$$

the result of partial order reducing $G$:

$$ample(G) =$$

the subgraph of $G$ that is reachable from init and only uses transitions $t = (s \rightarrow s') \in T$ in the corresponding ample set, i.e. $\text{statement}_s(t) \in \text{enabled}(s)$

<table>
<thead>
<tr>
<th>C0: Emptiness</th>
<th>$ample(s) = \emptyset \iff \text{enabled}(s) = \emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: Ample decomposition</td>
<td>No statement $\alpha \in \text{enabled}(s) \setminus approve(s)$ that is dependent on some statement in $ample(s)$ can be executed in the full state graph after reaching the state $s$ and before some statement in $ample(s)$ is executed.</td>
</tr>
<tr>
<td>C2: Invisibility</td>
<td>$ample(s) \neq \text{enabled}(s) \implies \forall \alpha \in \text{ample}(s): \alpha \text{ is invisible, meaning in our case that } \alpha \text{ does not change np.}$</td>
</tr>
<tr>
<td>C3: Cycle closing condition</td>
<td>If a cycle contains a state $s$ s.t. $\alpha \in \text{enabled}(s)$ for some statement $\alpha$, it also contains a state $s'$ s.t. $\alpha \in \text{ample}(s')$.</td>
</tr>
<tr>
<td>C3': NotInStack</td>
<td>$\alpha \in \text{ample}(s)$ and $\alpha(s)$ is on the stack $\Rightarrow \text{ample}(s) = \text{enabled}(s)$</td>
</tr>
</tbody>
</table>

Table 7.1: Constraints on $ample(s)$

**Lemma 7.3 (Non-progress cycle existence preservation).** An LTS $G$ has a non-progress cycle iff $ample(G)$ has a non-progress cycle.

**Proof.** Let $G$ be an LTS and $G' := approve(G)$.

If $G'$ has a non-progress cycle $\pi'$, which is reachable by path $\pi'_0$, then by (Peled, 1994, theorem 3.11.1) the interleaving sequence corresponding to $\pi'_0(\pi'_1)^w$ is also an interleaving sequence in $G$, so $\pi'_0(\pi'_1)^w$ is also a path in $G$. Thus $G$ has a reachable non-progress cycle, too.

If $G$ has a non-progress cycle $\pi_1$ which is reachable by path $\pi_0$, then let $v = v_0(v_1)^w$ be the interleaving sequence corresponding to $\pi_0(\pi_1)^w$. According to (Peled, 1994, theorem 3.11.2) there exists at least one sequence $v'$ that corresponds to some path of $G'$, s.t. $v \preceq v'$. According to $D'$ the dependency relation of statements complying with C3'), i.e. by removing from $v'$ visible statements (which are independent of all the following non-removed statements of $v'$) we can get a sequence $v''$ equivalent to $v$ w.r.t. $D'$. As $\pi_1$ is a non-progress cycle, $v_1$ only has visible statements, so $v$ only has $n \in \mathbb{N}_{\geq 0}$ visible (i.e. not invisible) statements, all in prefix $v_0$. As each visible statement alternates np and $\pi_1$ has only non-progress states, $n$ is even. Because of $v \equiv v''$, $v''$ is an infinite sequence with only $n$ visible statements in some prefix $v_0''$. Adding invisible statements to $v''$ to get $v'$ does not change the number $n$ of visible statements. Hence the path corresponding to $v'$ is infinite and does not make any progress after a finite number of transitions, thus eventually closing a non-progress cycle in $G'$. \qed
Note. So besides the variants of np cycle checks described in the following
section, the trivial DFS technique (cf. section 2.2.1) can also be used together
with po reduction with visible np_.

7.2.2 Overview of the variants

A search for non-progress cycles must consider the variable np_ which SPIN
uses to encode progress states. Hence np_ is visible in the nested DFS. But we
are not verifying an LTL property and are only interested whether np cycles
exist, not in the amount of progress the progress cycles make. So condition
C2, which guarantees that for every behaviour in the original graph there is
an equivalent behaviour modulo stuttering in the po reduced graph, is stronger
than necessary. We can abstract further by only distinguishing whether a cycle
contains no progress at all or the number of progresses is greater than zero.
Is po reduction with invisible np_ sufficient for this? Not in general, because
lemma 4.3 does not hold without visible np_ because the proof does not yield
the desired results with A containing progress transitions: v ≤_A v' only implies
that v' does not have less progress transitions, but it may have more, since
also progress transitions are allowed to be removed from v'. Hence np cycles
in the original graph can be destroyed by inserting progress transitions. Figure
4.4 shows a counterexample to the correctness of np cycle preservation by po
reduction with invisible np_. If the right states are progress states, β₁ and
β₂ change the value of np_. Hence the complete graph contains the np cycle
π := (α₁, α₂). If np_ is invisible, po reduction can restrict the graph to only the
thick transitions, therefore not containing any np cycles. With np_ visible, the
reduced graph must contain π.

![Diagram](image.png)

Figure 7.5: ample(s) without C2 is in general erroneous

But since we can prune progress transitions and as C3' demands regarding all
alternative transitions when closing a progress cycle, we might in some cases be
able to make np_ invisible and thus improve the po reduction.

Since there is no other variable we have to consider while searching for non-
progress cycles, the invisibility of np_ is equivalent to ignoring condition C2.

Another option is the order in which pruning and selection of the ample set
takes place:
If in each state \( s \) we first prune the outgoing transitions and then select \( \text{ample}(s) \), i.e. perform \( \text{ample}(\text{statement}_{\text{prune}}(\text{prune}(s))) \) (in short \( \text{ample}([\text{prune}(s)]) \)), the ample set can change with the complexity limit \( L \). This increases reduction if pruning statements reduces dependencies, and decreases reduction if some of the pruned statements had been selected as ample set.

**Lemma 7.4.** Let the ample set selection be the optimum or SPIN's heuristic, \( s \) be a reachable state and condition C2 be enabled. Then \( |\text{ample}([\text{prune}(s)])| \leq |\text{prune}(\text{ample}(s))| \), and \( |\text{ample}([\text{prune}(s)])| < |\text{prune}(\text{ample}(s))| \) is possible.

**Proof.** Let \( s \) be a reachable state. Since pruned statements are visible, either \( \text{ample}(s) = \text{enabled}(s) \) or those statements are not selected in the ample set, but can add dependencies.

If \( \text{ample}(s) = \text{enabled}(s) \), then we have \( \text{prune}([\text{ample}(s)]) = \text{prune}(s) \), but \( \text{ample}([\text{prune}(s)]) \subseteq \text{prune}(s) \) and for the following example a proper subset of \( \text{prune}(s) \):

\[
\text{enabled}(s) = \{a, b, c\}, \text{only } c \text{ makes progress, } (a, b) \in I, (a, c) \in D, (b, c) \in D. \text{ Therefore } \text{ample}(s) = \text{enabled}(s), \text{ but } \text{ample}([\text{prune}(s)]) = \{a\} \not\subseteq \{a, b\} \text{ is possible, cf. figure 7.4.}
\]

If \( \text{ample}(s) \not\subseteq \text{enabled}(s) \), then because of C2 all \( \alpha \in \text{ample}(s) \) do not make progress. Therefore \( \text{prune}([\text{ample}(s)]) = \text{ample}(s) \) and \( \text{ample}([\text{prune}(s)]) \) can also be chosen to be \( \text{ample}(s) \). SPIN's ample selection heuristic will not choose a worse ample set when visible transitions have already been pruned.

The following example shows that \( \text{ample}([\text{prune}(s)]) \) can sometimes be chosen smaller than \( \text{ample}(s) \):

\[
\text{enabled}(s) = \{a, b, c, d\}, \text{ only } d \text{ makes progress, } (a, d) \in D, (b, c) \in D, \{a, d\} \text{ and } \{b, c\} \text{ are pairwise independent. Then the optimal ample set is } \text{ample}(s) = \{b, c\}, \text{ but } \text{ample}([\text{prune}(s)]) = \{a\} \text{ is smaller (C1 holds because } d \text{ also gets pruned in } s_1 \text{ and } s_2), \text{ cf. figure 7.7.}
\]

So with C2 enabled, \( \text{ample}([\text{prune}(s)]) \) can yield a stronger reduction.

![Figure 7.6: ample([prune(s)]) \not\subseteq prune([ample(s)]) = prune(s)](image)

**Lemma 7.5.** Without condition C2 \( |\text{prune}([\text{ample}(s)])| \) and \( |\text{ample}([\text{prune}(s)])| \) are independent.

**Proof.** Let \( s \) be a reachable state. Without condition C2 pruned statements can also be selected in the ample set.
7.2. With partial order reduction

![Graph Diagram]

Figure 7.7: \( \text{ample}(\text{prune}(s)) \subseteq \text{prune}(\text{ample}(s)) = \text{ample}(s) \)

The following is an example for \( |\text{prune}(\text{ample}(s))| < |\text{ample}(\text{prune}(s))| \): Let \( \text{enabled}(s) = \{a, b\} \) and only \( b \) makes progress. The optimum ample selection and SPIN’s ample selection heuristic can choose \( \text{ample}(s) = \{b\} \). Then \( \text{prune}(\text{ample}(s)) = \emptyset \) but \( \text{ample}(\text{prune}(s)) = \{a\} \) because of C0.

An example for \( |\text{ample}(\text{prune}(s))| < |\text{prune}(\text{ample}(s))| \) is given in figure 7.9.

A third option is the choice of implementation: To be able to check C3’, we need the complete stack, which gets lost using DFS_FIFO. Figure 7.8 shows an example using \( \text{prune}(\text{ample}(s)) \) in which C3’ does not hold because the loop back to the previous, lost stack is not recognized. So if we are using \( \text{prune}(\text{ample}(s)) \) without C2 with the implementations DFS_FIFO, np cycles can get lost. Hence the implementation DFS starting over has to be chosen for \( \text{prune}(\text{ample}(s)) \) without C2. Using \( \text{prune}(\text{ample}(s)) \) with C2 guarantees considering all statements of a progress cycle anyway, so C3 already holds for that cycle. This observation shows a likely improvement for all incremental DFS algorithms with C2 enabled by ignoring condition C3’. If a np cycle is being closed, the algorithm has found an error anyway. If a progress cycle is being closed, C2 implies C3.

![Graph Diagram]

Figure 7.8: Error with DFS_FIFO and C3’
Using ample(prune(s)), condition C2 is being ignored with the implementations DFS_FIFO, since the ample selection algorithm only sees the pruned set without progress transitions, anyway. The progress transitions are added to the ample set when reconsidering the postponed states. Thereafter C2 does not hold anymore. So we have the same algorithm as without C2. The alternative of expanding the ample set to the complete enabled set to ensure C2 when adding progress transitions is probably too inefficient: Every state with an outgoing progress transition would have no po reduction at all.

If ample(s) instead of ample(prune(s)) would not select any progress transitions, the variant above simply adds progress transitions to the ample set. As this is done by SPIN's implementation of po reduction too, it is probably not a problem, even though the additional transitions and the order of traversal can alter the graph (e.g. in figure 7.16).

If ample(s) instead of ample(prune(s)) would select a progress transition, C2 causes full expansion of s, so DFS_FIFO is probably incorrect in this case.

So the correctness proofs become very complex because of the alternations of the graphs as in figure 7.16

### 7.2.3 Variants with prune(ample(s))

This variant is especially interesting if we can ignore C2. Figure 7.9 shows an example that this combination is not correct if SPIN's implementation of progress is used, because for statement $\beta$ the property of being a np cycle depends on the state the statement $\beta$ starts from. So if ample(s_1) = \{a_1\}, no np cycle will be found. The variant ample(prune(L(s))) is correct for this automaton since $a_1$ is pruned for the appropriate $L$ and then ample(prune(s_1)) = \{\beta\}. Because of this error, we will investigate alternative types of progress in the following section.

![Figure 7.9: An incorrect combination of options](image)

**Semantics of progress**

SPIN's semantics of progress entail several problems:

- They are not equal to the miniworld, since progress is made by some action, which corresponds to a transition and not to a state. For example,
if the action corresponding to the transition from $s_2$ to $s_1$ makes progress, PROMELA will model $s_2$ as progress state, so the cycle in the bottom of the global automaton in figure 7.10 is considered a progress cycle, although the system does not make any progress.

- Because of the asynchronous product, the global automaton has a lot of progress states, e.g. when a process stays in a local progress state for a long time. This makes the np cycle checks inefficient:
  - The incremental DFS algorithm DFS\_{starting \_over} has to start over much more often and needs a bigger progress\_counter.
  - The algorithm DFS\_{FIFO} needs a bigger FIFO and more time for memory management.
  - In all our algorithms as well as in SPIN the visibility condition C2 decreases the reduction's efficiency.

- The strongest po reduction variant for $\text{prune(ample(s))}$ without C2 is erroneous, as we have seen in the counterexample in figure 7.9

\[
\text{process 1} \times \text{process 2} = \text{global system}
\]

\[
\begin{array}{c}
\text{process 1} \\
\begin{array}{c}
S_1 \\
\text{progress}
\end{array}
\end{array}
\times
\begin{array}{c}
\text{process 2} \\
\begin{array}{c}
\text{transitions } s \rightarrow t \text{ switching np from false to true (e.g. implemented by increasing a progress\_counter the moment a progress transition is taken and by setting prune(s,t) to true iff progress\_counter=L and s \rightarrow t is a progress transition), avoiding all deficiencies from above:}

- We can tag as progress exactly those transitions corresponding to system activities making progress.
- The visible transitions are exactly the progress transitions. Thus we already have only half as many visible transitions as with the old semantics, which requires two visible transitions per local progress state: one for entering and one for exiting. If there are many transitions into a local progress state, the savings increase.

So using progress transitions is similar to po reduction since they can
be regarded as entering and exiting progress states atomically (cf. implementation with atomic{ ... } below). Therefore we made reduction by considering out of many combinations only representative paths w.r.t. the property we are interested in, i.e. whether a cycle makes progress.

- The counterexample of figure 7.9 for prune(ample(s)) without C2 translates to figure 7.11 for progress transitions. Po reduction now works correctly, since the property of being a np cycle solely depends on the transitions taken, i.e. $\beta$ is always a np cycle, independent of the state. If $\beta \in \text{ample}(s_1)$, the np cycle will be detected already from $s_1$. Otherwise $\beta \in \text{ample}(s_2)$ because of condition C3'.

![Diagram showing the combination is correct](image)

**Figure 7.11: Now the combination is correct**

The fundamental implementation of progress transitions is to change the semantics of PROMELA such that a progress label marks the following statement as a progress transition.

If we do not want to change the PROMELA semantics, we can use the construct $d_{\text{step}}\{...\}$ to define a progress transition. Instead of "statement; progress: statement;" we now use "statement; $d_{\text{step}}\{\text{skip; progress: statement;}\}". Figure 7.12 shows the difference in the automata: The progress moves from state $s$ to the following transition.

![Diagram showing progress transition with $d_{\text{step}}$](image)

**Figure 7.12: Progress transition with $d_{\text{step}}$**

Alternatively we can use "statement; $d_{\text{step}}\{\text{skip; progress: skip; statement;}\}" which splits $s$ into two states $s_1$ and $s_2$ and makes a progress transition between them, as depicted in figure 7.13.

We have moved the progress to an artificial transition just before statement;.

The np cycle check is still correct: The additional transition only gives more possibilities, so the only error could be false negatives. But every cycle in the new global automata using statement; must also use the progress transition to
return to \( s_2 \), so there are no additional np cycles. This solution for progress transitions adds one extra state, but makes the specification of progress easier. Using "\#define progress\_step(progressID)\{d\_step\{skip; progress; skip; \}\}" we can simply write "\( \text{statement}_i; \text{progress\_step}(progressID); \text{statement}_j \)". The parameter progressID is necessary, because expanding the macro more than once results in repeating identical labels, which PROMELA forbids within one process. If we do not want to bother with progressIDs, we can make sure the progress label only appears once by defining the alternative from listing 7.6 starting one instance of \( \text{progress\_proc}() \) at the start, and using "\( \text{statement}_i; \text{progress\_step0}; \text{statement}_j \)". In \( \text{progress\_proc}() \) we need to synchronously receive a message to perform a handshake with the process doing the progress\_step. This transition can be used to enter the progress state, so we can omit the first skip.

Experiments have shown that the introduction of this additional and synchronizing process does not amortize.

\[
\begin{align*}
\text{chan} & \quad \text{progress\_step} = [0] \text{ of } \{ \text{bit} \}; \\
\text{proctype} & \quad \text{progress\_proc}( ) \\
& \quad \{ \\
& \quad \quad \text{endStateIsValid}; \\
& \quad \quad \text{do} \\
& \quad \quad \quad \text{d\_step}\{ \text{progress\_step}; \text{progress}\_skip \} \\
& \quad \quad \text{od} \\
& \quad \} \\
\end{align*}
\]

Listing 7.6: Progress process

Because of the mentioned benefits, using \( d\_step \) constructs might also improve the np cycle check implemented by the nested DFS. But as the complete sequence within a \( d\_step \) is processed in one transition, and the Büchi automaton only considers states, the progress will be ignored. SPIN detects this and gives the warning "\( \text{progress label inside } d\_step - \text{is invisible} \)". But we can use "\( \text{atomic}(...) \)" instead of "\( d\_step \{ \ldots \} \)". This also guarantees that the progress state is left immediately after it was entered. Thus the amount of global progress states is still greatly reduced. Besides adding a redundant state, this solution is equivalent to the usage of progress transitions, hence showing that no reduction with \( C2 \) remains correct when using progress transitions instead of progress states. (Holzmann, 2004) chapter 7) states that the PROMELA semantics engine realizes atomicity by checking within the function executable()
whether the variable exclusive equals p.pid, which could be compatible with Büchi automata and SPIN’s progress techniques, i.e. using the variable np_.

But unfortunately, these progress transitions with the help of atomic is erroneous in SPIN. For example if we change line 28 in \( A_{\text{timi}} \) (cf. section 5.7.6) to ";:: else \to atomic\( \{ \text{skip; progress; skip} \} \)\';, SPIN claims to find a np cycle, but returns a trace which has a progress cycle. As this is also the case with 2o reduction and statement merging disabled, the reason must really be that SPIN does not interleave atomic sequences with the never claim, as is stated in [Gerth, 1997].

Using the other variants "atomic\( \{ \text{skip; progress; statement; } \} \); or progress\( \_poc() \) or \#define progress\( \_p\text{step}(\text{progressD}) \) for our algorithms either results in the error from above or gives the warning "progress label inside atomic - is invisible", which confirms the conclusion from above. SPIN’s inconsequent warning suggests that it does not always detect progress labels inside atomic.

So using progress transitions for the incremental DFS does not only solve the incorrectness of figure 7.9 it can also significantly reduce the number of visible transitions. Therefore we will prefer progress transitions.

Correctness

Correctness proofs would be easy with enabled condition C2 if pruning and ample set selection were independent, i.e. if using \texttt{prune}(ample(s)) in each state s on-the-fly led to the same graph as \texttt{prune}(ample(G)), which first completely po reduces the graph and only then prunes it. We could first use lemma 7.2 to show that ample(G) would have a np cycle if G had one. Therefore by proposition 7.2 the incremental DFS would find one in \texttt{prune}(ample(G)).

Unfortunately, pruning and ample set selection do interact. Figure 7.14 depicts that \texttt{prune}(ample(G)) leads to a different graph than \texttt{prune}(ample(s)): Pruning transitions before the complete graph is po reduced can change the path from which a state s is visited. C3’ can then cause a different ample set selection, which alters the graph.

Of course we can sacrifice on-the-fly model checking and perform two runs. The first to build ample(G), saving for every state its ample set (i.e. the number of the process whose statements are selected). Afterwards we can use this information to select the correct ample sets for \texttt{prune}(ample(G)) in the second run.

\textbf{On-the-fly model checking without condition C2.} Unfortunately, this variant is erroneous: If ample(s) without C2 does not contain any np cycles, pruning the graph will only provide a np cycle if the pruning alters the graph as in figure 7.14. But the graph in figure 7.13 is not altered by pruning. As \( \beta_2 \) is a progress transition, it is a counterexample to the correctness of \texttt{prune}(ample(s)) without C2.
On-the-fly model checking with condition C2. The counterexample from above without C2 cannot happen when C2 is enabled, as this guarantees that alternatives to progress transitions are considered, if possible.

The dependency of pruning and ample set selection prevents using lemma 7.6. The following lemma [7.6] shows that a main part of the correctness proof for po reduction (cf. [Clarke Jr. et al., 1999] section 10.6) can be lifted to our case, but becomes very complicated. We first give the following definitions:

Let $G$ be an LTS, $N := \{s \in G | s$ is on a np cycle of $G\}$, $L_0$ be the smallest number of progressions on any path of $G$ reaching $N$, and $G'$ be the graph built by the DFS with $\text{prune}_{L_0}(\text{ample}(s))$. In the rest of this section we need to reference both states and statements of paths, so a path $\rho$ will contain its states as well as the statements executed between them (resulting in the transitions): $\rho = r_0 \xrightarrow{\rho_0} r_1 \xrightarrow{\rho_1} \ldots$. A path $\rho$ of length $n \in \mathbb{N}_{\geq 0}$ and a path $\sigma = s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} \ldots$ can be concatenated if $r_n = s_0$: $\rho \circ \sigma := r_0 \xrightarrow{\rho_0} r_1 \xrightarrow{\rho_1} \ldots \xrightarrow{\rho_n} s_n \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} \ldots$. A path $\rho$ reaches $N$ if a state of $\rho$ is in $N$. A path $\sigma$ in $G$ starting from a state $s_0 \in G'$ is called a trial from $s_0$ if $\sigma$ reaches $N$. $s_0$ has a trial if there exists a trial from $s_0$. The notion of trials suggests what we are trying to do: to find a path in $G'$ starting in $s_0$ to some state in $N$ with the help of the trial $\sigma$. For this we will construct a sequence $\mu_0 \circ \sigma_0 \circ \pi_0, \mu_1 \circ \sigma_1 \circ \pi_1, \ldots$ of paths such that all $\mu_i$ lie in $G'$ and start in $r_i = \text{init}$, all $\sigma_i$ are trials that cannot be reduced (i.e. they are free of cycles and end as soon as they reach $N$), all $\pi_i$ are np cycles of $G$, and $|\sigma_{i+1}| \leq |\sigma_i|$. For this construction, we state the following lemma with a sketch of its proof:

**Lemma 7.6 (Trial monotony).** If there exists a trial $\sigma^i$ of length $n$ from $s_0$, such that the number of progress transitions on $\sigma^i$ plus the number of progresses the DFS made to reach $s_0$ is at most $L_0$, there exists a statement $\alpha \in \text{prune}_{L_0}(\text{ample}(s_0^i))$ so that a trial $\sigma^{i+1}$ of length at most $n$ exists from $\alpha(s_0^i)$ with $\alpha \circ \sigma^{i+1}$ having at most as much progress transitions as $\sigma^i$. 

![Figure 7.14: prune(ample(G)) ≠ prune(ample(s))](image-url)
Sketch of the proof. Let \( \pi^t = p_0^t \rightarrow p_1^t \rightarrow p_2^t \rightarrow \ldots \rightarrow p_{n-1}^t \rightarrow p_n^t \) be a np cycle of \( G \) starting in \( s_0^0 \), the end of \( \sigma^t \). Because of the precondition that the sum of progresses is at most \( L_0 \), one of the following two cases are met:

- The number of progresses the DFS made to reach \( s_0^0 \) is smaller than \( L_0 \). Hence progress transitions are not pruned in \( s_0^0 \) and \( \text{prune}_{L_0}(\text{ample}(s_0^0)) = \text{ample}(s_0^0) \).
- \( \sigma^t \circ \pi^t \) does not contain progress transitions.

So in both cases a transition from \( \sigma^t \circ \pi^t \) is in \( \text{prune}_{L_0}(\text{ample}(s_0^0)) \) iff it is in \( \text{ample}(s_0^0) \).

We have to prove the claim for three cases:

**Case basic:** If the statement \( \sigma_0^t \) is in \( \text{ample}(s_0^0) \), we can choose it and the trial \( \sigma^t+1 = s_1^0 \rightarrow \ldots \rightarrow \sigma^t n \rightarrow s_n^0 \) (which has length \( n-1 \)). Since \( \alpha \circ \sigma^t+1 = \sigma^t \), the number of progress transitions does not change. We call \( \sigma^t+1 \) the basic successor of \( \sigma^t \).

**Case independent:** If \( \sigma_0^0 \not= \text{ample}(s_0^0) \) and there does not exist \( \in \mathbb{N} \) with \( \sigma^t \circ \pi^t_0 \in \text{ample}(s_0^0) \), we choose an arbitrary statement \( \alpha \in \text{ample}(s_0^0) \). Condition C2 implies that \( \alpha \) is not a progress transition. Because of C1, all statements of \( \sigma^t \) and \( \pi^t \) are independent of \( \alpha \). Hence the path \( \sigma^{t+1} := \alpha(s_0^t) \rightarrow \alpha(s_1^t) \rightarrow \ldots \rightarrow \alpha(s_n^t) \) has length \( n \) and leads to \( N \), as \( \alpha(p_0^t) \rightarrow \pi^t_0 \rightarrow \alpha(p_1^t) \rightarrow \ldots \rightarrow \pi^t_{n-1} \rightarrow \alpha(p_n^t) \) is a np cycle, cf. figure 7.13. \( \sigma^{t+1} \) may contain cycles, so we first remove all of them before assigning the path to \( \sigma^t+1 \), resulting in a trial of length \( \leq n \). \( \sigma^t+1 \) has at most as much progress transitions as \( \sigma^t \), hence \( \alpha \circ \sigma^t+1 \) does so too. We call \( \sigma^t+1 \) the independent successor of \( \sigma^t \).

**Case confluent:** If \( \sigma_0^0 \not= \text{ample}(s_0^0) \) but there exist indices \( \in \mathbb{N} \) with \( \sigma^t \circ \pi^t_j \in \text{ample}(s_0^0) \), let \( j \in N \) be the smallest one. We select the statement \( \alpha := (\sigma^t \circ \pi^t_0) \) which is independent of \( \sigma^t \circ \pi^t_0 \), \( \ldots \), \( \sigma^t \circ \pi^t_{j-1} \) because of C1. If \( j < n \), \( \alpha \) is from \( \sigma^t \) and the path \( \alpha(s_0^t) \rightarrow \alpha(s_1^t) \rightarrow \ldots \rightarrow \alpha(s_{j-1}^t) \rightarrow \alpha(s_j^t) \rightarrow \alpha(s_{j+1}^t) \rightarrow \ldots \rightarrow \alpha(s_{n-1}^t) \rightarrow \alpha(s_n^t) \) has length \( n-1 \) and leads to \( N \). We get \( \sigma^t+1 \) by removing all cycles from this path, resulting in a trial of length at most \( n-1 \).

If \( j \geq n \), \( \alpha \) is from \( \pi^t \). So the path \( \sigma^{t+1} := \alpha(s_0^t) \rightarrow \alpha(s_1^t) \rightarrow \ldots \rightarrow \alpha(s_{j-1}^t) \rightarrow \alpha(s_j^t) \rightarrow \ldots \rightarrow \alpha(s_{n-1}^t) \rightarrow \alpha(s_n^t) \) has length \( n \) and leads to \( N \), as \( \alpha(p_0^t) \rightarrow \pi^t_0 \rightarrow \alpha(p_1^t) \rightarrow \ldots \rightarrow \pi^t_{j-1} \rightarrow \alpha(p_j^t) \rightarrow \pi^t_{j+1} \rightarrow \ldots \rightarrow \alpha(p_{j-n}^t) \rightarrow \pi^t_{j-n+1} \rightarrow \ldots \rightarrow \alpha(p_n^t) \) is a np cycle. We again get \( \sigma^t+1 \) by removing all cycles from \( \sigma^{t+1} \), hence \( \sigma^t+1 \) has length \( n \).

In both cases \( \sigma^t+1 \) has at most as much progress transitions as \( \sigma^t \). \( \alpha \) is not a progress transition because of C2. \( \sigma^t+1 \) is denoted the confluent successor of \( \sigma^t \).

So for each trial there is the possibility to advance on \( G \). Unfortunately, a sequence of trials might have to be aborted: As we did not make any statement about the order in which the transitions of an ample set are traversed, one of \( \pi^t_i \)'s
states $s$ can already be in the hash table. As the path that reached $s$ the first time can contain more progress states, $N$ might not be reachable by traversing $L_0$ progress transitions. So we have to increase $L$. If $N$ is reached after exactly $L$ progresses, a np cycle is found. This can be proved using a combination of theorem \[\la\] and lemma \[\lb\].

As a change in $L$ can modify the graph (similar to figure \[\la\]), $N$ might not be reachable for some $L$, but reachable for $L + 1$ with less than $L + 1$ progresses. Figure \[\la\] displays an example. The state $s \in N$, the destination of $\alpha_{13}$, can be reached for $L = 3$ after one progress transition, but is unreachable for $L < 3$. This is because for $L = 3$, the path with the progress transitions $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$ is traversed, and from the last state $s'$ only the transition $\alpha_4$, not $\alpha_8$. After backtracking to $init$, the path $\langle \alpha_5, \alpha_6 \rangle$ is aborted, because $s'$ is already in the hash table. Thereafter the path $\langle \alpha_{10}, \alpha_{11}, \alpha_{13} \rangle$ is traversed, reaching $s$. But for $L < 3$ the path $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$ is pruned, so $s'$ is reached through $\langle \alpha_5, \alpha_6 \rangle$. Then $\alpha_8$ can be traversed, and $\alpha_{13}$ deleted by po reduction. When finally the path $\langle \alpha_{10}, \alpha_{11} \rangle$ is traversed, it is aborted in $s$ because the state is already in the hash table.

But the smaller np cycles in the figure show that with each prevented path to a np cycle, there are alternative np cycles because of the conditions for po reduction. So the on-the-fly incremental DFS with po reduction and enabled condition C2 using pruneL(ample($s$)) is probably correct, but a proof thereof likely very complicated.

Complexity

The on-the-fly incremental DFS with po reduction and enabled condition C2 using pruneL(ample($s$)) will not necessarily find a np cycle for $L_0$, the minimal number of progresses on any path of $G$ reaching a state of a np cycle. Hence a complexity estimation cannot be stated easily.
Figure 7.16: The simplest counterexample for reduction of $L$

7.2.4 Variants with ample(prune(s))

Correctness

Pruning and ample set selection are dependent for ample(prune(s)), too. If they were independent and $G$ had a non-progress cycle, we could first use proposition 7.2 to show that there exists a smallest $L_0 \in \mathbb{N}_{>0}$ s.t. $G_{L_0}$ has a non-progress cycle if $G$ has one. Therefore by lemma $\text{prune}(G')_{L_0} = \text{ample}(G_{L_0})$ also would have one, and all $G'_{L} = \text{ample}(G_L)$ with $L < L_0$ would not.

Figure 7.11 shows that using ample(prune(s)) leads to a different graph than ample(prune(G)): Deleting transitions by po reduction before the complete graph is pruned, e.g. $a_1$ in init, can change the path from which a state is visited. If this causes a change in the number of progresses until $s$ is reached, the behaviour of pruning in $s$ can also change.

Sacrificing on-the-fly model checking and performing two runs, as in the case of prune(ample(s)), would first build part of the graph prune(G) and only
afterwards po reduce it. This does not make sense, as po reduction ought to restrict the needed memory by only considering the reduced graph.

![Diagram](image)

Figure 7.17: ample(prune(s)) \( \neq \) ample(prune(G))

The on-the-fly incremental DFS with po reduction using ample(prune(s)) might be correct with both C2 enabled or disabled. As in the case of prune(ample(s)), a proof of the correctness is probably very complicated.

Complexity

If the on-the-fly incremental DFS with po reduction and disabled condition C2 using ample(prune(s)) is correct, it is a very efficient np cycle check, since it does not restrict the strength of po reduction compared to the safety checks. Using DFS\text{Starting over}, we do not need more space than the safety check. Table 7.2 uses the inequations

\[
\text{Time}(\text{DFS}_{\text{Starting over}}) \leq (L_0 + 1) \cdot \text{Time}(\text{DFS}_{\text{safety}}) \leq \text{depth} \cdot \text{Time}(\text{DFS}_{\text{safety}})
\]

to give a very pessimistic estimation on the time requirements, illustrated with \( A_{\text{timing}} \). The safety checks and SPIN’s np cycle checks are listed for comparison, too.

<table>
<thead>
<tr>
<th>Prob. Size</th>
<th>DFS\text{nested, safety}</th>
<th>DFS\text{nested, np cycle test}</th>
<th>DFS\text{lat, over}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>depth</td>
<td>states</td>
</tr>
<tr>
<td>3</td>
<td>5&quot;</td>
<td>33</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>5&quot;</td>
<td>40</td>
<td>103</td>
</tr>
<tr>
<td>5</td>
<td>5&quot;</td>
<td>47</td>
<td>148</td>
</tr>
<tr>
<td>6</td>
<td>5&quot;</td>
<td>54</td>
<td>201</td>
</tr>
<tr>
<td>7</td>
<td>5&quot;</td>
<td>61</td>
<td>262</td>
</tr>
<tr>
<td>254</td>
<td>100°</td>
<td>1790</td>
<td>260353</td>
</tr>
</tbody>
</table>

Table 7.2: Estimation for DFS\text{Starting over}
7.3 Summary

Without partial order reduction, the correctness of the incremental DFS was proved. With po reduction enabled, there are several variants: \texttt{prune(ample(s))} working on-the-fly without condition C2 is erroneous. The conjecture of correctness for most variants is probably true, the proofs are likely to be difficult. Figure 7.18 summarizes the variants for on-the-fly verifications with po reduction.

For \texttt{prune(ample(s))} sacrificing on-the-fly model checking might result in a practical verification.

![Figure 7.18: Summary of variants](image)

In the optimal case of po reduction not being decreased, e.g. for \texttt{ample(prune(s))} and disabled condition C2, proving the correctness for our leader election algorithms would work up to the same problem size as the safety checks. Even with the requirement of starting over and over again, the runtime for the np cycle check by DFS\texttt{starting over} requires less than 50 hours for the maximum of \(n=254\) nodes, using a very conservative estimation.

Another improvement is using progress transitions instead of progress states, as this causes a first reduction and precisely models the miniworld.
Chapter 8

Closure

8.1 Summary

This thesis investigated six randomized leader election algorithms for anonymous, unidirectional rings:

- the two algorithms $A_{Fokkink, roundtrip}$ and $A_{Fokkink, noroundtrip}$ given in [Fokkink and Pang, 2004], which are derivatives of $A_{Itai}$ from (Itai and Rodhe, 1981),
- two new derivatives of $A_{Itai}$, namely $A_{Itai, round \leq K}$ and $A_{Itai, round \mod K}$,
- and two completely different, contrary algorithms $A_{sorting}$ and $A_{timing}$.

Since all but the last two algorithms have many similarities, the possible variations and their dependencies have been presented in chapter 3.

For the constructable problem instances the correctness of all algorithms was proved with PRISM. Its useful capability of computing probability values and expectations has shown that, besides $A_{Fokkink, roundtrip}$, each of our algorithms has its strength:

- $A_{Itai, round \leq K}$ works with message overtaking,
- $A_{Itai, round \mod K}$ has very high best case probability values for electing a leader within a given number of steps,
- $A_{Fokkink, noroundtrip}$'s worst case probabilities are the highest within the class $A_{compare IDs}$,
- $A_{sorting}$ has the best communication complexity for the problem sizes allowing comparison between all our algorithms,
- $A_{timing}$'s average case expected probabilities are very high.
Furthermore, several lemmata from chapter 7 were supported by verifications. Of course all concluded behaviours are only valid for the constructable problem instances, i.e. must be used with care for the general case.

Because of the low limits, the alternative SPIN was considered. After investigating how to verify our randomized algorithms with the non-probabilistic model checker, the correctness of the algorithms has been proved with this tool as well. For some problems SPIN was able to raise the limits. After several optimizations the improvements for the safety checks of $A_{\text{liming}}$ were even strong enough to avoid state space explosion altogether.

The non-progress cycle checks of SPIN cause the limitations of the correctness proofs, i.e. entail memory overflow for much smaller problem instances than the safety checks. Hence an alternative to SPIN's implementation by the nested depth-first search was proposed in the previous chapter: the incremental depth-first search. Several variants were investigated, their applicability is an open question.

### 8.2 Conclusions

We can now come back to the initial question from section 8.1 whether model checking avoids the requirement of mathematical expertise and supervision, as is needed by deductive proving. The answer is "no" if we do not want to restrict ourselves to only tiny problem instances: Although the process of model checking a property against a specific model is fully automatic, mathematical expertise is required for specifying all properties of interest and an optimized model that does not yield memory overflow already for tiny problem instances. If this optimization is done iteratively, user supervision while model checking is also needed.

Model checking behaves dually to theorem proving in several aspects, such as decidability, counter-example generation, and allowance of infinite state systems. This gives both areas their legitimacy and suggests combining them, as is being done in several current research projects.
Appendix A

PRISM specifications

A.1 Algorithm $A_{Itai, round \leq k}$

Model specification for $n=3, M \in \{2, 3\}$:

```plaintext
module model
    active : bool init true;
    round : [0..K];
    id : [1..M];
    state : [0..4];
    mesl_round : [0..K];
    mesl_id : [1..M];
    mesl_hop : [1..N];
    mesl_unique : bool;

//create message
[1] (state = 0) ->
    Prob: (state' = 1) & (id' = 1) & (mesl_id' = 1) & (mesl_hop' = 1)
    & (mesl_unique' = true) & (mesl_round' = round + 1) +
    Prob: (state' = 1) & (id' = 2) & (mesl_id' = 2) & (mesl_hop' = 1)
    & (mesl_unique' = true) & (mesl_round' = round + 1) +
    Prob: (state' = 1) & (id' = 3) & (mesl_id' = 3) & (mesl_hop' = 1)
    & (mesl_unique' = true) & (mesl_round' = round + 1);

//send message
[1] state = 1 -> (state' = 2) & (mesl_unique' = false)
    & (mesl_id' = 1) & (mesl_hop' = 1) & (mesl_round' = 0);

//receive message
[1] state = 2 & (Buffer3_id > 0) ->
    (state' = 3) & (mesl_round' = Buffer3_round) &
    (mesl_id' = Buffer3_id) & (mesl_hop' = Buffer3_hop) &
    (mesl_unique' = Buffer3_unique);
```

// message checks:
// incoming message > pl -> pl becomes passive
[] (state1 = 3) & (mesl_hop < N) & ((mesl_round > round1) |
  (mesl_round = round1) & (mesl_id > id1)) ->
  (state1' = 1) & (round1' = mesl_round) & (id1' = mesl_id) &
  (active1' = false) & (mesl_hop = mes1_hop + 1);

// incoming message equal pl -> unique bit dirty
[] (state1 = 3) & (mesl_hop < N) & (mesl_round = round1) &
  (mesl_id = id1) ->
  (state1' = 1) & (mesl_unique = false) &
  (mesl_hop = mesl_hop + 1);

// incoming message < pl -> message purged
[] (state1 = 3) & ((mesl_round < round1) | (mesl_id < id1)) &
  (mesl_unique = false) & (mesl_hop = 0) & (round1 = round1 - 0);

// incoming message roundtrip and active and not unique -> restart
[] (state1 = 3) & (round1 < K) & (mesl_hop = N) & (active1) &
  (mesl_unique = false) ->
  (state1 = 0) & (round1 = round1 + 1);

// incoming message roundtrip and active and unique -> LEADER
[] (state1 = 3) & (round1 = K) & (mesl_hop = N) & (active1) &
  (mesl_unique = true);

// selfloop for leader
[] (state1 = 4) -> true;

endmodule

module channel1
  Buffer1_round : [0..K];
  Buffer1_id : [0..M];            // 0 - empty buffer
  Buffer1_hop : [1..N];
  Buffer1_unique : bool;

// node1 into Buffer1
  [pl0 < 0] true ->
  (Buffer1_hop = mesl_hop) &
  (Buffer1_round = mesl_round) &
  (Buffer1_id = mesl_id) &
  (Buffer1_unique = mesl_unique);

// node2 out of Buffer1
  [cl0 < 2] (Buffer1_id > 0) ->
  (Buffer1_round = 0) &
  (Buffer1_id = 0) &
  (Buffer1_unique = false);

endmodule

module node2-node1 [active1 = active2, round2 = round21, id1 = id2, state2 = state2, mes2_round = mes2_round, mes2_hop = mes2_hop, mes2_unique = mes2_unique, Buffer3_round = Buffer1_round, Buffer3_id = Buffer1_id, Buffer3_hop = Buffer1_hop, Buffer3_unique = Buffer1_unique]

endmodule

module channel2-channel1 [Buffer1_round = Buffer2_round, Buffer1_id = Buffer2_id, Buffer1_hop = Buffer2_hop, Buffer1_unique = Buffer2_unique]

endmodule
A.2. Algorithm $A_{Itai, round \ mod \ K}$

Model description for $n=3$, $K=2$, $M \in \{2, 3\}$:

```plaintext
 module node3 - node3 [active1 = active3, round1 = round3, 
 id1 = id3, state1 = state3, mes1_round = mes3_round, 
 mes1_hop = mes3_hop, mes1_unique = mes3_unique] 
endmodule

 module channel3 - channel1 [Buffer1_round = Buffer3_round, 
 Buffer1_id = Buffer3_id, Buffer1_hop = Buffer3_hop, 
 Buffer1_unique = Buffer3_unique, pl1_to_c1 = pl3_to_c3, 
 cl1_to_p2 = cl3_to_p3, mes1_round = mes3_round, 
 mes1_id = mes3_id, mes1_hop = mes3_hop, mes1_unique = mes3_unique] 
endmodule
```

A.2 Algorithm $A_{Itai, round \ mod \ K}$

Model description for $n=3$, $K=2$, $M \in \{2, 3\}$:

```plaintext
 module node1
 active1 : bool init true;
 phase : [0..K];
 id1 : [1..M];
 state1 : [0..4];
 mes1_round : [0..K];
 mes1_id : [1..M];
 mes1_hop : [1..N];
 mes1_unique : bool;

 //create message
 [ state1 = 0 ] ->
 Prob: (state1 = 1) & (id1 = 1) & (mes1_id = 1) & (mes1_hop = 1) 
 & (mes1_unique = true) & (mes1_round = round1) +
 Prob: (state1 = 1) & (id1 = 2) & (mes1_id = 2) & (mes1_hop = 1) 
 & (mes1_unique = true) & (mes1_round = round1) +
 Prob: (state1 = 1) & (id1 = 3) & (mes1_id = 3) & (mes1_hop = 1) 
 & (mes1_unique = true) & (mes1_round = round1); 

 //send message
 [ pl1_to_c1 ] (state1 = 1) -> (state1 = 2) & (mes1_unique = false)
 & (mes1_id = 1) & (mes1_hop = 1) & (mes1_round = 0);

 //receive message
 [ c3_to_p1 ] (state1 = 2) & (Buffer3_id > 0) ->
 [ state1 = 3] & (mes1_round = Buffer3_round) & 
 (mes1_id = Buffer3_id) & (mes1_hop = Buffer3_hop) & 
```

149
[mes1_unique→Buffer3_unique];

// message checks:
// incoming message > p1 → p1 becomes passive
[] [state1 = 3] & (mes1_hop < N) & (mes1_round ! round1) |
[mes1_id = id1] -> [state1 = 1] &
[round1 ′ = mes1_round] & [id1 ′ = mes1_id] &
[active1 ′ = false] & [mes1_hop′ = mes1_hop + 1];

// incoming message equal p1 → unique bit dirty
[] [state1 = 3] & (mes1_hop < N) & (mes1_round = round1) &
[mes1_id = id1] ->
[state1 = 1] & [mes1_unique ′ = false] &
[mes1_hop′ = mes1_hop + 1];

// incoming message < p1 → message purged
[] [state1 = 3] & (mes1_id < id1) &
[mes1_round = round1] -> [state1 = 2] & [mes1_id = 1] &
[mes1_unique ′ = false] & [mes1_hop = 1] & [mes1_round = 0];

// incoming message roundtrip and active and not unique → restart
[] [state1 = 3] & (mes1_hop = N) & [active1] &
[mes1_unique ′ = false] -> [state1 = 0] & round1 ′ = func(mod, round1 + 1, K);  

// incoming message roundtrip and active and unique → LEADER
[] [state1 = 3] & (mes1_hop = N) & [active1] &
[mes1_unique] -> [state1 = 4];

// selfloop for leader
[] [state1 = 4] -> true;

endmodule

dmodule channel1
Buffer1_round : [0..K];
Buffer1_id : [0..M];        // 0 = empty buffer
Buffer1_hop : [1..N];      // 0 = empty buffer
Buffer1_unique : bool;

// node1 into Buffer1
[pl | <c1>] true -> [Buffer1_hop ′ = mes1_hop] &
(Buffer1_round ′ = mes1_round) & [Buffer1_id ′ = mes1_id] &
(Buffer1_unique ′ = mes1_unique);

// node2 out of Buffer1
[cl | c2] Buffer1_id > 0 -> [Buffer1_round ′ = 0] &
[Buffer1_id ′ = 0] & [Buffer1_hop ′ = 1] &
(Buffer1_unique ′ = false);

endmodule

dmodule node2-model[active1 = active2, round1 = round2,
id1 = id2, state1 = state2, mes1_round = mes2_round,
mes1_hop = mes2_hop, mes1_unique = mes2_unique,
mes1_id = mes2_id, pl_to_cl = p1_to_p2, c1_to_c2, c3_to_p1 = c1_to_p2,
Buffer3_round = Buffer1_round, Buffer3_id = Buffer1_id,
Buffer3_hop = Buffer1_hop, Buffer3_unique = Buffer1_unique]

dendmodule

dmodule channel2-channel1[Buffer1_round = Buffer2_round,
Buffer1_id = Buffer2_id, Buffer1_hop = Buffer2_hop,
Buffer1_unique = Buffer2_unique, pl_to_c1 = p1_to_c2,
c1_to_c2 = c1_to_p3, mes1_round = mes2_round, mes1_id = mes2_id,
mes1_hop = mes2_hop, mes1_unique = mes2_unique]
A.3 Algorithm $A_{Fokkink,roundtrip}$

Model description for $n=3$, $M \in \{2,3\}$:

```
module node3
  idl  : [0..1];
  state : [0..5];
  mes1_id : [0..M];
  mes1_hop : [0..N];
  mes1_unique : [0..1];
endmodule
```

```
module channel3
  Buffer1_round-Buffer3_round,
  Buffer1_id-Buffer3_id,
  Buffer1_unique-Buffer3_unique,
  p1_to-c1-p3_to-c3,
  c3_to-p1, mes1_round-mes3_round, mes1_id-mes3_id,
endmodule
```

```
A.3 Algorithm $A_{Fokkink,roundtrip}$
```

```
//create message
[] (state1 = 0) ->
  Prob: (state1 '1) & (idl '1) & (mes1_id '1) & (mes1_hop '1)
  & (mes1_unique '1) & (leader1 '0) +
  Prob: (state1 '1) & (idl '2) & (mes1_id '2) & (mes1_hop '1)
  & (mes1_unique '1) & (leader1 '0) +
  Prob: (state1 '1) & (idl '3) & (mes1_id '3) & (mes1_hop '1)
  & (mes1_unique '1) & (leader1 '0);
```

```
//send message
[p1_to_c1] (state1 = 1) -> (state1 '2) & (mes1_unique '0) &
  (mes1_id '0) & (mes1_hop '0);
```

```
//receive message
[c3_to_p1] (state1 = 2) ->
  [state1 '3] & (mes1_unique 'Buffer3a_unique) &
  (mes1_id 'Buffer3a_id) & (mes1_hop 'Buffer3a_hop);
```

```
//message checks:
//incoming message < p1 -> message purged
[] (state1 = 3) & (mes1_id<idl) -> [state1 '2] &
  (mes1_id '0) & (mes1_hop '0) & (mes1_unique '0);
```
/** incoming message \( p_1 \rightarrow p_1 \) becomes passive  
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1') \& (id1' \rightarrow 0) \& (mes1, hop' \rightarrow mes1, hop + 1);} 
\end{align*} \]

/** incoming message equal \( p_1 \rightarrow \text{unique bit dirty} \) 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1' \& (mes1, hop < N) \& (mes1, id = id1) \rightarrow} 
\text{(state1' \rightarrow) \& (id1' \rightarrow 0) \& (mes1, hop' \rightarrow mes1, hop + 1);} 
\end{align*} \]

/** then send the message, equal to state1-1 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1') \& (mes1, unique' \rightarrow 0)} 
\text{(state1' \rightarrow 2) \& (mes1, id' \rightarrow 0) \&} 
\text{(mes1, hop' \rightarrow 0);} 
\end{align*} \]

/** incoming message roundtrip, restart 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1) \& (mes1, hop = N) \& (mes1, unique = 0) \&} 
\text{(mes1, id' \rightarrow 0) \& (mes1, unique' \rightarrow 0) \&} 
\text{(state1' \rightarrow 0) \& (id1' \rightarrow 0) \&} 
\text{(mes1, hop' \rightarrow 0);} 
\end{align*} \]

/** incoming message roundtrip, leader 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1') \& (mes1, hop = N) \& (mes1, unique = 0)} 
\text{(state1' \rightarrow 4) \& (leader1' \rightarrow 1)} 
\text{(id1' \rightarrow 0) \& (mes1, id' \rightarrow 0) \&} 
\text{(mes1, hop' \rightarrow 0);} 
\end{align*} \]

/** passive send 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1')} 
\text{(mes1, id > 0) \& (mes1, hop > 0) \&} 
\text{(leader1' \rightarrow 0) \& (mes1, id' \rightarrow 0) \& (mes1, hop' \rightarrow 0) \&} 
\text{mes1, unique' \rightarrow 0);} 
\end{align*} \]

/** passive receive 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) state1') \& (mes1, id = 0) \& (mes1, hop = 0) \&} 
\text{(leader1 = 0) \&} 
\text{(Buffer3a, hop < N) \rightarrow} 
\text{(mes1, id' \rightarrow Buffer3a, id) \& (mes1, hop' \rightarrow Buffer3a, hop + 1) \&} 
\text{mes1, unique' \rightarrow Buffer3a, unique;} 
\end{align*} \]

/** selfloop 
\[ \text{(done) \& (sl \rightarrow 4) \rightarrow true;} \]

module channel 

/** the Buffer has size N=3 
\[ \begin{align*} 
\text{Bufferl, id: [0..M]; Bufferl, hop: [0..N]; Bufferl, id: [0..M]; Bufferl, hop: [0..N];} 
\text{Bufferl, unique: [0..1]; Bufferl, unique: [0..1];} 
\end{align*} \]

/** node1 adds to Buffer1 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) Bufferl, id = 0) \rightarrow} 
\text{Bufferl, id' \rightarrow mes1, id) \&} 
\text{Bufferl, hop' \rightarrow mes1, hop) \& (Bufferl, unique' \rightarrow mes1, unique);} 
\end{align*} \]

/** node2 out of Buffer1 
\[ \begin{align*} 
\text{(state1 \( \rightarrow \) Bufferl, id = 0) \rightarrow} 
\text{Bufferl, id' \rightarrow Bufferl, id) \&} 
\text{Bufferl, hop' \rightarrow Bufferl, hop) \&} 
\text{Bufferl, unique' \rightarrow Bufferl, unique) \&} 
\text{Bufferl, id' \rightarrow Bufferl, id) \& Bufferl, hop' \rightarrow Bufferl, hop) \&} 
\text{Bufferl, unique' \rightarrow Bufferl, unique) \& Bufferl, id' \rightarrow Bufferl, id) \& Bufferl, hop' \rightarrow Bufferl, hop) \& (Bufferl, unique' \rightarrow Bufferl, unique) \& (Bufferl, id' \rightarrow 0) \& Bufferl, hop' \rightarrow Bufferl, hop) \& Bufferl, unique' \rightarrow Bufferl, unique);} 
\end{align*} \]

/**
A.4. Algorithm $A_{Fokkink, no roundtrip}$

Model description for $n=3$, $M \in \{2,3\}$:

```plaintext
module node2
  state1 = state2, id1 = id2,
  mes1_unique = mes2_unique, leader1 = leader2, mes1_id = mes2_id,
  mes1_hop = mes2_hop, pl1o c1 = p2o c2, c3o p1 = c2o p2,
  Buffer3a_id = Bufferla_id, Buffer3a_hop = Bufferla_hop,
  Buffer3a_unique = Bufferla_unique
endmodule

module channel2
  Bufferla_id = Buffer2a_id,
  Bufferla_hop = Buffer2a_hop, Bufferla_unique = Buffer2a_unique,
  Bufferlb_id = Buffer2b_id, Bufferlb_hop = Buffer2b_hop,
  Bufferlb_unique = Buffer2b_unique, Bufferlc_id = Buffer2c_id,
  Bufferlc_hop = Buffer2c_hop, Bufferlc_unique = Buffer2c_unique,
  pl1o c1 = p2o c2, c3o p1 = c2o p2, mes1_id = mes2_id,
  mes1_hop = mes2_hop, mes1_unique = mes2_unique
endmodule

module node3
  state1 = state3, id1 = id3,
  mes1_unique = mes3_unique, leader1 = leader3, mes1_id = mes3_id,
  mes1_hop = mes3_hop, pl1o c1 = p3o c3, c3o p1 = c2o p3,
  Buffer3a_id = Buffer3a_id, Buffer3a_hop = Buffer3a_hop,
  Buffer3a_unique = Buffer2a_unique
endmodule

module channel3
  Bufferla_id = Buffer3a_id,
  Bufferla_hop = Buffer3a_hop, Bufferla_unique = Buffer3a_unique,
  Bufferlb_id = Buffer3b_id, Bufferlb_hop = Buffer3b_hop,
  Bufferlb_unique = Buffer3b_unique, Bufferlc_id = Buffer3c_id,
  Bufferlc_hop = Buffer3c_hop, Bufferlc_unique = Buffer3c_unique,
  pl1o c1 = p3o c3, c3o p1 = c2o p3, mes1_id = mes3_id,
  mes1_hop = mes3_hop, mes1_unique = mes3_unique
endmodule
```

```
const int N = 3; //number of nodes n
const int M; //number of IDs
const double Prob = 1/M;

module node1
  leader1: [0..1];
  id1: [0..M];
  state1: [0..4];
  mes1_id: [0..M];
  mes1_hop: [0..N];
//create message
[] {state1 = 0} ->
  Prob: {state1 = 1} & {id1 = 1} & {mes1_id = 1} & {mes1_hop = 1} & {leader1 = 0} +
  Prob: {state1 = 1} & {id1 = 2} & {mes1_id = 2} & {mes1_hop = 1} & {leader1 = 0} +
  Prob: {state1 = 1} & {id1 = 3} & {mes1_id = 3} & {mes1_hop = 1} & {leader1 = 0} +
```

```
A.4 Algorithm $A_{Fokkink, noroundtrip}$
```

```plaintext
```
```
& (leader1' = 0);

// send message
[pl.to x1] (state1' = 1) -> (state1' = 2) & (mes1_id' = 0) &
  (mes1_hop' = 0);

// receive message
[c3.to.p1] (state1' = 2) -> (state1' = 3) &
  (mes1_id' = Buffer3a_id) & (mes1_hop' = Buffer3a_hop);

// message checks:
// incoming message < p1 -> message purged
[1] (state1' = 3) & (mes1_id < id1) -> (state1' = 2) &
  (mes1_id' = 0) & (mes1_hop' = 0);

// incoming message > p1 -> p1 becomes passive
[1] (state1' = 3) & (mes1_hop < N) & (mes1_id = id1) ->
  (state1' = 0) & (mes1_hop' = 0) & (mes1_id' = 0) & (id1' = 0);

// incoming message equal p1 -> restart
[1] (state1' = 3) & (mes1_hop = N) & (mes1_id = id1) ->
  (state1' = 4) & [(leader1' = 1) & (id1' = 0) & (mes1_id' = 0) &
  (mes1_hop' = 0)];

// passive send
[pl.to x1] (state1' = 4) & (mes1_id > 0) & (mes1_hop > 0) &
  (leader1' = 0) -> (mes1_id' = 0) & (mes1_hop' = 0);

// passive receive
[c3.to.p1] (state1' = 4) & (mes1_id = 0) & (mes1_hop = 0) &
  (leader1' = 0) & (Buffer3a_hop < N) ->
  (mes1_id' = Buffer3a_id) & (mes1_hop' = Buffer3a_hop + 1);

// selloop
[done] (sl = 4) -> true;
endmodule

module channel
// the Buffer has size N=3
Buffera_id: [0..M]; Bufferb_id: [0..M]; Bufferc_id: [0..M];
Buffera_hop: [0..N]; Bufferb_hop: [0..N]; Bufferc_hop: [0..N];

// node1 adds to Buffer1
[pl.to x1] (Buffera_id' = 0) ->
  (Buffera_id' = mes1_id) & (Buffera_hop' = mes1_hop);
[pl.to x1] (Buffera_id' > 0) & (Buffera_id' = 0) ->
  (Bufferb_id' = mes1_id) & (Bufferb_hop' = mes1_hop);
[pl.to x1] (Bufferb_id' > 0) & (Bufferb_id' = 0) ->
  (Bufferc_id' = mes1_id) & (Bufferc_hop' = mes1_hop);

// node2 out of Buffer1
[c1.to x2] (Buffera_id' > 0) -> (Buffera_id' = Bufferb_id) &
  (Buffera_hop' = Bufferb_hop) &
  (Bufferb_id' = Bufferc_id) & (Bufferb_hop' = Bufferc_hop) &
  (Bufferc_id' = 0) & (Bufferc_hop' = 0);
endmodule

module node2 -> model (state1 = state2, id1 = id2, id2 = id1);
A.5. Algorithm $A_{\text{sorting}}$

Model description for $A_{\text{sorting, sync}}, n=3$:

```model
module model

p1\_var1 : [0..2];  // p1\_id, 0 - empty buffer
p1\_var2 : [0..2];  // p3\_id, because of clockwise transmis.
p1\_var3 : [0..2];  // p2\_id

state1 : [0..2];

//choose an ID
| (state1=0) -> 0.5: (p1\_var1=1) & (state1'=1) +
| 0.5: (p1\_var1=2) & (state1'=1); |

//read
| (state1=1) & (p1\_var1=0) & (p1\_var2=0) & (p3\_var1=0) ->
| (p1\_var2'=p3\_var1);
| (state1=1) & (p1\_var2=0) & (p1\_var3=0) & (p3\_var2=0) ->
| (p1\_var3'=p3\_var2);

//periodic -> synchronous restart
| [restart] (state1=1) & (p1\_var3=0) &
| (p1\_var1=p1\_var2) & (p1\_var1=p1\_var3) ->
| (state1'=0) & (p1\_var1'=0) & (p1\_var2'=0) & (p1\_var3'=0);

//aperiodic -> leader exists
| (state1=1) & (p1\_var3=0)
| [ (p1\_var1=p1\_var2) | (p1\_var1=p1\_var3) ] -> (state1'=2);

//selfloop
```

155
\[
\begin{align*}
[] \rightarrow true; \\
endmodule

dmodule node2 \rightarrow node1 \rightarrow p1 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
p1 \rightarrow p2 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
state1 \rightarrow state2, p3 \rightarrow p1, \\
p3 \rightarrow p2 \rightarrow p2 \\
endmodule

dmodule node3 \rightarrow node1 \rightarrow p1 \rightarrow p3, p1 \rightarrow p3 \rightarrow p3, \\
p1 \rightarrow p3 \rightarrow p3, state1 \rightarrow state3, p3 \rightarrow p1 \rightarrow p2, \\
p3 \rightarrow p2 \rightarrow p2 \\
endmodule
\]

Model description for \( A_{\text{sorting, a sync}} \), \( n=3 \):

module node1

\begin{align*}
pl_1 & : [0..2]: // pl_1.id, 0 = empty buffer \\
pl_2 & : [0..2]: // pl_2.id, because of clockwise transmission \\
pl_3 & : [0..2]: // pl_3.id \\
state & : [0..2] \\
pl_read & = init true; // local synchronization
\end{align*}

// choose an ID
\begin{align*}
[pl_{\text{made_reset}}, state] & \rightarrow [p1 \text{ pl_1 } \rightarrow 0.5: (pl_{\text{var1}} \rightarrow 1) & (state \rightarrow 1) + \\
& \rightarrow 0.5: (pl_{\text{var1}} \rightarrow 2) & (state \rightarrow 1)]; \\

[pl_{\text{made_reset}}, true] & \rightarrow pl_{\text{read_all}} \rightarrow false;
\end{align*}

// read
\begin{align*}
[] & \rightarrow [(state \rightarrow 1) & (pl_{\text{var1}} \rightarrow 0) & (pl_{\text{var2}} \rightarrow 0) & (pl_{\text{var3}} \rightarrow 0) & (pl_{\text{read_all}} \rightarrow (pl_{\text{var2}} \rightarrow 3) \\
& \rightarrow (pl_{\text{var3}} \rightarrow pl_{\text{var3}}) & (pl_{\text{read_all}} \rightarrow (pl_{\text{var2}} \rightarrow 0) & (pl_{\text{var3}} \rightarrow 0)); \\

// periodic \rightarrow local synchronous restart
[] & \rightarrow [(state \rightarrow 1) & (pl_{\text{var3}} \rightarrow 0) & (pl_{\text{read_all}} \rightarrow (pl_{\text{var1}} \rightarrow pl_{\text{var2}}) & (pl_{\text{var1}} \rightarrow pl_{\text{var3}}) & (state \rightarrow 0) & (pl_{\text{var1}} \rightarrow 0) & (pl_{\text{var2}} \rightarrow 0) & (pl_{\text{var3}} \rightarrow 0)); \\

// periodic \rightarrow leader exists
[] & \rightarrow [(state \rightarrow 1) & (pl_{\text{var3}} \rightarrow 0) & (pl_{\text{var3}} \rightarrow pl_{\text{var2}}) & (pl_{\text{var1}} \rightarrow pl_{\text{var3}}) & (state \rightarrow 2)];
\end{align*}

// selfloop
\begin{align*}
[] & \rightarrow true; \\
endmodule

dmodule node2 \rightarrow node1 \rightarrow p1 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
p1 \rightarrow p2 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
state1 \rightarrow state2, p3 \rightarrow p1, \\
p3 \rightarrow p2 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
p2 \rightarrow p2 \rightarrow p2, p2 \rightarrow p2 \rightarrow p2, \\
pl_{\text{made_reset}} \rightarrow pl_{\text{made_reset}}, pl_{\text{made_reset}} \rightarrow pl_{\text{made_reset}} \\
endmodule

dmodule node3 \rightarrow node1 \rightarrow p1 \rightarrow p3, p1 \rightarrow p3 \rightarrow p3, \\
p1 \rightarrow p3 \rightarrow p3, state1 \rightarrow state3, p3 \rightarrow p1, \\
p3 \rightarrow p2 \rightarrow p2, p1 \rightarrow p2 \rightarrow p2, \\
p3 \rightarrow p2 \rightarrow p2, p2 \rightarrow p2 \rightarrow p2, \\
pl_{\text{made_reset}} \rightarrow pl_{\text{made_reset}}, pl_{\text{made_reset}} \rightarrow pl_{\text{made_reset}} \\
endmodule
A.6  Algorithm $A_{\text{tim}}$

Model description for $n=3$ with formulae:

\begin{verbatim}
module node1
state1 : [0..5];
mes1_hop : [0..N]; // 0 = empty buffer
mes1_unique : bool;

// if no message is coming, then randomly wait
[top1] (state1=0) & ![preceding_hop_is_N] & (state1'=1) & (mes1_hop'=preceding_hop_plus_1) & (mes1_unique'=preceding_unique);

// passive, send
[top2] (state1=1) -> (state1'=2) & (mes1_hop'=0);

// passive, receive
[top1] (state1=2) & ![preceding_hop_is_N] & (state1'=1) & (mes1_hop'=preceding_hop_plus_1) & (mes1_unique'=preceding_unique);

// active, send
[top2] (state1=3) -> (state1'=4) & (mes1_hop'=0);

// active, receive no roundtrip
[top1] (state1=4) & ![preceding_hop_is_N] & (state1'=3) & (mes1_hop'=preceding_hop_plus_1) & (mes1_unique'=false);

// active, receive roundtrip, not unique -> restart
[top1] (state1=4) & ![preceding_unique] & (state1'=0);

// active, receive roundtrip, unique -> leader
[top1] (state1=4) & ![preceding_is_N] & (preceeding_unique) & (state1'=5);

// selfloop for leader
[] (state1=5) -> true;
endmodule

module node2-node [state1=state2, mes1_hop=mes2_hop,

p1_made_reset=p3_made_reset, p3_made_reset=p2_made_reset]
endmodule
\end{verbatim}
module node3 [state1 $\rightarrow$ state3, mes1 $\rightarrow$ mes3, mes1 $\rightarrow$ mes3, mes3 $\rightarrow$ mes1, mes3 $\rightarrow$ mes1, toP2 $\rightarrow$ toP3, toP1 $\rightarrow$ toP2]
endmodule

module node2 [state1 $\rightarrow$ state3, mes1 $\rightarrow$ mes3, mes1 $\rightarrow$ mes3, mes3 $\rightarrow$ mes1, mes3 $\rightarrow$ mes1, toP2 $\rightarrow$ toP3, toP1 $\rightarrow$ toP2]
endmodule
Appendix B

SPIN specifications

B.1 Algorithm $A_{Itai, round \leq K}$

```c
/* --promela--
for emacs promela mode */

#define N 7 /* number of nodes n */
#define K 4 /* number of rounds */
#define M 7 /* number of IDs (~ 0, ..., M-1) */
#define L 7 /* size of FIFO Buffers */

byte nrLeaders = 0;
bool limitReached = false;
typedef Msg {byte round; byte id; byte hop; bool unique};
chan q[N] = [L] of {Msg};

proctype node(chan in, out)
{
    bool nodeActive = true, leader = false;
    byte round, id;
    Msg curMsg;

    xx in;
    xx out;

START: /* random ID */
    id = 0;
do
    :: skip -> break
    :: id < M-1 -> id++
od;

    curMsg.id = id;
    curMsg.round = round;
    curMsg.hop = 1;
    curMsg.unique = true;

SEND AND RECEIVE:
    out! curMsg;
```

endRECEIVE:
  in?curMsg;

/* check message */
if
  :: curMsg.hop<N ->
  if
    :: [curMsg.round > round] ||
    [curMsg.round == round && curMsg.id > id] ->
      round = curMsg.round; id = curMsg.id; nodeActive = false;
    curMsg.hop++; goto SENDANDRECEIVE
  :: curMsg.round < round ||
    [curMsg.round == round && curMsg.id < id] ->
      goto endRECEIVE
  :: curMsg.round == round && curMsg.id == id ->
    curMsg.hop++; curMsg.unique = false;
  goto SENDANDRECEIVE
else ->
  printf("ERROR: hop<N and message cases not correct")
fi
:: curMsg.hop==N ->
if
  :: nodeActive && !curMsg.unique && round<K ->
    round++; goto START
  :: nodeActive && !curMsg.unique && round==K ->
    limitReached = true; printf("Round-Limit reached")
  :: nodeActive && curMsg.unique
    assert nrLeaders==1
  :: else -> printf("ERROR: hop==N and !nodeActive")
fi
fi


init {
  byte lv;
  atomic {
    lv = 0;
    do
      :: lv < N -> run node(q[lv], q[(lv+1)%N]); lv++
      :: lv==N -> break
    od
  }
  timeout -> assert nrLeaders == 1 || limitReached;
  printf("Asserted: exactly one leader or limit reached\n")
}

B.2 Algorithm A_{Itai,roundmodK}

Model specification for A_{Itai,roundmod2}:

#define N 6     /* number of nodes n */
#define K 2     /* rounds mod K-2 */
#define M 6     /* number of IDs = 0,...,N-1 */
#define L 6     /* size of FIFO Buffers */

byte nrLeaders = 0;
typedef Msg {byte round; byte id; byte hop; bool unique};
chan q[N] = [L] of {Msg};
proctype node {chan in, out}
{  
  bool nodeActive = true, leader = false;
  byte round, id;
  Msg curMsg;

  // in;
  // out;

START: /* random ID */
  id = 0;
  do
    skip => break
    id < M-1 => id++
  od;

  curMsg.id = id;
  curMsg.round = round;
  curMsg.hop = 1;
  curMsg.unique = true;

SENDANDRECEIVE:
  out! curMsg;
endRECEIVE:
  in? curMsg;

/* check message */
if
  : curMsg.hop < N =>
  if
    : (curMsg.round != round || curMsg.id > id) =>
      round = curMsg.round; id = curMsg.id; nodeActive = false;
      curMsg.hop++; goto SENDANDRECEIVE
    : curMsg.round == round && curMsg.id < id =>
      goto endRECEIVE
    : curMsg.round == round && curMsg.id == id =>
      curMsg.hop++; curMsg.unique = false;
      goto SENDANDRECEIVE
    : else =>
      printf("ERROR: hop<N and msg=cases not correct")
  fi
  : curMsg.hop == N =>
    if
      : nodeActive && !curMsg.unique && round < K-1 =>
        round++; goto START
      : nodeActive && !curMsg.unique && round == K-1 =>
        round = 0; goto START
      : nodeActive && curMsg.unique => leader = true;
        nrleaders++; assert (nrleaders == 1)
      : else => printf("ERROR: hop=N and !nodeActive")
    fi
  fi

init {
  byte lv;
  atomic {
    lv = 0;
    do
      : lv < N => run node(q[lv], q[[(lv+1)%N]]); lv++
      : lv == N => break
    od
  }
}
Model specification for $A_{Itai,round \ mod \ 2}$ with restricted progress labels:

```c
#define N 3 /* number of nodes n */
#define K 2 /* rounds mod K-2 */
#define M 3 /* number of IDs = 0, ..., M-1 */
#define L 3 /* size of FIFO Buffers */

byte nr leaders = 0;
typedef Msg {byte round; byte id; byte hop; bool unique};
chan q[N] — [L] of {Msg};

proctype node (chan in, out)
{
  bool nodeActive = true;
  bool leader = false, allSameID;
  byte round, id;
  Msg curMsg;

  xr in;
  xs out;

START: /* random ID */
  id = 0;
  do :: skip -> break
   :: id < M-1 -> id++
  od;
  allSameID = true;
  curMsg.id = id;
  curMsg.round = round;
  curMsg.hop = 1;
  curMsg.unique = true;

SENDANDRECEIVE:
  out! curMsg;

RECEIVE:
  in ? curMsg;

/* check message */
if :: curMsg.hop < N ->
  if :: (curMsg.round != round || curMsg.id > id) ->
    allSameID = false; round = curMsg.round; id = curMsg.id;
    nodeActive = false; curMsg.hop++;
    goto SENDANDRECEIVE
  :: curMsg.round = round && curMsg.id < id ->
    allSameID = false; goto RECEIVE
  :: curMsg.round = round && curMsg.id = id ->
    allSameID = allSameID && curMsg.id = id;
    curMsg.hop++; curMsg.unique = false;
    goto SENDANDRECEIVE
  :: else ->
    printf("ERROR: hop<N and msg-cases not correct")
  fi
```

Appendix B. SPIN specifications
B.2. Algorithm $A_{\text{fail\_round \ mod \ K}}$

```
:: curMsg. hop—N ->
  if
    :: allSameID -> progress ! skip
    :: ! allSameID -> skip
  fi;
  if
    :: nodeActive && ! curMsg.unique && round<K-1 ->
      round++; goto START
    :: nodeActive && ! curMsg.unique && round—K-1 ->
      round=0; goto START
    :: nodeActive && curMsg.unique -> leader=true;
      nr_leaders++; assert (nr_leaders==1)
    else -> printf("ERROR: hop=N and !nodeActive")
  fi

init {
  byte lv;
  atomic {
    lv=0;
    do
      :: lv < N -> run node(q[1lv], q[(lv+1)%N]); lv++
      :: lv==N -> break
    od
  }
  timeout -> assert (nr_leaders==1);
  printf("Asserted: nr_leaders==1")
}
```

Model specification for $A_{\text{fail\_round \ mod \ 2}}$ with message overwriting implemented by buffer processes:

```
define N 4        /* number of nodes n */
define K 2        /* rounds mod K—2 */
define M 4        /* number of IDs = 0, ..., M-1 */

byte nr_leaders=0;

typedef Msg [byte round; byte id; byte hop; bool unique];
chan preBuffer[N] = [1] of {Msg};
chan postBuffer[N] = [1] of {Msg};

proc type node (chan in, out)
{
  bool nodeActive = true, leader = false;
  byte round, id;
  Msg curMsg;

  x in;
  x out;

START: /* random ID */
  id=0;
  do
    :: skip -> break
    :: id < M-1 -> id++
  od;

  curMsg.id=id;
  curMsg.round—round;
  curMsg.hop—1;
  curMsg.unique=true;
```
SEND AND RECEIVE:
  out! curMsg;

endRECEIVE:
  in? curMsg;

/* check message */
if :
  curMsg.hop < N ->
  if :
    curMsg.round != round || curMsg.id > id ->
      round = curMsg.round; id = curMsg.id; nodeActive = false;
    curMsg.hop++; goto SEND AND RECEIVE
  else ->
    printf("ERROR: hop < N and msg_cases not correct")
  fi
  curMsg.hop-- N ->
  if :
    nodeActive && curMsg.unique && round < K_1 ->
      round++; goto START
    nodeActive && !curMsg.unique && round = K_1 ->
      round = 0; goto START
    nodeActive && curMsg.unique -> leader = true;
    nrLeaders++; assert nrLeaders = 1
  else ->
    printf("ERROR: hop = N and !nodeActive")
  fi
fi

proctype bufferproc (chan in, out; pid next)
{
  Msg buffer;
  bool bufferempty = true;

  xx in;
  xs out;

endvalid:
  do :
    in? buffer -> bufferempty = false;
    bufferempty = false && node[next]@sendRECEIVE ->
      out! buffer; bufferempty = true;
  od;
}

init {
  byte lv;
  pid currentNode;

  atomic {
    lv = N;
    do :
      lv > 0 ->
        currentNode = run node postBuffer [lv - 1],
        preBuffer [(lv)%N];
        run bufferproc (preBuffer [lv - 1],
        postBuffer [lv - 1], currentNode);
B.3 Algorithm \( A_{\text{Fokkink, roundtrip}} \)

```
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49

1
:: 1y==0 => break
od
}
timeout => assert(nr Leaders==1);
printf("Asserted: nr Leaders==1\n")
}
```

```
#define N 4     /* number of nodes n */
#define M 4     /* number of IDs = 0,...,M-1 */
#define L 4     /* size of FIFO Buffers */

byte nr Leaders=0;
typedef msg {byte id; byte hop; bool unique};
chan q[N] = [L] of {msg};

proctype node (chan in, out) {
  bool node Active = true, leader = false;
  byte id;
  msg curMsg;
  xx in;
  xx out;

START: /* random ID */
  id=0;
  do :: skip => break
    :: id < M-1 => id++
  od;
  curMsg.id=id;
  curMsg.hop=1;
  curMsg.unique=true;

SEND AND RECEIVE:
  out!curMsg;
end RECEIVE:
  in?curMsg;

/* check message */
if :: (!node Active) => curMsg.hop++;
goto SEND AND RECEIVE
:: else => if :: curMsg.hop<N =>
  if :: (curMsg.id > id) =>
    node Active=false; curMsg.hop++;
goto SEND AND RECEIVE
:: (curMsg.id < id) =>
goto end RECEIVE
:: curMsg.id==id =>
curMsg.hop++; curMsg.unique=false;
goto SEND AND RECEIVE
fi
```
:: curMsg.hop--N \rightarrow 
   if
   :: !curMsg.unique \rightarrow goto START
   :: curMsg.unique \rightarrow leader=true;
   nr_leaders++; assert(nr_leaders==1)
   fi
fi

init {
   byte lv;
   atomic {
      lv=0;
      do
      :: lv < N \rightarrow run node[q[lv], q[(lv+1)\%N]]; lv++
      :: lv==N \rightarrow break
      od
   }
timeout \rightarrow assert(nr_leaders==1);
   printf("Asserted: nr_leaders==1")
}

B.4 Algorithm \( \mathcal{A}_{Fokkink, no roundtrip} \)

#define N 4 /* number of nodes n */
#define M 4 /* number of IDs = 0, \ldots, M-1 */
#define L 4 /* size of FIFO Buffers */

byte nr_leaders=0;
typedef Msg { byte id; byte hop };
chan q[N] = [L] of {Msg};

proctype node (chan in, out) {
   bool nodeActive = true, leader = false;
   byte id;
   Msg curMsg;

   <in in;
   <out out;

   START: /* random ID */
   id=0;
   do
   :: skip \rightarrow break
   :: id < M-1 \rightarrow id++
   od;

   curMsg.id=id;
   curMsg.hop=1;

SEND AND RECEIVE:
   out!curMsg;

endRECEIVE:
   in?curMsg;

   /* check message */
   if
B.5. Algorithm $A_{\text{sorting}}$

:: (!nodeActive) => curMsg.hop++; goto SEND\_AND\_RECEIVE
:: else => if
   :: curMsg.hop<N =>
      if
        :: (curMsg.id > id) =>
           nodeActive=false; curMsg.hop++;
           goto SEND\_AND\_RECEIVE
        :: (curMsg.id < id) =>
           goto endRECEIVE
        :: curMsg.id==id =>
           goto START
      fi
      :: curMsg.hop==N => leader=true;
      nr_leaders++; assert(nr_leaders==1)
   fi
fi
}

init {
   byte lv;
   atomic {
      lv=0;
      do :: lv<N => run node[q[lv], q[(lv+1)%N]]; lv++
         :: lv==N => break
      od
   }
   timeout => assert(nr_leaders==1);
   printf("Asserted: nr_leaders==1")
}

B.5 Algorithm $A_{\text{sorting}}$

Model specification for $A_{\text{sorting}}, n$ prime:

```c
#define N 5 /* number of nodes n, prime */
#define L 5 /* size of FIFO-Buffers */

bool leader_exists = false;
chan q[N] = [L] of {bit};

proctype node (chan in, out)
{
   bit vector[N];
   byte index, sum;
   xx in;
   xx out;

   START:
   index=0;
   sum=0;
   if /* random ID */
      :: vector[index]=0
      :: vector[index]=1
   fi;
   out(vector[index];
   index++; do
```


:: atomic{index<N-1 &\& nempty(in) -> in?vector[index];
    out!vector[index]; index++}
:: atomic{index=N-1 &\& nempty(in) -> in?vector[index];
    index++}
:: index=N ->
    do
    : index>0 -> index--; sum=sum+vector[index]
    : index=0 -> break
    od;
    if /* for n prime: */
    : sum=0 || sum=N -> goto START /* periodic */
    : else => leader_exists = true; /* aperiodic */
    goto FINISH
fi
od;
FINISH: skip
}
init {
    byte lv;
    atomic {
        lv=0;
        do
        : lv<N -> run node(q[lv], q[(lv+1)%N]); lv++
        : lv=N -> break
        od
    }
timeout -> assert(leader_exists);
    printf("Asserted: some leader exists\n")
}

B.6 Algorithm \( \mathcal{A}_{\text{Timing}} \)

Model specification for \( \mathcal{A}_{\text{Timing}} \):

```c
#define N 18  /* number of nodes n */
#define L 18  /* ssc of FIFO Buffers */
byte nr leaders =0;
typedef Msg {byte hop; bool unique};
chan q[N] = [L] of {Msg};

proctype node (chan in, out) {
    bool leader=false;
    Msg curMsg;
    xr in;
    xs out;

    progressSTART:
    do
    : atomic{nempty(in) -> in?curMsg; curMsg.hop++};
    goto PASSIVE_SEND_AND_RECEIVE
    : else => if
    : skip -> progressWait: skip
    : skip -> curMsg.hop-=1; curMsg.unique=true;
    goto ACTIVE_SEND_AND_RECEIVE
    fi
```
B.6. Algorithm \(A_{\text{sim}}\)  

\[
\text{ACTIVE SEND AND RECEIVE:} \\
\text{do} \\
\quad \text{curMsg.hop} \leftarrow 0 \\
\quad \text{if} \ \text{in?curMsg} \\
\quad \quad \text{curMsg.hop} \leftarrow N \& \& \text{curMsg.unique} \\
\quad \quad \text{leader} \leftarrow \text{true}; n_r \_\text{leaders} \leftarrow \text{true} \\
\quad \quad \text{assert} \{ n_r \_\text{leaders} \leftarrow \text{false} \} \\
\quad \quad \text{curMsg.hop} \leftarrow 0; \text{curMsg.unique} \leftarrow \text{false}; \\
\quad \text{leader} \leftarrow \text{true}; n_r \_\text{leaders} \leftarrow \text{true} \\
\quad \quad \text{assert} \{ n_r \_\text{leaders} \leftarrow \text{false} \} \\
\quad \text{curMsg.hop} \leftarrow 0; \text{curMsg.unique} \leftarrow \text{false} \\
\quad \text{else} \rightarrow \text{skip} \\
\text{fi}; \\
\quad \text{curMsg.hop} \leftarrow \text{curMsg.hop} \leftarrow \text{curMsg.hop} \leftarrow \text{curMsg.hop} \leftarrow \text{false} \\
\text{od}; \\
\text{PASSIVE SEND AND RECEIVE:} \\
\text{do} \\
\quad \text{curMsg.hop} \leftarrow 0 \\
\quad \text{endVALIDSTATE:} \ \text{in?curMsg} \\
\quad \text{if} \ \text{curMsg.hop} \leftarrow N \\
\quad \quad \text{printf("Error: pas. node rec. roundtrip-msg")}; \\
\quad \quad \text{else} \rightarrow \text{skip} \\
\quad \text{fi}; \\
\quad \text{curMsg.hop} \leftarrow \text{curMsg.hop} \leftarrow \text{curMsg.hop} \leftarrow \text{false} \\
\text{od}; \\
\text{FINISH: skip} \\
\}
\]

\text{init} \\
\text{byte } lV; \\
\text{atomic} \\
\text{lv} = 0; \\
\text{do} \\
\quad \text{if } lV < N \rightarrow \text{run node} \{ q[lV], q[(lV+1)\%N] \}; lV++; \\
\quad \text{lv} = N \rightarrow \text{break} \\
\text{od} \\
\text{timeout} \rightarrow \text{assert} \{ n_r \_\text{leaders} \leftarrow \text{false} \}; \\
\text{printf("Asserted: n_r \_\text{leaders} \leftarrow \text{false}")}
\]

Model specification for \(A_{\text{sim}}\) with one testing node:

\[
\#\text{define } N 18 \quad /\!* \text{number of nodes } n */\]
\[
\#\text{define } L 18 \quad /\!* \text{size of FIFO Buffers */}\]
\[
\text{byte } n_r \_\text{leaders} = 0; \\
\text{byte } \text{nodeWaits} = 0; \\
\text{typedef } \text{Msg} \{ \text{byte } \text{hop}; \text{bool } \text{unique}; \}
\text{chan } q[N] = [L] \text{ of } \{ \text{Msg}; \}
\]

\text{proctype } \text{node} \{ \text{chan } \text{in}, \text{out} \}
\{
\text{bool } \text{leader} = \text{false};
\]
\begin{verbatim}
Msg curMsg;

xr in; xs out;

progressSTART:
  do :: atomic\{empty\} \rightarrow in?curMsg; curMsg.hop++;
       goto PASSIVE_SEND_AND_RECEIVE
  :: else \rightarrow if
       :: skip \rightarrow progressWait: skip
       :: skip \rightarrow curMsg.hop--1; curMsg.unique=true;
       goto ACTIVE_SEND_AND_RECEIVE
     fi

od;

ACTIVE_SEND_AND_RECEIVE:
  do :: curMsg.hop=0 \rightarrow
       in?curMsg; if
       :: curMsg.hop=N && curMsg.unique \rightarrow
           curMsg.hop=0; curMsg.unique=false;
           leader=true; nr_leaders++;
           assert\{nr_leaders==1\};
           goto FINISH
       :: curMsg.hop=N && !curMsg.unique \rightarrow
           curMsg.hop=0; goto progressSTART
       :: else \rightarrow skip
     fi;

   curMsg.hop++; curMsg.unique=false
   :: curMsg.hop>0 \rightarrow
      out!curMsg; curMsg.hop=0; curMsg.unique=false
od;

PASSIVE_SEND_AND_RECEIVE:
  do :: curMsg.hop=0 \rightarrow
endVALIDSTATE: in?curMsg;
  if
       :: curMsg.hop=N \rightarrow
           printf("Error: pas. node rec. roundtrip-msg");
       :: else \rightarrow skip
     fi;

   curMsg.hop++
   :: curMsg.hop>0 \rightarrow
      out!curMsg; curMsg.hop=0; curMsg.unique=false
od;

FINISH: skip
\end{verbatim}
:: skip → curMsg.hop--1; curMsg.unique=true; goto ACTIVE_SEND_AND_RECEIVE
fi;
if :: skip →
:: skip → goto nodeSwitchesIntoTestingMode
fi

ACTIVE_SEND_AND_RECEIVE:
do :: curMsg.hop--0 ⇒
in?curMsg; if :: curMsg.hop--N && curMsg.unique ⇒
curMsg.hop=0; curMsg.unique=false;
leader=true; nr_leaders++;
assert(nr_leaders==1); goto FINISH:: curMsg.hop--N && !curMsg.unique ⇒
curMsg.hop=0; goto progressSTART:: else → skip
fi;
curMsg.hop++; curMsg.unique=false
:: curMsg.hop>0 ⇒
out!curMsg; curMsg.hop=0; curMsg.unique=false
od;

PASSIVE_SEND_AND_RECEIVE:
do :: curMsg.hop--0 ⇒
endVALIDSTATE: in?curMsg;
if :: curMsg.hop--N ⇒
printf("Error: pas. node rec. roundtrip_msg");:: else → skip
fi;
curMsg.hop++
:: curMsg.hop>0 ⇒
out!curMsg; curMsg.hop=0; curMsg.unique=false
od;

nodeSwitchesIntoTestingMode:
curMsg.hop=1; curMsg.unique=true;
out!curMsg; nodeWaits=1;
in?curMsg; nodeWaits=0;

FINISH:: skip
}

init {
byte lv;
atomic {
run nodeLeavesProgress(q[0], q[1]);
lv=1;
do :: lv < N ⇒ run node(q[lv], q[(lv+1)%N]); lv++
:: lv=N ⇒ break
}
}

timeout ⇒ assert(nodeWaits==0);
printf("Node can definitely become passive or leader\n");
Bibliography


URL: http://www.dcs.warwick.ac.uk/~doron/ps/sttt98.ps (on page 129)


URL: http://spinroot.com/spin/Workshops/ws02/dams_tut.pdf (on page 10)


URL: http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0405.pdf (on page 74 91 38 16 90 89 90 50 50 104 129)

URL: http://www.cs.vu.nl/~wanf/pubs/avocs-leader.pdf (on page 27 92 50 70)

URL: http://www.springerlink.com/index/PN0805U9102624R2.pdf (on page 13)

URL: http://spinroot.com/spin/Man/Quick.html (on page 21 99 138)


URL: http://citeseer.ist.psu.edu/holzmann97model.html (on page 20)


URL: http://www.jcf.de/~jcf/uni/5aus8-Wahlpflicht/FS/FSSkript.ps (on page 11)


URL: http://netlib.bell-labs.com/netlib/spin/ws98/millet.ps.gz (on page 11)

URL: http://www.cs.rice.edu/~vardi/papers/foacs85.pdf.gz (on page 11)


URL: http://www.cmi.ac.in/~madhavan/papers/ps/isical97.ps.gz (on page 20)

URL: http://www.embedded.com/shared/printableArticle.html?articleID =17603352 (on page 19)

URL: http://www.cs.bham.ac.uk/~dxp/prism/casestudies/asynchronous_leader.php (on page 15)


URL: http://www.dcs.warwick.ac.uk/~doron/pr/ps/full.ps (on page 129)

URL: http://www.cs.bham.ac.uk/~dxp/prism (on page 15)
Bibliography


   URL: http://www.cs.utwente.nl/~ruys (on page 20)

   URL: http://www.cs.utwente.nl/~ruys (on page 20)

   URL: http://i12www.ira.uka.de/~pschmitt/NKLNKLSkript.ps (on page 11)

   URL: http://i12www.ira.uka.de/~pschmitt/FormSys/FSsKript.pdf (on page 10)

   URL: http://www.cs.cmu.edu/~emc/15817-s05/cegar.ppt (on page 11)

Spin (2007). The model checker Spin.
   URL: http://spinroot.com/spin/whatispin.html (on page 20)


   URL: http://www-i7.informatik.rwth-aachen.de/teaching/ws0304/modelchk/DivX-PDF.html (on page 13)

Uppaal (2007). The model checker Uppaal.
   URL: http://www.uppaal.com (on page 25)
# Index

## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>/{...}</td>
<td>17</td>
</tr>
<tr>
<td>□,</td>
<td>12</td>
</tr>
<tr>
<td>□ ≤k,</td>
<td>15</td>
</tr>
<tr>
<td>φ,</td>
<td>12</td>
</tr>
<tr>
<td>φ ≤k,</td>
<td>15</td>
</tr>
<tr>
<td>N_0,</td>
<td>2</td>
</tr>
<tr>
<td>N_0,</td>
<td>2</td>
</tr>
<tr>
<td>[ ... ],</td>
<td>17</td>
</tr>
<tr>
<td>[ ],</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ω-automaton,</td>
<td>25</td>
</tr>
<tr>
<td>ω-regular property,</td>
<td>25</td>
</tr>
<tr>
<td>P[ ... ],</td>
<td>10</td>
</tr>
<tr>
<td>σ,</td>
<td>67</td>
</tr>
<tr>
<td>t,</td>
<td>37</td>
</tr>
<tr>
<td>=,</td>
<td>21</td>
</tr>
<tr>
<td>A_compare IDs,</td>
<td>11</td>
</tr>
<tr>
<td>A_fokkink, no roundtrip,</td>
<td>31</td>
</tr>
<tr>
<td>A_fokkink, roundtrip,</td>
<td>32</td>
</tr>
<tr>
<td>Altai, round ≤K,</td>
<td>30</td>
</tr>
<tr>
<td>Altai, round mod K,</td>
<td>31</td>
</tr>
<tr>
<td>Altai,</td>
<td>30</td>
</tr>
<tr>
<td>A_compare IDs,</td>
<td>30</td>
</tr>
<tr>
<td>A_sorting, async,</td>
<td>50</td>
</tr>
<tr>
<td>A_sorting, sync,</td>
<td>50</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>54</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>54</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>48</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>48</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>48</td>
</tr>
<tr>
<td>A_sorting,</td>
<td>48</td>
</tr>
<tr>
<td>-&gt;,</td>
<td>22</td>
</tr>
<tr>
<td>-&gt;,</td>
<td>22</td>
</tr>
<tr>
<td>¬DBITSTATE,</td>
<td>108</td>
</tr>
<tr>
<td>DRANDSTOR,</td>
<td>108</td>
</tr>
<tr>
<td>,</td>
<td>22</td>
</tr>
<tr>
<td>last,</td>
<td>29</td>
</tr>
</tbody>
</table>

## A

- A, 12
- A □ ≤k, 15
- A φ, 15
- A φ ≤k, 15
- A¬ F ≤k, 15
- A¬ G ≤k, 15
- A¬ U ≤k, 15
- A¬ □ ≤k, 15
- A¬ φ ≤k, 15
- abortion of paths, 12
- abortion signal, 63
- absolute, 82
- abstract algorithm, 35
- abstract interpretation, 40
- abstraction, 40
- abstraction refinement, 10
- accept
  - label, 22
  - accept, 22
  - acceptance
    - label, 22
    - state, 11 22
  - acceptance cycle, 11 21
  - acceptance cycle check, 121
  - action, 5
    - action, 16
    - action label, 16
  - action, 5
    - active, 31
    - active node, 31
  - adverse schedule, 74
- AF, 13
- AF ≤k, 15
- AG, 13
- AG ≤k, 15
- algorithm, 5
- deterministic, 29
- Fokkink, no roundtrip, 51
- Fokkink, roundtrip, 52
- Itai, round mod K, 61
- Itai, round < K, 60
- Las-Vegas, 29
- random, 29
- Sorting, 51
- timing, 50

"allState2", 57
allState2 preservation property, 58
always, 12
ample(s), 128
ample decomposition condition, 130
ample set, 128
ample(E), 130
ample(G), 130
ampleC(s), 129
amplec(E), 130
amplecG(s), 129
amplec(s), 129
ample(prune(s)), 132
anonymity, 25
anonymous
- network, 25
- nodes, 25
aperiodicity property, 88
application domain, 9
approximation, 39
AR, 13
arbitrary transmissions, 50
array, 21
assert(), 24
assertion, 24
- x, 11
- s, 11
assertions, 11
associativity, 13
asynchronous
- execution, 23
- parallel composition, 17
- transmission, 21
asynchronous transmission, 29
atomic, 28
AU, 13
AU ≤A, 13
average expectation, 94
average probability, 84
AX, 22

B
Büchi automaton, 26
basic command, 21
basic statement, 21
basic successor, 140
BDD, 15
behaviour
- of components, 23
- of modules, 10
behaviours, 21
best case, 74
bibliographical references, 2
bibliography, 2
bidirectional, 27
bidirectional line, 28
bidirectional ring, 29
binary decision diagram, 10
bit rotation, 35
bitstate hashing, 108
blocked, 5
- sequence, 21
blocking never claim, 21
blow-up, 26
bold, 1
bounded branching-time operators, 15
bounded linear-time operator, 14
bounded until operator, 14
brace, 24
branching-time operators, 13
branching-time temporal logic, 11
break, 22
breaking symmetry, 28
buffer, 27
buffered channel, 21

C
C, 7
C0, 130
C1, 130
C2, 130
C3, 130
C3', 130
capacity, 21
CEGAR, 40
centralized protocol, 29
cchannel, 24
cchannel, 21
cchannel, overtaking, 54
channel
channel_{cal}), 55
channel_{imple}, 55
circular shift, 55
circular shifts property, 57
cleaning up all scratch data, 57
closed, 11
closes, 121
collapse compression, 39, 105
collide, 11
Colorado University Decision Diagram, 123
command, 16
communication, 21
commutativity, 13
Commutativity, 129
compact model representation, 10
compile time, 13
compile-time option, 115
complete graph, 23
complexity limit, 122
composed data type, 21
composed in parallel, 17
composition,
- asynchronous parallel, 17
- full parallel, 17
- restricted parallel, 17
compound command, 22
compound statement, 22
compression, 53
- collapse, 59
- minimized automaton, 40
- ratio, 60
computation tree
- of a graph, 7
- of an LTS, starting from $s$, 7
- of an STS, starting from $s$, 7
computation tree logic, 11
compute probabilities, 62
computing element, 27
concatenated, 139
concatenation, 22
concurrency, 16
conditional expression, 24
confluent successor, 140
connection, 27
constant
- in a PRISM model, 18
- in a PRISM property, 20
- in a SPIN model, 24
constraining execution, 23
continuous stochastic logic, 20
continuous time Markov chain, 10
control flow construct, 22
correct
- partially, 65
correct algorithm, 61, 74
correctly terminating paths, 98
correctly terminating paths property, 109
correctness properties, 79, 80
cost
- of administering IDs, 28
- of an error, 1
cost structure, 7
counterexample, 8, 13, 20
counterexample-guided abstraction refinement, 20
CSL, 20
CTL, 11, 13
CTL*, 11
- semantics, 83
- syntax, 12
ctl*, 11
CTMC, 16
cudd
- epsilon, 68
- max memory, 68
CUDD, 63
cycle closing condition, 130
D
data, 1, 10, 21
data type abstraction, 41
database theory, 5
de-anonymization, 28
deadlock, 11
"deadlock", 20, 38
decentralized environment, 29
decision procedure for $A_{sorting}$, 35
decoration, 16
deductive verification, 1
definition, 1
dependent, 129
depth, 108
depth-first search, 121
- incremental, 122
- nested, 121
destination, 4
deterministic, 5
deterministic algorithm, 29
DFS, \[127\]
- incremental, \[128\]
- nested, \[127\]
directed, \[27\]
disambiguation, \[27\]
discrete time Markov chain
- labelled, \[5\]
discrete-time Markov chain, \[7\]
distributed algorithms, \[27\]
distributive, \[27\]
do, \[22\]
do prob checks, \[62\]
DOT, \[1\]
d-step, \[23\]
dtmc, \[17\]
DTMC, \[7\]
- labelled, \[6\]
duplex, \[27\]
-DVECTORS, \[115\]

E, \[12\]
E[ ], \[13\]
\(E^{\leq k}\), \[15\]
EG, \[15\]
\(E^{\leq k}\), \[15\]
E(\(\Box f_1\)), \[15\]
E(\(\Box^{\leq k} f_1\)), \[15\]
E(\(\Diamond f_1\)), \[15\]
E(\(\Diamond^{\leq k} f_1\)), \[15\]
E(\(\neg F f_1\)), \[15\]
E(\(\neg F^{\leq k} f_1\)), \[15\]
E(\(\neg G f_1\)), \[15\]
E(\(\neg G^{\leq k} f_1\)), \[15\]
E(\(\neg X f_1\)), \[15\]
E(\(\neg (f_1 \cap f_2)\)), \[15\]
E(\(\neg (f_1 U f_2)\)), \[15\]
E(\(\neg (f_1 U^{\leq k} f_2)\)), \[15\]
E(\(\neg \Box f_1\)), \[15\]
E(\(\neg \Box^{\leq k} f_1\)), \[15\]
E(\(\neg \Diamond f_1\)), \[15\]
E(\(\neg \Diamond^{\leq k} f_1\)), \[15\]
E(\(\neg f_1 R f_2\)), \[15\]
E(\(f_1 U f_2\)), \[15\]
E(\(f_1 U^{\leq k} f_2\)), \[15\]
E(\(F f_1\)), \[15\]
E(\(F^{\leq k} f_1\)), \[15\]
E(\(G f_1\)), \[15\]
E(\(G^{\leq k} f_1\)), \[15\]
E(\(X f_1\)), \[15\]

early stage of development, \[6\]
edge, \[27\]
EF, \[13\]
EF\(^{\leq k}\), \[15\]
EG, \[15\]
EG\(^{\leq k}\), \[15\]
election, \[27\]
case, \[21\]
emptiness condition, \[130\]
empty buffer property, \[69\], \[100\]
enabled, \[6\], \[10\]
enabled(\(s\)), \[6\]
enabled(\(\cdot\)), \[23\]
Enanized, \[129\]
encapsulation behaviour, \[61\]
end, \[22\]
end label, \[22\]
end of a path, \[6\]
end state, \[7\]
end state validity, \[11\], \[21\]
end rewards, \[17\]
engine
- hybrid, \[62\]
- MTBDD, \[62\]
- sparse (matrices), \[22\]
Engine, \[62\]
ER, \[13\]
E [rounds], \[74\]
error signal, \[30\]
errors
- in SPIN, \[118\]
E [transmissions], \[76\]
EU, \[13\]
EU\(^{\leq k}\), \[15\]
eventually, \[12\]
EX, \[13\]
exact approximation, \[40\]
exception handling, \[24\]
exclusive
- read, \[11\]
- send, \[11\]
exclusive privilege, \[23\]
executable, \[21\]
exhaustive enumeration
- explicit, \[5\]
- implicit, \[5\]
expected number of message transmissions, \[76\]
expected number of rounds, \[74\]
experiment in PRISM, \[69\], \[63\]
explicit, 16
explicit exhaustive enumeration, 8
exploiting asymmetry, 28
explored state space, 121
exponential, 20
expression, 21

F, 12
F≤k, 15
fair, 18
fair computation tree, 58
fairness, 62, 23
false, 21
false negative, 40
false positive, 40
fault tolerance, 29
FCFS, 25
FCFS transmissions, 30, 11
f, 22
FIFO, 28
FIFO buffer, 21, 25
filter, 20, 44
finite, 4
finite state space, 4
first come, first served, 27
first in, first out, 28
flooding, 29
Fokkink, no roundtrip algorithm, 51
Fokkink, roundtrip algorithm, 52
formal deductive verification, 1
formula, 15, 20
full parallel composition, 17
fully expanded, 129
function, 13

G, 12
G≤k, 15
global progress state, 121
global variable
– in PRISM, 16
– in SPIN, 21
globally, 12
goto, 22
graph, 6, 24
– complete, 25
guard, 21
guard, 11

h, 37
hard-coded initialization, 59, 31, 109
hardware description, 5
hardware used for the verifications, 22
have a trial, 139
hidden variable, 24
hiding of actions, 17
hop, 59
hop counter, 59
hybrid
– max memory, 35
– num. levels, 63
hybrid engine, 22
hypercube, 28

I, 4
id, 31
ID, 23
identification numbers, 28
if, 22
if and only if, 4
iff, 4
implicit exhaustive enumeration, 8
incomplete, 10
incremental depth-first search, 122
incremental DFS, 122
incremental DFS, 122
independent, 129
independent successor, 140
indistinguishable nodes, 28
indivisibly, 23
induction
– on the number of active nodes,
51
– on the size of the ring, 29
"init", 20
init, 4, 21
init, 25
init process, 23
initial process instance definition, 24
initial states, 4
initial states definition, 18
initialized channel, 21
initializer, 21
inline, 24
interleavability, 29
interleaving
in PRISM, 10
– in SPIN, 23
interleaving diamond, 18
invariant under stuttering, 25
invisibility, 130
invisible, 130
Iterating mod K algorithm, 31
Iterating \leq K algorithm, 50
iterative method, 62
Jacobi, 62
K, 31
Kripke frame, 4
Kripke structure, 4
label
– ”deadlock”, 68
– ”limitReached”, 68
– ”allState2”, 87
– in PRISM, 20
– in SPIN, 22
– ”oneLeader”, 68 50
labelled discrete time Markov chain, 4
labelled DTMC, 8
labelled Markov decision process, 5
labelled MDP, 5
labelled transition system, 5
Las-Vegas algorithm, 26 30
leader, 27
leader, 31
leader election, 27
– stronger definition, 28
leader existence property, 57 111
leader preservation invariance property, 68 106
length, 5
length of a path, 3
”limitReached”, 65
linear equation system, 52
linear-time logic, 14
linear-time operator, 12
linear-time temporal logic, 14
link, 27
liveness check, 24
local algorithm, 24
local progress state, 121
local variable
– in PRISM, 10
– in SPIN, 21
lockstep, 21
logic
– branching-time temporal, 11
– computation tree, 11
– linear-time, 11
– linear-time temporal, 11
logically
– unsound, 40
LTL, 11 13
LTL formula, 22
LTS, 5
M, 7 31
MA, 106
macros, 24
Markov decision process, 8
– labelled, 8
matched, 106
maximum number of iterations, 65
mc, 3
MDP, 5
– labelled, 3
mem, 109
memory overflow, 65
message, 21 27
– complexity, 60
– overtaking, 44
– overwriting, 44
message driven, 31
minimized automata, 106
minimized automaton, 109
mini-world, 5 24
model, 3
– for a formula, 12
– of a network, 27
model checker, 3 12
model checking, 3
– of source code, 29
modularization, 16
module renaming, 10
modules, 10
monitor execution steps, 24
Monte-Carlo algorithm, 30 31
MTBDD, 15 16
MTBDD engine, 32
mtype, 24
multiterminal binary decision dia-
gram, 13

N

n, 24
N, 119
nested depth-first search, 121
nested DFS, 107
network, 27
never, 23
never claim, 11 12
never claim process, 23
nexttime operator, 12
n-fold composition, 29
no roundtrips, 24
node, 27
non-orthogonal action labels, 44
non-probabilistic algorithm, 29
non-progress action, 111 24 21
non-progress cycle, 111 24 21
non-progress cycle check, 121
non-progress cycle existence pres-
ervation, 180
normalization, 3 17
note, 17 50 53 58 59 39 38 31
107 122 125 128 131
NotInStack condition, 130
np_, 24 14
np cycle, 121
numerical computation, 62
numerical method, 72

O

od, 22
on-the-fly, 20 135 132
on-the-fly model checking with con-
dition c2, 139
on-the-fly model checking without
condition c2, 133
oneLeader property
  - for Las-Vegas algorithms, 73
  - for Monta-Carlo algorithms,
    97
"oneLeader"
  - for Fokkink algorithms, 84
"oneLeader"

 – for Itai algorithms, 65
 operator basis, 12
 operators
  - process algebraic, 17
 options for PRISM, 62
 output, 22
 over-approximate, 40

P

parallel composition, 16
parallelism, 16 26
parameterized with a property, 10
parse, 53
partial order reduction, 12 106 121
129
partially correct, 93
passive, 30 31 56
path, 6
  - end, 6
  - start, 6
path formula, 12
path quantifier, 12
pattern, 23
pc, 21
PCTL, 11 13 102
pctl, 13
pc_value(), 23
po reduction, 129
precomputation, 52
predicate abstraction, 41
premature termination, 22
preprocessor, 21
printn(), 22
PRISM, 15
  - experiment, 59 63
prism's fairness, 98
privilege, 28
probabilistic computation tree logic,
13
Probabilistic Computation Tree Logic,
102
probabilistic model checker, 2
probabilistic model checking, 2
probabilistic operators, 13
probabilistic statement, 8
process
  - in a network, 27
  - in SPIN, 22
Index

process algebraic composition operators, 17
process starvation, 29
processor, 27
proctype, 23
program counter, 21
progress, 121
  - label, 124 124 115
  - state, 121 22 121
  - transition, 125 195
progress, 24
progress counter, 124
PROMELA, 21
property, 1
  - allState2 preservation, 58
  - aperiodicity, 58
  - circular shifts, 57
  - correctly terminating paths, 25
  - correctness, 58
  - empty buffer, 59
  - Fokkink, 50
  - leader existence, 57
  - leader preservation, 58
  - leader preservation invariance, 25
  - oneLeader, 24 24
  - termination, 58
propositional temporal logic, 11 11
protocol, 11
prune
  - computation tree, 21
prune \((E)\), 124
prune \((G)\), 124
prune \((E)\), 124
prune \((G)\), 124
prune \((E)\), 124
prune \((G)\), 124
prune \((E)\), 124
prune \((s)\), 123
prune \((s)\), 123
prune \((s)\), 123
prune \((s)\), 123
prune \((s)\), 123
prune \((s)\), 123
prune \((s)\), 123
push-button technology, 119

- q, 116
queue, 28

R, 12

race condition, 74 115
random algorithm, 29
random ID, 51
random period of time, 50
random receive, 22
randomness, 20
range, 18
reachability test, 65
reachable, 47
reachable state space, 7
reaches \(N\), 139
Reactive Modules, 10
read exclusive, 11
receive, 24 27
reducing the state space, 50 57
reduction
  - partial order, 121
  - reference, 11
  - relative, 22
  - release operator, 112
  - relevant behaviours, 40
  - remoterefs, 23
  - renaming of actions, 17
  - rendezvous, 28
  - reordering of variables, 57
  - repetition, 22
  - restricted parallel composition, 17
  - rewards, 17
  - right-unique, 6
  - ring, 55 29
  - notation, 55 12
  - round, 51 60
  - round number, 13
  - roundtrips, 44
  - routing protocol, 29
  - run, 24 23
  - run-time option, 106
  - runtime, 12

S
safety check, 24
scheduling
  - in PRISM, 16
  - in SPIN, 23
second leader preservation property, 58
selection, 22
selective data hiding, 411
selective restriction, 31
self-loop, 28
semantics of CTL*, 13
send, 21 22
send exclusive, 11
sequence, 22
sequence blocks, 21
sequence of subproperties, 103
\( \sigma \)-algebra, 10
\( \sigma (Path^n) \), 10
signal 11, 83
signal 22, 83
simplex, 27
simulated parallelism, 10 23
size of a ring, 29
size of the problem instance, 29
skip, 21
slicing algorithm, 11
soft deadline, 81
sometimes, 12
sorted send, 22
Sorting algorithm, 31
source, 4
source code, 4
source code model checking, 29
space (matrices) engine, 12
special variable, 24
SPIN, 20
spurious errors, 10
standard receive, 22
standard send, 22
start of a path, 8
starving process, 99
state explosion, 20
state formula, 11
state machine, 20 10 30
state space, 4
- finite, 4
- reachable, 4
state space explosion, 20
state transition system, 4
state vector, 4 4
statement, 5
statement merging, 109
statement merging, 43
statement, (t), 123
states, 109
statespace, 121
statically, 43
strong fairness, 99
stronger definition of leader election, 28
strongly connected component, 99
STS, 4
stutter invariant, 25
stutter-closed, 25
subproperty, 113
super-exponential, 70
supertrace, 108
symbolic, 15
symbolic constant, 21
symbolic name, 25
symmetric, 28
- network, 28
symmetry breaking, 28
symmetry reduction, 22
synchronization, 10 14 17
synchronize, 10
synchronous transmission, 21 28
system, 23
system module, 17

T

T, 106

T. 4

temporal logic, 11
- branching-time, 11
- linear-time, 11
- propositional, 11
temporal operators, 11
temporally decoupled, 27
terminal, 4 54
termination
- criteria, 92
- epsilon, 92
- max iterations, 93
- of an algorithm, 93
termination property, 93 103
testing, 99
threshold, 92
timed automata, 29
timeout, 103
timeout, 21
timing algorithm, 86
token loss, 29
token ring, 29
topology, 28
transition, 41
transmission, 24
transmission complexity, 76
transmissions
  - arbitrary, 30,11
  - FCFS, 30,11
trapped path, 99
trial, 139
trial monotony, 139
trivial DFS technique, 107,122
type casting, 23
type of a channel, 21
typedef, 24
U
ubiquitous computing, 28
unbuffered, 28
under-approximate, 30
undirected, 27
unidirectional, 27
unidirectional ring, 29
uninitialized channel, 21
\textit{unique}, 31
unlabelled state transition system,
\textit{U}
unless, 22
unreachable code, 19,21
unreachable code existence, 19,105
unsound, 10
until operator, 12
update
  - in PRISM, 16
  - of the state vector, 5
UPPAAL, 15
use case, 1
use fairness, 92
use precomputation, 62
V
valid end state, 22
  - label, 22
variable, 4,16
  - global, in PRISM, 16
  - global, in SPIN, 21
  - local, in PRISM, 16
  - local, in SPIN, 21
variable assignment, 21
variable decoration, 16
verification, 4
verification time, 118
version
  - of PRISM, 65
W
weak fairness, 59
weak symmetry breaking, 50
worst case, 69,73
X
x, 12
xr assertion, 11,24,105
xs, 11,24
xs assertion, 11,24,105
xspin, 105