Modelling and Analysis of Communicating Systems

Jan Friso Groote & Michel Reniers
Department of Computer Science
Eindhoven University of Technology, Eindhoven
Email: J.F.Groote@tue.nl, M.A.Reniers@tue.nl

Revision: 2601
Date: 2010-08-26 21:52:10 +0200 (Thu, 26 Aug 2010)

Preface

Robin Milner observed in 1973 that the primary task of computers appeared to be interacting with their environment, yet the theory of programs and programming at that time seemed to ignore this fact completely [39, 40]. As a consequence, he set out working on his seminal book [41, 43] in which he developed the CCS, the Calculus of Communicating Systems. At the same time two other main process algebras were developed, namely ACP (Algebra of Communicating Processes, [6]) and CSP (Communicating Sequential Processes, [29, 30]).

Interesting as they were, these process algebras were too bare to be used for the description of actual systems, mainly because they lacked a proper integration of data. In order to solve this, process algebraic specification languages have been designed (most notably LOTOS [32] and PSF [38]) which contained both data and processes. A problem with these languages was that they were too complex to act as a basic carrier for the development of behavioural analysis techniques.

We designed an intermediate language, namely mCRL2 (and its direct predecessor μCRL [23, 20]) as a stripped down process specification language or an extended process algebra. It contains exactly those ingredients needed for a complete behavioural specification, and its (relative) simplicity allows to concentrate on proof and analysis techniques for process behaviour.

Throughout the years many of these techniques have been developed. To mention a few: the Recursive Specification Principle, Invariants, Confluence, Cones and Foci, Abstract Interpretation and Coordinate Transformations, Boolean Equation Systems, Proof by Patterns, etc. All these results together have constituted a mathematical framework suitable to launch a mathematical 'attack' on most phenomena that are not properly understood in process behaviour. They also form a very good framework to formulate and prove the correctness of complex and intricate protocols.

Up till now, all these results were lingering around in the literature. We combined them in this book, added exercises and examples to make the developed material suitable for self study and for teaching.

Acknowledgements

The first version of this book appeared as a handbook chapter [24]. This chapter formed the basis of a reader [15] used for courses at several universities (mostly written by Wan Fokkink). These earlier publications were based on the modelling language μCRL (micro Common Representation Language, [23, 20]) essentially developed in 1991. In 2003 we decided that it was time for a successor, to increase the usability of the μCRL, and we decided to baptise its successor mCRL2. The essential difference is that mCRL2 has the basic datatypes as part of the language, contrary to μCRL which contained only a mechanism to define datatypes. This book is solely based on mCRL2.

The development of mCRL2 builds upon the development work on process algebra's between 1970 and 1990. Especially the work on CCS (Calculus of Communicating Processes) by Robin Milner [41] and ACP (Algebra of Communicating Processes) by Jan Bergstra, Jan Willem Klop, Jos Baeten, Rob van Glabbeek and Frits Vaandrager [6, 2] formed an important basis. An essential step was the EC SPECS project, where a megalomane *Common Representation Language* had to be developed to represent all behavioural description languages that existed at that time (LOTOS, CHILL, SDL, PSF) and that still had to be developed. As a reaction a micro Common Representation Language (μCRL) had been developed in which Alban Ponse was instrumental. Bert Lisser was the main figure behind the maintenance and development of the tools to support μCRL.

The following people have contributed to the development of mCRL2, its tools and its theory: Muck van Weerdenburg, Aad Mathijssen, Bas Ploeger, Tim Willemse, Wieger Wesselink, Jeroen van der Wulp, Frank Stappers, Frank van Ham, Hannes Pretorius, Jaco van de Pol, Yaroslav Usenko, Jeroen Keiren, Carst Tankink and Tom Haenen.

This book is used as a reader for the course Requirements, Analysis, Design and Verification at Eindhoven University of Technology. Many thanks go to Jeroen Keiren for his careful proofreading. Valuable feedback also came from Muhammad Atif, Harsh Beohar, Debjyon Bera, Dwight Berendse, Anton Bilos, Michiel Bosveld, Gert-Jan van den Braak, Christoph Brandt, Mehmet Çubuk, Sven Goossens, Albert Hofkamp, Hossein Hojjat, Albert Hofkamp, Bas Kloet, Diana Koenraadt, Geert Kwintenberg, Koen van Langen, Mohammad Mousavi, Chidi Okwudire, Mathijs Opdam, Chidi Okwudire, Eva Ploum, André van Renssen, Marcel Roeloffzen, Koos Rooda, Frank Stappers, Carst Tankink, Sander Verdonschot, Amrita Vikas Sinha, Migiel de Vos and many others.

Jan Friso Groote and Michel Reniers
September 2007, Eindhoven, The Netherlands

Contents

I Modeling system behaviour **9**

1 Introduction **11**

2 Actions, behaviour, equivalence and abstraction **13**
2.1 Actions . . . 13
2.2 Labelled transition systems . . . 14
2.3 Equivalence of behaviours . . . 15
2.3.1 Strong bisimulation equivalence . . . 15
2.3.2 Trace equivalence . . . 18
2.3.3 Language and failures equivalence . . . 19
2.3.4 The Van Glabbeek linear time – branching time spectrum . . . 20
2.4 Behavioural abstraction . . . 22
2.4.1 The internal action τ . . . 22
2.5 Behavioural equivalence for the internal action . . . 23
2.5.1 Rooted branching bisimulation . . . 23
2.5.2 Rooted weak bisimulation . . . 26
2.5.3 Weak trace equivalence . . . 27

3 Timed process behaviour **29**
3.1 Timed actions and time deadlocks . . . 29
3.2 Timed transition systems . . . 31
3.3 Timed process equivalences . . . 32
3.3.1 Timed (strong) bisimulation . . . 32
3.3.2 Timed branching bisimulation . . . 32
3.3.3 Timed weak bisimulation . . . 33
3.3.4 Timed trace equivalence . . . 33
3.3.5 Timed weak trace equivalence . . . 33

4 Algebraic process descriptions **35**
4.1 Basic processes . . . 35
4.1.1 Actions . . . 35
4.1.2 Multi-actions . . . 36
4.1.3 Alternative and sequential composition . . . 37
4.1.4 Deadlock . . . 39
4.1.5 The conditional and sum operator . . . 39
4.1.6 Recursive processes . . . 40
4.2 Data types . . . 41
4.2.1 Basic data type definition mechanism . . . 41
4.2.2 Standard data types . . . 44
4.2.3 Booleans . . . 45
4.2.4 Numbers . . . 46
4.2.5 Lists . . . 48

4.2.6 Sets and bags . . . 49
4.2.7 Function types . . . 49
4.2.8 Structured types . . . 51
4.2.9 Terms and where expressions . . . 52
4.3 Timed processes . . . 53
4.4 Parallel processes . . . 55
4.4.1 The parallel operator . . . 55
4.4.2 Communication between parallel processes . . . 58
4.4.3 Blocking and renaming . . . 60
4.5 Hiding internal behaviour . . . 61
4.6 Alphabet axioms . . . 63

5 Describing properties in the modal μ-calculus 67
5.1 Hennessy-Milner logic . . . 67
5.2 Regular formulas . . . 69
5.3 Fixed point modalities . . . 70
5.4 Modal formulas with data . . . 73
5.5 Modal formulas with time . . . 74
5.6 Equations . . . 76

6 Modelling of system behaviour 79
6.1 Alternating bit protocol . . . 79
6.2 Sliding window protocol . . . 82
6.3 A patient support platform . . . 85

7 Semantics 93
7.1 Semantics of the datatypes . . . 93
7.1.1 Sorts . . . 93
7.1.2 Signatures . . . 94
7.1.3 Well-typed data expressions . . . 96
7.1.4 Free variables and substitutions . . . 98
7.1.5 Data specifications . . . 99
7.1.6 Semantics of datatypes . . . 100
7.2 Operational semantics of the mCRL2 processes . . . 101
7.2.1 Well-typed processes, action declarations and process equations . . . 101
7.2.2 Semantical multi-actions . . . 103
7.2.3 Operational semantics . . . 105
7.3 Validity of modal μ-calculus formulas . . . 107
7.4 Soundness and completeness . . . 109

II Model transformations 111

8 Basic manipulation of processes 113
8.1 Introduction . . . 113
8.2 Derivation rules for equations . . . 113
8.3 Derivation rules for formulas . . . 118
8.4 Induction for constructor sorts . . . 118
8.5 The sum elimination lemma . . . 120
8.6 Recursive specification principle . . . 121
8.7 Koomen's fair abstraction rule . . . 124
8.8 Parallel expansion . . . 125
8.8.1 Basic parallel expansion . . . 125
8.8.2 Parallel expansion with data: two one-bit buffers . . . 126

8.8.3 Parallel expansion with time . . . 128

9 Linear Process Equations and Linearisation 131
9.1 Linear process equations . . . 131
9.1.1 General linear process equations . . . 131
9.1.2 Clustered linear process equations . . . 133
9.2 Linearisation . . . 133
9.2.1 Linearisation of sequential processes . . . 134
9.2.2 Parallelization of linear processes . . . 138
9.2.3 Linearisation of n parallel processes . . . 139
9.3 Proof rules for linear processes . . . 141
9.3.1 τ-convergence . . . 141
9.3.2 Convergent Linear Recursive Specification Principle (CL-RSP) . . . 142
9.3.3 CL-RSP with invariants . . . 143

10 Confluence and τ-prioritisation 147
10.1 τ-confluence on labelled transition systems . . . 147
10.2 τ-prioritisation labelled transition systems . . . 149
10.3 Confluence and linear processes . . . 151
10.4 τ-prioritisation for linear processes . . . 153
10.4.1 Using confluence for state space generation . . . 154

III Checking conformance between specification and implementation 155

11 Cones and foci 157
11.1 Cones and foci . . . 157
11.2 Protocol verifications using the cones and foci proof technique . . . 160
11.2.1 Two unbounded queues form a queue . . . 160
11.2.2 Milner's scheduler . . . 162
11.2.3 The alternating bit protocol . . . 163

12 Verification of distributed systems 167
12.1 Tree identify protocol . . . 167
12.1.1 The correctness of the tree identify protocol . . . 169
12.2 Sliding window protocol . . . 171
12.2.1 Linearization . . . 172
12.2.2 Getting rid of modulo arithmetic . . . 172
12.2.3 Proving M equal to a bounded queue . . . 176
12.2.4 Correctness of the sliding window protocol . . . 178
12.3 Distributed summing protocol . . . 179
12.3.1 A description in mCRL2 . . . 180
12.3.2 Linearisation and invariants . . . 181
12.3.3 State mapping, focus points and final lemma . . . 183

IV Checking properties of systems 187

V Appendices 189

A Equational definition of built in datatypes 191
A.1 Bool . . . 191
A.2 Positive numbers . . . 192
A.3 Natural numbers . . . 193

A.4 Integers . . . 196
A.5 Reals . . . 198
A.6 Lists . . . 199
A.7 Sets . . . 200
A.8 Bags . . . 201
A.9 Structured types . . . 201

B Syntax of the formalisms **203**
B.1 Lexical part . . . 203
B.2 Conventions to denote the context free syntax . . . 203
B.3 Identifiers . . . 203
B.4 Sort expressions and sort declarations . . . 204
B.5 Declaration of constructors and mappings . . . 204
B.6 Declaration of equations . . . 204
B.7 Data expressions . . . 205
B.8 Communication and renaming sets . . . 206
B.9 Process expressions . . . 206
B.10 Action declaration . . . 206
B.11 Process and initial state declaration . . . 206
B.12 Syntax of an mCRL2 specification . . . 206

C Axioms for processes **207**

D Answers to exercises **211**

References **221**

Part I

Modeling system behaviour

Chapter 1

Introduction

In today's world virtually all designed systems contain computers and are connected via data networks. This means that contemporary systems behave in a complex way and are continuously in contact with their environment. For system architects, designing and understanding the behaviour of such systems is a major aspect of their task. This book deals with the question of how to model system interaction in a sufficiently abstract way, such that it can be understood and analysed. In particular, it provides techniques to prove that interaction schemes fullfill their intended purpose.

In order to appreciate this book, it is necessary to understand the complexity of contemporary system communication. As an extremely simple example, take the on/off switch of a modern computer. We do not have to go far back in history to find that the power switch had a very simple behaviour. After the power switch was turned off, the computer was dead, and it would not attempt to become involved in any communication of whatever kind anymore.

In a modern computer, the on/off switch is connected to the central processor. If the on/off switch is pressed, the processor will be signalled that it must switch off. The processor will finish current tasks, shut-down its hardware devices, and inform (or even ask permission from) others via its networks that it intends to go down. So, nowadays, an increased message traffic can be observed after the shut down button is pressed.

Given the complexity of systems, it is not at all self-evident anymore whether a system will respond and if it does so, what the correct interpretation of the response will be. The number of different message types generally is substantial. For large systems dozens of different message types are possible. The possible orderings of these messages in a component can be represented by an automaton. In a well designed system the number of states in such an automaton is small, but in practice the number tends to be huge. Especially, when the effect of the data transferred in messages is taken into account, the size of these automata quickly becomes draconic.

In general, the state space of a system is of the same order of magnitude as the product of the sizes of the automata of each component. The number of different message sequences (or traces) is in general exponential in the number of states of the system. The numbers indicating sizes of states spaces are of the kind 10^{1000} or even $10^{(10^{10})}$ and exceed the well-known astronomical numbers by far. It is completely justified to speak about a whole new class of numbers, i.e. the *computer engineering numbers*.

Of course such numbers would mean nothing if the design of communicating systems would not lead to problems. But unfortunately, on virtually all levels system communication is causing problems. Many distributed algorithms published in the literature turn out to be wrong, or if correct their proofs contain flaws. For most of the known standard communication protocols so many serious flaws have been revealed, that it is very likely to assume that the newest accepted international standards contain literally hundreds of serious — yet to be uncovered — bugs. Adding our own experience, we have far too often been confronted with subtle bugs in communication schemes that we designed.

This all leads to a strong belief that without proper mathematical theory, appropriate proof- and analysis methods and adequate computer tools, it is impossible to design correctly communicating systems. In this book we provide all these ingredients.

In chapter 2 the basic notion, namely an action is explained. Using transition systems, it is explained

how actions can be combined into behaviour. The circumstances under which behaviour can be considered the same are also investigated. In chapter 3 basic process theory is explained using time.

In chapter 4 behaviour is described in an algebraic way. This means that there are a number of behaviour combining operators of which the properties are characterized by axioms. In this chapter the standard constructs to describe data are also introduced.

The next chapter, i.e. chapter 5, explains how to describe behavioural properties in the modal mu-calculus. By integrating data in this calculus, we cannot only state simple properties, such as a system is deadlock-free, but also very complex properties that depend on fair behaviour or data processing.

The first part is concluded with chapter 6 that contains a number of example descriptions of the behaviour of some simple systems.

In chapter 7 the semantics of mCRL2 and the modal logic are described. In essence, this provides a mapping between syntactically or algebraically described processes on the one hand and a transition system on the other hand. Using this mapping it is possible to establish the relation between the process equivalences given in chapter 2 and the axioms in chapter 4.

The second part of the book deals with process manipulation. Chapter 8 provides basic technologies to transform one process into another. Typical techniques are induction, the recursive specification principle and the expansion theorem.

Chapter 9 describes how to transform processes to the so-called linear form, which will play an important role in all subsequent manipulations. Moreover, it will reformulate the principle RSP to the more concise principle CL-RSP (RSP for convergent linear processes).

Chapter 10 deals with confluence which is a typical behavioural pattern that originates from the parallel composition of two processes. If a process is found to be confluent, we can reduce it using so-called τ-priority. By giving priority to certain τ-actions the state space can be reduced considerably.

In chapter 11 the cones and foci technique is explained, which allows to prove that a specification and an implementation of certain behaviour is equivalent. In the subsequent chapter 12 the accumulated techniques are used to prove a number of typical protocols and distributed algorithms correct. In chapter 12 the verification techniques are used to validate some complex distributed algorithms. In chapter **??** it is shown how parameterised boolean equations can be used to verify modal formulas on linear processes.

In appendix A the equations characterising datatypes are given. In appendix B an overview of the syntaxes of all different formalisms is given. Appendix C summarises all process algebraic axioms. The last appendix D contains answers to the exercises.

An extensive toolset has been developed for mCRL2. It can be downloaded from `www.mcrl2.org`.

Chapter 2

Actions, behaviour, equivalence and abstraction

In this chapter the basic notions of (inter)action and behaviour are explained in terms of transition systems. It is discussed when different transition systems can behave the same and it is indicated how complex behaviour can be abstracted by hiding actions.

2.1 Actions

Interaction is everywhere. Computer systems, humans, machines, animals, plants, molecules, planets and stars are all interacting with their environment. Some interactions are continuous such as gravity pulling stellar objects towards each other. Other interactions take place pointwise in time, such as shaking hands or sending a message.

Within engineering continuously interacting systems were paramount. The forces on a bridge or building, the burning of fuel in combustion engines or the characteristics of an electronic circuit had to be mastered by a mathematical theory for continuous interaction.

However, with the advent of computers, systems tend to communicate in a pointwise manner. Slightly worrying, the complexity of message exchanges among computerized systems is currently exceeding the complexity of the more traditional engineering artefacts. This complexity needs to be tamed by making models and having the mathematical means to understand these models. The first purpose of this book is to provide the modelling means and mathematical analysis techniques to understand interacting systems. The second derived purpose is to provide the means to design such systems such that we know for sure that they work correctly.

Basic actions, also called interactions, are the basic ingredients of such models. We denote them abstractly by letters a, b, c or more descriptively by $read$, $deliver$, $timeout$, etc. They are generally referred to as actions and they represent some observable atomic event. The action $deliver$ can represent the event of a letter being dropped in a mailbox. An action *read* can consist of reading a message on a computer screen.

The fact that an action is atomic means that actions cannot overlap each other. For every pair of actions a and b, the one happens before the other, or vice versa. Only in rare cases they can happen exactly at the same moment. We write this as $a|b$ and call this a *multi-action*. It is possible to indicate that an action can take place at a specific time. E.g., $a{\cdot}3$ means that action a must take place at time 3. For the moment multi-actions and time are ignored. We come back to it in the chapters 3 and 4.

Exercise 2.1.1. What are the interactions of a CD-player? What are the actions of a text-editor? And what of a data-transfer channel?

2.2 Labelled transition systems

The order in which actions can take place is called behaviour. Behaviour is generally depicted as a labelled transition system, which is a directed labelled graph. A labelled transition system consists of a set of states, and a set of transitions labelled with actions that connect the states. Labelled transition systems must have an initial state, which is depicted by a small incoming arrow. They can also have terminating states, generally indicated with a small tick or square root symbol (✓). In figure 2.1 the behaviours of two simple processes are depicted. Both can perform the actions a, b, c and d. At the end, the lower one can terminate, whereas the upper one cannot do anything anymore. It is said to be in a *deadlock*, i.e. in a reachable state that does not terminate and has no outgoing transitions.

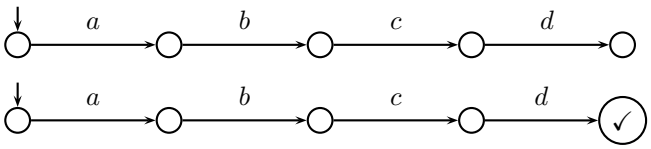


Figure 2.1: Two simple linear behaviours of which the lower one can terminate

Such simple diagrams are already useful to illustrate different behaviours. In figure 2.2 the behaviours of two alarm clocks are drawn. The behaviour on the left allows for repeated alarms, whereas the behaviour on the right only signals the alarm once. Note also that the behaviour at the left only allows a strict alternation between the set and the $reset$ actions, whereas this is not the case in the right diagram.

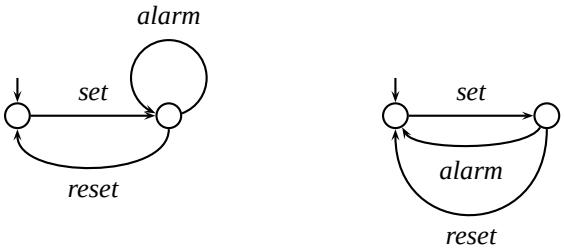


Figure 2.2: Two possible behaviours of an alarm clock

A state can have more than one outgoing transition with the same label to different states. The state is then called *nondeterministic*. A deterministic transition system contains no reachable nondeterministic states. Nondeterminism is a very strong modelling aid because it allows to model behaviour despite the fact that the exact behaviour is not clear. For instance if it is unclear how often the alarm can be repeated, this can be modelled by the behaviour of figure 2.3. If an alarm sounds, you cannot tell whether it is the last one, or whether there are more to follow. Even in case it is clear that the alarm sounds exactly 714

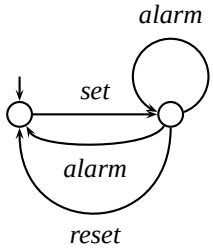


Figure 2.3: Nondeterministic behaviour of an alarm clock

times before stopping, it can be useful to describe it using the model of 2.3. Often the fact that the alarm sounds exactly 714 times does not outweigh the increased complexity of the model.

Milner was one of the early defenders of this use of nondeterminism [39, 41]. He called it the weather condition. The weather determines the temperature. The temperature influences the speed of processors and clocks in a computer. This may mean a timeout may come just too late, or just too early for some behaviour to happen. It generally is not effective to include a weather model to predict which behaviour will happen. It is much more convenient to describe all the behaviour in a nondeterministic way.

The general definition of a labelled transition system is the following.

Definition 2.2.1 (Labelled Transition System). A *labelled transition system* (*LTS*) is a five tuple $A = (S, Act, \longrightarrow, s, T)$ where

- S is a set of *states*.
- Act is a set of actions, possibly multi-actions.
- $\longrightarrow \subseteq S \times Act \times S$ is a *transition relation*.
- $s \in S$ is the *initial state*.
- $T \subseteq S$ is the set of *terminating states*.

It is common to write $t \xrightarrow{a} t'$ for $(t, a, t') \in \longrightarrow$.

Often, when not really relevant, the set T of terminating states is omitted, and it is also common that the initial state does not appear in the definition of an LTS.

Exercise 2.2.2. Make the following extensions to the alarm clock.

1. Draw the behaviour of an alarm clock where it is always possible to do a *set* or a *reset* action.
2. Draw the behaviour of an alarm clock with unreliable buttons. When pressing the *set* button the alarm clock can be set, but this does not need to be the case. Similarly for the *reset* button. Pressing it can reset the alarm clock, but the clock can also stay in a state where an alarm is still possible.
3. Draw the behaviour of an alarm clock where the alarm sounds at most three times when no other action interferes.

Exercise 2.2.3. Describe the transition system in figure 2.3 in the form of a labelled transition system conforming to definition 2.2.1.

2.3 Equivalence of behaviours

When do two systems have the same behaviour? Or stated differently, when are two labelled transition systems behaviourally equivalent? The initial answer to this question is simple. Whenever the difference in behaviour cannot be observed. The obvious next question is how behaviour is observed? The answer to this latter question is that there are many ways to observe behaviour and consequently many different behavioural equivalences. We present the most important ones here. For an overview see [18].

2.3.1 Strong bisimulation equivalence

Bisimulation equivalence (also referred to as strong bisimulation equivalence) or (strong) bisimilarity is the most important process equivalence [3, 42, 45]. The reason is that if two processes are bisimulation equivalent, they cannot be distinguished by any realistic form of behavioural observation. So, if two processes are bisimilar they can be considered equal.

The idea behind bisimulation is that two states are related if the actions that can be done in one state, can be done in the other, too. We say that the second action simulates the first. Moreover, if one action is simulated by another, the resulting states must be related also.

Definition 2.3.1 (Bisimulation). Let $A_1{=}(S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2{=}(S_2, Act, \longrightarrow_2, s_2, T_2)$ be labelled transition systems. A binary relation $R \subseteq S_1 \times S_2$ is called a *strong bisimulation relation* iff for all $s \in S_1$ and $t \in S_2$ such that sRt holds, it also holds for all actions $a \in Act$ that:

1. if $s \xrightarrow{a}_1 s'$, then there is a $t' \in S_2$ such that $t \xrightarrow{a}_2 t'$ with $s'Rt'$,
2. if $t \xrightarrow{a}_2 t'$, then there is a $s' \in S_1$ such that $s \xrightarrow{a}_1 s'$ with $s'Rt'$, and
3. $s \in T_1$ if and only if $t \in T_2$.

Two states s and t are *strongly bisimilar*, denoted by $s \leftrightarrows t$, if there is a strong bisimulation relation R such that sRt. The labelled transition systems A_1 and A_2 are *strongly bisimilar* iff their initial states are bisimilar, i.e. $s_1 R s_2$.

Often the adjective *strong* is dropped. Instead of speaking about a strong bisimulation relation we use the shorter bisimulation relation. However, we will see several other variants of bisimulation and in those cases the use of 'strong' helps us to stress the difference.

It is also possible to define bisimulation on the states of one single transition system. In this case the relation R is often referred to as an auto-bisimulation relation.

There are several techniques to show that one labelled transition system is bisimilar to another. Computer algorithms are generally based on the Relation Coarsest Partitioning Refinement [33, 44].

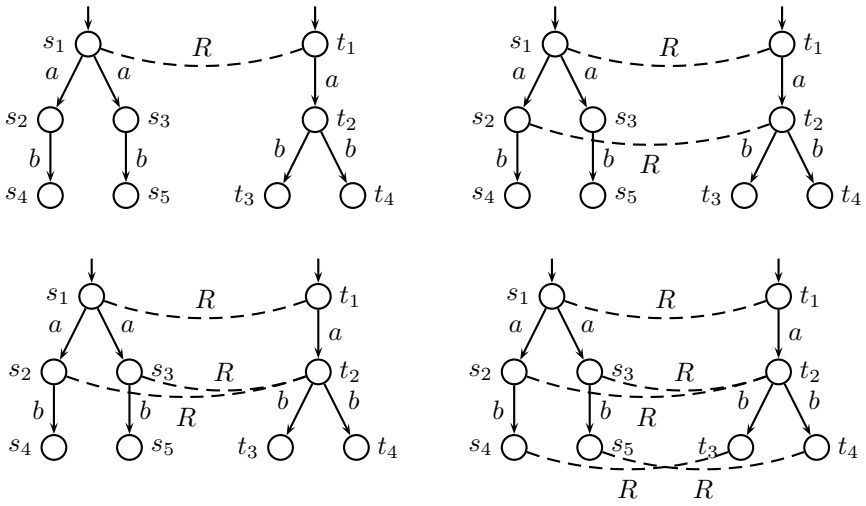


Figure 2.4: Showing two LTSs bisimilar

For small transition systems a more straightforward technique generally is adequate. Consider the transition systems in figure 2.4. In order to show that the initial states s_1 and t_1 are bisimilar, a bisimulation relation R must be constructed to relate these two states. We assume that this can be done. So, we draw an arc between s_1 and t_1 and label it with R. If R is a bisimulation, then every transition from s_1 must be mimicked by a similarly labelled transition from t_1. More concretely, the a-transition from s_1 to s_2 can only be mimicked by an a-transition from t_1 to t_2. So, s_2 and t_2 must be related, too. We also draw an arc to indicate this (see the second picture in figure 2.4). Now we can proceed by showing that the transition from s_1 to s_3 must also be mimicked by the a-transition from t_1 to t_2. Hence, s_3 is related to t_2 (see the third picture). Note that it is wise to choose the transitions to be simulated such that they are simulated by transitions in deterministic nodes. Otherwise, there might be a choice, and more than one possibility needs to be considered. E.g. the a-transition t_1 to t_2 can be simulated by either the transition from s_1 to s_2, or the one from s_1 to s_3.

The relation R needs to be extended to all reachable nodes. Therefore, we consider the relation between s_2 and t_2. We continue the process sketched above, but now let the transitions from the right transition system be simulated by the left one, because the states s_2 and s_3 are deterministic. The relation R is

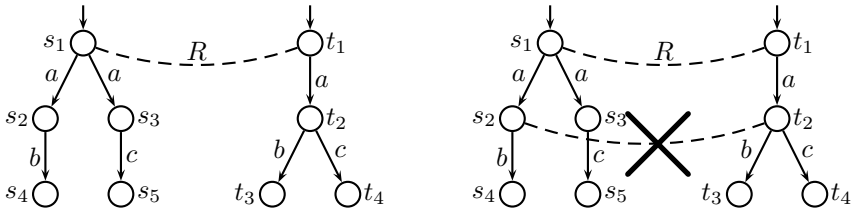


Figure 2.5: Two non bisimilar labelled transition systems

extended as indicated in the fourth picture of figure 2.4. Finally, it needs to be checked that all related states satisfy the requirements in definition 2.3.1.

Now consider the transition systems in figure 2.5. There are three actions a, b and c. These two transition systems are not bisimilar.

Before showing this formally, we first give an intuitive argument why these two processes are different. Let actions a, b and c stand for pressing a button. If a transition is possible, the button can be pressed. If a transition is not possible, the button is blocked.

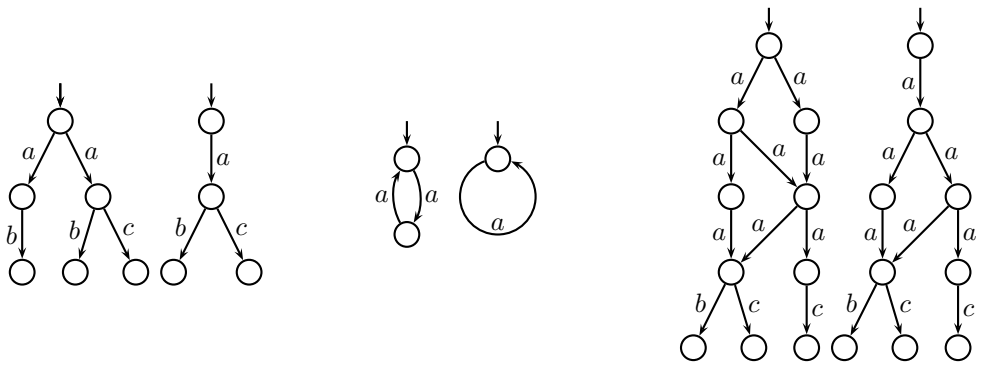
Now suppose a customer ordered the transition system at the right (with initial state t_1) and a 'malicious' supplier delivered a box with the behaviour of the transition system at the left. If the customer cannot experience the difference, the supplier did do an adequate job. However, the customer can first press an a button such that the box ends up being in state s_3. Now the customer, thinking that he is in state t_2 expects that both b and c can be pressed. He, however, finds out that b is blocked, from which he can conclude that he is deceived and he has an argument to sue the supplier.

Now note that in both behaviours in figure 2.5 the same sequence of actions can be performed, namely $a\,b$ and $a\,c$. Yet, the behaviour of both systems can be experienced to be different!

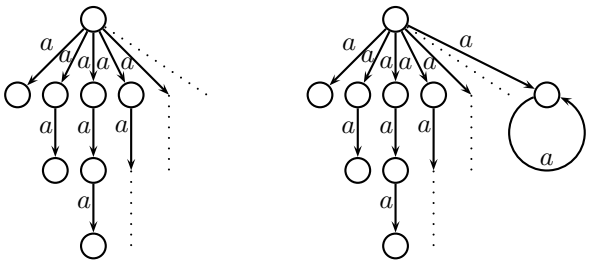
If one tries to show both transition systems bisimilar using the method outlined above, then in the same way as above, state s_2 must be related to state t_2. However, a c transition is possible from state t_2 that cannot be mimicked by state s_2 which has no outgoing c transition. So, s_2 cannot be related to t_2 and consequently, s_1 cannot be bisimilar to t_1.

A pleasant property of bisimulation is that for any labelled transition system, there is a unique minimal transition system which is bisimilar to it. Strictly speaking, it is unique except for the names of the states. But the names of the states are not really relevant for behavioural analysis.

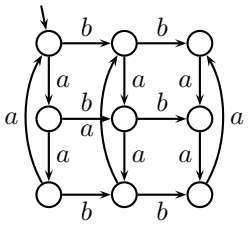
Exercise 2.3.2. Say for each of the following transition systems whether they are pairwise bisimilar:



Exercise 2.3.3. Show that the following transition systems are not bisimilar, where the transition system to the left consists of sequences of a-transitions with length n for each $n \in \mathbb{N}$. The transition system to the right is the same except that it can additionally do an infinite sequence of a-transitions.



Exercise 2.3.4. Give the unique minimal labelled transition system that is bisimilar to the following one:



2.3.2 Trace equivalence

A quite different behavioural equivalence is *trace equivalence*. The essential idea is that two transition systems are equivalent if the same sequences of actions can be performed from the respective initial states. Traces are sequences of actions, typically denoted as $a_1\, a_2\, a_3 \ldots a_n$. We typically use letters σ and ρ to represent traces. The termination symbol $\checkmark$ can also be part of a trace. The symbol ϵ represents the empty trace.

Definition 2.3.5 (Trace equivalence). Let $A = (S, Act, \longrightarrow, s, T)$ be a labelled transition system. The set of *traces* (*runs, sequences*) $Traces(t)$ for a state $t \in S$ is the minimal set satisfying:

1. $\epsilon \in Traces(t)$, i.e. the empty trace is a member of $Traces(t)$.
2. $\checkmark \in Traces(t)$ iff $t \in T$, and
3. if there is a state $t' \in S$ such that $t \xrightarrow{a} t'$ and $\sigma \in Traces(t')$ then $a\sigma \in Traces(t)$.

Two states $t, u \in S$ are called *trace equivalent* iff $Traces(t) = Traces(u)$. Two transition systems are *trace equivalent* if their initial states are trace equivalent.

Note that the two transition systems in figure 2.5 both have sets of traces $\{\epsilon, a, ac, ab\}$ and hence they are trace equivalent. Yet, as argued there, in an ordinary behavioural sense we cannot consider them equal. This is the reason why trace equivalence generally is not used. The two transition systems in figure 2.6

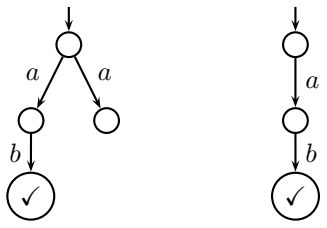


Figure 2.6: Trace equivalence does not preserve deadlocks

are also trace equivalent. Both have trace sets $\{\epsilon, a, ab, ab\checkmark\}$. However, the transition system at the left can deadlock after doing a single a. This is not possible in the transition system at the right. As deadlock freedom, i.e., the absence of a deadlock, is an important notion in processes, behavioural equivalences should preserve deadlocks. This is also an argument against the use of trace equivalence.

However, there are cases where trace equivalence is useful. If the only observations are that one can see what is happening without being able to influence this behaviour, trace equivalence is exactly the right notion. Also, many properties only regard the traces of processes. A property can for instance be that before every b an a action must be done. This property is preserved by trace equivalence. So, in order to determine this for the transition system at the left in figure 2.6, it is perfectly valid to first transform it into the transition system on the right of this figure, and then determine the property for this last transition system.

Exercise 2.3.6. Which of the labelled transition systems of exercise 2.3.2 are trace equivalent.

2.3.3 Language and failures equivalence

In language theory labelled transition systems are commonly used to help in parsing of languages. Generally, the word automaton is used for labelled transition systems in that context. Process theory, as described here, and language theory have a lot in common. For instance, grammars to describe languages are essentially the same as process expressions, described in the chapter 4.

There is however one difference. In the process world there are many different behavioural equivalences, whereas in the language world *language equivalence* is essentially the only one. In the process world one also refers to this equivalence as *completed trace equivalence*.

Every trace that cannot be extended is called a completed trace or a sentence. Two processes are language equivalent if their sets of sentences are the same. More formally:

Definition 2.3.7 (Language equivalence). Let $A = (S, Act, \longrightarrow, s, T)$ be a labelled transition system. We define the language $Lang(t)$ of a state $t \in S$ as the minimal set satisfying:

- $\epsilon \in Lang(t)$ if $t \notin T$ and there are no $t' \in S$ and $a \in Act$ such that $t \xrightarrow{a} t'$;
- $\checkmark \in Lang(t)$ if $t \in T$; and
- if $t \xrightarrow{a} t'$ and $\sigma \in Lang(t')$ then $a\sigma \in Lang(t)$.

Two states $t, u \in S$ are *language equivalent* iff $Lang(t) = Lang(u)$. Two labelled transition systems are *language equivalent* if their initial states are language equivalent.

Note that the transition systems in figure 2.6 are not language equivalent. The language of the one at the left is $\{ab\checkmark, a\}$ whereas the language of the one at the right is $\{ab\checkmark\}$.

The transition systems in figure 2.5 are language equivalent, both having the language $\{ab, ac\}$. However, suppose one would decide to block the action c in both diagrams, which is a normal operation on behaviour. Then the two transition diagrams of figure 2.6 are obtained, which are not language equivalent anymore. This is an annoying property that makes language equivalence quite unusable in the context of interaction.

The equivalence that is closest to language equivalence, but that can 'stand' interaction is *failures equivalence*. It preserves deadlocks, but relates far more processes than bisimulation equivalence. Therefore, some people prefer failures equivalence above bisimulation.

The definition of failures equivalence has two steps. First a refusal set of a state t is defined to contain those actions that cannot be performed in t. Then a failure pair is defined to be a trace ending in some refusal set.

Definition 2.3.8 (Failures equivalence). Let $A = (S, Act, \longrightarrow, s, T)$ be a labelled transition system. A set $F \subseteq Act \cup \{\checkmark\}$ is called a *refusal set* of a state $t \in S$,

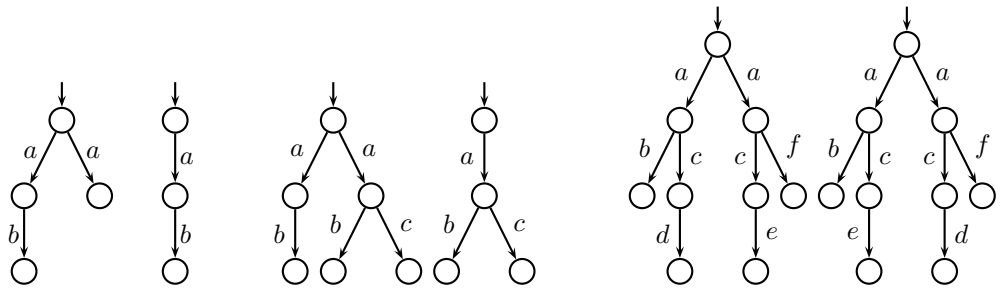
- if for all actions $a \in F$ there is no $t' \in S$ such that $t \xrightarrow{a} t'$, and
- if $\checkmark \in F$, then $t \notin T$, i.e. t cannot terminate.

The *set of failure pairs*, $FailurePairs(t)$ of a state $t \in S$ is inductively defined as follows

- $(\epsilon, F) \in \mathit{FailurePairs}(t)$ if F is a refusal set of t.
- $(\checkmark, F) \in \mathit{FailurePairs}(t)$ if $t \in T$ and F is a refusal set of t, and
- If $t \xrightarrow{a} t'$ and $(\sigma, F) \in \mathit{FailurePairs}(t')$ then $(a\sigma, F) \in \mathit{FailurePairs}(t)$.

Two states $t, u \in S$ are *failures equivalent* if $\mathit{FailurePairs}(t) = \mathit{FailurePairs}(u)$. Two transition systems are *failures equivalent* if their initial states are failures equivalent.

Exercise 2.3.9. State whether the following pairs of transition systems are language and/or failures equivalent.



2.3.4 The Van Glabbeek linear time – branching time spectrum

As stated before, there are very many process equivalences. A nice classification of some of these has been made by Van Glabbeek [18]. He produced the so-called linear time - branching time spectrum which is depicted in figure 2.7. At the top the finest – the less relating – equivalence is depicted and towards the bottom the coarser – the more relating – equivalences are found. The arrows indicate that an equivalence is strictly coarser. So, if processes are bisimulation equivalent, then they are also 2-nested simulation equivalent. Clearly, bisimulation equivalence is the finest equivalence and trace equivalence the coarsest. So, if two processes are bisimilar then they are equivalent in any sense. If processes are not bisimilar, but still appear to be behaviourally equal, then it makes sense to investigate whether they are equal with respect to another equivalence.

Each equivalence has its own properties, and it goes too far to treat them all. Some interesting properties can still be mentioned. Suppose that we can interact with a machine that is equipped with an undo button. So, after doing some actions, we can go back to where we came from. Then one can devise tests to distinguish between processes that are not ready simulation equivalent. So, ready simulation is tightly connected to the capability of undoing actions. In a similar way, possible future equivalence is strongly connected to the capability of predicting which actions are possible in the future and 2-nested simulation equivalence combines them both.

The Van Glabbeek spectrum is strongly related to nondeterminism. If transition systems are deterministic then the whole spectrum collapses. In that case two states are bisimulation equivalent if and only if they are trace equivalent. We state this theorem precisely here and provide the full proof as an example of how properties of bisimulation are proven.

Definition 2.3.10. We call a labelled transition system $A = (S, Act, \longrightarrow, s, T)$ *deterministic* iff for all states $t, t', t'' \in S$ and action $a \in Act$ it holds that if $t \xrightarrow{a} t'$ and $t \xrightarrow{a} t''$ then $t' = t''$.

Theorem 2.3.11. Let $A = (S, Act, \longrightarrow, s, T)$ be a deterministic transition system. For all states $t, t' \in S$ it holds that

$$\mathit{Traces}(t) = \mathit{Traces}(t') \quad \text{iff} \quad t \leftrightarrows t'.$$

Proof. We only prove the case from left to right, leaving the case from right to left as an exercise.

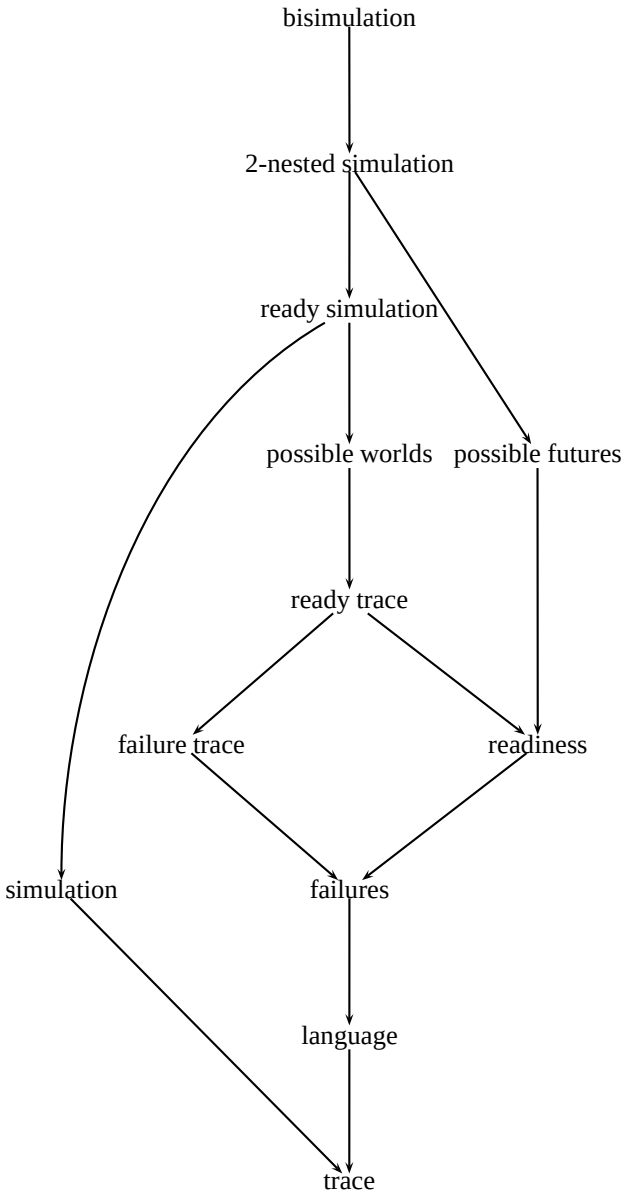


Figure 2.7: The Van Glabbeek linear – branching time spectrum

In order to show that $t \leftrightarrows t'$, we need to show the existence of a bisimulation relation R such that tRt'. We coin the following relation for any states u and u':

$$R(u, u') \quad \text{iff} \quad \mathit{Traces}(u) = \mathit{Traces}(u').$$

Finding the right relation R is generally the crux in such proofs. Note that R is indeed suitable, as R relates t and t'.

So, we are only left with showing that R is indeed a bisimulation relation. This boils down to checking the properties in definition 2.3.1. So, assume that for states u and v we have that uRv. Then

1. Suppose $u \xrightarrow{a} u'$. So, according to definition 2.3.5 $a\sigma \in \mathit{Traces}(u)$ for all traces $\sigma \in \mathit{Traces}(u')$. Furthermore, as $\mathit{Traces}(u) = \mathit{Traces}(v)$, it holds that $a\sigma \in \mathit{Traces}(v)$ or in other words, $v \xrightarrow{a} v'$ for some state $v' \in S$. So, we are left to show that $u'Rv'$, or in other words:

   $$\mathit{Traces}(u') = \mathit{Traces}(v').$$

   We prove this by mutual set inclusion, restricting to only one case, as both are almost identical. So, we prove $\mathit{Traces}(u') \subseteq \mathit{Traces}(v')$. So, assume some trace $\sigma \in \mathit{Traces}(u')$. So, $a\sigma \in \mathit{Traces}(u)$, and consequently $a\sigma \in \mathit{Traces}(v)$. So, there is a v'' such that $v \xrightarrow{a} v''$ and $\sigma \in \mathit{Traces}(v'')$. Now, as the transition system A is deterministic, $v \xrightarrow{a} v'$ and $v \xrightarrow{a} v''$, we can conclude $v' = v''$. Ergo, $\sigma \in \mathit{Traces}(v')$.

2. This second case is symmetric to the first case and is therefore omitted.

3. If $u \in T$, then $\checkmark \in \mathit{Traces}(u)$. As u and v are related, it follows by definition of R that $\checkmark \in \mathit{Traces}(v)$. So, $v \in T$. Similarly, it can be shown that if $v \in T$ then $u \in T$ must hold.

□

The definitions of bisimulation are much more complex than those for language or trace equivalence. This might lead to the wrong assumption that determining whether two transition systems are bisimilar is much harder than determining their trace or language equivalency. The contrary is true. Virtually all forms of bisimulation can be determined in polynomial time on finite state transition systems, whereas trace, failure and language equivalence are in general difficult (P-space hard).

Exercise 2.3.12. Prove that auto-bisimulation is an equivalence relation, i.e., the bisimulation relation on one single transition system is reflexive ($s \leftrightarrows s$ for any $s \in S$), symmetric (if $s \leftrightarrows t$ then $t \leftrightarrows s$ for all states s and t) and transitive (if $s \leftrightarrows t$ and $t \leftrightarrows u$, then $s \leftrightarrows u$ for all states s, t, u).

2.4 Behavioural abstraction

Although the examples given up till now may give a different impression, the behaviour of systems can be utterly complex. The only way to obtain insight in such behaviour is to use abstraction. The most common and extremely powerful abstraction mechanism is to declare an action as non observable or internal. Milner introduced this notion [41] together with an associated process equivalence, called weak bisimulation. We are mainly using branching bisimulation which was defined by Van Glabbeek and Weijland [19]. Branching bisimulation and weak bisimulation serve the same purpose, namely relating processes with internal actions, and are exchangeable for practical purposes.

2.4.1 The internal action τ

An action is internal, if we have no way of observing it directly. We use the special symbol τ to denote any internal action. We generally assume that it is available in a labelled transition system, i.e., $\tau \in Act$. Typical for an internal action is that if it follows another action, it is impossible to say whether it is there. So, the transition systems in figure 2.8 cannot be distinguished, because the τ after the a cannot be observed. Such internal actions are called *inert*.

Figure 2.8: The internal action τ is not visible

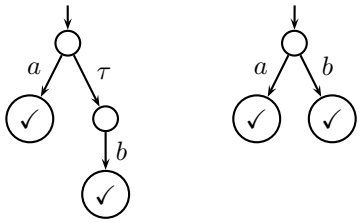


Figure 2.9: The internal action τ is indirectly visible

However, in certain cases the presence of an internal action can be observed, although the action by itself cannot be seen. Suppose one expects the behaviour of the transition system at the right of figure 2.9. It is always possible to do an a-action, as long as neither an a or a b have been done. Now suppose the actual behaviour is that of the transition system at the left. After a while, if the internal action has silently happened, it is impossible to do a anymore. Hence, it is observed that the behaviour cannot be the same as that of the transition system at the right. In this case the τ is not inert, and it cannot be removed without altering the behaviour.

2.5 Behavioural equivalence for the internal action

With the internal action present, equivalences for processes change slightly, to take into account that we cannot observe the internal action directly. Here the most important of such equivalences are given.

2.5.1 Rooted branching bisimulation

In [17] it is shown that the observable equality of the two behaviours in figure 2.8 leads to rooted branching bisimulation as the finest equivalence incorporating internal actions [19]. The argument uses the presence of parallelism.

The idea is very similar to that of strong bisimulation. But now, instead of letting a single action be simulated by a single action, an action can be simulated by a sequence of internal transitions, followed by

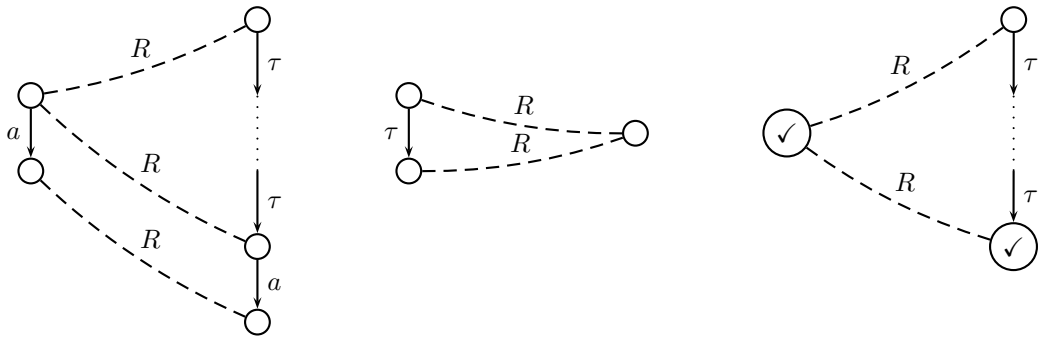


Figure 2.10: Branching bisimulation

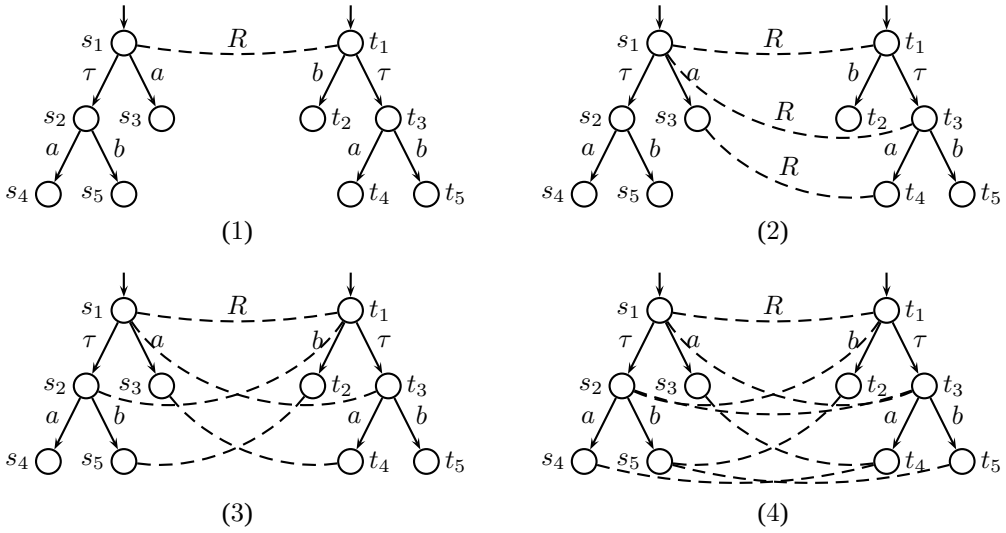


Figure 2.11: Two branching bisimilar transition systems

that single action. See the diagram at the left of figure 2.10. Note that all states that are visited via τ actions are related.

If the action to be simulated is a τ it can be simulated by any number of internal transitions. Even by none, as the diagram in the middle of figure 2.10 shows.

If a state can terminate, it does not need to be related to a terminating state. It suffices if a terminating state can be reached after a number of internal transitions, as shown at the right of figure 2.10.

Definition 2.5.1 (Branching bisimulation). Consider the labelled transition systems $A_1 = (S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow_2, s_2, T_2)$. We call a relation $R \subseteq S_1 \times S_2$ a *branching bisimulation relation* if for all $s \in S_1$ and $t \in S_2$ such that sRt, the following conditions hold for all actions $a \in Act$:

1. If $s \xrightarrow{a}_1 s'$, then
   - either $a = \tau$ and $s'Rt$, or
   - there is a sequence $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ of (zero or more) τ-transitions such that sRt' and $t' \xrightarrow{a}_2 t''$ with $s'Rt''$.
2. Symmetrically, if $t \xrightarrow{a}_2 t'$, then
   - either $a = \tau$ and sRt', or
   - there is a sequence $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ of (zero or more) τ-transitions such that $s'Rt$ and $s' \xrightarrow{a}_1 s''$ with $s''Rt'$.
3. If $s \in T_1$, then there is a sequence of (zero or more) τ-transitions $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ such that sRt' and $t' \in T_2$.
4. Again, symmetrically, if $t \in T_2$, then there is a sequence of (zero or more) τ-transitions $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ such that $s'Rt$ and $s' \in T_1$.

Two states s and t are *branching bisimilar*, denoted by $s \leftrightarrow_b t$, if there is a branching bisimulation relation R such that sRt. Two labelled transition systems are *branching bisimilar* if their initial states are branching bisimilar.

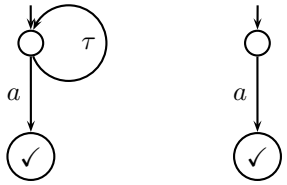


Figure 2.12: Branching bisimulation does not preserve τ-loops

Example 2.5.2. In figure 2.11 two transition systems are depicted. We can determine that they are branching bisimilar in the same way as for strong bisimulation. So, first assume that the initial states must be related, via some relation R. For R to be a branching bisimulation, the transition $s_1 \xrightarrow{a} s_3$ must be mimicked. This can only be done by two transitions $t_1 \xrightarrow{\tau} t_3 \xrightarrow{a} t_4$. So, as depicted in the second diagram, s_1 must be related to the intermediate state t_3 and s_3 must be related to t_4. Now, by letting the transition $t_1 \xrightarrow{b} t_2$ be simulated by $s_1 \xrightarrow{\tau} s_2 \xrightarrow{b} s_5$ the relation is extended as indicated in the third diagram. Ultimately, the relation R must be extended as indicated in the fourth diagram. It requires a careful check that this relation is indeed a branching bisimulation relation.

Branching bisimulation equivalence satisfies a notion of *fairness*. That is, if a τ-loop exists, then no infinite execution sequence will remain in this τ-loop forever if there is a possibility to leave it. The intuition is that there is zero chance that no exit from the τ-loop will ever be chosen. It is straightforward to show that the initial states in the two labelled transition systems in figure 2.12 are branching bisimilar.

Branching bisimulation has an unpleasant property. If an alternative is added to the initial state then processes that were branching bisimilar are not branching bisimilar anymore. In chapter 4, we will see that adding an alternative to the initial state is a common operation. The following figure illustrates the problem:

The two transition systems at the left are branching bisimilar, but the diagrams at the right reflect the transition systems of figure 2.5. They are not branching bisimilar, because doing the τ in the third transition system means that the option to do a b disappears, and this is not possible in the fourth graph.

Milner [43] showed that this problem can be overcome by adding a rootedness condition: initial τ-transitions are never inert. In other words, two processes are considered equivalent if they can simulate each other's initial transitions, such that the resulting processes are branching bisimilar. This leads to the notion of *rooted branching bisimulation*, which is presented below.

Definition 2.5.3 (Rooted branching bisimulation). Let $A_1 = (S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow_2, s_2, T_2)$ be two labelled transition systems. A relation $R \subseteq S_1 \times S_2$ is called a *rooted branching bisimulation relation* iff it is a branching bisimulation relation and it satisfies for all $s \in S_1$ and $t \in S_2$ such that sRt:

1. if $s \xrightarrow{a}_1 s'$, then there is a $t' \in S_2$ such that $t \xrightarrow{a}_2 t'$ and $s' \leftrightarrows_b t'$,
2. symmetrically, if $t \xrightarrow{a}_2 t'$, then there is an $s' \in S_1$ such that $s \xrightarrow{a}_1 s'$ and $s' \leftrightarrows_b t'$.

Two states $s \in S_1$ and $t \in S_2$ are *rooted branching bisimilar*, denoted by $s \leftrightarrows_{rb} t$, if there is a rooted branching bisimulation relation R such that sRt. Two transition systems are *rooted branching bisimilar* iff their initial states are rooted branching bisimilar.

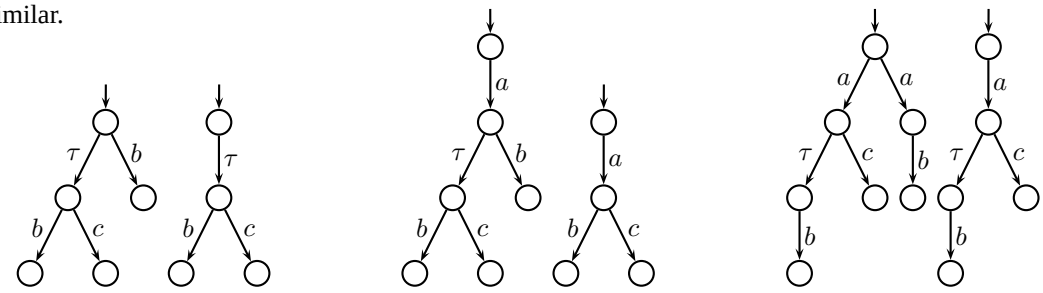
Branching bisimulation equivalence strictly includes rooted branching bisimulation equivalence, which in turn strictly includes bisimulation equivalence:

$$\leftrightarrows \subset \leftrightarrows_{rb} \subset \leftrightarrows_{b} .$$

In the absence of τ, bisimulation and branching bisimulation coincide.

Exercise 2.5.4. Show using the definition of rooted branching bisimulation that the two labelled transition systems in figure 2.8 are rooted branching bisimilar. Show also that the two transition systems in figure 2.9 are neither rooted branching bisimilar nor branching bisimilar.

Exercise 2.5.5. Which of the following pairs of transition systems are branching and/or rooted branching bisimilar.



Exercise 2.5.6. With regard to the examples in the previous exercise 2.5.5 which τ-transitions are inert with respect to branching bisimulation, i.e., for which τ-transitions $s \xrightarrow{\tau} s'$ are the states s and s' branching bisimilar.

2.5.2 Rooted weak bisimulation

A slight variation of branching bisimulation is weak bisimulation. We give its definition here, because weak bisimulation was defined well before branching bisimulation was invented and therefore weak bisimulation is much more commonly used in the literature.

The primary difference between branching and weak bisimulation is that branching bisimulation preserves 'the branching structure' of processes. For instance the last pair of transition systems in exercise 2.5.5 are weakly bisimilar, although the initial a in the transition system at the left can make a choice that cannot be mimicked in the transition system at the right. The branching structure is not respected.

It is useful to know that (rooted) branching bisimilar processes are also (rooted) weakly bisimilar. Furthermore, from a practical perspective, it hardly ever matters whether branching or weak bisimulation is used, except that the algorithms to calculate branching bisimulation on large graphs are more efficient than those for weak bisimulation.

Definition 2.5.7 (Weak bisimulation). Consider the labelled transition systems $A_1 = (S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow_2, s_2, T_2)$. We call a relation $R \subseteq S_1 \times S_2$ a *weak bisimulation relation* if for all $s \in S_1$ and $t \in S_2$ such that sRt, the following conditions hold:

1. If $s \xrightarrow{a}_1 s'$, then
   - either $a = \tau$ and $s'Rt$, or
   - there is a sequence $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 \xrightarrow{a}_2 \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ such that $s'Rt'$.
2. Symmetrically, if $t \xrightarrow{a}_2 t'$, then
   - either $a = \tau$ and sRt', or
   - there is a sequence $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 \xrightarrow{a}_1 \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ such that $s'Rt'$.
3. If $s \in T_1$, then there is a sequence $t \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ such that $t' \in T_2$.
4. Again, symmetrically, if $t \in T_2$, then there is a sequence $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ such that $s' \in T_1$.

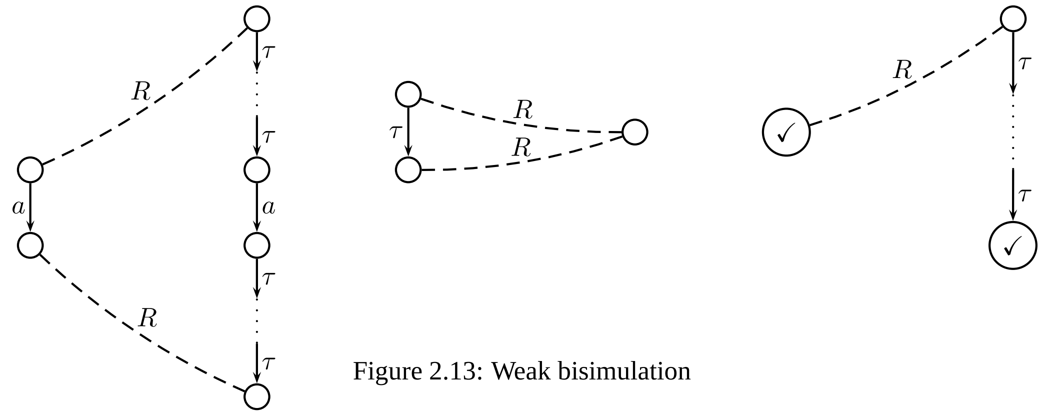


Figure 2.13: Weak bisimulation

Two states s and t are *weakly bisimilar*, denoted by $s \leftrightarrow_w t$, iff there is a weak bisimulation relation R such that sRt. Two labelled transition systems are *weakly bisimilar* iff their initial states are weakly bisimilar.

In figure 2.13 weak bisimulation has been depicted. Compare this figure with figure 2.10 for branching bisimulation. Note that weak bisimulation is more relaxed in the sense that R does not have to relate that many states.

The notion of rooted weak bisimulation is defined along exactly the same lines as rooted branching bisimulation. The underlying motivation is exactly the same.

Definition 2.5.8 (Rooted weak bisimulation). Let $A_1 = (S_1, Act, \longrightarrow_1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow_2, s_2, T_2)$ be two labelled transition systems. A relation $R \subseteq S_1 \times S_2$ is called a *rooted weak bisimulation relation* if R is a weak bisimulation relation and it satisfies for all $s \in S_1$ and $t \in S_2$ such that sRt:

1. if $s \xrightarrow{\tau}_1 s'$, then there is a sequence $t \xrightarrow{\tau}_2 \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t'$ of at least length 1 and $s' \leftrightarrow_w t'$, and

2. symmetrically, if $t \xrightarrow{\tau}_2 t'$, then there is sequence of at least length 1 of τ-steps $s \xrightarrow{\tau}_1 \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ and $s' \leftrightarrow_w t'$.

Two states $s \in S_1$ and $t \in S_2$ are *rooted weak bisimilar*, denoted by $p \leftrightarrow_{rw} q$, if there is a rooted weak bisimulation relation R such that sRt. Two transition systems are *rooted weak bisimilar* iff their initial states are rooted weak bisimilar.

We finish this section by showing the relationships between the various bisimulation relations defined hitherto.

$$\leftrightarrow \subset \leftrightarrow_{rb} \subset \leftrightarrow_b \subset \leftrightarrow_w, \qquad \leftrightarrow \subset \leftrightarrow_{rb} \subset \leftrightarrow_{rw} \subset \leftrightarrow_w .$$

Note that rooted weak bisimulation and branching bisimulation are incomparable.

Exercise 2.5.9. Which of the pairs of transition systems of exercise 2.5.5 are (rooted) weakly bisimilar. Which τ-transitions are inert with respect to weak bisimulation.

Exercise 2.5.10. Prove that any branching bisimulation is a weak bisimulation relation.

2.5.3 Weak trace equivalence

Two processes are weakly trace equivalent, if their sets of traces in which τ-transitions are made invisible are the same.

Definition 2.5.11 (Weak trace equivalence). Let $A = (S, Act, \longrightarrow, s, T)$ be a labelled transition system. The set of *weak traces* $WTraces(t)$ for a state $t \in S$ is the minimal set satisfying:

1. $\epsilon \in WTraces(t)$.

2. $\checkmark \in WTraces(t)$ iff $s \in T$, and

3. if there is a state $t' \in S$ such that $t \xrightarrow{a} t'$ ($a \neq \tau$) and $\sigma \in WTraces(t')$ then $a\sigma \in WTraces(t)$.

4. if there is a state $t' \in S$ such that $t \xrightarrow{\tau} t'$ and $\sigma \in WTraces(t')$ then $\sigma \in WTraces(t)$.

Two states $t, u \in S$ are called *weak trace equivalent* if $WTraces(t) = WTraces(u)$. Two transition systems are *weak trace equivalent* if their initial states are trace equivalent.

Weak trace equivalence does not preserve deadlocks, or any other branching behaviour and it is the weakest of all behavioural equivalences that are generally considered. Weak trace equivalence is hard to calculate for a given transition system, and the smallest transition system obtained by applying weak trace equivalence is not unique. But as calculating the minimal transition system modulo weak trace equivalence is very hard, generally a deterministic transition system that is not minimal is calculated.

But in those cases where one is only interested whether certain sequential orderings of actions is preserved, and the investigated transition system modulo say branching bisimulation is not too large, applying a weak trace reduction can be of great value.

Chapter 3

Timed process behaviour

The labelled transition systems as presented in the previous chapter can easily be transformed into timed transition systems by making it explicit at which time an action can take place. This chapter introduces timed transition systems and extends the equivalences of the previous chapter to a setting with time.

3.1 Timed actions and time deadlocks

Timed actions are actions with a time tag. Time tags are positive real numbers. We say that process behaviour starts at time 0, and this means that the first action must take place after time 0. We typically write

$a \leftarrow t$ or $\mathtt{a@t}$

for action a that takes place at time t. Graphically, this is denoted by process (a) in figure 3.1. Just as in the transition systems in the previous chapter, the initial state is marked with a small incoming arrow, and the terminating state is marked with a small ✓ symbol. Process (b) in the same figure, we see a process that performs an action a at time 2 followed by a b at time 4. In a timed transition system all outgoing transitions have higher time tags than all incoming transitions to indicate that time is increasing. So, a transition system as in figure 3.1 (c) is not allowed. An action a is just an action that can happen at any time. It is depicted in diagram (d). As we can only draw a finite number of transitions, the diagram can only be suggestive, using grey-shades to suggest an infinite number of transitions. Alternatives, are the use of timeless actions as in (e) to indicate that the action take place at any time, or the use of intervals as done in (f).

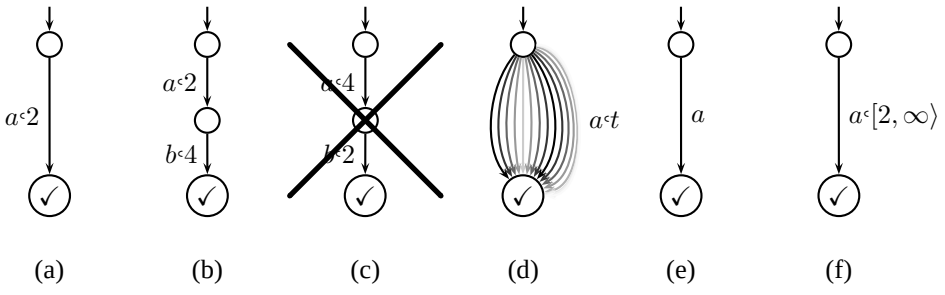


Figure 3.1: Timed transition systems

An important aspect of time is idling. This means that no action is done while time is proceeding. If an action a can be done at time t, it is self evident that idling is possible until time t. But if an action a can be done at time t, it is not necessarily the case that it must be done. In figure 3.2 (a) and (b) the situations are depicted where an action a must happen at time 3, and where an action a must happen at time 3, but

alternatively idling is possible until time 4. This is indicated by the curly arrow with a time label. Note that the idle arrow does not have an endstate.

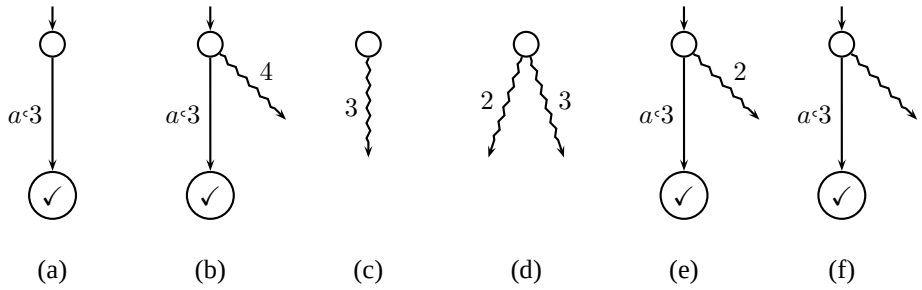


Figure 3.2: Timed transition systems with idling arrows

What if the action a in figure 3.2 (a) cannot be performed for whatever reason. Then at time 3 there is no alternative. We resolve this situation by introducing the notion of a *time deadlock* at time 3. Time cannot proceed. Clearly, a process that has a time deadlock does not represent a real life process, as time cannot stop. So, a good check to find out whether a timed process can be built, is to prove that it is free of time deadlocks.

The process with a time deadlock is represented in figure 3.2 (c). If it is possible to idle in a state until time t, it is also possible to idle to a shorter moment in time. So, process (d) is equivalent to process (c), as the idle transition with time 2 is redundant. Redundant idle transitions need not be drawn. Similarly, the process in (e) is equivalent to to the process in (a). In (f) it is indicated using a curly arrow without time label that waiting can be done indefinitely.

A somewhat larger example is depicted in figure 3.3. It describes a process that can send at times 1,2 and 3 using an s action, but if nothing is sent at time 4 a *timeout* at time 4 takes place. After sending, it can also receive a message using action r at any time, but it can also idle till infinity.

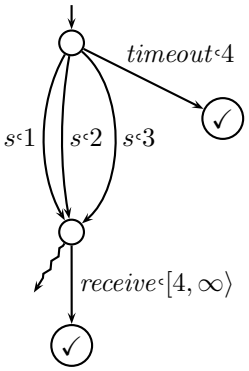
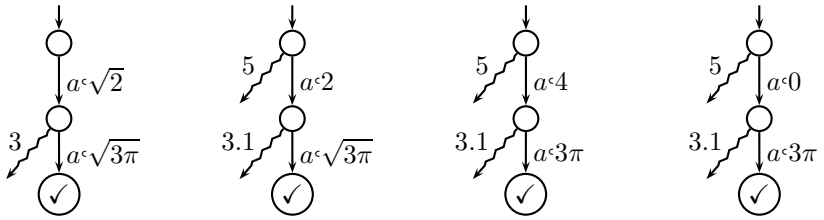


Figure 3.3: A simple timeout process

In general timed transition systems have an infinite number of states and transitions. Therefore, they are not really useful to describe real processes. This can much more conveniently be done using the algebraic notation of the next chapter. The main goal of timed transition systems is to provide a very straightforward basic model to study the elementary properties of timed systems. Using the semantic mapping of chapter 7 the algebraic process descriptions are mapped to timed labelled transition systems.

Exercise 3.1.1. Which of the following are proper timed labelled transition systems and which have time deadlocks?



(a) (b) (c) (d)

Exercise 3.1.2. Draw a timed labelled transition system that represents a process that ticks every second using a timed *tick* action.

Exercise 3.1.3. Draw a timed transition system where if an alarm is sounded between time 0 and 10, a response will come at time 11.

3.2 Timed transition systems

We now give the definition of a general timed transition system. It differs from a normal transition system in that there is an extra idle relation, and transitions additionally carry a positive real number indicating when the transition occurs. Moreover, there are some restrictions on which transitions and idle relations are allowed. We use $\mathbb{R}^{>0}$ for the set of the real numbers larger or equal than 0.

Definition 3.2.1 (Timed transition system). A timed transition system is a five tuple $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ where

- S is a set of states.
- Act is a set of actions.
- $\longrightarrow \subseteq S \times Act \times \mathbb{R}^{>0} \times S$ is a *transition relation*. The expression $s \xrightarrow{a}_t s'$ says that a traversal is made from state s to state s' by executing action a at time t.
- $\rightsquigarrow \subseteq S \times \mathbb{R}^{>0}$ is the *idle relation*. The predicate $s \rightsquigarrow_t$ expresses that it is possible to idle until and including time t in state s.
- s_0 is the *initial state*.
- $T \subseteq S$ is the set of *terminating states*.

Every timed transition system must satisfy the *progress* and *density* requirements. The progress requirement says that if

$$s' \xrightarrow{a}_t s \xrightarrow{a'}_{t'} s'' \text{ or } s' \xrightarrow{a}_t s \rightsquigarrow_{t'}$$

for states s, s', s'', actions a and a' and times t and t', then $t' > t$. The density requirement expresses that for any action a, states s, s' and time t

$$\text{if } s \xrightarrow{a}_t s' \text{ or } s \rightsquigarrow_t, \text{ then } s \rightsquigarrow_{t'}$$

for any $0 < t' \leq t$.

Every labelled transition system can easily be transformed into a timed transition system. Let $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ be a labelled transition system. The timed transition system that describes the same process is $(S \times \mathbb{R}^{>0}, Act, \longrightarrow^T, \rightsquigarrow^T, \langle s_0, 0\rangle, T \times \mathbb{R}^{>0})$. As states we take all states of S paired with a real number indicating when this state was entered. The initial state gets 0 as starting time, and all termination states are paired with an arbitrary positive real number. The new timed transition relation is defined as follows:

$$\langle s, t\rangle \xrightarrow{a}{}^{T}_{u} \langle s', t'\rangle \text{ iff } s \xrightarrow{a} s', u{>}t \text{ and } t'{=}u.$$

The idle relation expresses that in any state it is possible to idle indefinitely:

$\langle s, t\rangle \rightsquigarrow^{T}_{t'}$ iff $t' > t$.

In order to see that this translation is well defined, it is necessary to check that the progress and density requirements are satisfied.

Exercise 3.2.2. Give explicit definitions of two timed transition systems that have only the initial state. In the first transition system it is is possible to wait to time 2, whereas in the second transition system it is possible to wait up till and including time 2. Which of the two transition systems is hard to depict using idle transitions?

3.3 Timed process equivalences

The equivalences defined in section 2.3 can straightforwardly been lifted to the setting with time. We treat strong, branching and weak bisimulation, trace and weak trace equivalence.

3.3.1 Timed (strong) bisimulation

We start out with the definition of timed strong bisimulation, which is the straightforward extension of strong bisimulation in the untimed setting.

Definition 3.3.1 (Timed strong bisimulation). Let $A_1 = (S_1, Act, \longrightarrow^1, \rightsquigarrow^1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow^2, \rightsquigarrow^2, s_2, T_2)$ be timed transition systems. A binary relation $R \subseteq S_1 \times S_2$ is called a timed (strong) *bisimulation relation* iff for all $s \in S_1$ and $u \in S_2$ such that sRu holds, it also holds for all actions $a \in Act$ and $t \in \mathbb{R}^{>0}$ that:

1. if $s \xrightarrow{a}{}^{1}_{t} s'$, then there is a $u' \in S_2$ such that $u \xrightarrow{a}{}^{2}_{t} u'$ with $s'Ru'$,
2. if $u \xrightarrow{a}{}^{2}_{t} u'$, then there is a $s' \in S_1$ such that $s \xrightarrow{a}{}^{1}_{t} s'$ with $s'Ru'$,
3. $s \rightsquigarrow^1_t$ iff $u \rightsquigarrow^2_t$,
4. $s \in T_1$ if and only if $u \in T_2$.

Two states s and u are *timed* (*strongly*) *bisimilar*, denoted by $s \leftrightarrows u$, if there is a timed strong bisimulation relation R such that sRu. Timed transition systems A_1 and A_2 are *timed* (*strongly*) *bisimilar* iff their initial states are timed bisimilar, i.e. $s_1 R s_2$.

Note that the symbol $\leftrightarrows$ for timed strong bisimulation is the same as the symbol for ordinary bisimulation. As one works on timed transition systems and the other on ordinary transition systems, this overloading does not cause confusion.

3.3.2 Timed branching bisimulation

Definition 3.3.2 (Timed branching bisimulation). Consider two timed transition systems $A_1 = (S_1, Act, \longrightarrow^1, \rightsquigarrow^1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow^2, \rightsquigarrow^2, s_2, T_2)$. We call a relation $R \subseteq S_1 \times S_2$ a *timed branching bisimulation relation* iff for all states $s \in S_1$ and $u \in S_2$ such that sRu, the following conditions hold for all actions $a \in Act$ and $t \in \mathbb{R}^{>0}$:

1. If $s \xrightarrow{a}{}^{1}_{t} s'$, then
   - either $a = \tau$ and $s'Ru$, or
   - there is a sequence $u \xrightarrow{\tau}{}^{2}_{t_1} \cdots \xrightarrow{\tau}{}^{2}_{t_n} u'$ of (zero or more) τ-transitions such that sRu' and $u' \xrightarrow{a}{}^{2}_{t} u''$ with $s'Ru''$.

2. Symmetrically, if $u \xrightarrow{a}{}^{2}_{t} u'$, then
   - either $a = \tau$ and sRu', or
   - there is a sequence $s \xrightarrow{\tau}{}^{1}_{t_1} \cdots \xrightarrow{\tau}{}^{1}_{t_n} s'$ of (zero or more) τ-transitions such that $s'Ru$ and $s' \xrightarrow{a}{}^{1}_{t} s''$ with $s''Ru'$.
3. If $s \rightsquigarrow^{1}_{t}$, then there is a sequence of (zero or more) τ-transitions $u \xrightarrow{\tau}{}^{2}_{t_1} \cdots \xrightarrow{\tau}{}^{2}_{t_n} u'$ such that sRu' and $u' \rightsquigarrow^{2}_{t}$.
4. Again, symmetrically, if $u \rightsquigarrow^{2}_{t}$, then there is a sequence of (zero or more) τ-transitions $s \xrightarrow{\tau}{}^{1}_{t_1} \cdots \xrightarrow{\tau}{}^{1}_{t_n} s'$ such that $s'Ru$ and $s' \rightsquigarrow^{1}_{t}$.
5. If $s \in T$, then there is a sequence of (zero or more) τ-transitions $u \xrightarrow{\tau}{}^{2}_{t_1} \cdots \xrightarrow{\tau}{}^{2}_{t_n} u'$ such that sRu' and $u' \in T$.
6. Again, symmetrically, if $u \in T$, then there is a sequence of (zero or more) τ-transitions $s \xrightarrow{\tau}{}^{1}_{t_1} \cdots \xrightarrow{\tau}{}^{1}_{t_n} s'$ such that $s'Ru$ and $s' \in T$.

Two states s and u are *timed branching bisimilar*, denoted by $s \leftrightsquigarrow_b u$, if there is a timed branching bisimulation relation R such that sRu. Two timed transition systems are *timed branching bisimilar* if their initial states are timed branching bisimilar.

Definition 3.3.3 (Timed rooted branching bisimulation). Let $A_1 = (S_1, Act, \longrightarrow^1, \rightsquigarrow^1, s_1, T_1)$ and $A_2 = (S_2, Act, \longrightarrow^2, \rightsquigarrow^2, s_2, T_2)$ be two timed transition systems. A relation $R \subseteq S_1 \times S_2$ is called a *timed rooted branching bisimulation relation* iff for all states $s \in S_1$ and $u \in S_2$ such that sRu, it satisfies the conditions below for all actions $a \in Act$ and $t \in \mathbb{R}^{>0}$:

1. if $s \xrightarrow{a}{}^{1}_{t} s'$, then there is a $u' \in S_2$ such that $u \xrightarrow{a}{}^{2}_{t} u'$ and $s' \leftrightsquigarrow_b u'$,
2. symmetrically, if $u \xrightarrow{a}{}^{2}_{t} u'$, then there is an $s' \in S_1$ such that $s \xrightarrow{a}{}^{1}_{t} s'$ and $s' \leftrightsquigarrow_b u'$,
3. $s \rightsquigarrow^{1}_{t}$ if and only if $u \rightsquigarrow^{2}_{t}$, and
4. $s \in T_1$ if and only if $s \in T_2$.

Two states $s \in S_1$ and $u \in S_2$ are *timed rooted branching bisimilar*, denoted by $s \leftrightsquigarrow_{rb} u$, if there is a rooted branching bisimulation relation R such that sRu. Two transition systems are *timed rooted branching bisimilar* iff their initial states are timed rooted branching bisimilar.

3.3.3 Timed weak bisimulation

3.3.4 Timed trace equivalence

3.3.5 Timed weak trace equivalence

Chapter 4

Algebraic process descriptions

In the previous chapter we described behaviour by labelled transition systems. If behaviour becomes more complex this technique falls short. In this chapter we describe processes using an extended process algebra. This language is much richer than that of the previous chapter. We use multi-actions with data and time. There are operators to combine behaviour sequentially and in parallel. The language in this chapter provides us with all the means to model reactive systems.

4.1 Basic processes

In this section basic processes are defined, without parallelism and time. Only very basic data types are used. In section 4.2 a full account is given to define and use data types. Axioms are used to characterise the meaning of the various constructs.

4.1.1 Actions

As in the previous chapter, actions are the basic ingredients of processes. More precisely, every action is a (elementary) process. Differently from the previous chapter, actions can carry data. E.g. a $receive$ action can carry a message, and an $error$ action can carry a natural number, for instance indicating its severity. Actions can have any number of parameters. They are declared as follows:

act $timeout$;
$error : \mathbb{N}$;
$receive : \mathbb{B} \times \mathbb{N}^+$;

This declares parameterless action name *timeout*, action name *error* with a data parameter of sort $\mathbb{N}$ (natural numbers), and action name *receive* with two parameters of sort $\mathbb{B}$ (booleans) and $\mathbb{N}^+$ (positive numbers) respectively. For the above action name declaration, *timeout*, *error*(0) and *receive*$(false, 6)$ are valid actions. The data parameters of actions cannot be a process.

Actions are events that happen atomically in time. They have no duration. In case duration of activity is important, it is most convenient to think of an action as the beginning of the activity. If that does not suffice, activity can be modelled by two actions that mark its beginning and end. A declaration of actions describing the beginning and end of an activity a could look like:

act a_{begin}, a_{end};

From now on, we write $a, b, \ldots$ to denote action names. In concrete models we attach the required number of data arguments in accordance with the declaration. In more abstract treatments we assume that actions have only a single parameter generally denoted using letters d and e. If the number of parameters is of explicit concern, we use $\vec{d}, \vec{e}, \ldots$ to denote vectors of data arguments.

MA1	$\alpha\|\beta = \beta\|\alpha$
MA2	$(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$
MA3	$\alpha\|\tau = \alpha$
MD1	$\tau \setminus \alpha = \tau$
MD2	$\alpha \setminus \tau = \alpha$
MD3	$\alpha \setminus (\beta\|\gamma) = (\alpha \setminus \beta) \setminus \gamma$
MD4	$(a(d)\|\alpha) \setminus a(d) = \alpha$
MD5	$(a(d)\|\alpha) \setminus b(e) = a(d)\|(\alpha \setminus b(e))$ if $a \not\equiv b$ or $d \not\approx e$
MS1	$\tau \sqsubseteq \alpha = \mathit{true}$
MS2	$a \sqsubseteq \tau = \mathit{false}$
MS3	$a(d)\|\alpha \sqsubseteq a(d)\|\beta = \alpha \sqsubseteq \beta$
MS4	$a(d)\|\alpha \sqsubseteq b(e)\|\beta = a(d)\|(\alpha \setminus b(e)) \sqsubseteq \beta$ if $a \not\equiv b$ or $d \not\approx e$
MAN1	$\underline{\tau} = \tau$
MAN2	$\underline{a(d)} = a$
MAN3	$\underline{\alpha\|\beta} = \underline{\alpha}\|\underline{\beta}$

Table 4.1: Axioms for multi-actions

4.1.2 Multi-actions

Multi-actions represent a collection of actions that occur at the same time instant. Multi-actions are constructed according to the following BNF[1] grammar:

$$\alpha ::= \tau \mid a \mid a(\vec{d}) \mid \alpha|\beta,$$

where a denotes an action name and $\vec{d}$ a vector of data parameters. The term τ represents the empty multi-action, which contains no actions and as such cannot be observed. It is exactly the internal action introduced in the previous chapter. The multi-actions a and $a(\vec{d})$ consist of a single action, respectively without and with data arguments. The multi-action $\alpha|\beta$ consists of the actions from both the multi-actions α and β, which all must happen simultaneously.

Typical examples of multi-actions are τ, $\mathit{error}|\mathit{error}|\mathit{send}(\mathit{true})$, $\mathit{send}(\mathit{true})|\mathit{receive}(\mathit{false}, 6)$ and $\tau|\mathit{error}$. We generally write $\alpha, \beta, \ldots$ as variables for multi-actions. Multi-actions are particularly interesting for parallel behaviour. If sequential behaviour is described, multi-actions generally do not occur.

In table 4.1 the basic properties about multi-actions are listed by defining which multi-actions are equal to each other using the equality symbol ($=$). In particular the first three are interesting, as they express that multi-actions are associative, commutative and have τ as its unit element. This structure is called a monoid. Note that we use the convention of writing exactly one data argument for each action. There are a few operators on multi-actions that turn out to be useful. There is an operator $\underline{\alpha}$ that associates with a multi-action α the multiset of action names that is obtained by omitting all data parameters that occur in α. We also define operators $\setminus$ and $\sqsubseteq$ on multi-actions that represents removal and inclusion of multi-actions. Here $\equiv$ denotes syntactic equality on action names and $\approx$ denotes equality on data.

Exercise 4.1.1. Simplify the following expressions using the equations in table 4.1.

1. $(a(1)|b(2)) \setminus (b(2)|a(1))$.
2. $(a(1)|b(2)) \setminus (b(3)|c(2))$.

[1] BNF stands for Backus-Naur Form which is a popular notation to denote context free grammars.

A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y)\cdot z = x\cdot z + y\cdot z$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$
A6	$\alpha + \delta = \alpha$
A7	$\delta\cdot x = \delta$
Cond1	$true \rightarrow x \diamond y = x$
Cond2	$false \rightarrow x \diamond y = y$
SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = X(e) + \sum_{d:D} X(d)$
SUM4	$\sum_{d:D}(X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
SUM5	$(\sum_{d:D} X(d))\cdot y = \sum_{d:D} X(d)\cdot y$

Table 4.2: Axioms for the basic operators

3. $a(1)|b(2) \sqsubseteq b(2)$.

Exercise 4.1.2. Prove using induction on the structure of multi-actions that the following three properties hold for all multi-actions α and β:

1. $\alpha\backslash\alpha = \tau$.
2. $\alpha \sqsubseteq \alpha = true$.
3. $(\alpha|\beta|\gamma)\backslash\beta = \alpha|\gamma$.

4.1.3 Alternative and sequential composition

There are two main operators to combine multi-actions into behaviour. These are the alternative and sequential composition operators. For processes p and q we write $p\cdot q$ to indicate the process that first performs the behaviour of p and after p terminates, continues to behave as q. Note that the dot is often omitted when denoting concrete processes.

If a, b and c are actions, the action a is the process that can do an a-action and then terminate. The process $a\cdot b$ can do an a followed by a b and then terminate. The process $a\cdot b\cdot c$ can do three actions in a row before terminating. The three processes are depicted in figure 4.1

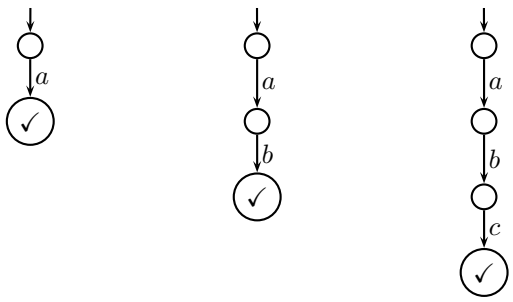


Figure 4.1: Three sequential processes

The process $p + q$ is the alternative composition of processes p and q. This expresses that either the behaviour of p or that of q can be chosen. The actual choice is made by the first action in either p or q. So, the process $a + b$ is the process that can either do an a or a b and the process $a{\cdot}b + c{\cdot}d$ can either do a followed by b, or c followed by d as shown in figure 4.2. The alternative composition operator $+$ is also called the choice operator. In table 4.2 some axioms are given, that indicate which processes are equal to

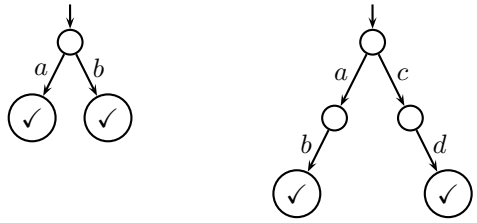


Figure 4.2: Two processes with a choice

other processes. In the axioms symbols x and y are variables that can be substituted by processes. For the alternative and sequential composition, the axioms A1 to A5 are particularly important. A1 and A2 say that the alternative composition is commutative and associative. Practically, this means that is does not matter whether behaviour stands at the left or right of the choice, and that brackets to group more than one choice operator can be omitted. An interesting axiom is A3, which says that the choice is idempotent. If a choice can be made between two identical processes, there is no choice to make at all.

The axiom A4 says that sequential composition right distributes over the choice. The left distribution, namely $x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$ is not valid, as it would imply that $a{\cdot}(b + c)$ would be equal to $a{\cdot}b + a{\cdot}c$ which are in general not behaviourally equivalent, as argued in the previous chapter (see figure 2.5). The axiom A5 says that sequential composition is associative. So, we can as well write $a{\cdot}b{\cdot}c$ instead of $(a{\cdot}b){\cdot}c$, as the position of the brackets is immaterial.

The axioms listed in table 4.2 and elsewhere are valid for strong bisimulation. If we provide axioms that hold for other process equivalences, this will be explicitly stated. Using the axioms we can show the processes $a{\cdot}b + a{\cdot}b$ equal to $a{\cdot}(b + b)$ (cf. 2.4). This goes as follows:

$$a{\cdot}(b + b) \stackrel{\text{A3}}{=} a{\cdot}b \stackrel{\text{A3}}{=} a{\cdot}b + a{\cdot}b.$$

In the first step we take the subexpression $b + b$. It is reduced to b using axiom A3 by substituting b for x. Henceforth b is used to replace $b + b$. In the second step, axiom A3 is used again, but now by taking $a{\cdot}b$ for x.

As a more elaborate example, we show that $((a + b){\cdot}c + a{\cdot}c){\cdot}d$ and $(b + a){\cdot}(c{\cdot}d)$ are equal.

$$((a + b){\cdot}c + a{\cdot}c){\cdot}d \stackrel{\text{A4}}{=} (a{\cdot}c + b{\cdot}c + a{\cdot}c){\cdot}d \stackrel{\text{A1,A3}}{=} (a{\cdot}c + b{\cdot}c){\cdot}d \stackrel{\text{A4}}{=} ((a + b){\cdot}c){\cdot}d \stackrel{\text{A5}}{=} (b + a){\cdot}(c{\cdot}d).$$

Exercise 4.1.3. Derive the following equations from the axioms A1-A5:

1. $((a + a){\cdot}(b + b)){\cdot}(c + c) = a{\cdot}(b{\cdot}c)$;
2. $(a + a){\cdot}(b{\cdot}c) + (a{\cdot}b){\cdot}(c + c) = (a{\cdot}(b + b)){\cdot}(c + c)$.

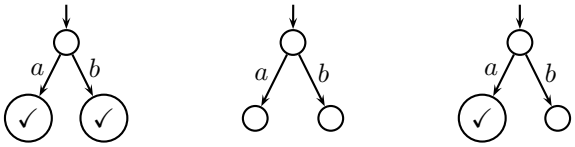
We use the shorthand $x \subseteq y$ for $x + y = y$, and write $x \supseteq y$ for $y \subseteq x$. This notation is called *summand inclusion*. It is possible to divide the proof of an equation into proving two inclusions, as the following exercise shows.

Exercise 4.1.4. Prove that if $x \subseteq y$ and $y \subseteq x$, then $x = y$.

4.1.4 Deadlock

A remarkable but very essential process is *deadlock*, also called *inaction*. This is denoted as δ and cannot do any action. In particular, it cannot terminate. The properties of deadlock are best illustrated by the axioms A6 and A7 in table 4.2. Axiom A7 says that it is impossible to go beyond a deadlock. So, the x in $\delta{\cdot}x$ is irrelevant because it cannot be reached. The axiom A6 says that if we can choose between a multi-action α and δ, we must choose α because δ has no first action that would cause the δ to be chosen.

The deadlock can be used to prevent processes from terminating. So, the process $a + b$ can terminate, whereas the process $a{\cdot}\delta + b{\cdot}\delta$ cannot, and $a + b{\cdot}\delta$ has both a terminating branch and one that cannot terminate. The tree graphs belonging to these processes are depicted below.



Deadlock is not very often used as a specification primitive, as specifying that a system has a deadlock is strange because this is undesired behaviour. The deadlock is generally the result of some communicating parallel processes. It shows that there is some incompatibility in these processes such that at a certain moment no actions can be done anymore. Sometimes, deadlock is used to indicate that a problematic situation is reached. For instance, when an error occurs, an erroneous situation is reached. This can be modelled by $error{\cdot}\delta$. Sometimes, after performing a task, a deadlock is put at the end of a process to prevent explicit termination.

Later when we introduce time, it will turn out that a stronger deadlock than δ exists. One feature of δ is that it lets time pass. The stronger variant can even let time come to a halt. This means that there are incompatible time constraints in the processes. In case there would not be any time constraint on x, the axiom A6 can be formulated more generally as $x + \delta = x$.

4.1.5 The conditional and sum operator

Data influences the run of processes using the conditional operator. For a condition c of sort $\mathbb{B}$ (boolean) and processes p and q we write $c \to p \diamond q$ to express *if c then p else q*. The condition c must consist of data, and it is not allowed to use processes in c. The axioms Cond1 and Cond2 in table 4.2 are obvious. If c is $true$, then the behaviour is p, and otherwise it is q.

The else part of the condition can be omitted, meaning that nothing can be done in the else part. We come back to provide the exact definition later.

So, to say that if the water level is too high, an alarm is sounded, and otherwise an ok message is sent, is described as follows:

$$(waterLevel{>}limit) \to soundAlarm \diamond sendOK\,.$$

Case distinction can also neatly be described. Suppose there is a data variable *desiredColor* which indicates which colour a signal should get. Using actions such as *setSignalToRed* the desire can be transformed in messages to set the signal to the desired colour:

$$\begin{aligned}&(desiredColor{\approx}Red) \to setSignalToRed\\ +\ &(desiredColor{\approx}Yellow) \to setSignalToYellow\\ +\ &(desiredColor{\approx}Green) \to setSignalToGreen\end{aligned}$$

The axioms Cond1 and Cond2 are very handy, when used in combination with *case distinction* or *induction* on booleans. There are exactly two booleans, $true$ and $false$, which means that to prove a property for all booleans, it suffices to prove it only for $true$ and $false$. More concretely, in order to show $c \to x \diamond x = x$ we must show $true \to x \diamond x = x$, which follows directly from Cond1, and $false \to x \diamond x = x$ which follows directly from Cond2.

Exercise 4.1.5. Derive the following equations:

(1) $c \to x \diamond y = \neg c \to y \diamond x$;

(2) $c \vee c' \to x \diamond y = c \to x \diamond (c' \to x \diamond y)$;

(3) $x+y \trianglerighteq c \to x \diamond y$;

(4) if assuming that c holds, we can prove that $x = y$ then $c \to x \diamond z = c \to y \diamond z$.

The sum operator $\sum_{d:D} p(d)$ is a generalisation of the choice operator. The notation $p(d)$ is used to stress that d can occur in the process p. Where $p + q$ allows a choice among processes p and q, $\sum_{d:D} p(d)$ allows to choose any $p(d)$ for some value d from D. If D is finite, e.g. equal to $\mathbb{B}$, then the sum operator can be expressed using the choice. The following is valid:

$$\sum_{c:\mathbb{B}} p(c) = p(true) + p(false).$$

For sums over infinite domains, e.g. $\sum_{n:\mathbb{N}} p(n)$, it is not possible anymore to expand the sum operator using the choice operator.

The sum operator can be used for many purposes, but the most important one is to model the reading of data values. So, modelling a (one time usable) buffer that can read a message to be forwarded at a later moment, can be done as follows:

$$\textstyle\sum_{m:Message} read(m) \cdot forward(m).$$

A commonly made mistake is to not place the sum operator directly around the action in which the reading takes place. Compare the following two processes, where reading takes place with actions $read_1$ and $read_2$.

$$\textstyle\sum_{m_1:Message} read_1(m_1) \cdot \sum_{m_2:Message} read_2(m_2) \cdot forward(m_1, m_2).$$
$$\textstyle\sum_{m_1,m_2:Message} read_1(m_1) \cdot read_2(m_2) \cdot forward(m_1, m_2).$$

In the first (correct) process, the message m_2 is chosen when the action $read_2$ takes place. In the second process, the message m_2 to be read is chosen when action $read_1$ takes place. When doing $read_2$, the value to be read is already fixed. If this fixed value is not equal to the value to be read, a deadlock occurs.

The axioms for the sum operator given in table 4.2 are quite subtle. We defer full treatment of them to chapter 8. In order to use them it is necessary that data variables that occur in the sum operator must not bind variables in terms that are substituted for process variables as x, y and z. For variables written as $X(d)$ it is allowed to substitute a term with a data variable d even if d becomes bound by a surrounding sum.

So, for the axiom SUM1 no process containing the variable d can be substituted for x. This is another way of saying that d must occur free for any process p that we substitute for x. Therefore, the sum operator can be omitted.

Exercise 4.1.6. Specify a (one time usable) buffer that reads a natural number, and forwards it if the number is smaller than 100. Otherwise it should flag an overflow.

4.1.6 Recursive processes

With the description of one time usable buffers in the previous section it already became apparent that continuing behaviour must also be described. This is done by introducing process variables and defining their behaviour by equations. So, consider the following, which describes the alarm clock at the left in figure 2.2:

act $set, alarm, reset$;
proc $P = set \cdot Q$;
$Q = reset \cdot P + alarm \cdot Q$;

This declares process variables P and Q (often just called processes). Note that we use the keyword **proc** to indicate that we define a process. The process variable P corresponds to the situation where the alarm clock is switched off, and the process variable Q corresponds to the state where the alarm clock is set.

If in a set of equations defining a process there are only single variables at the left we speak of a *recursive specification*. The variables at the left are called the defined process variables. If every occurrence of a defined process variable at the right is preceded by an action, we speak about a *guarded recursive specification*. It has the property that the behaviour of the defined process variables is indeed uniquely defined. So, in the example above, the behaviour of P and Q is neatly defined.

The keyword **init** can be used to indicated the initial behaviour. In accordance with figure 2.2 this ought to be variable P.

init P;

While interacting with their environment, processes store information that can later influence their behaviour. For this purpose process variables can contain parameters in which this information can be stored. Data and processes are strictly distinguished. This means that there cannot be any reference to processes in data parameters.

We can transform the alarm clock such that it sounds its alarm after a specified number of *tick* actions have happened.

act $set{:}\mathbb{N}; alarm, reset, tick;$
proc $P = \sum_{n:\mathbb{N}} set(n)\cdot Q(n) + tick\cdot P;$
$Q(n{:}\mathbb{N}) = reset\cdot P + (n{\approx}0) \rightarrow alarm\cdot Q(0) \diamond tick\cdot Q(n{-}1);$
init $P;$

Note that the value of n is used in process Q to determine whether an alarm must sound or whether a *tick* action is still possible.

A guarded recursive specification with data also uniquely defines a process. More precisely, they define a function from the data parameters to processes. E.g. the 'process' Q above is actually a function from natural numbers to processes. The equation must be understood to hold for any concrete value for the parameters. So, given the equation for Q above, the following are also valid by taking for n respectively 0, m, $n+1$ and $23k+7$ and simplifying the result.

$$Q(0) = reset\cdot P + alarm\cdot Q(0);$$
$$Q(m) = reset\cdot P + (m{\approx}0) \rightarrow alarm\cdot Q(0) \diamond tick\cdot Q(m{-}1);$$
$$Q(n{+}1) = reset\cdot P + tick\cdot Q(n);$$
$$Q(23k+7) = reset\cdot P + tick\cdot Q(23k+6).$$

This finishes the treatment of sequential processes. We have now seen all the operators to specify sequential behaviour. Using recursion we can specify iterative behaviour and by using data parameters in these equations they are suitable to describe even the most complex real life systems.

Exercise 4.1.7. Describe the behaviour of a buffer with capacity 1 that iteratively reads and forwards a message. Add the option to empty the buffer when it is full, by a specific *empty* action.

4.2 Data types

There are four sorts of predefined data types, standard data types, structured data types, function types and constructed types. But before explaining these, the general mechanism to define new data types, also called data sorts, is explained.

4.2.1 Basic data type definition mechanism

Basically, we have a straightforward data definition mechanism using which all data sorts are built. One can declare arbitrary sorts using the keyword **sort**. Sorts are non-empty, possibly infinite sets with data

elements. For a sort one can define constructor functions using the keyword **cons**. These are functions by which exactly all elements in the sort can be denoted. For instance

sort S;
cons $c, d : S$;

declares sort S in which all elements can be denoted by either c or d. So, S has either two elements, or in case c and d are the same, it has one element.

Using constructor functions, it is possible to declare a sort Nat representing the natural numbers. This is not the built-in sort $\mathbb{N}$, which for efficiency purposes has a different internal structure.

sort Nat;
cons $zero : Nat$;
$successor : Nat \to Nat$;

In this case we have a domain Nat of which all elements can be denoted by an expression of the form:

$$successor(successor(\ldots successor(zero)\ldots)).$$

Without explicitly indicating so, these elements are not necessarily different. Below it is shown how it can be guaranteed that all elements of the sort *Nat* must differ.

Similarly to the definition of sort *Nat*, a sort $\mathbb{B}$ of booleans can be defined. The standard definition for $\mathbb{B}$ is as follows:

cons $true, false : \mathbb{B}$;

The sort $\mathbb{B}$ plays a special role. In the first place the semantics of the language prescribes that the constructors $true$ and $false$ must be different. So, there are exactly two booleans. In the second place, conditions in processes and conditional equations must be of sort $\mathbb{B}$. In appendix A additional operators for this sort are defined.

The following example does not define a proper sort, because it can only be empty. All data elements in S must be denoted as a term consisting of applications of the function f only. But as there is no constant, such a term cannot be constructed, and hence the sort S must be empty. However, empty sorts are not permitted.

sort S;
cons $f : S \to S$;

It is possible to declare sorts without constructor functions. In this case the sort can contain an arbitrary number of elements. In particular, the sort can contain elements that cannot be denoted by a term. As an example it is possible to declare a sort $Message$ without constructors. In this way, one can model for instance data transfer protocols without assuming anything about messages being transferred.

sort $Message$;

Auxiliary functions can be declared using the keyword **map**. For instance, the equality, addition and multiplication operator on natural numbers are not necessary to construct all the numbers, but they are just useful operations. They can be declared as follows:

map $eq : Nat \times Nat \to \mathbb{B}$;
$plus, times : Nat \times Nat \to Nat$;

This is not yet sufficient. It must also be defined how these operate on the numbers. This can be done by introducing equations (using the keyword **eqn**). Often the equations have the form of rewrite rules and they are strictly applied from left to right. Although this is not strictly necessary, this is convenient, because in general the tools interpret the equations as rewrite rules. For addition and multiplication the equations are as follows. Using the keyword **var** the variables needed in the next equation section are declared:

var $n, m : Nat$;
eqn $eq(n, n) = true$;
$eq(zero, successor(n)) = false$;
$eq(successor(n), zero) = false$;
$eq(successor(n), successor(m)) = eq(n, m)$;
$plus(n, zero) = n$;
$plus(n, successor(m)) = successor(plus(n, m))$;
$times(n, zero) = zero$;
$times(n, successor(m)) = plus(n, times(n, m))$;

By applying these equations, one can show which terms are equal to others. For instance showing that $2 * 2 = 4$ goes as follows (where the numbers 2 and 4 represent $successor(successor(zero))$ and $successor(successor(successor(s$ respectively):

$$
\begin{aligned}
&times(successor(successor(zero)), successor(successor(zero))) = \\
&plus(successor(successor(zero)), times(successor(successor(zero)), successor(zero))) = \\
&plus(successor(successor(zero)), plus(successor(successor(zero)), \\
&\qquad times(successor(successor(zero)), zero))) = \\
&plus(successor(successor(zero)), plus(successor(successor(zero)), zero)) = \\
&plus(successor(successor(zero)), successor(successor(zero))) = \\
&successor(plus(successor(successor(zero)), successor(zero))) = \\
&successor(successor(plus(successor(successor(zero)), zero))) = \\
&successor(successor(successor(successor(zero))))
\end{aligned}
$$

There is much to say about whether these equations suffice or whether their symmetric variants should be included, too. These equations are essentially sufficient to prove properties, but for the tools adding more equations can make a huge difference in performance.

When defining functions, it is a good strategy to define them on terms consisting of the constructors applied to variables. In the case above these constructors are $zero$ and $successor(n)$ and the constructor patterns are only used in one argument. But sometimes such patterns are required in more arguments, and it can even be necessary that the patterns are more complex. As an example consider the definition of the function $even$ below which in this form requires patterns $zero$, $successor(zero)$ and $successor(successor(n))$.

map $even : Nat \to \mathbb{B}$;
var $n : Nat$;
eqn $even(zero) = true$;
$even(successor(zero)) = false$;
$even(successor(succcessor(n))) = even(n)$;

It is very well possible to only partly define a function. Suppose the function $even$ should yield $true$ for even numbers, and it is immaterial whether it should deliver $true$ or $false$ for odd numbers. Then the second equation can be omitted. This does not mean that $even(successor(zero))$ is undefined. It has a very precise value (either $true$ or $false$), except that we do not know what it is.

The equations can have conditions that must be valid before they can be applied. The definition above can be rephrased as:

var $n : Nat$;
eqn $eq(n, 0) \to even(n) = true$;
$eq(n, successor(zero)) \to even(n) = false$;
$even(successor(succcessor(n))) = even(n)$;

The equations in an equation section can only be used to show that certain data elements are equal. In order to show that $zero$ and $successor(zero)$ are not equal another mechanism is required. We assume that $true$ and $false$ are different. This is within this basic data definition mechanism the only assumption about terms not being equal. In order to show that other data elements are not equal, we use 'reductio at absurdum', in this case reduction to $true = false$. This always requires an auxiliary function from such a data sort to booleans. In case of the sort Nat we can define the function $less$ to do the job:

Sort	*Rich*	`Plain`
Booleans	$\mathbb{B}$	`Bool`
Positive numbers	$\mathbb{N}^+$	`Pos`
Natural numbers	$\mathbb{N}$	`Nat`
Real numbers	$\mathbb{R}$	`Real`
Structured types	$\mathbf{struct} \ldots \mid \ldots \mid \ldots .$	`struct ... | .... | ....`
Functions	$S \to T$	`S -> T`
Lists	$List(S)$	`List(S)`
Sets	$Set(S)$	`Set(S)`
Bags	$Bag(S)$	`Bag(S)`

Table 4.3: The predefined sorts (S and T are sorts)

map $less : Nat \times Nat \to \mathbb{B}$;
var $n, m : Nat$;
eqn $less(n, zero) = false$;
$less(zero, successor(n)) = true$;
$less(successor(n), successor(m)) = less(n, m)$;

Now assume that $zero$ and $successor(zero)$ are equal, more precisely,

$$zero = successor(zero).$$

Then, we can derive:

$$true = less(zero, successor(zero)) \overset{\text{assumption}}{=} less(zero, zero) = false.$$

So, under this assumption, $true$ and $false$ coincide, leading to the conclusion that $zero$ and $successor(zero)$ must be different. In a similar way it can be proven that that any pair of different natural numbers are indeed different when the function $less$ is present.

Exercise 4.2.1. Give equational specifications of 'greater than or equal' $\geq$, 'smaller than' $<$ and 'greater than' $>$ on the natural numbers.

Exercise 4.2.2. Give specifications of $max{:}Nat \times Nat \to Nat$, $minus{:}Nat \times Nat \to Nat$ and $power{:}Nat \times Nat \to Nat$ where $power(m, n)$ equals m^n.

Exercise 4.2.3. Define a sort *List* on an arbitrary non-empty domain D, with as constructors the empty list $[] :\to List$ and $in : D \times List \to List$ to insert an element of D at the beginning of a list. Extend this with the following non-constructor functions: $append : D \times List \to List$ to insert an element of D at the end of a list; $top : List \to D$ and $toe : List \to D$ to obtain the first and the last element of a list, respectively; $tail : List \to List$ and $untoe : List \to List$ to remove the first and the last element from a list, respectively; $nonempty : List \to \mathbb{B}$ to check whether a list is empty, $length : List \to Nat$ to compute the length of a list, and $+\!\!+ : List \times List \to List$ to concatenate two lists.

4.2.2 Standard data types

When modelling communicating systems, often the same data types are used, namely booleans, numbers, structured types, lists, functions and sets. Therefore, these are predefined. For these data types we use common mathematical notation. The sorts are summarized in table 4.3.

All predefined sorts have functions for if-then-else ($if(\_, \_, \_)$), data equality ($\approx$) and data inequality ($\not\approx$). These equality and inequality functions are functions from their respective data domains to booleans. Applied to a pair of terms they either yield $true$ or $false$, just as every other binary function to sort $\mathbb{B}$ would do, such as e.g., less-than on natural numbers. The equality function should not be confused with equality

among terms ($=$), which indicates which terms are equal. It is the power of the defining equations for data equality, that allows us to use term equality and data equality interchangeably.

The defining equations for $\approx$, $\not\approx$ and $if(\_,\_,\_)$ are as follows:

map $\approx, \not\approx : S \times S \to \mathbb{B}$;
$if : \mathbb{B} \times S \times S \to S$;
var $x, y : S$;
$b : \mathbb{B}$;
eqn $x \approx x = true$;
$x \not\approx y = \neg(x \approx y)$;
$if(true, x, y) = x$;
$if(false, x, y) = y$;
$if(b, x, x) = x$;
$if(x \approx y, x, y) = y$;

The last equation is called *Bergstra's axiom*. As said above equality on terms is strongly related to data equality $\approx$. More precisely, the following lemma holds:

Lemma 4.2.4. For any data sort S for which the equations above are defined, it holds that:

$$x \approx y = true \quad \text{iff} \quad x = y.$$

Proof. For the direction from left to right, we derive:

$$x = if(true, x, y) = if(x \approx y, x, y) = y.$$

For the direction from right to left, we derive:

$$true = x \approx x = x \approx y.$$

□

Bergstra's axiom is generally not used by tools since the shape of the axiom is not very convenient for term rewriting.

4.2.3 Booleans

The sort boolean is already introduced as $\mathbb{B}$. It consists of exactly two different constructors $true$ and $false$. For this sort, the operations are listed in table 4.4, including the syntax used by the tools in the mCRL2 language. The equations with which all the operators on booleans are defined are found in appendix A. Most functions on $\mathbb{B}$ are standard and do not need an explanation. Within booleans, it is possible to use quantifiers $\forall$ and $\exists$. They add a substantial amount of expressivity to the language. This is important because compact, insightful behavioural specifications reduce the number of errors, increase the comprehensibility and in general lead to better balanced behaviour of designs.

We illustrate the expressiveness with two examples, already using the notation for numbers introduced in the next section. The last conjecture of Fermat says that there is no positive number $n > 2$ such that $a^n + b^n = c^n$ for natural number a, b and c. The following process can perform the action $valid$ if the conjecture holds, and *invalid* if the conjecture fails to hold ($\mathbb{N}^+$ is the sort of positive numbers):

proc $Fermat = (\exists a, b, c, n{:}\mathbb{N}^+.(a^n + b^n \approx c^n \wedge n > 2)) \to invalid \diamond valid$;

As the conjecture holds, this process is bisimilar to $valid$ but any tool that wants to figure this out, must be sufficiently strong to prove Fermat's last conjecture.

Using quantifiers a process that repeatedly reads numbers and checks whether they are prime can straightforwardly be specified. Using an action $read$ a natural number is read. If it is a prime, the next action is yes. Otherwise, the reaction is no.

Operator	*Rich*	`Plain`
true	$true$	`true`
false	$false$	`false`
negation	$\neg\_$	`!_`
conjunction	$\_\wedge\_$	`_ && _`
disjunction	$\_\vee\_$	`_ || _`
implication	$\_\Rightarrow\_$	`_ => _`
equality	$\_\approx\_$	`_ == _`
inequality	$\_\not\approx\_$	`_ != _`
conditional	$if(\_,\_,\_)$	`if(_,_,_)`
universal quantification	$\forall\_:\_.\_$	`forall _:_._`
existential quantification	$\exists\_:\_.\_$	`exists _:_._`

Table 4.4: Operators on booleans

proc $PrimeCheck = \sum_{n:\mathbb{N}} read(n)\cdot((\forall m{:}\mathbb{N}^+.n{>}m \wedge m{>}1 \Rightarrow n|_m \not\approx 0) \rightarrow yes \diamond no)\cdot PrimeCheck;$

The notation $n|_m$ means n modulo m.

The downside of the expressivity of quantifiers is that tools generally have difficulties to handle them. This may mean that a specification with quantifiers cannot even be simulated. It is the subject of continuous research to make the tools more effective in dealing with these primitives.

4.2.4 Numbers

Positive numbers, natural numbers, integers and reals are represented by the sorts $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$, respectively. These numbers are unbounded. So, there is no largest natural number, and there are no smallest and largest integers. There is an implicit type conversion. Any positive number can become a natural number, which in turn can become an integer, which can become a real number. These automatic conversions apply to any object, not only to constants but also to variables and terms.

The operators on numbers are given in table 4.5. They are all well known. Most operators are defined for all possible types of numbers. So, there are additional operators for $\mathbb{N}^+ \times \mathbb{N}^+$, $\mathbb{N} \times \mathbb{N}$, $\mathbb{Z} \times \mathbb{Z}$ and for $\mathbb{R} \times \mathbb{R}$. The resulting type is the most restrictive sort possible. Addition on $\mathbb{N}^+ \times \mathbb{N}^+$ has as resulting sort $\mathbb{N}^+$, but subtraction on $\mathbb{N}^+ \times \mathbb{N}^+$ has as result sort $\mathbb{Z}$, as the second number can be larger than the first. Some operators have restricted sorts. For instance, for the modulo operator the sort of the second operator must be $\mathbb{N}^+$ as $x|_0$ is generally not defined. In accordance with common usage, we write multiplication generally as a dot, or leave it out completely, instead of writing a '$*$'.

In some cases the sort of the result must be upgraded. For numbers, we have the explicit type conversion operations $A2B$ (pronounce A to B) where $A, B \in \{\mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$. For instance, the expression $n{-}1$ has sort $\mathbb{Z}$, because n can be zero. However, if it is known that n is larger than 0, it can be retyped to $\mathbb{N}$ by writing $\mathbb{Z}2\mathbb{N}(n{-}1)$. These operators are generally only written in specs intended for tools. In textual specifications we generally leave them out.

The reals $\mathbb{R}$ are used to denote time. As it stands the implementation of the reals is very limited. The tools currently deal with it as if they were integers. This of course does not limit the use of reals in processes and it does not limit its use in manual manipulation of processes.

The equations that the tools use for these operators are given in appendix A. The numbers have been designed such that each number has a unique and efficient internal representation.

The operator $x|_n$ is the modulo operation. For a natural number x and a positive number n, the expression $x|_n$ yields a natural number, being the remainder of $x \operatorname{div} n$. This situation is best characterised by the valid equation $x = n(x \operatorname{div} n) + x|_n$.

Calculations with expressions involving modulo operations is very much simplified using the following properties (contributed to Gauss [50]).

1. $(x|_{nm})|_n = x|_n$;

Operator	*Rich*	`Plain`
positive numbers	$\mathbb{N}^{+}\ (1, 2, 3, \ldots)$	`Pos`, `(1,2,3,...)`
natural numbers	$\mathbb{N}\ (0, 1, 2, \ldots)$	`Nat`, `(0,1,2,...)`
integers	$\mathbb{Z}\ (\ldots, -2, -1, 0, 1, 2, \ldots)$	`Int`, `(...,-2,-1,0,1,2,...)`
reals	$\mathbb{R}$	`Real`
equality	$\_ \approx \_$	`_ == _`
inequality	$\_ \not\approx \_$	`_ != _`
conditional	$\mathit{if}(\_, \_, \_)$	`if(_,_,_)`
conversion	$A2B(\_)$	`A2B(_)`
less than or equal	$\_ \leq \_$	`_ <= _`
less than	$\_ < \_$	`_ < _`
greater than or equal	$\_ \geq \_$	`_ >= _`
greater than	$\_ > \_$	`_ > _`
maximum	$\max(\_, \_)$	`max(_,_)`
minimum	$\min(\_, \_)$	`min(_,_)`
absolute value	$\mathrm{abs}(\_)$	`abs(_)`
negation	$-\_$	`-_`
successor	$\mathit{succ}(\_)$	`succ(_)`
predecessor	$\mathit{pred}(\_)$	`pred(_)`
addition	$\_ + \_$	`_ + _`
subtraction	$\_ - \_$	`_ - _`
multiplication	$\_ * \_$	`_ * _`
integer div	$\_ \ \mathrm{div}\ \_$	`_ div _`
integer mod	$\_ \vert \_$	`_ mod _`
exponentiation	$\_^{\wedge}\_$	`exp(_,_)`

Table 4.5: Operations on numbers

Operator	*Rich*	`Plain`
construction	$[\_, \ldots, \_]$	`[_,...,_]`
element test	$\_ \in \_$	`_ in _`
length	$\#\_$	`#_`
cons	$\_ \triangleright \_$	`_ |> _`
snoc	$\_ \triangleleft \_$	`_ <| _`
concatenation	$\_{+\!\!+}\_$	`_ ++ _`
element at position	$\_ . \_$	`_ . _`
the first element of a list	$head(\_)$	`head(_)`
list without its first element	$tail(\_)$	`tail(_)`
the last element of a list	$rhead(\_)$	`rhead(_)`
list without its last element	$rtail(\_)$	`rtail(_)`
equality	$\_ \approx \_$	`_ == _`
inequality	$\_ \not\approx \_$	`_ != _`
conditional	$if(\_, \_, \_)$	`if(_,_,_)`

Table 4.6: Operations on lists

2. $(x|_n + y)|_n = (x + y)|_n$;
3. $(x|_n * y)|_n = (x * y)|_n$;
4. $x|_n = x$ if $0 \leq x < n$;
5. $x|_n < n$.

Exercise 4.2.5. What are the sorts of the successor function $succ(\_)$, and what are the sorts of the predecessor function $pred(\_)$.

Exercise 4.2.6. Prove using the equations in appendix A that the numbers 0 and 1 are different.

Exercise 4.2.7. Prove that $(1+x|_n)|_n = (1+x)|_n$.

4.2.5 Lists

Lists, where all elements are of sort A, are declared by the sort expression $List(A)$. The operations in table 4.6 are predefined for this sort. Lists consist of constructors $[\,]$, the empty list, and $\triangleright$, putting an element in front of a list. All other functions on lists are internally declared as mappings.

Lists can also be denoted explicitly, by putting the elements between square brackets. For instance for lists of natural numbers $[1, 5, 0, 234, 2]$ is a valid list. Using the $.$ operator, an element at a certain position can be obtained where the first element has index 0 (e.g. $[2, 4, 1].1$ equals 4). The concatenation operator $+\!\!+$ can be used to append one list to the end of another. The $\triangleleft$ operator can be used to put an element to the end of a list. So, the lists $[a, b]$, $a \triangleright [b]$, $[a] \triangleleft b$ and $[a]{+\!\!+}[] \triangleleft b$ are all equivalent. The precise equations for lists are given in appendix A.

Exercise 4.2.8. Specify a function $stretch$ that given a list of lists of some sort S, concatenates all these lists to one single list.

Exercise 4.2.9. Define an $insert$ operator on lists of some sort S such that the elements in the list occur at most once. Give a proof that the insert operation is indeed correct.

Exercise 4.2.10. Define an $insert$ function on lists of naturals such that the list is sorted. The smallest elements must occur first in a list. Define a function is_in that checks whether a number n occurs in an ordered list l using the smallest number of steps necessary. Prove that the functions $insert$ and is_in behave exactly the same as the functions $\triangleright$ and $\in$ from appendix A.

Operator	*Rich*	`Plain`
set enumeration	$\{\_,\ldots,\_\}$	`{ _,...,_ }`
bag enumeration	$\{\_:\_,\ldots,\_:\_\}$	`{ _:_,...,_:_}`
comprehension	$\{\_:\_ \mid \_\}$	`{ _:_ | _ }`
element test	$\_ \in \_$	`_ in _`
bag multiplicity	$count(\_,\_)$	`count(_,_)`
subset/subbag	$\_ \subseteq \_$	`_ <= _`
proper subset/subbag	$\_ \subset \_$	`_ < _`
union	$\_ \cup \_$	`_ + _`
difference	$\_ - \_$	`_ - _`
intersection	$\_ \cap \_$	`_ * _`
set complement	$\overline{\_}$	`!_`
convert set to bag	$Set2Bag(\_)$	`Set2Bag(_)`
convert bag to set	$Bag2Set(\_)$	`Bag2Set(_)`
equality	$\_ \approx \_$	`_ == _`
inequality	$\_ \not\approx \_$	`_ != _`
conditional	$if(\_,\_,\_)$	`if(_,_,_)`

Table 4.7: Operations on sets and bags

4.2.6 Sets and bags

Mathematical specifications often use sets or bags. These are declared as follows (for an arbitrary sort A):

sort $S = Set(A)$;
$B = Bag(A)$;

The essential difference between lists and sets (or bags) is that lists are inherently finite structures. It is impossible to build a list of all natural numbers, whereas the set of all natural numbers can easily be denoted as $\{n{:}\mathbb{N} \mid true\}$. Similarly, the infinite set of all even numbers is easily denoted as $\{n{:}\mathbb{N} \mid n|_2{\approx}0\}$. The difference between bags and sets is that elements can occur at most once in a set whereas they can occur with any multiplicity in a bag.

The empty set or bag is represented by an empty enumeration, i.e. $\{\}$. A set enumeration declares a set, not a bag. So e.g. $\{a, b, c\}$ declares the same set as $\{a, b, c, c, a, c\}$. In a bag enumeration the number of times an element occurs has to be declared explicitly. So e.g. $\{a{:}2, b{:}1\}$ declares a bag consisting of two a's and one b. Also $\{a{:}1, b{:}1, a{:}1\}$ declares the same bag. A set comprehension $\{\, x{:}A \mid P(x) \,\}$ declares the set consisting of all elements x of sort A for which predicate $P(x)$ holds, i.e. $P(x)$ is an expression of sort $\mathbb{B}$. A bag comprehension $\{\, x{:}A \mid f(x) \,\}$ declares the bag in which each element x occurs $f(x)$ times, i.e. $f(x)$ is an expression of sort $\mathbb{N}$.

Exercise 4.2.11. Specify the set of all prime numbers.

Exercise 4.2.12. Specify the set of all lists of natural numbers that only contain the number 0. This is the set $\{[], [0], [0, 0], \ldots\}$. Also specify the set of all lists with length 2.

4.2.7 Function types

Just like sets and bags, functions are objects in common mathematical use, and very convenient for abstract modelling of data in behaviour. Therefore, it is possible to use function types in specifications. So,

sort $F = \mathbb{N} \to \mathbb{N}$;
$G = \mathbb{R} \times \mathbb{N} \to \mathbb{R}$;
$H = \mathbb{R} \to F \to List(G)$;

Operator	*Rich*	`Plain`
function application	$\_(\_,\ldots,\_)$	`_(_,...,_)`
lambda abstraction	$\lambda\_{:}D_0,\ldots,\_{:}D_n.\_$	`lambda _:D0,...,_:Dn._`
equality	$\_\approx\_$	`_ == _`
inequality	$\_\not\approx\_$	`_ != _`
conditional	$if(\_,\_,\_)$	`if(_,_,_)`

Table 4.8: Lambda abstraction and function application

declares that F is the sort of functions from natural numbers to natural numbers and G is the sort of functions from real and nat to real. Functions of the complex sort H map reals to functions from F to $List(G)$. The brackets of function types associate to the right. Hence, the sort H equals $\mathbb{R} \to (F \to List(G))$. If the sort $(\mathbb{R} \to F) \to List(G)$ were required, explicit bracketing is needed.

Functions can be made using the lambda abstraction and application (see table 4.8). Lambda abstraction is used to denote functions. E.g.,

$$\lambda n{:}\mathbb{N}.n^2$$

represents a function of sort $\mathbb{N}$ to $\mathbb{N}$ that yields for each argument n its square. This function can be applied to an argument by putting it directly behind the function. For instance

$$(\lambda n{:}\mathbb{N}.n^2)\,4$$

equals 16. More common notation would require brackets around the argument, which is always possible, and would yield

$$(\lambda n{:}\mathbb{N}.n^2)(4)$$

Equality, its negation and an if-then-else function are also defined. By default brackets of function application associate to the left. So, $(f)(g)(h)$ must be read as $((f)(g))(h)$.

Functions are very suitable to describe arrays. We describe a process that maintains an unbounded array in which natural numbers can be stored. There are actions set, get and $show$. The action $set(n,m)$ sets the nth entry of the array to value m. After an action $get(n)$ an action $show(m)$ shows the value m stored at position n in the array.

act $set : \mathbb{N} \times \mathbb{N}$;
$get, show : \mathbb{N}$;
proc $P(a{:}\mathbb{N} \to \mathbb{N})$
$= \sum_{n,m:\mathbb{N}} set(n,m)\cdot P(\lambda z{:}\mathbb{N}.if(z{\approx}n, m, a(z)))$
$+ \sum_{n:\mathbb{N}} get(n)\cdot show(a(n))\cdot P(a)$;

Another example is the specification of a sorting machine. This machine reads arrays of natural numbers, and delivers sorted arrays with exactly the same numbers. The predicate *sorted* expresses that the numbers in an array are increasing and the predicate *equalcontents* expresses that each of the arrays a and a' contain the same elements. But note that *equalcontents* does not preserve the number of occurrences of numbers.

act $read, deliver : \mathbb{N} \to \mathbb{N}$;
map $sorted, equalcontents, includes : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}$;
var $a, a' : \mathbb{N} \to \mathbb{N}$;
eqn $sorted(a) = \forall i{:}\mathbb{N}.a(i) \leq a(i{+}1)$;
$equalcontents(a,a') = includes(a,a') \wedge includes(a',a)$;
$includes(a,a') = \forall i{:}\mathbb{N}.\exists j{:}\mathbb{N}.a(i){\approx}a'(j)$;
proc $P = \sum_{a:\mathbb{N}\to\mathbb{N}} read(a)\cdot \sum_{a'\mathbb{N}\to\mathbb{N}}(sorted(a')\wedge equalcontents(a,a')) \to deliver(a')\cdot P$;

Exercise 4.2.13. Specify a function map that gets a function and applies it to all elements of a given list.

Exercise 4.2.14. Adapt the predicate $equalcontents(a,a',n)$ such that it also preserves the number of occurrences of data elements, assuming that the arrays contain n elements.

4.2.8 Structured types

Structured types, also called functional or recursive types, find their origin in functional programming. The idea is that the elements of a data type are explicitly characterised. For instance, an enumerated type $Direction$ with elements up, $down$, $left$ and $right$ can be characterised as follows:

sort $Direction = \mathbf{struct}\ up?isUp \mid down?isDown \mid left?isLeft \mid right?isRight;$

This says the sort $Direction$ has four constructors characterising different elements. The optional recognisers such as $isUp$ are functions from $Direction$ to $\mathbb{B}$ and yield true iff they are applied to the constructor to which they belong. E.g. $isUp(up) = true$ and $isUp(down) = false$.

It is possible to let the constructors in a structured sort depend on other sorts. So, pairs of elements of sort A and B can be declared as follows:

sort $Pair = \mathbf{struct}\ pair(fst{:}A, snd{:}B);$

This says that any term of sort $Pair$ can be denoted as $pair(a, b)$ where a and b are data elements of sort A and B. The functions fst and snd are so called projection functions. They allow to extract the first and second element out of a pair. They satisfy the equations:

$$fst(pair(a, b)) = a;$$
$$snd(pair(a, b)) = b;$$

Projection functions are optional, and can be omitted.

In structured sorts it is even possible to let sorts depend on itself. Using this, well known recursive data types such as lists and trees can be constructed. A sort *Tree* for binary trees has the following minimal definition:

sort $Tree = \mathbf{struct}\ leaf(A) \mid node(Tree, Tree);$

By adding projection and recogniser functions it looks like:

sort $Tree = \mathbf{struct}\ leaf(val{:}A)?isLeaf \mid node(left{:}Tree, right{:}Tree)?isNode;$

As an example we define a function HE, short for 'holds everywhere', that gets a function of sort $A \to \mathbb{B}$ and checks whether the function yields true in every leaf of the tree.

map $HE : (A \to \mathbb{B}) \times Tree \to \mathbb{B};$
var $f : A \to \mathbb{B};$
$t, u : Tree;$
$a : A;$
eqn $HE(f, leaf(a)) = f(a);$
$HE(f, node(t, u)) = HE(f, t) \wedge HE(f, u);$

The following definition of sort $Tree$ allows the definition of operation HE without pattern matching.

var $f : A \to \mathbb{B};$
$t : Tree;$
eqn $HE(f, t) = if(isLeaf(t), f(val(t)), f(HE(f, left(t))) \wedge f(HE(f, right(t))));$

This last definition has as disadvantage that the equation is not a terminating rewrite rule. Under certain circumstances, tools will have difficulties dealing with such an equation.

The general form of a structured type is the following, where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$ with $1 \leq i \leq n$:

$$\begin{aligned}&\mathbf{struct}\ c_1(pr_{1,1} : A_{1,1},\ \ldots, pr_{1,k_1} : A_{1,k_1})?isC_1\\ &\quad \mid c_2(pr_{2,1} : A_{2,1},\ \ldots, pr_{2,k_2} : A_{2,k_2})?isC_2\\ &\qquad\quad \vdots\\ &\quad \mid c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?isC_n;\end{aligned}$$

Operator	*Rich*	`Plain`
constructor of summation i	$c_i(\_,\ldots,\_)$	`ci(_,...,_)`
recogniser for constructor i	$is\_c_i(\_)$	`is_ci(_)`
projection (i,j), if declared	$pr_{i,j}(\_)$	`prij(_)`
equality	$\_\approx\_$	`_ == _`
inequality	$\_\not\approx\_$	`_ != _`
conditional	$if(\_,\_,\_)$	`if(_,_,_)`

Table 4.9: Operators for structured types

This declares n constructors c_i, projection functions $pr_{i,j}$ and recognisers isC_i. All names have to be chosen such that no ambiguity can arise. The operations in table 4.9 are available after declaring the sort above.

Exercise 4.2.15. Define the sort $Message$ that contains message frames with a header containing the type of the message (ack, $ctrl$, mes), checksum field, and optionally a data field. Leave the data and checksums unspecified.

Exercise 4.2.16. Describe a coffee and tea machine that accepts coins (5 cent, 10 cent, 20 cent, 50 cent, 1 euro and 2 euro). Coffee costs 45 cent, tea costs 25 cent. The machine can pay back, but there is only a limited amount of coins (it knows exactly how many). It is necessary to develop a way how to accept coins, return change and deliver beverages when the machine is low on cash.

4.2.9 Terms and where expressions

There are three kinds of expressions in mCRL2, namely expressions over sorts, over data and over processes. Sort expressions are built using existing sort names and representations of predefined data types and type constructors. Data expressions are terms constructed from operations and variables.

For the construction of data terms the following priority rules apply. The prefix operators have the highest priority, followed by the infix operators, followed by the lambda operator together with universal and existential quantification, followed by the where clause. Table 4.10 lists the infix operators by decreasing priority. The symbols are shown in plain text format and may represent multiple rich text symbols. Operators on the same line have the same priority and associativity. Note that the list operations $\triangleright$, $\triangleleft$ and $+\!\!+$ are split into three priority levels such that expressions with one of these operations as their head symbol are allowed if and only if they match the following pattern, where $b,\ldots,c$, $d,\ldots,e$ and $s,\ldots,t$ are expressions with a priority level greater than $+\!\!+$:

$$b \triangleright \ldots \triangleright c \triangleright s +\!\!+ \ldots +\!\!+ t \triangleleft d \triangleleft \ldots \triangleleft e$$

Operator	plain	associativity
$*$, div , $\mid$, .	`*`, `div`, `mod`, `.`	left
$+, -$	`+`, `-`	left
$\triangleright$	`|>`	right
$\triangleleft$	`<|`	left
$+\!\!+$	`++`	left
$<, >, \leq, \geq, \in$	`<`, `>`, `<=`, `>=`, `in`	none
$\approx, \not\approx$	`==`, `!=`	right
$\wedge, \vee$	`&&`, `||`	right
$\Rightarrow$	`=>`	right

Table 4.10: Precedence of infix operators on data

Where clauses may be used as an abbreviation mechanism in data expressions. A where clause is of the form e **whr** $a_1 = e_1, \ldots, a_n = a_n$ **end**, with $n \in \mathbb{N}$. Here, e is a data expression and, for all i,

$1 \leq i \leq n$, a_i is an identifier and e_i is a data expression. Expression e is called the body and each equation $a_i = e_i$ is called a *definition*. Each identifier a_i is used as an abbreviation for e_i in e, even if a_i is already defined in the context. Also, an identifier a_i may not occur in any of the expressions e_j, $1 \leq j \leq n$. As a consequence, the order in which definitions occur is irrelevant.

4.3 Timed processes

T1	$x + \delta \lhd 0 = x$
T2	$c \to x = c \to x \diamond \delta \lhd 0$
T3	$x = \sum_{t:\mathbb{R}} x \lhd t$
T4	$x \lhd t \cdot y = x \lhd t \cdot (t \gg y)$
T5	$x \lhd t = t > 0 \to x \lhd t$
TA1	$\alpha \lhd t \lhd u = (t \approx u) \to \alpha \lhd t \diamond \delta \lhd \min(t, u)$
TA2	$\delta \lhd t \lhd u = \delta \lhd \min(t, u)$
TA3	$(x + y) \lhd t = x \lhd t + y \lhd t$
TA4	$(x \cdot y) \lhd t = x \lhd t \cdot y$
TA5	$(\sum_{d:D} X(d)) \lhd t = \sum_{d:D} X(d) \lhd t$
TI1	$t \gg \alpha \lhd u = t < u \to \alpha \lhd u \diamond \delta \lhd t$
TI2	$t \gg \delta \lhd u = \delta \lhd \max(t, u)$
TI3	$t \gg (x + y) = t \gg x + t \gg y$
TI4	$t \gg (x \cdot y) = (t \gg x) \cdot y$
TI5	$t \gg \sum_{d:D} X(d) = \sum_{d:D} t \gg X(d)$

Table 4.11: Timed axioms for the basic operators

It is possible to describe precisely at what time an action takes place. Because the analysis of explicitly timed processes is much harder than showing the properties of untimed processes, it is advisable to avoid the use of explicit time. Only in rare cases, explicit time is really needed.

However, if necessary, it is possible to express the moment where a multi-action takes place by the 'at' operator. We write $a \lhd t$ to express that action a takes place at time t where t is a real number. We pronounce this as 'a at t'. The machine readable notation is `a@t`. It is assumed that processes start at time 0 and actions happen only after time 0. So, $a \lhd 0 = \delta @ 0$.

Consecutive actions must take place at consecutive moments in time. So, $a \lhd 1 \cdot b \lhd 2 \cdot c \lhd 3$ indicates a perfectly executable sequence of actions. But in $a \lhd 1 \cdot b \lhd 2 \cdot c \lhd 1$ the last c cannot be executed without grossly violating the laws of physics by reversing time. There is a timing inconsistency. We avoid this inconsistency by saying that time cannot proceed after the b action. We do not let time proceed beyond time 2. We call this a *time deadlock* at time 2, which is denoted by $\delta \lhd 2$. The process $a \lhd 1 \cdot b \lhd 2 \cdot c \lhd 1$ is equivalent to $a \lhd 1 \cdot b \lhd 2 \cdot \delta \lhd 2$.

Time deadlocks typically occur when the interaction of parallel processes is studied, and both processes have incompatible time constraints. Typically, one process wants to interact before a certain time t while the other only wants to join in at a later time instant. In this case the system will deadlock at time t. When studying behavioural models with explicit time, it always makes sense to prove the specification free of time deadlocks.

Actually, the at operator does not only operate on actions but also on processes. The expression $p \lhd t$ says that the first action of process p must take place at time t.

With timed actions we can describe a very precise clock that ticks every time unit:

act $tick$;
proc $Clock(t{:}\mathbb{R}) = tick \lhd (t{+}1) \cdot Clock(t{+}1)$;

For a any $u > 0$ of sort $\mathbb{R}$, the process $Clock(u)$ exhibits sequence $tick \lhd (u{+}1) \cdot tick \lhd (u{+}2) \cdot tick \lhd (u{+}3) \cdot \cdots$. For $u \leq 1$, the first action of the sequence takes place at time ≤ 0. As actions must take place after time 0, the first action deadlocks at time 0 in that case.

It is also possible to model a *drifting* clock. This is a clock where each subsequent tick can come up to ϵ too early or too late.

act $tick$;
proc $DriftingClock(t{:}\mathbb{R}) = \sum_{u:\mathbb{R}} (t{+}1{-}\epsilon \leq u \leq t{+}1{+}\epsilon) \rightarrow tick \lhd u \cdot DriftingClock(u)$;

There are two approaches to describing time constraints. The first one is by prescribing that something must happen before a certain time. So, we can describe that after an a action a b must happen within 5 time units:

$$\sum_{t:\mathbb{R}} a \lhd t \cdot \sum_{u:\mathbb{R}} (u \leq t{+}5) \rightarrow b \lhd u.$$

An alternative is to describe that if the action b does not happen in time, some time out action must take place:

$$\sum_{t:\mathbb{R}} a \lhd t \cdot \sum_{u:\mathbb{R}} (u \leq t{+}5) \rightarrow b \lhd u \diamond timeOut \lhd u.$$

In table 4.11 the axioms for timed sequential processes are given. The axiom T1 says that $\delta \lhd 0$ is not an selectable alternative. Note that $x + \delta \lhd 1$ and x are not equal. It might be that x is a process where an action must be done before time 1. For instance, x can be equal to $a \lhd 0.5$. So, the process $x + \delta \lhd 1$ has an option to wait until time 1, whereas in x an action must be done at time 0.5. This is also the reason why the processes $x + \delta$ and x are not the same in a timed setting. The deadlock δ can wait until eternity, and hence can $x + \delta$, whereas x could have to do an action before a certain time.

The axiom T2 indicates that the if-then is an abbreviated form of the if-then-else. One might expect an axiom of the form $c \rightarrow x = c \rightarrow x \diamond \delta$. This axiom is perfectly valid if x does not refer to time. But in case there is a reference to time, this axiom would equate the following two processes:

$$\sum_{t:\mathbb{R}} (t \approx 2) \rightarrow a \lhd t \diamond \delta, \qquad \sum_{t:\mathbb{R}} (t \approx 2) \rightarrow a \lhd t \diamond \delta \lhd 0.$$

However, the left process equals $a \lhd 2 + \delta$, whereas the right process is equal to $a \lhd 2$. So, in the first process doing an action a at time 2 is an option, whereas in the second process it is unavoidable. Clearly, these processes have different behaviour.

The axiom T3 shows that a process equals a process where the first action occurs at an arbitrary time. The axiom T4 indicates the essence of time, using the auxiliary *initialisation* operator $\gg$. The process $t \gg p$ is the process p that must start after time t. The axiom T4 says that if process x has performed an action at time t, the actions of y must come after time t. The axioms TI1-TI5 characterise the initialisation operator.

The axiom T5 expresses that all actions must take place after time 0. This is clearly illustrated in the following identity, which is derivable using T5.

$$x = \sum_{t:\mathbb{R}} (t > 0) \rightarrow x \lhd t.$$

The axioms TA1-TA5 express how the 'at' operator distributes over basic processes. As indicated by TA1 the time tags of a multi-action must coincide. Inconsistent time tags lead to a timed deadlock at the earliest convenience.

The axioms characterising time look quite straightforward, but it can still be tricky to derive expected properties. For instance one may want to show that there is no real choice between an early timed deadlock and a late action. In an equation (with $t \geq u$):

$$a \lhd t = a \lhd t + \delta \lhd u.$$

We can show this equation by distinguishing between $t > u$ and $t = u$, respectively:

$$a \lhd t \overset{\text{T3}}{=} (\textstyle\sum_{v:\mathbb{R}} a \lhd v) \lhd t \overset{\text{SUM3}}{=} (\textstyle\sum_{v:\mathbb{R}} a \lhd v + a \lhd u) \lhd t \overset{\text{T3}}{=} (a + a \lhd u) \lhd t \overset{\text{TA3}}{=} a \lhd t + a \lhd u \lhd t \overset{t>u,\text{TA1}}{=} a \lhd t + \delta \lhd u,$$
$$a \lhd t \overset{\text{A6}}{=} (a + \delta) \lhd t \overset{\text{TA3}}{=} a \lhd t + \delta \lhd t.$$

Exercise 4.3.1. Specify a drifting clock where the drifting influences when $tick$ happens, but which does not influence the accuracy of the clock.

Exercise 4.3.2. Describe the behaviour of a coffee machine that returns tea or coffee after entering a coin. If after entering a coin no choice is made in 10 seconds, the coin is returned. Furthermore, after making a choice, coffee or tea is served within 5 seconds.

Exercise 4.3.3. Prove the following implication. This implication is known as *Fer-Jan's Lemma*. It was the first process algebraic identity verified using proof checkers:

1. If $x + y = \delta \lhd 0$, then show $x = \delta \lhd 0$,
2. $b \to x \diamond y = b \to x \diamond \delta \lhd 0 + \neg b \to y \diamond \delta \lhd 0$,
3. $a = a + \delta \lhd t$.

Exercise 4.3.4. Prove that $x \lhd t \cdot a \lhd t = x \lhd t \cdot \delta \lhd t$.

4.4 Parallel processes

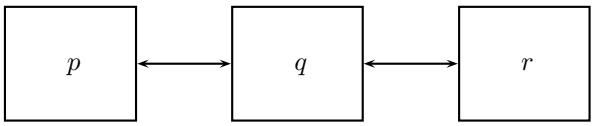
It is now possible to describe sequential processes that can interact with their environment. In this section we describe how to put these in parallel to describe and study the interaction between different processes.

The actions in two parallel processes happen independently of each other. Recall that we consider actions as atomic events in time. Hence, an action from the first process can happen before, after, or at the same time with an action of the second process. The actions happen in an interleaved fashion.

Below we first discuss the interleaving of actions. In the subsequent section we show how actions that happen simultaneously can be synchronised. By synchronising data values they can pass information to each other. Therefore, this is also called communication.

4.4.1 The parallel operator

Two processes p and q are put in parallel by placing the parallel operator between them, i.e. $p \parallel q$. This means that the actions in p happen independently of those in q. The process $p \parallel q$ terminates if both p and q can terminate. Three processes can simply be put in parallel by writing $p \parallel q \parallel r$. As the parallel operator is commutative and associative, the brackets can be omitted. Typically, parallel processes are depicted as follows, where the arrows indicate how processes communicate.



In table 4.12 the axioms governing the behaviour of processes are found. It turns out that it is only possible to give a finite set of axioms, if auxiliary operators are introduced. The necessary operators are the leftmerge ($\between$), the synchronisation merge ($|$) and the before operator ($\ll$), which is specially introduced to deal with time.

The process $p \between q$ (say p left merge q) is almost the same as the process $p \parallel q$ except that the first action must come from p.

The process $p|q$ (say p synchronises with q) is also the same as the process $p \parallel q$, except that the first action must happen simultaneously in p and q. Note that the symbol that we use for synchronisation between processes, is the same as the symbol used for combination of actions to multi-actions. Although,

M	$x \parallel y = x \between y + y \between x + x \mid y$
LM1	$\alpha \between x = (\alpha \ll x) \cdot x$
LM2	$\delta \between x = \delta \ll x$
LM3	$\alpha \cdot x \between y = (\alpha \ll y) \cdot (x \parallel y)$
LM4	$(x + y) \between z = x \between z + y \between z$
LM5	$(\sum_{d:D} X(d)) \between y = \sum_{d:D} X(d) \between y$
LM6	$x \lhd t \between y = (x \between y) \lhd t$
S1	$x \mid y = y \mid x$
S2	$(x \mid y) \mid z = x \mid (y \mid z)$
S3	$x \mid \tau = x$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot x) \mid \beta = \alpha \mid \beta \cdot x$
S6	$(\alpha \cdot x) \mid (\beta \cdot y) = \alpha \mid \beta \cdot (x \parallel y)$
S7	$(x + y) \mid z = x \mid z + y \mid z$
S8	$(\sum_{d:D} X(d)) \mid y = \sum_{d:D} X(d) \mid y$
S9	$x \lhd t \mid y = (x \mid y) \lhd t$
TB1	$x \ll \alpha = x$
TB2	$x \ll \delta = x$
TB3	$x \ll y \lhd t = \sum_{u:\mathbb{R}} u < t \to (x \lhd u) \ll y$
TB4	$x \ll (y + z) = x \ll y + x \ll z$
TB5	$x \ll y \cdot z = x \ll y$
TB6	$x \ll \sum_{d:D} Y(d) = \sum_{d:D} x \ll Y(d)$
TC1	$(x \between y) \between z = x \between (y \parallel z)$
TC2	$x \between \delta = x \cdot \delta$
TC3	$(x \mid y) \between z = x \mid (y \between z)$

Table 4.12: Axioms for the parallel composition operators

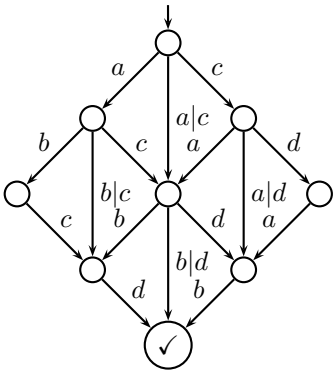


Figure 4.3: The behaviour of $a{\cdot}b \parallel c{\cdot}d$

these are two different operators, we use them interchangeably as their meaning in both cases is the same. More concretely, $a|b$ both represents a multiaction and a synchronisation of two processes both consisting of a single action.

The process $p \ll q$ (pronounce p before q) describes the part of process p that can happen before q is forced to perform an action.

The first axiom marked M in table 4.12 characterises our view on parallelism. The first action in $x \parallel y$ can either come from x, come from y or is an action that happens simultaneously in both of them. In axiom LM1 it is expressed that multi-action α must happen before the process x must do an action. Consider the process $a \between b^{\frown}2$. Then the a action must happen before the b action, and hence it must happen before time 2.

Consider the following process $a{\cdot}b \parallel c{\cdot}d$. Using the axioms it can be rewritten to an expression in which the parallel operator does not occur anymore. This is called expansion. We get:

$$
\begin{aligned}
&a{\cdot}b \parallel c{\cdot}d = \\
&a \between (b \parallel c{\cdot}d) + c \between (a{\cdot}b \parallel d) + a{\cdot}b|c{\cdot}d = \\
&a{\cdot}(b \between c{\cdot}d + c{\cdot}d \between b + b|c{\cdot}d) + c{\cdot}(a{\cdot}b \between d + d \between a{\cdot}b + a{\cdot}b|d) + (a|c){\cdot}(b \parallel d) = \\
&a{\cdot}(b{\cdot}c{\cdot}d{+}c{\cdot}(b \parallel d){+}(b|c){\cdot}d) + c{\cdot}(a{\cdot}(b \parallel d){+}d{\cdot}a{\cdot}b{+}(a|d){\cdot}b) + (a|c){\cdot}(b \between d{+}d \between b{+}b|d) = \\
&a{\cdot}(b{\cdot}c{\cdot}d{+}c{\cdot}(b{\cdot}d{+}d{\cdot}b{+}b|d){+}(b|c){\cdot}d) + c{\cdot}(a{\cdot}(b{\cdot}d{+}d{\cdot}b{+}b|d){+}d{\cdot}a{\cdot}b{+}(a|d){\cdot}b) + (a|c){\cdot}(b{\cdot}d{+}d{\cdot}b{+}b|d)
\end{aligned}
$$

In this expansion quite a number of axioms have been applied each time. We have not even made applications of the before operator visible. Expansion is a very time consuming activity that shows how many options there are possible when parallel behaviour is involved. Later on, we treat ways to get rid of the parallel operator, without getting entangled in parallel expansion. Although not evident from the expansion above, parallel processes have a very typical structure, which becomes clear if the behaviour is plotted in a labelled transition system (see figure 4.3).

The synchronisation operator binds stronger than all other binary operators. The parallel composition and left merge bind stronger than the sum operator but weaker than the conditional operator: $|$, $^{\frown}$, $\cdot$, $\{\gg, \ll\}$, $\rightarrow$, $\{\parallel, \between\}$, $\sum$, $+$.

Exercise 4.4.1. Expand the process $a{\cdot}b \parallel c$. Indicate precisely which axioms have been used.

Exercise 4.4.2. Expand the process $a^{\frown}1{\cdot}b^{\frown}3 \parallel c^{\frown}2$.

Exercise 4.4.3. Prove that the parallel operator is both commutative and associative, i.e. $x \parallel y = y \parallel x$ and $x \parallel (y \parallel z) = (x \parallel y) \parallel z$.

Exercise 4.4.4. Prove that $x \ll (c \rightarrow y) = c \rightarrow (x \ll y)$.

C1	$\Gamma_C(\alpha) = \gamma_C(\alpha)$	C4	$\Gamma_C(x{\cdot}y) = \Gamma_C(x){\cdot}\Gamma_C(y)$
C2	$\Gamma_C(\delta) = \delta$	C5	$\Gamma_C(\sum_{d:D} X(d)) = \sum_{d:D} \Gamma_C(X(d))$
C3	$\Gamma_C(x+y) = \Gamma_C(x) + \Gamma_C(y)$	C6	$\Gamma_C(x \lhd t) = \Gamma_C(x) \lhd t$

Table 4.13: Axioms for the communication operator

4.4.2 Communication between parallel processes

Processes that are put in parallel can execute actions simultaneously, resulting in multi-actions. The communication operator $\Gamma_C(p)$ takes some actions out of a multi-action and replaces them with a single action, provided their data is equal. In this way it is made clear that these actions communicate or synchronise. Here C is a set of allowed communications of the form $a_1|\cdots|a_n{\rightarrow}c$, with $n{>}0$ and a_i and c action names. For each communication $a_1|\cdots|a_n{\rightarrow}c$, the part of a multi-actions consisting of $a_1(d)|\cdots|a_n(d)$ (for some d) in p is replaced by $c(d)$. Note that the data parameter is retained in action c. For example $\Gamma_{\{\,a|b\rightarrow c\,\}}(a(0)|b(0)) = c(0)$, but also $\Gamma_{\{\,a|b\rightarrow c\,\}}(a(0)|b(1)) = a(0)|b(1)$. Furthermore, $\Gamma_{\{\,a|b\rightarrow c\,\}}(a(1)|a(0)|b(1)) = a(0)|c(1)$. The axioms are given in table 4.13.

The function $\gamma_C(\alpha)$ applies the communications described by C to a multi-action α. It replaces every occurrence of a left-hand side of a communication it can find in α with the appropriate result. More precisely:

$$
\begin{array}{lcl}
\gamma_{\emptyset}(\alpha) & = & \alpha \\
\gamma_{C_1 \cup C_2}(\alpha) & = & \gamma_{C_1}(\gamma_{C_2}(\alpha)) \\
\gamma_{\{a_1 \mid \cdots \mid a_n \rightarrow b\}}(\alpha) & = & \begin{cases} b(d) \mid \gamma_{\{a_1 \mid \cdots \mid a_n \rightarrow b\}}(\alpha \setminus (a_1(d)|\cdots|a_n(d))) \\ \qquad \text{if } a_1(d)|\cdots|a_n(d) \sqsubseteq \alpha \text{ for some } d. \\ \alpha \qquad \text{otherwise.} \end{cases}
\end{array}
$$

For example, $\gamma_{\{a|b\rightarrow c\}}(a|a|b|c) = a|c|c$ and $\gamma_{\{a|a\rightarrow a,b|c|d\rightarrow e\}}(a|b|a|d|c|a) = a|a|e$. An action cannot occur in two left hand sides of allowed communications (e.g. $C = \{a|b{\rightarrow}c, a|d{\rightarrow}e\}$ is not allowed). Otherwise, $\gamma_{C_1}(\gamma_{C_2}(\alpha)) = \gamma_{C_2}(\gamma_{C_1}(\alpha))$ would not necessarily hold. Hence, $\gamma_{C_1 \cup C_2}(\alpha)$ is not uniquely defined and γ_C would not be a properly defined function.

V1	$\nabla_V(\alpha) = \alpha$	if $\underline{\alpha} \in V \cup \{\tau\}$	V4	$\nabla_V(x+y) = \nabla_V(x) + \nabla_V(y)$	
V2	$\nabla_V(\alpha) = \delta$	if $\underline{\alpha} \notin V \cup \{\tau\}$	V5	$\nabla_V(x{\cdot}y) = \nabla_V(x){\cdot}\nabla_V(y)$	
V3	$\nabla_V(\delta) = \delta$		V6	$\nabla_V(\sum_{d:D} X(d)) = \sum_{d:D} \nabla_V(X(d))$	
			V7	$\nabla_V(x \lhd t) = \nabla_V(x) \lhd t$	
TV1	$\nabla_V(\nabla_W(x)) = \nabla_{V \cap W}(x)$				

Table 4.14: Axioms for the allow operator

The communication operator lets actions communicate when their data parameters are equal. But it cannot enforce communication. We must explicitly allow those actions that we want to see (e.g. communications), and implicitly block other actions.

The *allow operator* $\nabla_V(p)$ is used for this purpose, where V is a set of multi-action names that specifies exactly which multi-actions from p are allowed to occur. The operator $\nabla_V(p)$ disregards the data parameters of the multi-actions in p, e.g. $\nabla_{\{b|c\}}(a(0){+}b(true,5)|c) = b(true,5)|c$. The empty multi-action τ is not allowed to occur in set V because it cannot be blocked. The axioms are given in table 4.14.

So, consider again the process $a{\cdot}b \parallel c{\cdot}d$ as depicted in figure 4.3. Assume we want that action a communicates with c to e and b communicates with d to f . Then we can first apply the operator $\Gamma_{\{a|c\rightarrow e,b|d\rightarrow f\}}$

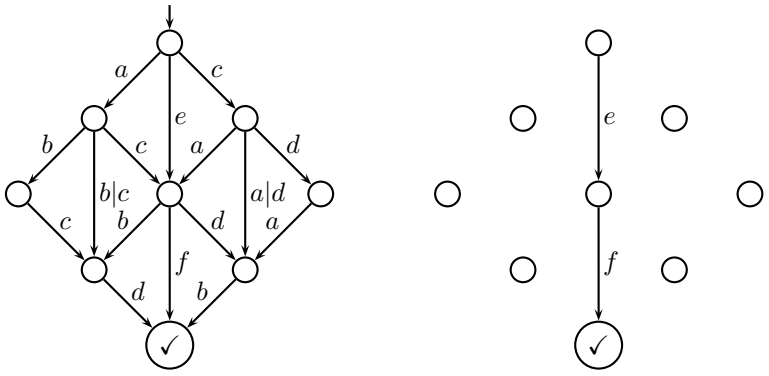
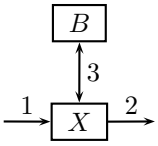


Figure 4.4: The behaviour of $\Gamma_{\{a|c\to e,b|d\to f\}}(a{\cdot}b \parallel c{\cdot}d)$ and $\nabla_{\{e,f\}}(\Gamma_{\{a|c\to e,b|d\to f\}}(a{\cdot}b \parallel c{\cdot}d))$

to this process. We get the state space as depicted in figure 4.4 at the left. As a next operation we say that we only allow communications e and f to occur, effectively blocking all (multi-)actions in which an a, b, c or d occurs. The labelled transition system in figure 4.4 at the right belongs to the expression $\nabla_{\{e,f\}}(\Gamma_{\{a|c\to e,b|d\to f\}}(a{\cdot}b \parallel c{\cdot}d))$.

As a more realistic example we describe a system with a switching buffer X and a temporary store B. Data elements can be received by X via channel 1. An incoming datum is either sent on via channel 2, or stored in a one-place buffer B via channel 3. For sending an action via channel i we use the action s_i and for receiving data via channel i we use r_i.



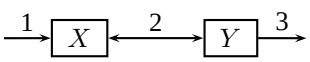
The processes X and B are defined as follows:

act $r_1, s_2, s_3, r_3, c_3{:}D$;
proc $X = \sum_{d:D}(r_1(d) + r_3(d)){\cdot}(s_2(d) + s_3(d)){\cdot}X$;
$B = \sum_{d:D} r_3(d){\cdot}s_3(d){\cdot}B$;

Consider the behaviour $S = \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(X \parallel B))$. In order to depict the labelled transition system we let D be equal to $\{d_1, d_2\}$. In figure 4.5 the behaviour of the processes X, B and S are drawn. Note that it is not straightforward to combine the behaviour of X and B and apply the communication and allow operator. In subsequent chapters we will provide different techniques to do this.

As we have the transition system of S we can answer a few questions about its behaviour. For instance, it is obvious that there are no deadlocks. It is also easy to see that reading at channel 1 and delivery at channel 2 does not necessarily have to take place in sequence. Reading more than two times at channel 1 without any intermediate delivery at channel 2 is also not possible. The system S can store at most two data elements.

Exercise 4.4.5. Data elements (from a set D) can be received by a one-place buffer X via channel 1, in which case they are sent on to a one-place buffer Y via channel 2. Y either forwards an incoming datum via channel 2, or it returns this datum to X via channel 2. In the latter case, X returns the datum to Y via channel 2.



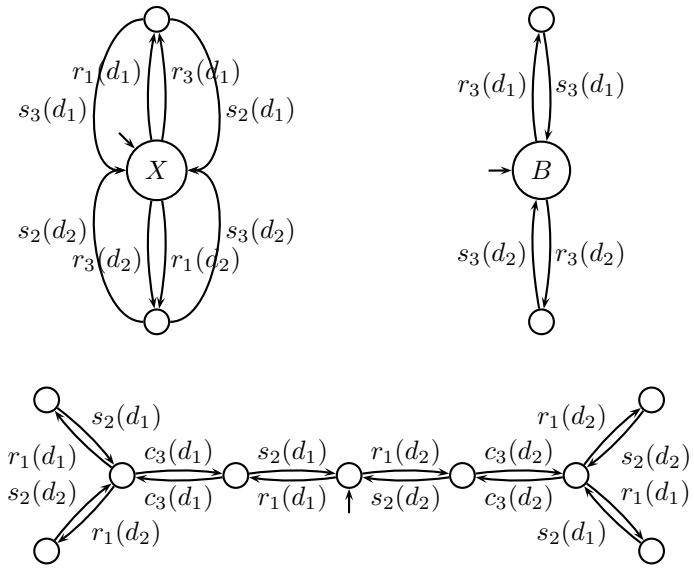


Figure 4.5: The LTSs of X, B and $\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(X \parallel B))$

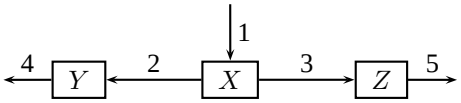
X and Y are defined by the following recursive specification:

act $r_1, s_2, r_2, c_2, s_3{:}D$;
proc $X = \sum_{d:D}(r_1(d) + r_2(d))\cdot s_2(d)\cdot X$;
$Y = \sum_{d:D} r_2(d)\cdot(s_3(d) + s_2(d))\cdot Y$;

Let S denote $\nabla_{\{r_1,c_2,s_3\}}(\Gamma_{\{s_2|r_2\to c_2\}}(X \parallel Y))$, and let D consist of $\{d_1, d_2\}$.

- Draw the state space of S.
- Are data elements read via channel 1 and sent in the same order via channel 3?
- Does $\nabla_{\{r_1,c_2\}}(S)$ contain a deadlock? If yes, then give an execution trace to a deadlock state.

Exercise 4.4.6. Data elements (from a set D) can be received by a one-bit buffer X via channel 1, in which case they are sent on in an alternating fashion to one-bit buffers Y and Z via channels 2 and 3, respectively. So the first received datum is sent to Y, the second to Z, the third to Y, etc. Y and Z send on incoming data elements via channels 4 and 5, respectively.



(1) Specify the independent processes X, Y en Z and the parallel composition with the right communication and allow functions around it.

(2) Let D consist of a single element. Draw the state space.

4.4.3 Blocking and renaming

The *blocking operator* $\partial_B(p)$ (also known as *encapsulation*) has the opposite effect of the allow operator. The set B contains action names that are not allowed. Any multi-action containing an action in B is blocked. Blocking $\partial_B(p)$ disregards the data parameters of the actions in p when determining if an action should be blocked, e.g., $\partial_{\{b\}}(a(0) + b(true, 5)|c) = a(0)$. The blocking operator is sometimes used as an auxiliary operator, by blocking certain actions when analysing processes. For instance blocking the

E1	$\partial_B(\tau) = \tau$		E6	$\partial_B(x + y) = \partial_B(x) + \partial_B(y)$
E2	$\partial_B(a(d)) = a(d)$	if $a \notin B$	E7	$\partial_B(x{\cdot}y) = \partial_B(x){\cdot}\partial_B(y)$
E3	$\partial_B(a(d)) = \delta$	if $a \in B$	E8	$\partial_B(\sum_{d:D} X(d)) = \sum_{d:D} \partial_B(X(d))$
E4	$\partial_B(\alpha\|\beta) = \partial_B(\alpha)\|\partial_B(\beta)$		E9	$\partial_B(x \lhd t) = \partial_B(x) \lhd t$
E5	$\partial_B(\delta) = \delta$		E10	$\partial_H(\partial_{H'}(x)) = \partial_{H \cup H'}(x)$

Table 4.15: Axioms for the blocking operator

R1	$\rho_R(\tau) = \tau$	
R2	$\rho_R(a(d)) = b(d)$	if $a{\to}b \in R$ for some b
R3	$\rho_R(a(d)) = a(d)$	if $a{\to}b \notin R$ for all b
R4	$\rho_R(\alpha\|\beta) = \rho_R(\alpha)\|\rho_R(\beta)$	
R5	$\rho_R(\delta) = \delta$	
R6	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$	
R7	$\rho_R(x{\cdot}y) = \rho_R(x){\cdot}\rho_R(y)$	
R8	$\rho_R(\sum_{d:D} X(d)) = \sum_{d:D} \rho_R(X(d))$	
R9	$\rho_R(x \lhd t) = \rho_R(x) \lhd t$	

Table 4.16: Axioms for the renaming operator

possibility to lose messages allows to get insight in the 'good weather' behaviour of a communication protocol more easily. The blocking operator is characterised by the axioms in table 4.15.

The *Renaming operator* ρ_R is used to rename action names. The set R contains renamings of the form $a{\to}b$. For a process $\rho_R(p)$ this means that every occurrence of action name a in p is replaced by action name b. Renaming $\rho_R(p)$ also disregards the data parameters. When a renaming is applied the data parameters are retained, e.g. $\rho_{\{a \to b\}}(a(0) + a) = b(0) + b$. To avoid unclarities, every action name may only occur once as a left-hand side of $a \to b$ in R. All renamings are applied simultaneously, i.e., a renamed action cannot be renamed twice in one application of the renaming operator. So $\rho_{\{a \to b, b \to c\}}$ renames action label a to b, not to c. The axioms are given in table 4.16.

4.5 Hiding internal behaviour

H1	$\tau_I(\tau) = \tau$		H6	$\tau_I(x{+}y) = \tau_I(x) + \tau_I(y)$
H2	$\tau_I(a(d)) = \tau$	if $a \in I$	H7	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$
H3	$\tau_I(a(d)) = a(d)$	if $a \notin I$	H8	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$
H4	$\tau_I(\alpha\|\beta) = \tau_I(\alpha)\|\tau_I(\beta)$		H9	$\tau_I(x \lhd t) = \tau_I(x) \lhd t$
H5	$\tau_I(\delta) = \delta$		H10	$\tau_I(\tau_{I'}(x)) = \tau_{I \cup I'}(x)$

Table 4.17: Axioms for the hiding operator

As indicated in the previous chapter, hiding information is very important to obtain insight in the behaviour of processes. For this purpose the hiding operator τ_I is defined. The action names in the set I are removed from multi-actions. So, $\tau_{\{a\}}(a|b) = b$ and $\tau_{\{a\}}(a) = \tau$. The axioms for hiding are listed in table 4.17.

U1	$\Upsilon_U(\tau) = \tau$		U5	$\Upsilon_U(\delta) = \delta$
U2	$\Upsilon_U(a(d)) = int$	if $a \in U$	U6	$\Upsilon_U(x{+}y) = \Upsilon_U(x) + \Upsilon_U(y)$
U3	$\Upsilon_U(a(d)) = a(d)$	if $a \notin U$	U7	$\Upsilon_U(x{\cdot}y) = \Upsilon_U(x){\cdot}\Upsilon_U(y)$
U4	$\Upsilon_U(\alpha\|\beta) = \Upsilon_U(\alpha)\|\Upsilon_U(\beta)$		U8	$\Upsilon_U(\sum_{d:D} X(d)) = \sum_{d:D} \Upsilon_U(X(d))$
			U9	$\Upsilon_U(x \lhd t) = \Upsilon_U(x) \lhd t$

Table 4.18: Axioms for the pre-hiding operator

It is convenient to be able to postpone hiding of actions, by first renaming them to a special visible action int which is subsequently renamed to τ. For this purpose the straightforward *pre-hide* operator Υ_U is defined where U is a set of action labels. All actions with labels in U are renamed to the action int and the data is removed. The important property of the pre-hide operator is that $\tau_{I\cup\{int\}}(x) = \tau_{\{int\}}(\Upsilon_I(x))$. The axioms for the pre-hide operator are in table 4.18.

In order to get the required insight in systems, we need axioms to get rid of the internal action. Within the context of strong bisimulation this is expressed in axiom S3 (and axiom MA3), saying that τ is the empty multi-action.

In the context of branching bisimulation, the axioms in table 4.19 can be used, provided no time is used. But note that these axioms do not hold in strong bisimulation. When using axioms that are not valid in strong bisimulation, we always explicitly state which equivalence is intended. The axiom B1 is pretty obvious. It says that a trailing τ after a process can be omitted. The axiom B2 is more intriguing and generally seen as the typical axiom for branching bisimulation. It says that behaviour does not have to be apparent at once, but may gradually become visible. Concretely, instead of seeing $y + z$ at once, it is possible to see first the behaviour of y, and after some internal rumble (read τ), the total behaviour of $x + y$ becomes visible.

B1	$x{\cdot}\tau = x$
B2	$x{\cdot}(\tau{\cdot}(y + z) + y) = x{\cdot}(y + z)$

Table 4.19: Axioms for τ, valid in rooted branching bisimulation for untimed processes

In [41] Robin Milner proposed weak bisimulation. The characterizing axioms for weak bisimulation are found in table 4.20.

W1	$x{\cdot}\tau = x$
W2	$\tau{\cdot}x = \tau{\cdot}x + x$
W3	$a{\cdot}(\tau{\cdot}x + y) = a{\cdot}(\tau{\cdot}x + y) + a{\cdot}x$

Table 4.20: Axioms for τ, valid in weak bisimulation for untimed processes

All equivalences in the Van Glabbeek spectrum have their equational characterisation. For us the equations in the table 4.21 are relevant, for failures equivalence, trace equivalence and weak trace equivalence. As an example we apply hiding to the example in section 4.4.2 with a switching buffer and a temporary store. We may be interested in the communication at channels 1 and 2, but we are not interested how X and B exchange information on channel 3. So, we hide the action c_3. So, we are interested in the behaviour of $\tau_{\{c_3\}}(S)$. In the first labelled transition system in figure 4.6 the hiding operator has been applied on the

Failures equivalence	$a\cdot(b\cdot x+u)+a\cdot(b\cdot y+v)=a\cdot(b\cdot x+b\cdot y+u)+a\cdot(b\cdot x+b\cdot y+v)$ $a\cdot x+a\cdot(y+z)=a\cdot x+a\cdot(x+y)+a\cdot(y+z)$
Trace equivalence Weak trace equivalence	$x\cdot(y+z)=x\cdot y+x\cdot z$ $x\cdot(y+z)=x\cdot y+x\cdot z$ $\tau\cdot x=x$ $x\cdot\tau=x$

Table 4.21: Axioms for some other equivalences for untimed processes

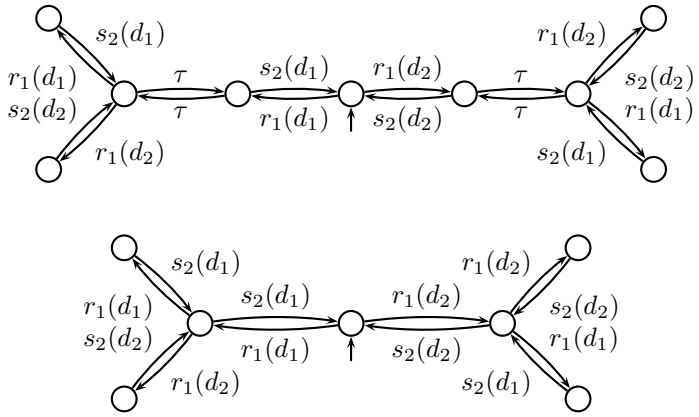


Figure 4.6: The full and reduced LTSs of $\tau_{\{c_3\}}(\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_2|s_2\to c_2\}}(X \parallel B)))$

behaviour as given in figure 4.5. In the second transition system the states connected with τ's have been joined, because they are branching bisimilar.

Exercise 4.5.1. Derive the following equations using the axioms valid in rooted branching bisimulation.

(1) $a\cdot(\tau\cdot b+b)=a\cdot b$;

(2) $a\cdot(\tau\cdot(b+c)+b)=a\cdot(\tau\cdot(b+c)+c)$;

(3) $\tau_{\{a\}}(a\cdot(a\cdot(b+c)+b))=\tau_{\{d\}}(d\cdot(d\cdot(b+c)+c))$;

(4) If $y\subseteq x$, then $\tau\cdot(\tau\cdot x+y)=\tau\cdot x$.

See exercise 4.1.4 for the definition of $\subseteq$.

Exercise 4.5.2. Consider the labelled transition system drawn for the system in example 4.4.5 where c_2 is hidden. Draw this transition system modulo branching bisimulation. Would it make sense to reduce this transition system further using weak bisimulation or weak trace equivalence?

4.6 Alphabet axioms

The parallel operator and its associated operators such as the hiding and allow operator have many relations that can fruitfully be exploited. They are for instance useful when linearising parallel processes. By distributing the communication and allow operator as far as possible over the parallel operator, the generation of many multi-actions can be avoided, that will be blocked later on.

VL1	$\nabla_V(x) = x$	if $\alpha(x) \subseteq V$
VL2	$\nabla_V(x \parallel y) = \nabla_V(x \parallel \nabla_{V'}(y))$	if $\Downarrow(V) \subseteq V'$
DL1	$\partial_H(x) = x$	if $H \cap \mathcal{N}(\alpha(x)) = \emptyset$
DL2	$\partial_H(x \parallel y) = \partial_H(x) \parallel \partial_H(y)$	
TL1	$\tau_I(x) = x$	if $I \cap \mathcal{N}(\alpha(x)) = \emptyset$
TL2	$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	
CL1	$\Gamma_C(x) = x$	if $dom(C) \cap \Downarrow(\alpha(x)) = \emptyset$
CL2	$\Gamma_C(\Gamma_{C'}(x)) = \Gamma_{C \cup C'}(x)$	if $\mathcal{N}(dom(C)) \cap \mathcal{N}(dom(C')) = \emptyset \wedge$ $\mathcal{N}(dom(C)) \cap rng(C') = \emptyset$
CL3	$\Gamma_C(x \parallel y) = x \parallel \Gamma_C(y)$	if $\Downarrow(dom(C)) \cap \Downarrow(\alpha(x)) = \emptyset$
CL4	$\Gamma_C(x \parallel y) = \Gamma_C(x \parallel \Gamma_C(y))$	if $\mathcal{N}(dom(C)) \cap rng(C) = \emptyset$
RL1	$\rho_R(x) = x$	if $dom(R) \cap \mathcal{N}(\alpha(x)) = \emptyset$
RL2	$\rho_R(\rho_{R'}(x)) = \rho_{R \cup R'}(x)$	if $dom(R) \cap dom(R') = \emptyset \wedge dom(R) \cap rng(R') = \emptyset$
RL3	$\rho_R(\rho_{R'}(x)) = \rho_{R''}(x)$	if $R''=\{\langle a,b\rangle \mid (\langle a,b\rangle \in R \wedge a \notin (dom(R') \cup rng(R'))) \vee$ $(\langle c,b\rangle \in R \wedge \langle a,c\rangle \in R') \vee$ $(\langle a,b\rangle \in R' \wedge b \notin dom(R))\}$
RL4	$\rho_R(x \parallel y) = \rho_R(x) \parallel \rho_R(y)$	
VC1	$\nabla_V(\Gamma_C(x)) = \nabla_V(\Gamma_C(\nabla_{V'}(x)))$	if $V' = \{\alpha \mid \beta \mid \ C(\alpha) \mid \beta \in V\}$
VC2	$\Gamma_C(\nabla_V(x)) = \nabla_V(x)$	if $dom(C) \cap \Downarrow(V) = \emptyset$
VD1	$\nabla_V(\partial_H(x)) = \partial_H(\nabla_V(x))$	
VD2	$\nabla_V(\partial_H(x)) = \nabla_{V'}(x)$	if $V' = \{\alpha \mid \alpha \in V \wedge \mathcal{N}(\{\alpha\}) \cap H = \emptyset\}$
VD3	$\partial_H(\nabla_V(x)) = \nabla_{V'}(x)$	if $V' = \{\alpha \mid \alpha \in V \wedge \mathcal{N}(\{\alpha\}) \cap H = \emptyset\}$
VT	$\nabla_V(\tau_I(x)) = \tau_I(\nabla_{V'}(x))$	if $V' = \{\alpha \mid \theta(\alpha, I) \in V\}$
VR	$\nabla_V(\rho_R(x)) = \rho_R(\nabla_{V'}(x))$	if $V' = \{\alpha \mid R(\alpha) \in V\}$
CD1	$\partial_H(\Gamma_C(x)) = \Gamma_C(\partial_H(x))$	if $(\mathcal{N}(dom(C)) \cup rng(C)) \cap H = \emptyset$
CD2	$\Gamma_C(\partial_H(x)) = \partial_H(x)$	if $\mathcal{N}(dom(C)) \subseteq H$
CT1	$\tau_I(\Gamma_C(x)) = \Gamma_C(\tau_I(x))$	if $(\mathcal{N}(dom(C)) \cup rng(C)) \cap I = \emptyset$
CT2	$\Gamma_C(\tau_I(x)) = \tau_I(x)$	if $\mathcal{N}(dom(C)) \subseteq I$
CR1	$\rho_R(\Gamma_C(x)) = \Gamma_C(\rho_R(x))$	if $dom(R) \cap rng(C) = dom(R) \cap \mathcal{N}(dom(C)) =$ $rng(R) \cap \mathcal{N}(dom(C)) = \emptyset$
CR2	$\Gamma_C(\rho_R(x)) = \rho_R(x)$	if $\mathcal{N}(dom(C)) \subseteq dom(R) \wedge \mathcal{N}(dom(C)) \cap rng(R) = \emptyset$
DT	$\partial_H(\tau_I(x)) = \tau_I(\partial_H(x))$	if $I \cap H = \emptyset$
DR	$\partial_H(\rho_R(x)) = \rho_R(\partial_{H'}(x))$	if $H' = \{\alpha \mid R(\alpha) \in H\}$
TR	$\tau_I(\rho_R(x)) = \rho_R(\tau_{I'}(x))$	if $I = \{R(a) \mid a \in I'\}$

Table 4.22: Alphabet Axioms

These relations are characterised by the so called alphabet axioms. The reason is that they are very dependent on the actions labels that occur in a process. The set of action labels in a process p is often called its alphabet and denoted by $\alpha(p)$ and is defined as follows on basic processes.

Definition 4.6.1. Let p be a process expression. We define the alphabet of p, notation $\alpha(p)$ inductively by:

- $\alpha(\alpha) = \{\underline{\alpha}\}$ if $\alpha \not\equiv \tau$.
- $\alpha(\tau) = \emptyset$.
- $\alpha(\delta) = \emptyset$.
- $\alpha(x + y) = \alpha(x) \cup \alpha(y)$.
- $\alpha(x{\cdot}y) = \alpha(x) \cup \alpha(y)$.
- $\alpha(\sum_{d:D} X(d)) = \alpha(X(e))$ where e is an arbitrary data term of sort D.
- $\alpha(x \lhd t) = \alpha(x)$.

In table 4.22 the alphabet axioms are given (inspired by [51]). They depend heavily on operations of action labels. To phrase these action label constraints, we require the following notations.

Definition 4.6.2. Let V be a set containing multi-sets of action names. We define the set $\mathcal{N}(V)$ of actions as follows:

$$\mathcal{N}(V) = \{a \mid a \in \alpha \wedge \alpha \in V\}$$

We define the set with multi-sets of action names

$$\Downarrow(V) \text{ by } \Downarrow(V) = \{\beta \sqsubseteq \alpha \mid \alpha \in V\}.$$

Let $C = \{a_1^1|\ldots|a_{m_1}^1 \to a^1, \ldots, a_1^n|\ldots|a_{m_n}^n \to a^n\}$ be a set of allowed communications or a set of renamings (in which case only one action occurs at the left hand side of the arrow). We write $dom(C)$ and $rng(C)$ as follows:

$$\begin{array}{lcl} dom(C) & = & \{a_1^1|\ldots|a_{m_1}^1, \ldots, a_1^n|\ldots|a_{m_n}^n\} \\ rng(C) & = & \{a^1, \ldots, a^n\} \end{array}$$

If R is a renaming we write $R(a(d_1, \ldots, d_n)) = b(d_1, \ldots, d_n)$ if $a \to b \in R$. If α is a multi-action, we apply R to all individual action names. Ie., if α is an action then $R(\alpha)$ is as indicated above. If $\alpha = \alpha_1|\alpha_2$, then $R(\alpha_1|\alpha_2) = R(\alpha_1)|R(\alpha_2)$. For C a set of communications, we write $C(\alpha) = b$ if $\underline{\alpha} \to b \in C$.

Example 4.6.3. We show how the alphabet axioms can be used to reduce the number of multi-actions when simplifying a process. Consider

$$\nabla_{\{a,d\}}(\Gamma_{\{b|c\to d\}}(a \parallel b \parallel c)). \tag{4.1}$$

Straightforward expansion of the parallel operators yields the following term:

$$\nabla_{\{a,d\}}(\Gamma_{\{b|c\to d\}}(a{\cdot}(b{\cdot}c{+}c{\cdot}b{+}b|c){+}b{\cdot}(a{\cdot}c{+} \\ c{\cdot}a{+}a|c){+}c{\cdot}(a{\cdot}b{+}b{\cdot}a{+}a|b){+}(a|b){\cdot}c{+}(a|c){\cdot}b{+}(b|c){\cdot}a{+}(a|b|c)).$$

Via a straightforward but laborious series of applications of axioms this term can be shown to be equal to $a{\cdot}d + d{\cdot}a$. But using the alphabet axioms CL3 and VL2 we can rewrite equation (4.1) to:

$$\nabla_{\{a,d\}}(a \parallel \nabla_{\{a,d\}}(\Gamma_{\{b|c\to d\}}(b \parallel c))).$$

Expansion of the innermost $b \parallel c$ yields $b{\cdot}c{+}c{\cdot}d{+}b|c$ and application of the communication and allow operator shows that (4.1) is equal to

$$\nabla_{\{a,d\}}(a \parallel d).$$

This is easily shown to be equal to $a{\cdot}d + d{\cdot}a$, too.

Exercise 4.6.4. Simplify $\nabla_{\{e,f\}}(\Gamma_{\{a|b\to e,c|d\to f\}}(a \parallel b \parallel c \parallel d))$.

Chapter 5

Describing properties in the modal μ-calculus

In this chapter we discuss how to denote properties of a reactive system. A property describes some aspect of the behaviour of a system. For instance, deadlock freedom is a simple, but generally a very desired property. Also the property that every message that is sent, will ultimately be received is a typical property of a system.

There are three main reasons to formulate properties of systems:

- Reactive systems are often so complex that its behaviour cannot be neatly characterised. Only certain properties can be characterised. For instance in a leader election protocol processes can negotiate to select one, and only one, leader. In the more advanced protocols it is very hard or even not a priori determined to predict which process will become the leader [14]. Hence, describing the behaviour of the protocol is hard. It is only possible to denote the property that exactly one leader will be chosen.
- In the early design stages, it is unclear what the behaviour of a system will be. Hence, writing down basic properties can help to establish some of the essential aspects of the system behaviour before commencing a detailed behavioural design. In UML *use cases* are used for this purpose. They are examples of potential runs of the system. The property language described here allows use cases to be denoted, but also allows to denote properties which all runs must adhere to.
- It is very common that behavioural descriptions contain mistakes. By checking that a behavioural specification satisfies desirable properties, an extra safeguard is built in to guarantee the correctness of the specification.

It is not easy to characterise the properties of a system. It often happens that what initially appears to be a neat and global property, turns out to be a property that is only valid most of the time. There are exceptions that require the property to be relaxed. Verification of the property is generally the only way to bring such exceptions to light. Techniques to do so are treated in chapter **??**.

5.1 Hennessy-Milner logic

Hennessy-Milner logic is the underlying modal logic for our property language [28]. Its syntax is given by the following BNF grammar:

$$\phi \ ::= \ true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi.$$

The modal formula $true$ is true in each state of a process and $false$ is never true. The connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (not) have their usual meaning. E.g. the formula $\phi_1 \wedge \phi_2$ is valid wherever both ϕ_1 and ϕ_2 hold. It is perfectly valid to use other connectives from propositional logic such as implication ($\Rightarrow$) and bi-implication ($\Leftrightarrow$), as these can straightforwardly be expressed using the connectives above.

The diamond modality $\langle a\rangle\phi$ is valid whenever an a-action can be performed such that ϕ is valid after this a has been done. So, the formula $\langle a\rangle\langle b\rangle\langle c\rangle true$ expresses that a process can do an a followed by b followed by c.

Using the connectives more complex properties can be formulated. Expressing that after doing an a action both a b and c *must* be possible is done by the formula $\langle a\rangle(\langle b\rangle true \wedge \langle c\rangle true)$. Expressing that after an a action no b is possible can be done by $\langle a\rangle\neg\langle b\rangle true$.

The box modality $[a]\phi$ is more involved. It is valid when for every action a that can be done, ϕ holds after doing that a. So, the formula $[a]\langle b\rangle true$ says that whenever an a can be done, a b action is possible afterwards. The formula $[a]false$ says that whenever an a is done, a situation is reached where $false$ is valid. As this cannot be, the formula expresses that an action a is not possible. Likewise, $[a][b]false$ holds when a trace $a\,b$ does not exist.

Although $[a]\phi$ and $\langle a\rangle\phi$ look somewhat similar they are very different. A good way to understand the differences is by giving two transition systems, one where $\langle a\rangle\phi$ holds and $[a]\phi$ is invalid, and vice versa, one where $[a]\phi$ is valid, and $\langle a\rangle\phi$ is invalid. In the first transition system below $\langle a\rangle\phi$ and not $[a]\phi$ are valid. In the second transition system the situation is reversed, namely $[a]\phi$ is valid and not $\langle a\rangle\phi$. Both formulas are true in the third transition system and both are invalid in the fourth.

In the labelled transition system at the left, an a action is possible to a state where ϕ holds, and one to a state where ϕ does not hold. So, $\langle a\rangle\phi$ is valid, and $[a]\phi$ is not. In the labelled transition diagram there is no a-transition at all, so certainly not one to a state where ϕ is valid. So, $\langle a\rangle\phi$ is not valid. But all a-transitions (which are none) go to a state where ϕ is valid. So, $[a]\phi$ holds.

The following identities are valid for Hennessy-Milner formulas. These identities are not only useful to simplify formulas, but can also help to reformulate a modal formula to check whether its meaning matches intuition. For instance the formula $\neg\langle a\rangle[b]false$ can be hard to comprehend. Yet the equivalent $[a]\langle b\rangle true$ clearly says that whenever an action a can be done, it must be followed by an action b. Note that the equations show that the box and diamond modalities are dual to each other, just like the $\wedge$ and the $\vee$ are each other duals.

$\neg\langle a\rangle\phi = [a]\neg\phi$	$\neg[a]\phi = \langle a\rangle\neg\phi$
$\langle a\rangle false = false$	$[a]true = true$
$\langle a\rangle(\phi \vee \psi) = \langle a\rangle\phi \vee \langle a\rangle\psi$	$[a](\phi \wedge \psi) = [a]\phi \wedge [a]\psi$
$\langle a\rangle\phi \wedge [a]\psi \Rightarrow \langle a\rangle(\phi \wedge \psi)$	

Besides these identities, the ordinary identities of propositional logic are also valid.

Exercise 5.1.1.

1. Give a modal formula that says that in the current state an a can be done, followed by a b. Moreover, after the a no c is allowed.

2. Give a modal formula that expresses that whenever an a action is possible in the current state, it cannot be followed by an action b or an action c.

3. Give a modal formula that expresses that whenever in the current state an a action can be done when a b is also possible, the a cannot be followed by a b. In other words the action a cancels a b.

Exercise 5.1.2. Give an argument why the following two formulas are equivalent.

$$\langle a\rangle\,(\langle b\rangle true \vee \langle c_1\rangle false), \qquad\qquad \neg[a]\,([b]false \wedge [c_2]true).$$

5.2 Regular formulas

It is often useful to allow more than just a single action in a modality. For instance to express that after two arbitrary actions, a specific action must happen. Or to say that after observing one or more *receive* actions, a deliver must follow.

A very convenient way to do this, as put forward by [36], is the use of regular formulas within modalities. Regular formulas are based on action formulas, which we define first:

Action formulas have the following syntax:

$$\alpha \ ::= \ a_1|\cdots|a_n \mid \mathit{true} \mid \mathit{false} \mid \overline{\alpha} \mid \alpha \cap \alpha \mid \alpha \cup \alpha.$$

Action formulas define a set of actions. The formula $a_1|\cdots|a_n$ defines the set with only the multi-action $a_1|\cdots|a_n$ in it. The formula true represents the set of all actions and the formula false represents the empty set. For example the modal formula $\langle \mathit{true}\rangle\langle a\rangle \mathit{true}$ expresses that an arbitrary action followed by an action a can be performed. The formula $[\mathit{true}]\mathit{false}$ expresses that no action can be done.

The connectives $\cap$, $\cup$ in action formulas denote intersection and union of sets of action. The notation $\overline{\alpha}$ denotes the complement of the set of actions α with respect to the set of all actions. The formula $\langle \overline{a}\rangle\langle b \cup c\rangle \mathit{true}$ says that an action other than an a can be done, followed by either a b or a c. The formula $[\overline{a}]\mathit{false}$ says that only an a action is allowed.

The precise definitions of modalities with action formulas in them is the following. Let α be a set of actions then:

$$\langle\alpha\rangle\phi = \bigvee_{a\in\alpha} \langle a\rangle\phi \qquad\qquad [\alpha]\phi = \bigwedge_{a\in\alpha} [a]\phi.$$

Regular formulas extend the action formulas to allow the use of sequences of actions in modalities. The syntax of regular formulas, with α an action formula, is:

$$R ::= \ \varepsilon \mid \alpha \mid R{\cdot}R \mid R{+}R \mid R^\star \mid R^+.$$

The formula ε represents the empty sequence of actions. So, $[\varepsilon]\phi = \langle\varepsilon\rangle\phi = \phi$. In other words, it is always possible to perform no action and by doing so, one stays in the same state.

The regular formula $R_1{\cdot}R_2$ represents the concatenation of the sequences of actions in R_1 and R_2. For instance, $\langle a{\cdot}b{\cdot}c\rangle \mathit{true}$ is the same as $\langle a\rangle\langle b\rangle\langle c\rangle \mathit{true}$ expresses that the sequence of actions a, b and c can be performed. The regular formula $R_1{+}R_2$ denotes the union of the sequences in R_1 and R_2. So, $[a{\cdot}b + c{\cdot}d]\mathit{false}$ expresses that neither the sequence $a\,b$ nor the sequence $c\,d$ is possible.

The definitions of both operators is the following:

$$\begin{array}{ll} \langle R_1{+}R_2\rangle\phi = \langle R_1\rangle\phi \vee \langle R_2\rangle\phi & [R_1{+}R_2]\phi = [R_1]\phi \wedge [R_2]\phi \\ \langle R_1{\cdot}R_2\rangle\phi = \langle R_1\rangle\langle R_2\rangle\phi & [R_1{\cdot}R_2]\phi = [R_1][R_2]\phi. \end{array}$$

All the modal formulas described up till now are rather elementary, and not of much use to formulate requirements of real system behaviour. By allowing $R^\star$ and R^+ this improves substantially, because they allow iterative behaviour.

The regular formula $R^\star$ denotes zero or more repetitions of the sequences in R. Similarly, the formula R^+ stands for one or more repetitions. So, $\langle a^\star\rangle \mathit{true}$ expresses that any sequence of a actions is possible. And $[a^+]\phi$ expresses that the formula ϕ must hold in any state reachable by doing one or more actions a.

Two formulas, the *always* and *eventually* modalities, are commonly used. The always modality is often denoted as $\Box\phi$ and expresses that ϕ holds in all reachable states. The eventually modality is written as $\Diamond\phi$ and expresses that there is a sequence of actions that leads to a state in which ϕ holds. Using regular formulas these can be written as follows:

$$\Box\phi = [\mathit{true}^\star]\phi \qquad\qquad \Diamond\phi = \langle \mathit{true}^\star\rangle\phi.$$

The always modality is a typical instance of a so-called safety property. These properties typically say that something bad will never happen. A typical example is that two processes cannot be in a critical region

at the same time. Entering the critical region is modelled by the action $enter$ and leaving the critical region is modelled by an action $leave$. So, in a modal formula we want to say that it is impossible to do two consecutive $enter$s without a $leave$ action in between:

$$[true^\star \cdot enter \cdot \overline{leave}^\star \cdot enter] false.$$

Another typical safety property is that there is no deadlock in any reachable state:

$$[true^\star]\langle true\rangle true.$$

Liveness properties say that something good will eventually happen. For instance the following formula expresses that after sending a message, it can eventually be received:

$$[send]\langle true^\star \cdot receive\rangle true.$$

Compare this to the following formula

$$[send \cdot \overline{receive}^\star]\langle true^\star \cdot receive\rangle true$$

which says that after a $send$ a $receive$ is possible as long as it has not happened.

Exercise 5.2.1. Give modal formulas for the following properties:

1. As long as no *error* happens, a deadlock will not occur.
2. Whenever an a can happen in any reachable state, a b action can subsequently be done unless a c happens cancelling the need to do the b.
3. Whenever an a action happens, it must always be possible to do a b after that, although doing the b can infinitely be postponed.

Exercise 5.2.2. Show that the identities $[R_1 \cdot (R_2 + R_3)]\phi = [R_1 \cdot R_2 + R_1 \cdot R_3]\phi$ and $\langle R_1 \cdot (R_2 + R_3)\rangle\phi = \langle R_1 \cdot R_2 + R_1 \cdot R_3\rangle\phi$ hold. This shows that regular formulas satisfy the left distribution of sequential composition over choice, which justifies to say that regular formulas represent sequences.

5.3 Fixed point modalities

Although regular expressions are very expressive and suitable for stating most behavioural properties, they do not suit every purpose. By adding explicit minimal and maximal fixed point operators to Hennessy-Milner logic, a much more expressive language is obtained. This language is called the modal μ-calculus. As a sign of its expressiveness, it is possible to translate regular formulas to the modal μ-calculus. However, the expressiveness comes at a price. Formulating properties using the modal μ-calculus is far from easy.

The modal μ-calculus in its basic form is given by the following syntax. Note that Hennessy-Milner logic is included in this language.

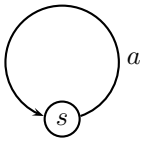
$$\phi \ ::= \ true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \langle a\rangle\phi \mid [a]\phi \mid \mu X.\phi \mid \nu X.\phi \mid X.$$

The formula $\mu X.\phi$ is the minimal fixed point and $\nu X.\phi$ stands for the maximal fixed point. Typically, the variable X is used in fixed points, but other capitals, such as Y, Z are used as well.

A good way to understand fixed point modalities is by considering X as a set of states. The formula $\mu X.\phi$ is valid for all those states in the smallest set X that satisfies the equation $X = \phi$, where X generally occurs in ϕ. Here we abuse notation, by thinking of X as the set of states where ϕ is valid. Similarly, $\nu X.\phi$ is valid for the states in the largest set X that satisfies $X = \phi$.

We can illustrate this by looking at two simple fixed point formulas, namely $\mu X.X$ and $\nu X.X$. So, we are interested in respectively the smallest and largest set of states X that satisfies the equation $X = X$. Now, any set satisfies this equation. So, the smallest set to satisfy it, is the empty set. This means that $\mu X.X$ is not valid for any state. This is equivalent to saying that $\mu X.X = false$. The largest set to satisfy the equation $X = X$ is the set of all states. So, $\nu X.X$ is valid everywhere. In other words, $\nu X.X = true$.

As another example consider the formulas $\mu X.\langle a\rangle X$ and $\nu X.\langle a\rangle X$. One may wonder whether these hold for state s in the following transition system:



The only sets of states to be considered are the empty set $X = \emptyset$ and the set of all states $X = \{s\}$. Both satisfy the 'equation' $X = \langle a \rangle X$. Namely, if $X = \emptyset$, then the equation reduces to $\mathit{false} = \langle a \rangle \mathit{false}$, which is valid. If $X = \{s\}$ it is also clear that this equation holds.

So, $\mu X.\langle a \rangle X$ is valid for all states in the empty set. Hence, this formula is not valid in s. However, $\nu X.\langle a \rangle X$ is valid for all states in the largest set, being $\{s\}$ in this case. So, $\nu X.\langle a \rangle X$ is valid.

In the previous section we have seen that $\Diamond \phi$ means that ϕ can eventually become valid. More precisely, there is a run starting in the current state on which ϕ becomes valid. Very often a stronger property is required, namely that ϕ will eventually become valid along every path. The formula to express this is:

$$\mu X.([\mathit{true}]X \vee \phi).$$

Strictly speaking, this formula will also become true for paths ending in a deadlock, because in such a state $[\mathit{true}]X$ becomes valid. In order to avoid this anomaly, the absence of a deadlock must explicitly be mentioned:

$$\mu X.(([\mathit{true}]X \wedge \langle \mathit{true} \rangle \mathit{true}) \vee \phi).$$

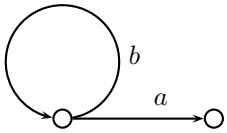
A variation of this is that an a action must unavoidably be done, provided there is no deadlock before the action a.

$$\mu X.[\overline{a}]X.$$

In order to express that a must be done anyhow, the possibility for a deadlock before an action a must explicitly be excluded. This can be expressed by the following formula:

$$\mu X.([\overline{a}]X \wedge \langle \mathit{true} \rangle \mathit{true}).$$

The last two formulas are not valid for the following transition system. The reason is that the b can infinitely often be done, and hence, an a action can be avoided.



The formula $\mu X.([\overline{a}]X \vee \langle a \rangle \mathit{true})$ is valid in the previous transition system. So, this transition system distinguishes between the last formula and the two before that.

Safety properties are generally formulated using the maximal fixed point operator. Dually, liveness properties are formulated using the minimal fixed point operator.

An effective intuition to understand whether or not a fixed point formula holds is by thinking of it as a graph to be traversed, where the fixed point variables are states and the modalities $\langle a \rangle$ and $[a]$ are seen as transitions. A formula is true when it can be made true by passing a finite number of times through the minimal fixed point variables, whereas it is allowed to traverse an infinite number of times through the maximal fixed point variables. In the example with a single a-loop above, the formulas $\mu X.\langle a \rangle X$ and $\nu X.\langle a \rangle X$ can only be made true by passing an infinite number of times through X and/or s. So, the minimal fixed point formula does not hold, and the maximal one is valid.

Consider the formulas $\mu X.\phi$ and $\nu X.\phi$. As the maximal fixed point formula is valid for the largest set of states satisfying $X = \phi$ and the minimal fixed point only for the smallest set, $\nu X.\phi$ is valid whenever $\mu X.\phi$ is. This can concisely be formulated as follows:

$$\mu X.\phi \Rightarrow \nu X.\phi.$$

The minimal and maximal fixed point operators are each other's duals. This boils down to the following two equations:

$$\neg\nu X.\phi = \mu X.\neg\phi \qquad\qquad \neg\mu X.\phi = \nu X.\neg\phi.$$

Note that using these equations it is always possible to remove the negations from modal formulas, provided they have solutions (i.e. variables occur in the scope of an even number of negations).

In order to be sure that a fixed point $\mu X.\phi$ or $\nu X.\phi$ exists, X must occur positively in ϕ. This means that X in ϕ must be preceded by an even number of negations. For counting negations $\phi_1 \to \phi_2$ ought to be read as $\neg\phi_1 \vee \phi_2$. So, for instance $\mu X.\neg X$ and $\nu X.\neg([a]\neg X \vee X)$ are not allowed. In the first case, the variable X is preceded by one negation. But there is no set of states that is equal to its complement, and therefore there is no solution for the equation '$X = \neg X$', and certainly no minimal solution. So, $\mu X.\neg X$ is not properly defined. In the second formula the first occurrence of X is preceded by two negations, which is ok, but the second is preceded by only one. In this case the formula has only a well defined meaning on transition systems without states with outgoing a-transitions. The formula $\nu X.\neg([a]\neg X \vee \neg X)$ has a well defined solution as both occurrences of the variable X are preceded by two negations.

Regular formulas containing a $\star$ or a $+$ are straightforwardly translated to fixed point formulas. The translation is:

$$\begin{array}{ll} \langle R^\star\rangle\phi = \mu X.(\langle R\rangle X \vee \phi) & [R^\star]\phi = \nu X.([R]X \wedge \phi) \\ \langle R^+\rangle\phi = \langle R\rangle\langle R^\star\rangle\phi & [R^+]\phi = [R][R^\star]\phi \end{array}$$

Note that with the rules given above every regular formula can be translated to a fixed point modal formula. From a strictly formal standpoint, regular formulas are unnecessary. However, they turn out to be a very practical tool to formulate many commonly occurring requirements on practical systems.

Until now, we have only addressed fixed point formulas where the fixed point operators are used in a straightforward way. However, by nesting fixed point operators, a whole new class of properties can be stated. These properties are often called *fairness* properties, because these can express that some action must happen, provided it is unboundedly often enabled, or because some other action happens only a bounded number of times.

Consider for instance the formula

$$\mu X.\nu Y.((\langle a\rangle true \wedge [b]X) \vee (\neg\langle a\rangle true \wedge [b]Y)).$$

It says that it is not possible that in the states of each infinite b-trail, a-transitions are always enabled. In other words, a state without a-transitions must infinitely often be enabled. Because the X is preceded by a minimal fixed point, the X can only finitely often be 'traversed'. Within that the variable Y, can be traversed infinitely often, as it is preceded by a maximal fixed point.

By exchanging the minimal and maximal fixed point symbol, the meaning of the formula can become quite different. The formula

$$\nu X.\mu Y.((\langle a\rangle true \wedge [b]X) \vee (\neg\langle a\rangle true \wedge [b]Y))$$

says that on each sequence states reachable via b actions, only finite substretches of states can not have an outgoing a transition. So, typically, if a is enabled in every state, this formula holds.

Exercise 5.3.1. Consider the formulas $\phi_1 = \mu X.[a]X$ and $\phi_2 = \nu X.[a]X$. If possible, give transition systems where ϕ_1 is valid in the initial state and ϕ_2 is not valid and vice versa.

Exercise 5.3.2. Give a labelled transition system that distinguishes between the following formulas

$$\mu X.([\overline{a}]X \vee \langle true^\star\cdot a\rangle true) \quad \text{and} \quad \mu X.[\overline{a}]X.$$

Exercise 5.3.3. Are the following formulas equivalent, and if not, explain why:

$$[send\cdot\overline{receive}^\star]\langle true^\star\cdot receive\rangle true \quad \text{and} \quad [send]\nu X.[\overline{receive}]X.$$

Exercise 5.3.4. What do the following formulas express:

$$\mu X.\nu Y.([a]Y \vee [b]X) \quad \text{and} \quad \nu X.\mu Y.([a]Y \vee [b]X).$$

Is there a process that shows that these formulas are not equivalent?

5.4 Modal formulas with data

Similar to the situation with processes, we also need data, and sometimes time in modal formulas to describe real world phenomena.

Modal formulas are extended with data in three ways, similar to processes. In the first place, modal variables can have arguments. Secondly, actions can carry data arguments and time stamps. And finally, existential and universal quantification is possible. The extensions lead to the following extensions of the syntax, where α stands for a multi-action, R represents a regular formula, ϕ stands for a modal formula and af stands for an action formula. We provide the full syntax, including time constructs ($\lhd t$, Δ, ∇) explained in the next section.

$$
\begin{array}{lll}
\alpha & ::= & \tau \mid a(t_1, \ldots, t_n) \mid \alpha|\alpha. \\
af & ::= & t \mid \mathit{true} \mid \mathit{false} \mid \alpha \mid \overline{af} \mid af \cap af \mid af \cup af \mid \forall d{:}D.af \mid \exists d{:}D.af \mid af \lhd u. \\
R & ::= & \varepsilon \mid af \mid R{\cdot}R \mid R{+}R \mid R^{\star} \mid R^{+}. \\
\phi & ::= & \mathit{true} \mid \mathit{false} \mid t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \forall d{:}D.\phi \mid \exists d{:}D.\phi \mid \langle R\rangle\phi \mid [R]\phi \mid \\
 & & \Delta \mid \Delta \lhd u \mid \nabla \mid \nabla \lhd u \mid \mu X(d_1{:}D_1{:=}t_1, \ldots, d_n{:}D_n{:=}t_n).\phi \mid \nu X(d_1{:}D_1{:=}t_1, \ldots, d_n{:}D_n{:=}t_n).\phi \mid \\
 & & X(t_1, \ldots, t_n).
\end{array}
$$

So, any expression t of sort $\mathbb{B}$ is an action formula. If t is true, it represents the set of all actions and if false, it represents the empty set. The action formula $\exists n{:}\mathbb{N}.a(n)$ represents the set of actions $\{a(n) \mid n \in \mathbb{N}\}$. More specifically, the action formula $\exists d{:}D.af$ represents $\bigcup_{d:D} af$. Dually, $\forall d{:}D.af$ represents $\bigcap_{d:D} af$.

These quantifications are useful to express properties involving certain subclasses of actions. E.g. the formula

$$[\mathit{true}^{\star}{\cdot}\exists n{:}\mathbb{N}.\mathit{error}(n)]\mu X.([\overline{\mathit{shutdown}}]X \wedge \langle \mathit{true}\rangle \mathit{true})$$

says that whenever an error with some number n is observed, a shutdown is inevitable.

There can be a side condition on the error, for instance using a predicate fatal. A shutdown should only occur if the error is fatal:

$$[\mathit{true}^{\star}{\cdot}\exists n{:}\mathbb{N}.(\mathit{fatal}(n) \cap \mathit{error}(n))]\mu X.([\overline{\mathit{shutdown}}]X \wedge \langle \mathit{true}\rangle \mathit{true}).$$

So, if $\mathit{fatal}(n)$ holds, $\mathit{fatal}(n)$ is true and represents the set of all actions. So, $\mathit{fatal}(n) \cap \mathit{error}(n)$ is the set with exactly the action $\mathit{error}(n)$, provided $\mathit{fatal}(n)$ is valid. So, $\exists n{:}\mathbb{N}.(\mathit{fatal}(n) \cap \mathit{error}(n))$ is exactly the set of all those $\mathit{error}(n)$ actions which are fatal.

Reversely, one may be interested in saying that as long as no fatal error occurs, there will be no deadlock.

$$[(\forall n{:}\mathbb{N}.\overline{(\mathit{fatal}(n) \cap \mathit{error}(n))})^{\star}]\langle \mathit{true}\rangle \mathit{true}.$$

In such cases the universal quantifier can be used in action formulas.

In modal formulas it is also allowed to use universal and existential quantifications over data with the standard meaning. So, $\forall d{:}D.\phi$ is true, if ϕ holds for all values from the domain D substituted for d in ϕ. For the existential quantifier, ϕ only needs to hold for some value in D substituted for d. Quantification allows to use data that stretches throughout a formula. For instance saying that the same value is never delivered twice can be done as follows:

$$\forall n{:}\mathbb{N}.[\mathit{true}^{\star}{\cdot}\mathit{deliver}(n){\cdot}\mathit{true}^{\star}{\cdot}\mathit{deliver}(n)]\mathit{false}.$$

Note that this is not possible using the quantifiers of action formulas, as their scope is only limited to a single action formula.

Using the existential quantifier we can express that some action takes place, about which we do not have all information. For instance, after sending a message that message can eventually be delivered with some error code n. The error code is irrelevant for this requirement, but as it is a parameter of the action *deliver*, it must be included in the formula.

$$\forall m{:}\mathit{Message}.[\mathit{true}^{\star}{\cdot}\mathit{send}(m)]\langle \mathit{true}^{\star}{\cdot}\exists n{:}\mathbb{N}.\mathit{deliver}(m,n)\rangle \mathit{true}.$$

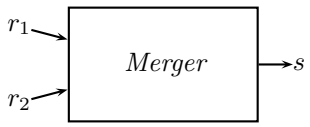


Figure 5.1: A merger reading natural numbers at r_1 and r_2 and delivering at s

Note that it does not really matter where the quantifiers are put. I.e.

$$\forall d{:}D.[true^\star \cdot a(d)]false = [true^\star]\forall d{:}D[a(d)]false$$

A very powerful feature is the ability of putting data in fixed point variables as parameters. Using this it is possible to for instance count the number of events. Saying that a buffer may never deliver more messages than it received can be done as follows:

$$\nu X(n{:}\mathbb{N}{:=}0).[\overline{deliver \cup receive}]X(n) \wedge [receive]X(n{+}1) \wedge [deliver](n > 0 \wedge X(n{-}1)).$$

Here n counts the number of received messages that have not been delivered. The notation $n{:}\mathbb{N}{:=}0$ says that n is a natural number that is initially set to 0. The core of the formula is in its last conjunct, which is false if a deliver is possible while $n{\approx}0$. This conjunct then becomes false, turning the whole modal formula into false.

Note that the fixed point variables that we are using now, are effectively variables ranging over functions from data elements to sets of states. Although this turns modal formulas into rather advanced mathematical objects, the use of data in variables is generally quite intuitive and straightforward.

Another interesting example consists of a merger process (see figure 5.1). It reads two streams of natural numbers, and delivers a merged stream. Reading goes via actions r_1 and r_2 and data is delivered via stream s. The property that the merger must satisfy is that as long as the input streams at r_1 and r_2 are ascending, the output must be ascending too. This can be formulated as follows. The variables in_1, in_2 and out contain the last numbers read and delivered.

$$\begin{aligned}\nu X(in_1{:}\mathbb{N}{:=}0, in_2{:}\mathbb{N}{:=}0, out{:}\mathbb{N}{:=}0).\forall l{:}\mathbb{N}.([r_1(l)](l \geq in_1 \rightarrow X(l, in_2, out)) \wedge \\ [r_2(l)](l \geq in_2 \rightarrow X(in_1, l, out)) \wedge \\ [s(l)](l \geq out \wedge X(in_1, in_2, l))).\end{aligned}$$

It does not appear to be possible to phrase this property without using data in the fixed point variables.

Exercise 5.4.1. Specify a unique number generator that works properly if it does not generate the same number twice.

Exercise 5.4.2. Express the property that a sorting machine only delivers sorted arrays. Arrays are represented by a function $f{:}\mathbb{N} \rightarrow \mathbb{N}$.

Exercise 5.4.3. Specify that a store with products of sort $Prod$ is guaranteed to refresh each product. The only way to see this, is that the difference in the number of $enter(p)$ and $leave(p)$ is always guaranteed to become zero within a finite number of steps.

5.5 Modal formulas with time

In modal formulas it is also possible to refer to time using the '*at*' operator. The action formula $\alpha{@}u$, where u is of sort $\mathbb{R}$, expresses that the action α must take place at time u. More generally, for a set of multi-actions A, $A{@}u$ expresses that all multi-actions in the set must take place at time u. So, the following

formula says that whenever an a action takes place at time 2, it can immediately be followed by a b action at time 3:

$$[true^\star \cdot a\langle 2]\langle b\langle 3\rangle true.$$

If actions do not have a time stamp, they can take place at any time.

In combination with a quantifier, it is possible to express that actions must take place within a certain time interval. For instance, after an emergency call, an ambulance can arrive within ten minutes:

$$[true^\star]\forall t{:}\mathbb{R}.[call\langle t]\langle true^\star\rangle\exists u{:}\mathbb{R}.(u{\leq}t{+}10 \wedge \langle ambulance\langle u\rangle true).$$

A slightly stronger, and very common real time requirement is that some action must be performed within a certain time interval. After switching the system on, a led must light up within 0.1 second:

$$\forall u_1{:}\mathbb{R}.[true^\star \cdot on\langle u_1]\mu X.[\forall u_2{:}\mathbb{R}.\overline{u_2 \leq u_1{+}0.1 \cap led\langle u_2}]X.$$

Stronger requirements with time often require much more stringent formulations. If after a standby, no action is allowed to be done for five seconds, the formula can look like this:

$$\forall u_1, u_2{:}\mathbb{R}.[true^\star \cdot standby\langle u_1 \cdot true\langle u_2](u_2 > u_1 + 5).$$

If time is involved in processes, a requirement can be that it is possible to wait indefinitely, without having to perform an action. For this the *delay* Δ can be used. It simply expresses that time can proceed eternally. The timed delay $\Delta\langle u$ says that time can go on until and including time u without necessarily having to perform an action. Delay can straightforwardly be expressed in terms of timed delays by $\Delta = \forall t{:}\mathbb{R}.\Delta\langle t$.

So, the requirement that after going to standby mode, it is possible to idle indefinitely is expressed by:

$$[true^\star \cdot standby]\Delta.$$

The requirement that after a standby, one must be able to wait for at least five seconds is formulated as follows:

$$\forall u{:}\mathbb{R}.[true^\star \cdot standby\langle u]\Delta\langle(u{+}5).$$

The dual of the delay formula, $\neg\Delta\langle t$, is denoted as $\nabla\langle t$. We call ∇ *yaled* and $\nabla\langle t$ *timed yaled*, lacking better terms (yaled is the reverse of delay). The formula $\nabla\langle t$ says that before time t some action must be done, or time cannot proceed to time t. In the latter case we have a timed deadlock before time t. The duality of timed delay and timed yaled is expressed by:

$$\neg\Delta\langle t = \nabla\langle t \quad \text{and} \quad \neg\nabla\langle t = \Delta\langle t.$$

(Untimed) yaled ∇ says that sometime in the future an action (or a time deadlock) must happen: $\nabla = \exists t{:}\mathbb{R}.\nabla\langle t$.

So, we can express that after an *emergency* action, there must be some response before time t or time must deadlock before time t, using the following formula

$$[true^* \cdot emergency]\nabla\langle t.$$

Exercise 5.5.1. Show that delay and yaled are also dual to each other.

Exercise 5.5.2. Specify that a process cannot have a time deadlock and conversely that it must have a time deadlock.

Exercise 5.5.3. Specify a requirement on a machine that sequentially processes products. Each product entering the machine must leave within five time units. In addition specify that in the long run the time needed to process a product is on average at most 3 time units.

5.6 Equations

In table 5.1 equations are enumerated that hold between modal formulas. This is not a complete list. As it stands, it is unknown which equations must be added to make the list complete.

Note that a the identity $\langle a \cup b \rangle \phi = \langle a \rangle \phi \vee \langle b \rangle \phi$ is *not* valid. This is caused by the fact that $a \cup b$ is the empty action formula, i.e., $false$. The formula $\langle false \rangle \phi$ equals $false$ as it says that in the current state a step can be done, not carrying any action label. Such a step does not exist (τ is an action label). The formula $\langle a \rangle \phi \vee \langle b \rangle \phi$ is not equal to false. It for instance holds for the process $a + b$.

A weaker version, in casu $\langle a \cup b \rangle \phi \Rightarrow \langle a \rangle \phi \vee \langle b \rangle \phi$ is valid. This is also the reason that the following identities have weaker formulations:

$[af_1 \cap af_2]\phi \Leftarrow [af_1]\phi \vee [af_2]\phi$,
$\langle \forall d{:}D.AF(d) \rangle \phi \Rightarrow \forall d{:}D.\langle AF(d) \rangle \phi$,
$[\forall d{:}D.AF(d)]\phi \Leftarrow \exists d{:}D.[AF(d)]\phi$.

Action formulas	
$\overline{true} = false$	$\overline{false} = true$
$\overline{\alpha_1 \cup \alpha_2} = \overline{\alpha_1} \cap \overline{\alpha_2}$	$\overline{\alpha_1 \cap \alpha_2} = \overline{\alpha_1} \cup \overline{\alpha_2}$
$\overline{\exists d{:}D.A(d)} = \forall d{:}D.\overline{A(d)}$	$\overline{\forall d{:}D.A(d)} = \exists d{:}D.\overline{A(d)}$
Regular formulas	
$\langle\varepsilon\rangle\phi = \phi$	$[\varepsilon]\phi = \phi$
$\langle false\rangle\phi = false$	$[false]\phi = true$
$\langle af_1 \cup af_2\rangle\phi = \langle af_1\rangle\phi \vee \langle af_2\rangle\phi$	$[af_1 \cup af_2]\phi = [af_1]\phi \wedge [af_2]\phi$
$\langle af_1 \cap af_2\rangle\phi \Rightarrow \langle af_1\rangle\phi \wedge \langle af_2\rangle\phi$	$[af_1 \cap af_2]\phi \Leftarrow [af_1]\phi \vee [af_2]\phi$
$\langle\exists d{:}D.AF(d)\rangle\phi = \exists d{:}D.\langle AF(d)\rangle\phi$	$[\exists d{:}D.AF(d)]\phi = \forall d{:}D.[AF(d)]\phi$
$\langle\forall d{:}D.AF(d)\rangle\phi \Rightarrow \forall d{:}D.\langle AF(d)\rangle\phi$	$[\forall d{:}D.AF(d)]\phi \Leftarrow \exists d{:}D.[AF(d)]\phi$
$\langle R_1 + R_2\rangle\phi = \langle R_1\rangle\phi \vee \langle R_2\rangle\phi$	$[R_1 + R_2]\phi = [R_1]\phi \wedge [R_2]\phi$
$\langle R_1{\cdot}R_2\rangle\phi = \langle R_1\rangle\langle R_2\rangle\phi$	$[R_1{\cdot}R_2]\phi = [R_1][R_2]\phi$
$\langle R^\star\rangle\phi = \mu X.(\langle R\rangle X \vee \phi)$	$[R^\star]\phi = \nu X.([R]X \wedge \phi)$
$\langle R^+\rangle\phi = \langle R\rangle\langle R^\star\rangle\phi$	$[R^+]\phi = [R][R^\star]\phi$
Proposition logic	
$\phi \wedge \psi = \psi \wedge \phi$	$\phi \vee \psi = \psi \vee \phi$
$(\phi \wedge \psi) \wedge \chi = \phi \wedge (\psi \wedge \chi)$	$(\phi \vee \psi) \vee \chi = \phi \vee (\psi \vee \chi)$
$\phi \wedge \phi = \phi$	$\phi \vee \phi = \phi$
$\neg true = false$	$\neg false = true$
$\phi \wedge true = \phi$	$\phi \vee true = true$
$\phi \wedge false = false$	$\phi \vee true = \phi$
$\phi \wedge (\psi \vee \chi) = (\phi \wedge \psi) \vee (\phi \wedge \chi)$	$\phi \vee (\psi \wedge \chi) = (\phi \vee \psi) \wedge (\phi \vee \chi)$
$\neg(\phi \wedge \psi) = \neg\phi \vee \neg\psi$	$\neg(\phi \vee \psi) = \neg\phi \wedge \neg\psi$
$\neg\neg\phi = \phi$	$\phi \rightarrow \psi = \neg\phi \vee \psi$
$\phi \Rightarrow \psi = \neg\phi \vee \psi$	$\phi \Leftrightarrow \psi = \phi \Rightarrow \psi \wedge \psi \Rightarrow \phi$
Predicate logic	
$\forall d{:}D.\phi = \phi$	$\exists d{:}D.\phi = \phi$
$\neg\forall d{:}D.\Phi(d) = \exists d{:}D.\neg\Phi(d)$	$\neg\exists d{:}D.\Phi(d) = \forall d{:}D.\neg\Phi(d)$
$\forall d{:}D.(\Phi(d) \wedge \Psi(d)) = \forall d{:}D.\Phi(d) \wedge \forall d{:}D.\Psi(d)$	$\exists d{:}D.(\Phi(d) \vee \Psi(d)) = \exists d{:}D.\Phi(d) \vee \exists d{:}D.\Psi(d)$
$\forall d{:}D.(\Phi(d) \vee \psi) = \forall d{:}D.\Phi(d) \vee \psi$	$\exists d{:}D.(\Phi(d) \wedge \psi) = \exists d{:}D.\Phi(d) \wedge \psi$
$\forall d{:}D.\Phi(d) \Rightarrow \Phi(e)$	$\Phi(e) \Rightarrow \exists d{:}D.\Phi(d)$
Hennessy-Milner logic	
$\neg\langle a\rangle\phi = [a]\neg\phi$	$\neg[a]\phi = \langle a\rangle\neg\phi$
$\langle a\rangle false = false$	$[a]true = true$
$\langle a\rangle(\phi \vee \psi) = \langle a\rangle\phi \vee \langle a\rangle\psi$	$[a](\phi \wedge \psi) = [a]\phi \wedge [a]\psi$
$\langle a\rangle\phi \wedge [a]\psi \Rightarrow \langle a\rangle(\phi \wedge \psi)$	
Fixed point equations	
$\mu X.\phi \Rightarrow \nu X.\phi$	
$\neg\mu X.\phi = \nu X.\neg\phi$	$\neg\nu X.\phi = \mu X.\neg\phi$
$\mu X.\phi(X) = \phi(\mu X.\phi(X))$	$\nu X.\phi(X) = \phi(\nu X.\phi(X))$
if $\phi(\psi) \Rightarrow \psi$ then $\mu X.\phi(X) \Rightarrow \psi$	if $\psi \Rightarrow \phi(\psi)$ then $\psi \Rightarrow \nu X.\phi(X)$
Time	
$[a]\phi = \forall t{:}\mathbb{R}.[a{<}t]\phi$	$\langle a\rangle\phi = \exists t{:}\mathbb{R}.\langle a{<}t\rangle\phi$
$\Delta = \forall t{:}\mathbb{R}.\Delta{<}t$	$\nabla = \exists t{:}\mathbb{R}.\nabla{<}t$
$\neg\Delta = \nabla$	$\neg\nabla = \Delta$

Table 5.1: Equivalences between modal formulas

Chapter 6

Modelling of system behaviour

Modelling of system behaviour is generally not too difficult. However, making compact, insightful models that can serve as a means of communication and that can be shown to satisfy all desirable properties, is generally not so easy. This craftmanship can only be obtained by making and analysing many models. There is no other way to learn this than by doing. This chapter provides a number of examples of what such models could look like. But every situation is unique, requiring a model with its own characteristics.

Our extensive modelling experience taught us one rather disconcerting lesson. Systems that are built without an underlying behavioural design are so complex that it is impossible to get them right. In general it does not even suffice to model before building. Without thorough mathematical analysis of its behavioural properties, systems generally exhibit unexpected and undesired behaviour. We call this the 100% rule: all system behaviours that have not been proven to be correct have at least one, but generally many unintended behavioural aspects. Typical examples are unintentional loss or duplication of data, issuing conflicting commands, letting the system get into a global or partial deadlock. In a partial deadlock some of the functionality of the system is not accessible anymore.

This is also the most important motivation for this book. It is not surprising that, as it is hard to get the models of reactive systems right, it is virtually impossible to program reactive systems and hope that they do exactly what is desired. Bugs are detected while testing or using the systems, but are hardly reproducable. The explanation for this is that the number of states in a system typically vary from 10^{10} to $10^{10^{10}}$, although many reactive systems have effectively an infinite state space, due to unbounded buffers or data stores. Testing all states to detect all errors is impossible. But generally, most states are reachable within a few hundred steps, so every error can exhibit itself at essentially any moment.

The examples in this chapter are used throughout the rest of this book where we provide the techniques to verify the correctness. Note that the examples given here are minimal in the sense that realistic systems often have many more features. It requires advanced modelling skills to find the right balance between making the model as close as possible to the real system, and keeping it sufficiently simple such that the model is understandable. Ideally, the model and the real system coincide, and the model fits on a single page.

6.1 Alternating bit protocol

We start out by providing a model of a simple system, namely the alternating bit protocol (ABP) [5, 35]. The alternating bit protocol ensures successful transmission of data through lossy channels. The protocol is depicted in Figure 6.1. The processes K and L are channels that transfer messages from 2 to 3 and from 5 to 6 respectively. However, the data can be lost in transmission, in which case the processes K and L deliver an error. The sender process S and receiver R must take care that despite this loss of data, transfer between 1 and 4 is reliable, in the sense that messages sent at 1 are received at 4 exactly once in the same order in which they were sent.

In order to model the protocol, we assume some sort D with data elements to be transferred. We model the external behaviour of the Alternating Bit Protocol using the following process equation which defines

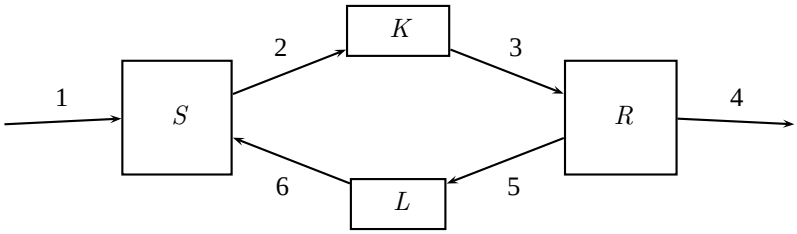


Figure 6.1: Alternating bit protocol

a simple buffer B and we expect that the modelled protocol is branching bisimulation equivalent to this buffer:

proc $\quad B = \sum_{d:D} r_1(d) \cdot s_4(d) \cdot B.$

Action $r_1(d)$ represents "read datum d from channel 1", and action $s_4(d)$ represents "send datum d into channel 4". Strictly spoken, B is a process variable that satisfies the equation above. Note that the external behaviour is actually the simplest conceivable behaviour for a data transfer protocol.

In order to develop the sender and the receiver we must have a good understanding of the exact behaviour of the channels K and L. As will be explained below, the process K sends pairs of a message and a bit, and the process L only forwards bits. We can introduce a data sort *Bits*, but it is more efficient to use the booleans to represent bits. The processes choose internally whether data is delivered or lost using the action i. If it is lost an error message $\bot$ is delivered:

$$\begin{aligned}\textbf{proc} \quad K &= \sum_{d:D, b:\mathbb{B}} r_2(d,b) \cdot (i \cdot s_3(d,b) + i \cdot s_3(\bot)) \cdot K \\ L &= \sum_{b:\mathbb{B}} r_5(b) \cdot (i \cdot s_6(b) + i \cdot s_6(\bot)) \cdot L\end{aligned}$$

Note that the action i cannot be omitted. If it would be removed, the choice between delivering the correct data or the error is made while interacting with the receiver of the message. The receiver can henceforth determine whether the data will be lost or not. This is not what we want to model here. We want to model that whether or not data is lost, is determined internally in K and L. Because the factors that cause the message to be lost are outside our model, we use a nondeterministic choice to model data loss.

We model the sender and receiver using the protocol proposed in [5, 35]. The first aspect of the protocol is that the sender must guarantee that despite data loss in K, data eventually arrives at the receiver. For this purpose, it iteratively sends the same messages to the sender. The receiver sends an acknowledgment to the sender whenever it receives a message. If a message is acknowledged the sender knows that the message is received and it can proceed with the next message.

A problem of this protocol is that data may be sent more than once, and the receiver has no way of telling whether the data stems from a single message which is resent, or whether it stems from two messages that contain the same data. In order to resolve extra control information must be added to the message. A strong point made in [5, 35] is that adding a single bit already suffices for the job. For consecutive message the bit is alternated for each subsequent datum to be transferred. If data is resent, the old bit is used. This explains the name Alternating Bit Protocol.

After receiving a message at the receiver, its accompanying bit is sent back in the acknowledgment. When the bit differs from the bit associated with the last message, the receiver knows that this concerns new data and forwards it to gate 4. If an error $\bot$ arrives at the receiver, it does not know whether this regards an old or new message, and it sends the old bit to indicate that resending is necessary.

Whenever a unexpected bit or an error message arrives at the sender, it knows that the old data must be resent. Otherwise, it can proceed to read new data from 1 and forward it:

First, we specify the sender S in the state that it is going to send out a datum with the bit b attached to it, represented by the process name $S(b)$ for $b \in \{0,1\}$:

proc $S(b{:}\mathbb{B}) = \sum_{d:D} r_1(d)\cdot T(d,b)$,
$T(d{:}D, b{:}\mathbb{B}) = s_2(d,b)\cdot(r_6(b)\cdot S(\neg b) \;+\; (r_6(\neg b) + r_6(\bot))\cdot T(d,b))$.

In state $S(b)$, the sender reads a datum d from channel 1. Next, the system proceeds to state $T(d,b)$, in which it sends this datum into channel 2, with the bit/boolean b attached to it. Then expects to receive the acknowledgement b through channel 6, ensuring that the pair (d,b) has reached the Receiver unscathed. If the correct acknowledgement b is received, then the system proceeds to state $S(\neg b)$, in which it is going to send out a datum with the bit $\neg b$ attached to it. If the acknowledgement is either the wrong bit $\neg b$ or the error message $\bot$, then the system sends the pair (d,b) into channel B once more.

Next, we specify the receiver in the state that it is expecting to receive a datum with the bit b attached to it, represented by the process name $R(b)$ for $b \in \{0,1\}$:

proc $R(b{:}\mathbb{B}) = \sum_{d:D} r_3(d,b)\cdot s_4(d)\cdot s_5(b)\cdot R(\neg b) \;+\; (\sum_{d:D} r_3(d,\neg b) + r_3(\bot))\cdot s_5(\neg b)\cdot R(b)$.

In state $R(b)$ there are two possibilities.

1. If in $R(b)$ the receiver reads a pair (d,b) from channel 3, then this constitutes new information, so the datum d is sent into channel 4, after which acknowledgement b is sent to the sender via channel 5. Next, the receiver proceeds to state $R(\neg b)$, in which it is expecting to receive a datum with the bit $\neg b$ attached to it.

2. If in $R(b)$ the receiver reads a pair $(d,\neg b)$ or an error message $\bot$ from channel 3, then this does not constitute new information. So then the Receiver sends acknowledgement $\neg b$ to the sender via channel 5 and remains in state $R(b)$.

The desired concurrent system is obtained by putting $S(true)$, $R(true)$, K and L in parallel, blocking send and read actions over internal channels, and abstracting away from communication actions over these channels and from the action i. That is, the Alternating Bit Protocol (ABP) is defined by the process term

proc $ABP = \nabla_I(\Gamma_G(S(true) \parallel K \parallel L \parallel R(true)))$

where $I = \{r_1, s_4, c_2, c_3, c_5, c_6, i\}$ and $G = \{r_2|s_2 \to c_2, r_3|s_3 \to c_3, r_5|s_5 \to c_5, r_6|s_6 \to c_6\}$.

Does the scheme with alternating bits work correctly? Or in other words do the buffer B and the alternating bit protocol ABP behave the same when only the actions r_1 and s_4 are visible? This question is concisely stated by the following equation to hold in branching bisimulation:

$$B \;=\; \tau_{\{c_2,c_3,c_5,c_6,i\}}(ABP) \tag{6.1}$$

In section 11.2.3 the proof of this equation is given, using the explicit assumption that i is fair. This says that the channels do not always lose the data. Note that with the equation above, it is possible to use B when proving the correctness of a system employing the alternating bit protocol. This makes reasoning about such systems considerably simpler.

Exercise 6.1.1. In the mCRL2 toolset a description of the alternating bit protocol can be found in the directory `examples/academic/abp.mcrl2`. Use the tool `mcrl22lps abp.mcrl2 abp.lps` to transform it into linear process. This linear process can be simulated using the `xsim` tool. Transfer several data items from the sender to the receiver and familiarize yourself that the observed actions indeed match with the specified behaviour in this section.

Exercise 6.1.2. Make a copy of the file `abp.mcrl2` from exercise 6.1.1 and remove the bits. Show with the simulator `xsim` that data elements can be duplicated and/or lost.

6.2 Sliding window protocol

In the ABP, the Sender sends out a datum and then waits for an acknowledgement before it sends the next datum. In situations where transmission of data is relatively time consuming, this procedure tends to be unacceptably slow. In sliding window protocols [12] (see also [47]), this situation has been resolved as the Sender can send out multiple data elements before it requires an acknowledgement. This protocol is so effective that it is one of the core protocols of the internet.

The most complex sliding window protocol (SWP) described in [47] was modelled in 1991 using techniques as described in this book [10]. This model revealed a deadlock. When confronted with this, the author of [47] indicated that this problem remained undetected for a whole decade, despite the fact that the protocol had been implemented a number of times. There is some evidence, that this particular deadlock occurs in actual implementations of internet protocols, but this has never been systematically investigated. In recent editions of [47] this problem has been removed.

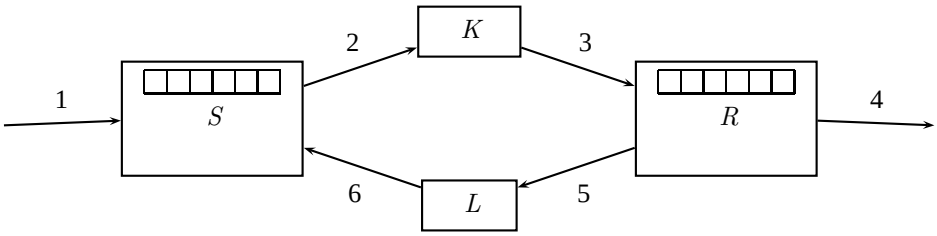


Figure 6.2: Sliding window protocol

We concentrate on a variant of the sliding window protocol which is unidirectional, to keep the model to its essential minimum. The essential feature of the sliding window protocol is that it contains buffers in the sender and the receiver to keep copies of the data in transit. This is needed to be able to resend this data if it turns out after a while that the data did not arrive correctly. Both buffers have size n. This means that there can be at most $2n$ data elements under way, i.e., there can at most be $2n$ data elements that have been received at gate 1, but have not been delivered at gate 4. This suggests that the external behaviour of the sliding window protocol is a bounded first-in-first-out (fifo) queue of length $2n$.

As in the ABP elements from a nonempty data domain D are sent from a Sender to a Receiver. The external behaviour of the sliding window protocol is a fifo queue defined by the following equation, where the sliding window protocol with buffer size n behaves as $FIFO([], 2n)$:

proc $FIFO(l{:}List(D), m{:}\mathbb{N}^+)$
$= \sum_{d:D} \#(l){<}m \rightarrow r_1(d){\cdot}FIFO(l{\lhd}d, m)$
$+ \#(l) > 0 \rightarrow s_4(head(l)){\cdot}FIFO(tail(l), m);$

Note that $r_1(d)$ can be performed until the list l contains m elements, because in that situation the sending and receiving windows will be filled. Furthermore, $s_4(head(l))$ can only be performed if l is not empty.

We now give a model of the sliding window protocol which implements the bounded buffer on top of two unreliable channels K and L. The setup is similar to that of the alternating bit protocol. See figure 6.2.

The channels differ from those of the ABP because they do not deliver an error in case of data loss. An indication of an error is necessary for the ABP to work correctly, but it is not very realistic. A better model of a channel is one where data is lost without an explicit indication. In this case we still assume that the channels can only carry a single data item, and have no buffer capacity. However, the channels can be replaced by others, for instance with bounded, or unbounded capacity. As long as the channels transfer a message every now and then, the sliding window protocol can be used to transform these unreliable channels into a reliable fifo queue.

proc $K = \sum_{d:D,k:\mathbb{N}} r_2(d,k)\cdot(i\cdot s_3(d,k)+i)\cdot K;$
$L = \sum_{k:\mathbb{N}} r_5(k)\cdot(i\cdot s_6(k)+i)\cdot L;$

The sender and the receiver in the SWP both maintain a buffer of size n containing the data being transmitted. The buffers are represented by a function from $\mathbb{N}$ to D indicating which data element occurs at which position. Only the first n places of these functions are used. In the receiver we additionally use a buffer of booleans of length n to recall which of the first n positions in the buffer contain valid data.

sort $DBuf = \mathbb{N} \to D;$
$BBuf = \mathbb{N} \to \mathbb{B};$

The sliding window protocol uses a numbering scheme to number the messages that are sent via the channels. It turns out that if the sequence numbers are issued modulo $2n$, messages are not confused and are transferred in order. Each message with sequence number j is put at position $j|_n$ in the buffers.

We use the following auxiliary functions to describe the sliding window protocol. The function $empty$ below represents a boolean buffer that is false everywhere, indicating that there is no valid data in the buffer. We use notation $q[i{:=}d]$ to say that position i of buffer q is filled with datum d.

The most involved function is $nextempty_{mod}(i,b,m,n)$. It yields the first position in buffer b starting at $i|_n$ that contains $false$. If the first m positions from $i|_n$ of b are all $true$, it yields the value $(i{+}m)|_{2n}$. The variable m is used to guarantee that the function $nextempty$ is well defined if b is false at all its first n positions. The variables have the following sorts: $d{:}D$, $i,j,m{:}\mathbb{N}$, $n{:}\mathbb{N}^+$, $q{:}DBuf$, $c{:}\mathbb{B}$ and $b{:}BBuf$.

eqn $empty = \lambda j{:}\mathbb{N}.false;$
$q[i{:=}d] = \lambda j{:}\mathbb{N}.if(i{\approx}j, d, q(j));$
$b[i{:=}c] = \lambda j{:}\mathbb{N}.if(i{\approx}j, c, b(j));$
$nextempty_{mod}(i,b,m,n) = if(b(i|_n) \wedge m{>}0, nextempty_{mod}((i{+}1)|_{2n}, b, m{-}1, n), i|_{2n});$

Below we model the sender process S. The variable ℓ contains the sequence number of the oldest message in sending buffer q and m is the number of items in the sending buffer. If data arrives via gate 1, it is put in the sending buffer q, provided there is place. There is the possibility to send the kth datum via gate 2 with sequence number $(\ell{+}k)|_{2n}$ and an acknowledgement can arrive via gate 6. This acknowledgment is the index of the first message that has not yet been received by the receiver.

proc $S(\ell, m{:}\mathbb{N}, q{:}DBuf, n{:}\mathbb{N}^+)$
$= \sum_{d:D} m < n \to r_1(d)\cdot S(\ell, m{+}1, q[((\ell{+}m)|_n){:=}d], n)$
$+ \sum_{k:\mathbb{N}} k < m \to s_2(q((\ell{+}k)|_n), (\ell{+}k)|_{2n})\cdot S(\ell, m, q, n)$
$+ \sum_{k:\mathbb{N}} r_6(k)\cdot S(k, (m{-}k{+}\ell)|_{2n}, q, n);$

The receiver R is modelled by the process $R(\ell', q', b, n)$ where ℓ' is the sequence number of the oldest message in the receiving buffer q'. Data can be received via gate 3 from channel K and is only put in the receiving buffer q' if its sequence number k is in the receiving window. If sequence numbers and buffer positions would not be considered modulo $2n$ and n this could be stated by $\ell' \le k < \ell' + n$. The condition $(k-\ell')|_{2n} < n$ states exactly this, taking the modulo boundaries into account.

The second summand in the receiver says that if the oldest message position is valid (i.e., $b(\ell'|_n)$ holds), then this message can be delivered via gate 4. Moreover, the oldest message is now $(\ell'{+}1)|_{2n}$ and the message at position $\ell'|_n$ becomes invalid.

The last summand says that the index of the first message that has not been received at the receiver is sent back to the sender as an acknowledgement that all lower numbered message have been received.

proc $R(\ell'{:}\mathbb{N}, q'{:}DBuf, b{:}BBuf, n{:}\mathbb{N}^+)$
$= \sum_{d:D,k:\mathbb{N}} r_3(d,k)\cdot((k{-}\ell')|_{2n} < n$
$\to R(\ell', q'[(k|_n) := d], b[(k|_n) := true], n)$
$\diamond R(\ell', q', b, n))$
$+ b(\ell'|_n) \to s_4(q'(\ell'|_n))\cdot R((\ell'{+}1)|_{2n}, q', b[(\ell'|_n) := false], n)$
$+ s_5(nextempty_{mod}(\ell', b, n, n))\cdot R(\ell', q', b, n);$

The behaviour of the SWP is characterized by:

proc $SWP(q, q'{:}DBuf, n{:}\mathbb{N}^+) = \nabla_H(\Gamma_G(S(0,0,q,n) \parallel K \parallel L \parallel R(0,q',empty,n)))$;

where the set $H = \{c_2, c_3, c_5, c_6, i, r_1, s_4\}$ and $G = \{r_2|s_2{\rightarrow}c_2, r_3|s_3{\rightarrow}c_3, r_5|s_5{\rightarrow}c_5, r_6|s_6{\rightarrow}c_6\}$. The contents of q and q' can be chosen arbitrarily without affecting the correctness of the protocol. This is stressed by not instantiating these variables. The sliding window protocol behaves as a bounded first-in-first-out buffer for any $n{:}\mathbb{N}^+$ and $q, q'{:}DBuf$:

$$\tau.FIFO([], 2n) = \tau.\tau_I(SWP(q, q', n))$$

where $I = \{c_2, c_3, c_4, c_5, i\}$. In section 12.2 it is proven that this equivalence holds. Due to the tricky nature of modulo calculation, this proof is quite involved.

Exercise 6.2.1. Suppose the buffer size is two and messages are sent modulo 4 (see the file `swp.mcrl2` in the directory `examples/academic`). Give an execution trace of $SWP(q, q', n)$ where a single datum is read at gate 1 and delivered at 4. Verify that this trace is indeed a correct execution trace using the simulator of mCRL2.

Now make a copy of the file with the sliding window protocol and adapt it such that messages are sent modulo 2. Find a trace where a datum d_1 is sent first, and d_2 is received first. Note that it is necessary to read more than one datum to achieve the above effect.

In the *two-way* SWP [9, 11], not only the Sender reads data elements from gate 1 and passes them on to the Receiver, but also the Receiver reads data elements from channel 4 and passes them on to the Sender (see figure 6.3). In the two-way SWP, the Sender has two buffers, one to store incoming data elements from channel A, and one to store incoming data elements from channel F; likewise for the Receiver. Note that in the two-way SWP, the Sender and the Receiver are symmetric identities, and likewise for the mediums K and L.

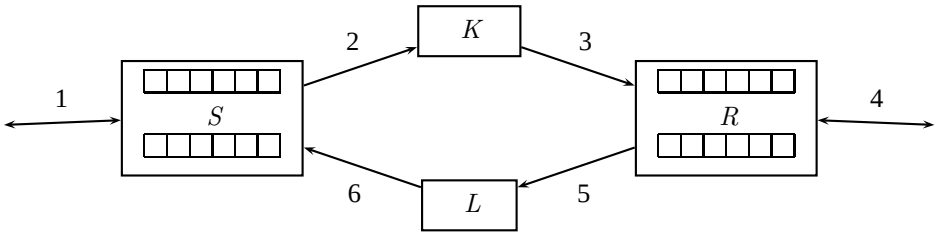


Figure 6.3: Two-way SWP

Exercise 6.2.2. Give a specification of the two-way SWP. Note that due to symmetry, the sender and the receiver and also the channels K and L are equal, except for a renaming of the input and output gates. So, you can use renaming to extract the Receiver and L from the process declarations of the Sender and K, respectively.

In the two-way SWP, acknowledgements that are sent from the Sender to the Receiver, and vice versa, can get a free ride by attaching them to data packets. This technique, which is commonly known as *piggybacking* (as the acknowledgement gets a free ride on the back of a 'pig'), helps to achieve a better use of available channel bandwidth.

Exercise 6.2.3. Give a specification of the Sender in the two-way SWP with piggybacking.

Piggybacking as described in exercise 6.2.3 slows down the two-way SWP, since an acknowledgement may have to wait for a long time before it can be attached to a data packet. Therefore, the Sender and the Receiver ought to be supplied with a timer, which sends a time-out message if an acknowledgement must be sent out without further delay; see [47] for more details.

Exercise 6.2.4. Model the two-way SWP with piggybacking supplied with timers. Note that there are two ways to do this. The first one is by using explicit time. But in this case analysis of the protocol might be harder. It is also possible to model time-outs as actions that can take place at any time after the timer has been set. This avoids the use of explicit time and makes the model much easier to analyse.

6.3 A patient support platform

We show how to develop a controller for a typical embedded system. The system given here is much simpler than its original, or more in general any embedded system, but it will still give a flavour of how such a system can be developed.

The system is a movable patient support unit that is used to move patients into magnetic-resonance (MR) scanners. These scanners are used to look inside a patient using magnetic radiation. The patient is put on the trolley, the trolley is docked into the MR-scanner, and using a motor on the trolley the patient is moved into the scanner.

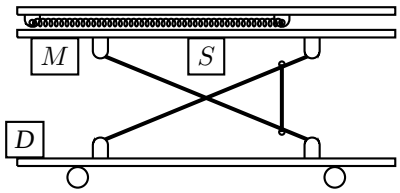


Figure 6.4: A movable patient support unit.

The patient support unit is depicted in figure 6.4. The motor M has a brake which is applied when the patient support is undocked, or when in normal mode, is not moving.

There is a sensor S that senses the position of the movable bed. It can sense whether the bed is at the left, at the right or somewhere in between.

The bed contains a docking station D which can sense whether the bed is docked onto the MR scanner. Furthermore, it has a mechanical lock which applies automatically, when the bed is docked. If the unit must be undocked, the docking unit must first get a signal to unlock.

The bed is controlled using a panel, which is depicted in figure 6.5. This interface has 5 buttons. The stop button brings the patient support unit in emergency mode. In this mode no movement of the bed is allowed anymore. In case the bed is docked, the brakes must be released to allow to move the patient out of the scanner manually. In case the unit is undocked the brakes must be applied. The resume button is used to bring the bed back in normal mode.

The buttons marked with the arrows, are used to move the bed to the left and to the right. They have only effect if the bed is docked and can still move into the indicated direction.

The undock button is used to unlock the docking unit. This can only be done when the bed is moved completely on top of the support unit. In order to allow a patient to be removed in case of an emergency, the undock button can be applied, even if the platform is in emergency mode. Also in this case the bed must be on top of the unit. When the lock of the docking unit is released, the brakes must be applied.

As indicated above, actual docking units are much more complex. They can also move up and down, and often have means of moving sideways. Generally, such platforms can be remotely controlled, either from the scanner or from an operating room. Besides normal and emergency modes, such platforms have other modes, such as manual mode where the bed can be moved manually, without the need to press the

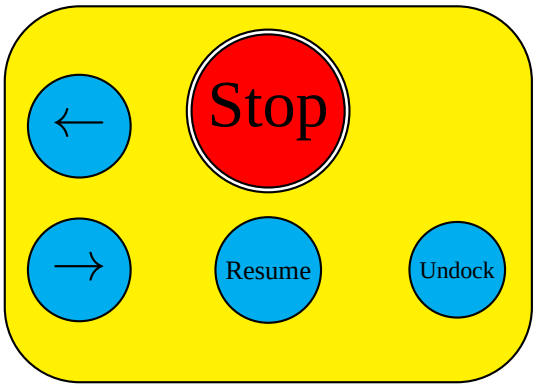


Figure 6.5: The user console on the patient support unit.

emergency buttons. There are also means for logging and remote maintenance. For the exposition here, we keep our setup relatively simple. We also ignore the possibilities of errors, such as non-functioning sensors and broken motors.

The first question to address is what we must model. In this case we must develop a controller for the patient support platform. So, it makes sense to identify the actions by which the controller communicates with the outside world. These actions are listed in table 6.1. Note that there are fundamentally two ways of communication. The controller can poll the devices continuously, or the devices can inform the controller on an interrupt basis. For the design here, we choose the latter. Before commencing to detail the behaviour

pressStop	The stop button is pressed
pressResume	The resume button is pressed
pressUndock	The undock button is pressed
pressLeft	The button marked with a left arrow is pressed
pressRight	The button with a right arrow is pressed
motorLeft	The motor is switched on and makes an inward movement
motorRight	The motor is switched on and makes an outward movement
motorOff	The motor is switched off
applyBrake	Apply the brake of the motor
releaseBrake	Release the brake of the motor
isDocked	Indicates that the platform has been docked
unlockDock	Unlock the lock in the docking unit, to enable undocking
atInnermost	An indication that the bed is completely inside the bore of the MR scanner
atOutermost	Indicates that the bed is completely on top of the patient support unit

Table 6.1: The interactions of the MPSU controller

of the controller it is a good habit to first list all behavioural requirements or properties on the behaviour. Alternatively, the external behaviour could be made explicit as done with the alternating bit protocol (section 6.1) and the sliding window protocol (section 6.2). There are three reasons not to do this. The first one is that the behaviour of such embedded systems is often quite complex, which makes it very hard to give a neat description of the external behaviour. The second one is that such controllers are not really used as part of a more complex behavioural description. The last one is that the actual external behaviour is not of real interest, as long as a number of essential properties are satisfied.

Finding the list of properties is generally one of the more difficult tasks of behavioural modelling. The reason is that properties must be formulated on the basis of general understanding of the problem domain. There is no way to check whether the properties cover all relevant aspects, and are correct. It is very common that properties must be rephrased in a later phase of the project because they turn out to be

incorrect.

It is helpful to formulate the behavioural requirements by iteratively considering the following groups of properties.

- Safety properties. What must absolutely not happen! It is useful to first formulate this for the whole system, and later translate this into concrete requirements on the interactions of the controller to be designed.
- Liveness properties. Here the basic question is what the purpose of an interaction from the outside world with the system is. E.g. if the *pressStop* button is pressed, the system must go to emergency mode.

As a rule of thumb, each interaction should at least occur once in a property. This is a practical way to check the completeness of the requirements.

The requirement must ultimately be stated in terms of the interaction of the system to be designed. The way to achieve this is to state the properties at increased levels of detail. Below, we state the requirements in three stages, indicated with **a**, **b** and **c**. At **a**, the property is stated in ordinary English referring to intuitive notions such as emergency mode and 'being undocked'. At **b** the properties are reformulated in terms of interactions only. Instead of speaking about emergency mode, we must now speak about 'having pressed the stop button, without having pressed the release button yet'. At **c** we translate these properties to a modal formula. It is good habit to at least describe the properties informally (as done under **a**) and describe them at least once precisely as under **b** or **c**. The informal description is needed for quick human understanding, whereas the other is required as the first description is often ambiguous.

We have the following safety requirements for the patient platform. We intensionally provide the list of requirements that sprang to our mind at the initial design of the platform. We also provide a model of the controller as we initially thought the controller should behave. The purpose of not giving a definitive list of requirements is to show how deceptive initial ideas about system behaviour are.

For each we list the **a**, **b** and **c** formulations grouped together, but for first reading, it is best to skip **b** and **c**.

1**a**. For the patient platform we want that the platform can not tumble over. Concretely this means that when undocked, the bed is in the rightmost position, the brakes are applied and the motor is off.

1**b**. Before sending an *unlockDock* action, an *applyBrake*, an *atOutermost* and a *motorOff* action must have taken place. Moreover, between an *unlockDock* and an *applyBrake* action, no *releaseBrake* action is allowed. Similarly, between an *unlockDock* and a *motorOff* action, no *motorLeft* or *motorRight* action is allowed. Finally, between an *atOutermost* and the *unlockDock* actions no *motorLeft* should take place.

1**c**. We can formulate this requirement as six separate modal formulas. As smaller formulas are easier to check, it is always wise to make the formulas as compact as possible.

- $[\overline{applyBrake}^{\star} \cdot unlockDock]false$.
- $[\overline{atOutermost}^{\star} \cdot unlockDock]false$.
- $[\overline{motorOff}^{\star} \cdot unlockDock]false$.
- $[true^{\star} \cdot releaseBrake \cdot \overline{applyBrake}^{\star} \cdot unlockDock]false$.
- $[true^{\star} \cdot motorLeft \cdot \overline{atOutermost}^{\star} \cdot unlockDock]false$.
- $[true^{\star} \cdot (motorLeft + motorRight) \cdot \overline{motorOff}^{\star} \cdot unlockDock]false$.

2**a**. By pressing the stop button, a patient on the platform can manually be moved out of the scanner and scanning room.

2**b**. After a *pressStop* event takes place, a *motorOff*, a *releaseBrake* and an *unlockDock* action must inevitably follow.

2**c**. We can write this requirement as three compact modal formulas.

- $[true^\star \cdot pressStop]\mu X.[\overline{motorOff}]X.$
- $[true^\star \cdot pressStop]\mu X.[\overline{releaseBrake}]X.$
- $[true^\star \cdot pressStop]\mu X.[\overline{unlockDock}]X.$

3**a**. In order to protect the motor, the motor will not attempt to push the bed beyond its outermost and innermost positions.

3**b**. If an *atOutermost* or an *atInnermost* action takes place, a *motorOff* event will follow.

3**c**. $[true^\star(atOutermost + atInnermost)]\mu X.[\overline{motorOff}]X.$

Interaction properties are the following:

4**a**. If *pressUndock* takes place, and the patient platform is in rightmost position, an *unlockDock* event takes place.

4**b**. After a *pressUndock*, if there has been a preceding *atInnermost*, and between those no *motorLeft* and *releaseBrake* has taken place, then an *unlockDock* will take place, except if there is a prior *motorLeft* or *releaseBrake*.

4**c**. $[true^\star \cdot atInnermost \cdot \overline{motorLeft \cup releaseBrake}^\star \cdot pressUndock]$
$\mu X.[\overline{unlockDock} \cap \overline{motorLeft} \cap \overline{releaseBrake}]X.$

5**a**. If *pressLeft* takes place, the platform is docked, not in emergency mode, the bed is not completely inside the bore and not moving already to the left, the motor is switched on to move the bed inside the bore.

5**b**. If a *pressLeft* takes place, with a preceding *isDocked* without an intermediary *unlockDock*, no preceding *pressStop* without an intermediary *pressResume* took place, no *atInnermost* took place without intermediary *motorRight*, and no *motorLeft* happened without any *motorOff* or *motorRight* in between, then a *motorLeft* will take place, unless a *unlockDock*, a *pressStop* or a *atInnermost* takes place before that.

5**c**. In the booleans b_1, b_2, b_3 and b_4 we record that the four conditions that must hold before the *pressLeft* to lead to a *motorLeft* are favourable. Then the formula becomes:

$$\begin{array}{l}\nu X(b_1{:}\mathbb{B}{:=}\mathit{false}, b_2{:}\mathbb{B}{:=}\mathit{true}, b_3{:}\mathbb{B}{:=}\mathit{true}, b_4{:}\mathbb{B}{:=}\mathit{true}).\\ \quad [\mathit{isDocked}]X(\mathit{true}, b_2, b_3, b_4) \wedge [\mathit{unlockDock}]X(\mathit{false}, b_2, b_3, b_4) \wedge\\ \quad [\mathit{pressResume}]X(b_1, \mathit{true}, b_3, b_4) \wedge [\mathit{pressStop}]X(b_1, \mathit{false}, b_3, b_4) \wedge\\ \quad [\mathit{atInnermost}]X(b_1, b_2, \mathit{true}, b_4) \wedge [\mathit{motorRight}]X(b_1, b_2, \mathit{false}, b_4) \wedge\\ \quad [\mathit{motorOff} \cup \mathit{motorRight}]X(b_1, b_2, b_3, \mathit{true}) \wedge [\mathit{motorLeft}]X(b_1, b_2, b_3, \mathit{false}) \wedge\\ \qquad (b_1 \wedge b_2 \wedge b_3 \wedge b_4 \Rightarrow [\mathit{pressLeft}]\\ \qquad\quad \mu Y.[\overline{\mathit{motorLeft}} \cap \overline{\mathit{unlockDock}} \cap \overline{\mathit{pressStop}} \cap \overline{\mathit{atInnermost}}]Y).\end{array}$$

6**a**. If *pressRight* happens, the platform is docked, not in emergency mode, the bed is not completely on top of the platform, not already moving to the right, the motor is switched on for an outward movement.

6**b**. This requirement is completely similar to requirement 5. We reformulate it into actions and into a modal formula along exactly the same lines. If a *pressRight* takes place, with a preceding *isDocked* without an intermediary *unlockDock*, no preceding *pressStop* without an intermediary *pressResume* took place, no *atOutermost* took place without intermediary *motorLeft*, and no *motorRight* happened without any *motorOff* or *motorLeft* in between, then a *motorRight* will take place, unless a *unlockDock*, a *pressStop* or a *atInnermost* takes place before that.

6**c**. This formula very much resembles the formula at 5**c**.

$$
\begin{aligned}
&\nu X(b_1{:}\mathbb{B}{:=}\mathit{false}, b_2{:}\mathbb{B}{:=}\mathit{true}, b_3{:}\mathbb{B}{:=}\mathit{true}, b_4{:}\mathbb{B}{:=}\mathit{true}).\\
&\quad [\mathit{isDocked}]X(\mathit{true}, b_2, b_3, b_4) \wedge [\mathit{unlockDock}]X(\mathit{false}, b_2, b_3, b_4) \wedge\\
&\quad [\mathit{pressResume}]X(b_1, \mathit{true}, b_3, b_4) \wedge [\mathit{pressStop}]X(b_1, \mathit{false}, b_3, b_4) \wedge\\
&\quad [\mathit{atOutermost}]X(b_1, b_2, \mathit{true}, b_4) \wedge [\mathit{motorLeft}]X(b_1, b_2, \mathit{false}, b_4) \wedge\\
&\quad [\mathit{motorOff} \cup \mathit{motorLeft}]X(b_1, b_2, b_3, \mathit{true}) \wedge [\mathit{motorRight}]X(b_1, b_2, b_3, \mathit{false}) \wedge\\
&\qquad (b_1 \wedge b_2 \wedge b_3 \wedge b_4 \Rightarrow [\mathit{pressRight}]\\
&\qquad\quad \mu Y.[\overline{\mathit{motorRight}} \cap \overline{\mathit{unlockDock}} \cap \overline{\mathit{pressStop}} \cap \overline{\mathit{atOutermost}}]Y).
\end{aligned}
$$

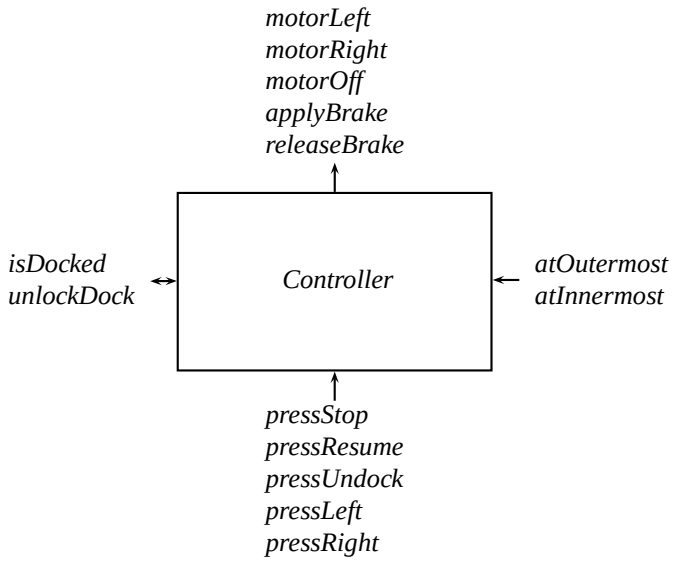


Figure 6.6: Input/output diagram of the MPSU controller

A possible behaviour of the controller is given below. The idea is that there are two primary modes, namely Normal and $\mathit{Emergency}$. In $\mathit{Emergency}$ mode the bed the brakes are released and the motors are off and the bed only responds to pressing the resume button. In *Normal* mode all functions operate normally. Within the controller it is recalled whether the unit is *docked*, and whether it is signalled to be at the *leftmost* or *rightmost* position. Also the status of the motor is recalled to *moveleft, moveright* and *stopped*. The process itself is rather straightforward and is therefore not explained.

sort $\mathit{Mode} = \mathbf{struct}\ \mathit{Normal} \mid \mathit{Emergency}$;
$\mathit{MotorStatus} = \mathbf{struct}\ \mathit{moveleft} \mid \mathit{moveright} \mid \mathit{stopped}$;

proc $Controller(m{:}Mode, docked, rightmost, leftmost{:}\mathbb{B}, ms{:}MotorStatus)$
$= pressStop{\cdot}releaseBrake{\cdot}motorOff{\cdot}$
$\quad Controller(Emergency, docked, rightmost, leftmost, stopped)$
$+ pressResume{\cdot}Controller(Normal, docked, rightmost, leftmost, ms)$
$+ pressUndock{\cdot}$
$\quad (docked \wedge rightmost)$
$\quad\quad \rightarrow applyBrake{\cdot}unlockDock{\cdot}Controller(m, false, rightmost, leftmost, ms)$
$\quad\quad \diamond Controller(m, docked, rightmost, leftmost, ms)$
$+ pressLeft{\cdot}$
$\quad (docked \wedge ms \not\approx moveleft \wedge \neg leftmost \wedge m \approx Normal)$
$\quad\quad \rightarrow releaseBrake{\cdot}motorLeft{\cdot}Controller(m, docked, false, leftmost, moveleft)$
$\quad\quad \diamond Controller(m, docked, rightmost, leftmost, ms)$
$+ pressRight{\cdot}$
$\quad (docked \wedge ms \not\approx moveright \wedge \neg rightmost \wedge m \approx Normal)$
$\quad\quad \rightarrow releaseBrake{\cdot}motorRight{\cdot}Controller(m, docked, rightmost, false, moveright)$
$\quad\quad \diamond Controller(m, docked, rightmost, leftmost, ms)$
$+ isDocked{\cdot}Controller(m, true, rightmost, leftmost, ms)$
$+ atInnermost{\cdot}motorOff{\cdot}applyBrake{\cdot}Controller(m, docked, true, false, stopped)$
$+ atOutermost{\cdot}motorOff{\cdot}applyBrake{\cdot}Controller(m, docked, false, true, stopped).$

Initially, the support unit starts in *Normal* mode, the motor is Off, and the position of the bed is not known. It is assumed that when the unit is switched on, it is always docked.

init $Controller(Normal, true, false, false, stopped);$

Controllers are generally much more complex than the one above, and often distributed over different computing units. So, in general controllers consist of a number of parallel components that all have a well defined task. Between the components communications take place to keep the components in par with each other. This parallel behaviour generally makes it much harder to understand the behaviour, than in the case above.

Exercise 6.3.1. Which of the modal formulas are valid for the given model of the patient support platform? You may want to use the tools `mcrl22lps`, `lps2pbes` and `pbes2bool` to verify the modal formulas automatically. Also the state space generation tool `lps2lts` and the visualisation tools `ltsgraph` and `ltsview` can be used to understand whether the patient support platform works properly.

Exercise 6.3.2. Are the requirements consistent in the sense that there exists a behavioural description such that all requirements are valid?

Exercise 6.3.3. In parallel programming synchronization operators can become costly. In order to avoid the problems, synchronization free programming has been invented. Unfortunately, without synchronisation operators and with only the possibility to read from and write to memory, the *consensus problem* cannot be solved. In this problem several parallel processes must agree on a common value [34].

In order to overcome this problem extra hardware primitives are available, such as *LoadLock* and *Compare&Swap*. In LoadLock a memory location is read and locked. When this memory location is written by the same process, it can see whether this location has been written by another process in the mean time. Compare&Swap is a single instruction that can be used to write a value to a memory location if this location has an expected value. More precisely:

$$CMPSWP(n, m_1, m_2) = \text{if } M[n] \approx m_1 \text{ then } M[n] := m_2; \text{ return } true \text{ else return } false$$

Here the array M represents the memory. It says that if memory location $M[n]$ equals m_1 then m_2 is written to this location and success is reported. Otherwise, nothing is written and false is returned.

Using the Compare&Swap instruction it is possible to construct a free list of nodes that can be accessed by several processes in parallel. The parallelism is on the level of instructions, so all reads, writes and Compare&Swaps can be executed in an interleaved manner.

Free lists typically have two operations, namely release and get. Release puts a node on the list, effectively making this node available to others. The get operation allows a process to claim a node for its own use. Typically, nodes may not be given to more than one process without being released in the mean time, and nodes that are released become available at some later time.

A straightforward way of doing this is by implementing the following procedures to access the list. The description is in some pseudo programming language. There is some globally accessible head of the list *hd*. Furhtermore, there is a sort **node** of objects to be stored on the list. For each **node** n there is a **node** $n.next$ indicating the next node in the list. The special node *NULL* is used as an end marker of the list. $NULL.next$ is not defined.

list $hd = NULL$;

void release(**node** n) =
 bool b;
 repeat
  $n.next := hd$;
  $b := CMPSWP(hd, n.next, n)$;
 until b;

node get(**node** n) =
 bool b; **node** n;
 repeat
  $n := hd$;
  if $n \approx NULL$ **return** $NULL$;
  $b := CMPSWP(hd, n, n.next)$;
 until b;
 return n;

This implementation is incorrect. The question is to find out why. A systematic technique is to model the memory in mCRL2, together with multiple release and get processes. Verifying the properties for a limite memory size and a limited set of processes should reveal the problem. In the literature, this problem is known as the false positive, or ABA problem. Due to this problem, several microprocessors contain the double compare and swap instruction.

Chapter 7

Semantics

In this chapter we give a detailed account of the meaning of data, processes and modal formulas. In the previous chapters we have presented their syntax and using some hand waving claimed all kind of equalities and properties about them. A rightful question is whether these properties and equalities are correct. In order to answer that, we must first explain in detail what the objects represent.

For data we explain which data domains are defined by data specifications and how data expressions relate to elements in these domains. We define how a process determines a timed transition system. Finally, given a timed transition system, we explain in which states a modal formula is valid.

7.1 Semantics of the datatypes

As the data expressions as we are using it, are quite complex, we provide the semantics in an abstract way. We first define sorts, signatures, expressions, substitutions, data specifications and finally we provide the semantics of data types. While doing so, we give restrictions on all these to guarantee that they are well-typed. Furthermore, we indicate how the concrete data types of mCRL2 fit into the given frame.

7.1.1 Sorts

We assume that we have a given set of basic sorts BS with a sort inclusion relation. Typically, basic sorts are booleans, natural numbers, bits, etc. The sort inclusion relation allows data expressions of a certain sort to change their type to a more general sort as indicated by it. Furthermore, there is a set $\mathcal{E}$ of sort constructors, containing functions that given some sorts, can be used to construct a new sort. A nice example of a sort constructor is a list, that turns any sort D into a list of that sort.

In the next definition we extend the basic sorts and the sort constructors to form a set of sorts that include functions. After this general definition it is made precise what the basic sorts, the sort constructors and the sort inclusion relation for mCRL2 are. To keep definitions concise and general, we use the set of sorts $\mathcal{S}$ and the sort inclusion relation $\subseteq$ in the remainder of this chapter, without subscripts, and hardly refer to the concrete basic sorts and sort constructors that are used in mCRL2.

Definition 7.1.1 (Sorts and sort inclusion). Let BS be a set of basic sorts with a sort inclusion relation $\subseteq$ which is a preorder. Let $\mathcal{E}$ be a set of sort constructors. The *set of sorts* over BS and $\mathcal{E}$, notation $\mathcal{S}_{BS,\mathcal{E}}$ is the minimal set satisfying the following rules:

- $BS \subseteq \mathcal{S}_{BS,\mathcal{E}}$. For every $D, D' \in BS$, $D \subseteq_{BS,\mathcal{E}} D'$ if $D \subseteq D'$.
- if $C \in \mathcal{E}$ is an n-ary sort constructor and $D_1, \ldots, D_n \in \mathcal{S}_{BS,\mathcal{E}}$ then $C(D_1, \ldots, D_n) \in \mathcal{S}_{BS,\mathcal{E}}$.
- if $D_1, \ldots, D_n, D \in \mathcal{S}_{BS,\mathcal{E}}$, then $D_1 \times \cdots \times D_n \to D \in \mathcal{S}_{BS,\mathcal{E}}$. This sort is called the *function sort* from $D_1, \ldots, D_n$ to D.

Sort inclusion on sorts is the reflexive relation extended with $\subseteq$ on basic sorts.

In mCRL2 the set of basic sorts contain the primary sorts $\mathbb{B}$, $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. The sort inclusion relation is reflexive, e.g., it contains $\mathbb{B} \subseteq \mathbb{B}$, $\mathbb{N}^+ \subseteq \mathbb{N}^+$, etc. Furthermore, it contains only

$$\mathbb{N}^+ \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}.$$

The primary motivation for sort inclusion is that it allows to conveniently write down numbers, without having to write explicit conversions. E.g., if a real number 1 is required we can write the positive number 1. Using sort inclusion this positive number can also become a data expression of sort real. With explicit conversion we had to write $\mathbb{N}^+2\mathbb{R}(1)$ or `Pos2Real(1)` which is awkward. As booleans are written differently than numbers, there is no sort inclusion from the booleans to the numbers.

Any sortname that is declared in a specification, e.g.,

sort D;

is a also a basic sort. Each sort can only be declared once in a specification. The sort inclusion relation is extended by $D \subseteq D$.

The basic sort constructors are $List$, Set and Bag that all require one argument. So, $List(\mathbb{N})$, $Set(\mathbb{N} \to \mathbb{N})$ and $Bag(Set(\mathbb{N} \to \mathbb{N}))$ are sorts.

Furthermore, any definition of a structured datatype is a sort constructor. Consider,

sort $Tree = \mathbf{struct}\ leaf(\mathbb{N}) \mid node(Tree, Tree)$;

The sort $Tree$ is constructed based on the sort $\mathbb{N}$. Typically, it is also possible to define

sort $Tree_f = \mathbf{struct}\ leaf(\mathbb{N} \to \mathbb{N}) \mid node(Tree_f, Tree_f)$;

for a sort containing trees with functions. The rules for sort inclusions carry over to structured sorts as indicated above.

At times it is useful to introduce abbreviations for sorts. We call these *sort aliases* or just *aliases* for short. For instance the declaration

sort $L = List(D)$;

is made, then L is an alias for $List(D)$. We consider aliases as different denotations for the same basic sort. Sort aliases can always be exchanged. More precisely, we consider two sorts equal if they can be rewritten by exchanging aliases, or by applying the definition of a recursive type. For instance the basic sort $Tree$ and $\mathbf{struct}\ leaf(\mathbb{N}) \mid node(Tree, Tree)$ are equal and can be used interchangeably. After the definition of

sort $Tree' = \mathbf{struct}\ leaf(\mathbb{N}) \mid node(Tree, Tree)$;

the sorts $Tree$ and $Tree'$ are aliases. But after the definition of

sort $altTree = \mathbf{struct}\ leaf(\mathbb{N}) \mid node(altTree, altTree)$;

the sorts $Tree$ and $altTree$ are not equal. This is because they cannot be shown equal by only folding and unfolding definitions of sort aliases and structured sorts.

7.1.2 Signatures

In the data types for mCRL2 a number of typed function symbols of the form $f{:}D_1 \times \cdots \times D_n \to D$ are declared. There are essentially two kinds of functions symbols, namely constructors and mappings. Constructors are those functions that span up a sort. If there are constructors in a sort any element of the sort can be denoted using constructors. Mappings represent the 'other' functions. The combination of sorts, constructors and mappings is called a signature.

Definition 7.1.2 (Signature). Let $\mathcal{S}$ be a set of sorts, $\mathcal{C}_\mathcal{S}$ a set of function symbols over $\mathcal{S}$, called *constructors* and $\mathcal{M}_\mathcal{S}$ a set of function symbols over $\mathcal{S}$ called *mappings*. We call the triple $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ a *signature*.

Besides the mappings and constructors that are explicitly declared in an mCRL2 specification, standard constructors and mappings are defined. These can be found in appendix A.

An important concept in a data specification is a constructor sort. This is a basic sort such that there is a constructor with this sort as target sort.

Definition 7.1.3 (Constructor sort). Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ be a data signature. A sort D in $\mathcal{S}_\mathcal{S}$ is called a *constructor sort* iff there is a constructor $f{:}D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}$.

In constructor sorts all elements can be written by an application of a constructor function. In the following example there are three constant constructors:

sort $Enum$;
cons $e_1, e_2, e_3 : Enum$;

The elements of the sort $Enum$ can be denoted by e_1, e_2 and e_3. But note that we have no evidence that these elements are different. The semantics of the sort $Enum$ can be any set with one to three elements.

When using constructor functions sorts such as natural numbers can be denoted as shown in section 4.2.1. This use is typical when denoting domains with a countable elements.

It is also possible to use constructors that map elements of a non-constructor domain into a constructor domain. Then a constructor domain D can be constructed using a constructor function i_1 that embeds the real numbers. Similarly, a constructor domain E containing all functions from natural numbers to natural numbers can easily be defined. Note that the D and E represent uncountable data types.

sort D, E;
cons $i_1 : \mathbb{R} \to D$;
$i_2 : (\mathbb{N} \to \mathbb{N}) \to E$;

As said above, the data types in mCRL2 can completely be translated to a signature (following appendix A). The signature that is obtained must be well-typed, meaning that it must satisfy the following properties.

Definition 7.1.4 (Well-typed signature). We say that a data signature $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ is *well-typed* iff

- $\mathcal{C}_\mathcal{S} \cap \mathcal{M}_\mathcal{S} = \emptyset$.
- $\mathbb{B}$ is a basic sort, and $true{:}\mathbb{B}$ and $false{:}\mathbb{B}$ are exactly the constructors of sort $\mathbb{B}$.
- Function sorts are not constructor sorts: if $f{:}D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}$ then D is not a function sort.
- Constructor sorts are *syntactically non empty*. This is inductively defined as follows. A sort D is called *syntactically non empty* iff there is a constructor $f{:}D_1 \times \cdots \times D_n \to D \in \mathcal{C}_\mathcal{S}$ ($n \geq 0$) such that for all $1 \leq i \leq n$ if D_i is a constructor sort, it must also be syntactically non empty.

Specific for mCRL2, we require that for sorts declared in appendix A there are no additional constructors than those occurring in appendix A. This applies to $\mathbb{B}$, $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$, as well a all sorts constructed by structured sorts, lists, sets and bags.

Note that *syntactically non empty* exactly implies that a constructor domain is not empty. This is done to avoid specifications of the form:

sort D;
cons $f : D \to D$;

In this case data expressions constructed with f are necessarily infinite: $f(f(f(\cdots)))$. As such expressions do not exist, there are no expressions to represent elements in D. But all elements in D should be written using an expression starting with a constructor symbol. Consequently, the sort D is empty, which is inconsistent with the requirements in section 7.1.6 below. To circumvent this situation, we introduce that constructor sorts must be *syntactically non empty* which exactly excludes all these undesired situations.

The requirement that for mCRL2 no other constructors than those declared in appendix A can be used is needed to avoid that (by accident) the structure of the built-in data-types are changed. If it would be allowed to declare:

cons $one : \mathbb{R}$;

then all real numbers would be reduced to one. Similarly, one can add non-standard elements to the natural numbers, or alter the structure of lists or structured data types. This generally is undesired and very confusing, and therefore forbidden.

7.1.3 Well-typed data expressions

In this section we formally introduce data expressions – although we have already used them widely –, and we say when we call an expression well-typed, i.e., allowable in a specification. We are quite liberal, in the sense that we call an expression well-typed if a way can be found to type it in a unique way.

In data expressions we use variables. Therefore we pose the existence is a set of $\mathcal{S}$-typed variable symbols $\mathcal{X}_\mathcal{S}$. In $\mathcal{X}_\mathcal{S}$ there are typed variables of of the form $x{:}D$ similar to function symbols. In this case we say x has sort D. In an mCRL2 specification, the set of variable symbols is constantly changing, because variables are locally declared. We take as a ground rule that at any time a variable occurs only once with a unique type in a set of variables. Typically, it is not possible that $\mathcal{X}_\mathcal{S}$ contains both $x{:}\mathbb{N}$ and $x{:}\mathbb{Z}$ at the same time. Of course this does not exclude that in one section the variable x is used of one type and at another place x is used with another type. It is also not allowed that a variable takes the same name as a mapping or a constructor, independent of its type.

Data expressions essentially consists of applications of function symbols to constants and variables. As we use function types, we also allow *lambda abstraction* and *application*. Lambda abstraction turns a data expression into a function. So, if $f(x)$ is a data expression with x a variable of sort D, $\lambda x{:}D.f(x)$ is a function that can be applied to an expression t (of sort D), yielding the result $f(t)$, namely $f(x)$ with t put in place of the variable x. Note that in order to avoid problems with variables, x can only be used with sort D if it also had sort D in the context. This is to avoid very confusing data expressions such as $\lambda b{:}\mathbb{B}.\lambda b{:}\mathbb{N}.\neg b$ where due to typing the commonly accepted rules of lambda bindings do not apply anymore.

Furthermore, we allow universal and existential quantification, because these are very useful for compact specification. The same holds for the **whr** construct which allows to define an abbreviation for a subexpression. This can be convenient when the subexpression is used more often in a data expression.

Definition 7.1.5 (Data expressions). Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ a signature and $\mathcal{X}_\mathcal{S}$ a set of $\mathcal{S}$-typed variable symbols. We inductively define typed data expressions (over $\mathcal{X}_\mathcal{S}$) as follows:

- every variable symbol $x{:}D \in \mathcal{X}_\mathcal{S}$ of sort D is a data expression of sort D.
- every function symbol $f{:}D \in \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$ of sort D is a data expression of sort D.
- If p is a data expression of sort $D_1 \times \cdots \times D_n \to D$ and p_i are data expressions of sort $E_i \subseteq D_i$ for $1 \leq i \leq n$, then $p(p_1, \ldots, p_n)$ is a data expression of sort D.
- If $x_i{:}D_i \in \mathcal{X}_\mathcal{S}$ or x_i does not occur in $\mathcal{X}_\mathcal{S} \cup \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$, and p is a data expression of sort D over $\mathcal{X}_\mathcal{S} \cup \{x_i{:}D_i | 1{\leq}i{\leq}n\}$, then $\lambda x_1{:}D_1, \ldots, x_n{:}D_n.p$ is a data expression of sort $D_1 \times \cdots \times D_n \to D$.
- If $x{:}D \in \mathcal{X}_\mathcal{S}$ or x does not occur in $\mathcal{X}_\mathcal{S} \cup \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$, and p is a data expression of sort $\mathbb{B}$ over $\mathcal{X}_\mathcal{S} \cup \{x{:}D\}$, then $\forall x{:}D.p$ and $\exists x{:}D.p$ are data epxressions of sort $\mathbb{B}$.
- If p_i are data expressions of sorts D_i ($1{\leq}i{\leq}n$), and p is a data expression of sort D over $\mathcal{X}_\mathcal{S} \cup \{x_i{:}D_i | 1{\leq}i{\leq}n\}$ and $x_i{:}D_i \in \mathcal{X}_\mathcal{S}$ or x_i does not occur in $\mathcal{X}_\mathcal{S} \cup \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$, then p $\mathbf{whr}$ $x_1 = p_1, \ldots, x_n = p_n$ $\mathbf{end}$ is a data expression of sort D.

Althought the definition above suggests otherwise, we do not strictly stick to prefix notation, but use infix operators freely for some operators of the predefined datatypes. For example, we write $n + m$ as the addition for the natural numbers n and m. This requires brackets to disambiguate the parsing of some data expressions. To avoid brackets, we use the precedence conventions in table 4.10.

We also avoid repeated occurrences of quantifiers. So, we write $\forall x_1{:}D_1, x_2{:}D_2, \ldots x_n{:}D_n.p$ for $\forall x_1{:}D_1.\forall x_2{:}D_2. \ldots \forall x_n{:}D_n.p$. Similarly for exists and lambda abstraction.

Definition 7.1.6 (Well-typed data expression). Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ a signature. We inductively define whether a data expression p of sort D is well-typed data expression over a set $\mathcal{S}_\mathcal{S}$ of variable symbols as follows:

- every variable symbol $x{:}D \in \mathcal{X}_\mathcal{S}$ of sort D is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S}$.
- every function symbol $f{:}D \in \mathcal{C}_\mathcal{S} \cup \mathcal{M}_\mathcal{S}$ of sort D is a well typed data expression of sort D over $\mathcal{X}_\mathcal{S}$.
- The data expression $p(p_1, \ldots, p_n)$ is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S}$ if there are sorts D_i and $E_i \subseteq D_i$ for $1 \leq i \leq n$ such that
  - p is a well-typed data expression of sort $D_1 \times \cdots \times D_n \to D$ over $\mathcal{X}_\mathcal{S}$ and p_i are well-typed data expressions of sort E_i over $\mathcal{X}_\mathcal{S}$, and
  - if there are sorts D'_i and E'_i ($1 \leq i \leq n$) such that p is a well-typed data expression of sort $D'_1 \times \cdots \times D'_n \to D$ over $\mathcal{X}_\mathcal{S}$ and p_i are well-typed data expressions of sort $E'_i \subseteq D'_i$ over $\mathcal{X}_\mathcal{S}$ then for all $1 \leq j \leq n$ it holds that $D_j \subseteq D'_j$ and $E_j \subseteq E'_j$.

  We call the sort E_i the lower well-typed sort of arguments p_i and D_i the upper well-typed sort of p_i in $p(p_1, \ldots, p_n)$.
- If $x_i{:}D_i \in \mathcal{X}_\mathcal{S}$ are variables of sort D_i and if p is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$, then $\lambda x_1{:}D_1, \ldots, x_n{:}D_n.p$ is a well-typed data expression of sort $D_1 \times \cdots \times D_n \to D$ over $\mathcal{X}_\mathcal{S}$.
- If $x{:}D \in \mathcal{X}_\mathcal{S}$ is a variable and p is a well-typed data expression of sort $\mathbb{B}$ over $\mathcal{X}_\mathcal{S} \cup \{x{:}D\}$, then $\forall x{:}D.p$ and $\exists x{:}D.p$ are well-typed data expressions of sort $\mathbb{B}$ over $\mathcal{X}_\mathcal{S}$.
- The data expression p **whr** $x_1 = p_1, \ldots, x_n = p_n$ **end** is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S}$ if there are sorts D_i for $1 \leq i \leq n$ such that
  - if p is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$ and p_i are well-typed data expressions of sorts D_i over $\mathcal{X}_\mathcal{S}$, and
  - if there are sorts D'_i ($1 \leq i \leq n$) such that p is a well-typed data expression of sort D over $\mathcal{X}_\mathcal{S} \cup \{x_i{:}D'_i | 1 \leq i \leq n\}$ and p_i are well-typed data expressions of sort D'_i over $\mathcal{X}_\mathcal{S}$, it holds that $D_i \neq D'_i$ implies $D_i \subseteq D'_i$.

  We call the sorts D_i the well-typed sorts of variables x_i in p.

We say that a data expression p is well-typed over a set of variables $\mathcal{X}_\mathcal{S}$ iff there is a sort D such that p is well-typed of sort D over $\mathcal{X}_\mathcal{S}$.

Example 7.1.7. Consider the data expressions

$$f(1,1), \quad f(x,x) \textbf{ whr } x{=}1 \textbf{ end} \quad \text{and} \quad g(y,y) \textbf{ whr } y = [1] \textbf{ end}$$

where $f : \mathbb{N} \times \mathbb{R} \to \mathbb{N}$ and $g : List(\mathbb{N}) \times List(\mathbb{R}) \to \mathbb{N}$. In the first data expression 1 has sort $\mathbb{N}^+$. Using sort inclusion it converts to $\mathbb{N}$ and $\mathbb{R}$ and the data expression is well-typed. In the second example, the variable x can have sorts $\mathbb{N}^+$, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. The minimal type that makes the data expression well-typed is $\mathbb{N}^+$. Within $f(x,x)$ the first x is converted from $\mathbb{N}^+$ to $\mathbb{N}$ and the second from $\mathbb{N}^+$ to $\mathbb{R}$. In the third data expression y can have types $List(\mathbb{N}^+)$, $List(\mathbb{N})$, $List(\mathbb{Z})$ and $List(\mathbb{R})$. In order for g to be well-typed, y must both have sort $List(\mathbb{N})$ and $List(\mathbb{R})$. As sort inclusion does not apply to $List$, the third data expression is not well-typed.

Note that with the definition of well-typed data expressions of a particular sort, we can always uniquely determine the the sorts of the subexpressions. In the sequel we assume that all expressions are well-typed and we know the sort of each variable and subexpression. If the lower well-typed sorts and the upper well-typed sorts differ, we can insert sort transformation operators such as $\mathbb{N}2\mathbb{N}^+$ such that the lower and upper well-typed sorts are the same. In this case we say the data expression is *strongly typed*.

Example 7.1.8. Consider the following two well-typed data expressions where x is a variable of sort $\mathbb{R}$.

$$x + (3 + 0), \quad \{x, y, 5\} \textbf{ whr } y = 17 \textbf{ end}.$$

Written with explicit type information and sort transformation function, these data expressions become strongly typed as follows:

$$x_{\mathbb{R}} + \mathbb{N}2\mathbb{R}(\mathbb{N}^+2\mathbb{N}(3) + 0), \quad \{x_{\mathbb{R}}, \mathbb{N}^+2\mathbb{R}(y_{\mathbb{N}^+}), \mathbb{N}^+2\mathbb{R}(5)\} \textbf{ whr } y_{\mathbb{N}^+} = 17 \textbf{ end}.$$

Exercise 7.1.9. Determine whether the following data expressions are well-typed, and if so, determine the possible sorts that the expression has.

1. $0 + 1$.
2. $\{0, 1, 2/3\}$.
3. $[]$ (the empty list).
4. $\forall l_1{:}List(\mathbb{N}^+).\exists l_2{:}List(\mathbb{N}).l_1 \approx l_2$.

7.1.4 Free variables and substitutions

The lambda operator, both quantifiers and the **whr** clause are called binders of variables. For instance, the data expression $\lambda x{:}D.p(x)$ is said to bind the variable x in $p(x)$. It is said to be *bound* in $\lambda x{:}D.p(x)$. If a variable occurs in a data expression, and it is not bound it is said to occur *freely* (or to be free). So, typically, x is free and y is bound in $\lambda y{:}\mathbb{N}^+.x{+}y$. The following definition gives a precise account when we call a variable free in a data expression.

Definition 7.1.10. Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ a signature and let $\mathcal{X}_\mathcal{S}$ a set of variable symbols. We inductively define when a variable $x{:}E$ is free in a well-typed data expression p of sort D over $\mathcal{X}_\mathcal{S}$ as follows:

- The variable $x{:}E$ is free in a data expression y iff x is not equal to y or the sort of y is not equal to E.
- The variable $x{:}E$ is free in $p(p_1, \ldots, p_n)$, iff it is free in p or it is free in some p_i ($1{\leq}i{\leq}n$).
- The variable $x{:}E$ is free in $\lambda y_1{:}D'_1, \ldots, y_n{:}D_n.p$ iff x is not equal to y_i or $E \neq D'_i$ for $1{\leq}i{\leq}n$, and $x{:}E$ is free in p.
- The variable $x{:}E$ is free in $\forall y{:}D'.p$ and $\exists y{:}D'.p$ iff x is not equal to y or $E \neq D'$ and $x{:}E$ is free in p.
- The variable $x{:}E$ is free in p **whr** $y_1{=}p_1, \ldots, y_n{=}p_n$ **end** iff $x{:}E$ is free in p_i, or $x{:}E$ is free in p and x is not equal to y_i or $E \neq D_i$.

If $x{:}E$ occurs in the form $\lambda x{:}E.\ldots$, $\forall x{:}E.\ldots$, $\exists x{:}E.\ldots$ or $\ldots$ **whr** $\ldots x{=}p \ldots$ **end**, p, we say that $x{:}E$ is bound in p.

Note that if a variable occurs in an expression, but is not free, it must be bound. Also note that a variable can occur both bound and free in an expression. E.g., x is both bound and free in $x + \lambda x{:}\mathbb{N}.x$.

Exercise 7.1.11. Does $x{:}\mathbb{N}^+$ occur freely in

1. $x + y$.
2. $\lambda x{:}\mathbb{N}^+.x + y$.
3. $\lambda x{:}\mathbb{N}^+.x + x$.
4. $(\lambda x{:}\mathbb{N}^+.x) + x$.

Substitutions, generally written using the letter ξ, are functions that map variables in $\mathcal{X}_\mathcal{S}$ to data expressions of the same sort. I.e., substitutions are *sort preserving*. We use special notation for simple substitutions. The substitution $[x{:}D := p]$ maps all variables to itself, except that it replaces free occurrences of variable $x{:}D$ to p. The substitution $[x_i{:}D_i{:=}p_i]_{1\leq i\leq n}$ maps all variables x_i to data expresions p_i. We write such substitutions after a data expression. We extend the definition of substitutions to more complex data expressions as follows.

Definition 7.1.12. Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ a signature and let $\mathcal{X}_\mathcal{S}$ a set of variable symbols. Let ξ be a substitution from $\mathcal{X}_\mathcal{S}$ to well-typed data expressions. We inductively extend ξ to a function from data expressions to data expressions as follows. So, consider a data expression p of sort D:

- if $p = f$ for a function symbol $f \in \mathcal{F}_\mathcal{S}$ then $\xi(p) = f$.
- If $p = p'(p_1, \ldots, p_n)$ then $\xi(p) = \xi(p')(\xi(p_1), \ldots, \xi(p_n))$.
- If $p = \lambda x_1{:}D_1, \ldots, x_n{:}D_n.p'$, then $\xi(p) = \lambda y_1{:}D_1, \ldots, y_n{:}D_n.\xi(p[x_i{:}D_i := y_i]_{1\leq i\leq n})$, provided $y_i{:}D_i$ do not occur freely in p' and $\xi(y_i{:}D_i) = y_i$. We assume that we have plenty of variables, such that the required variables y_i can always be found.
- If $p = \forall x{:}D'.p'$ or $p = \exists x{:}D'.p'$, then $\xi(p) = \forall y{:}D'.\xi(p[x{:}D' := y])$ and $\xi(p) = \exists y{:}D'.\xi(p[x{:}D' := y])$, respectively, provided y does not occur freely in p' and $\xi(y{:}D) = y$. Again with an infinite supply of variables, such an y can always be found.
- If $p = p'$ $\mathbf{whr}$ $x_1{=}p_1, \ldots, x_n{=}p_n$ $\mathbf{end}$, then

  $$\xi(p) = \xi(p'[x_i{:=}y_i]_{1\leq i\leq n})\ \mathbf{whr}\ y_1{=}\xi(p_1), \ldots, y_n{=}\xi(p_n)\ \mathbf{end}$$

  where y_i are variables that do not occur freely in p' and p_i for all $1 \leq i \leq n$.

Exercise 7.1.13. Apply the following substitutions.

1. $(\lambda x{:}\mathbb{N}.\lambda y{:}\mathbb{N}.x + y + z)[z := u]$.
2. $(\lambda x{:}\mathbb{N}.y)[y{:=}x]$.
3. $(\lambda x{:}\mathbb{N}.\lambda y{:}\mathbb{N}.x)[x{:=}z]$.
4. $(\lambda x, y{:}\mathbb{N}.x)[x{:=}z]$.

7.1.5 Data specifications

Given a signature, we want that certain data expressions are equal to others. For instance, we want that $1 + 1$ equals 2. Using a set of conditional equations we define which data expressions are equal to each other. In appendix A all the basic data-types are defined using such conditional equations.

Definition 7.1.14 (Data specification). Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ be a well-typed signature. We call a tuple $\mathcal{D} = (\Sigma, E)$ a *data specification* where E is a set of *conditional equations*. Each equation in E is a pair $\langle \mathcal{X}, c \to p_1 = p_2 \rangle$ where $\mathcal{X}$ is a set of variables and c, p_1 and p_2 are data data expressions.

Definition 7.1.15 (Well-typed data specification). We say that a data specification $\mathcal{D} = (\Sigma, E)$ where $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$ is well-typed iff for each pair $\langle \mathcal{X}, c{\to}p_1 = p_2 \rangle \in E$ it holds that

- For each variable $d{:}D$ in $\mathcal{X}$ the sort D is in $\mathcal{S}$ and d does not occur in $\mathcal{C}$ and in $\mathcal{M}$.
- c is a well-typed data expression of sort $\mathbb{B}$ over $\mathcal{X}$ and p_1 and p_2 are well-typed data data expressions of sort D and D' respectively over $\mathcal{X}$ where $D \subseteq D'$ or $D' \subseteq D$.

Note that the sorts D and D' are subsorts. In general D and D' are equal sorts.

7.1.6 Semantics of datatypes

Data specifications are complex objects, in which variables, sorts, mappings, constructors, equations and functions play a role. We want to define when two expressions in this world are or are not equal in a concise fashion. This is done using so-called model class semantics. We first define an applicative D-structure as a collection of sets where each set constitutes the counterpart of a sort. We also show how each data expression can be interpreted as an element in such a set. Subsequently, we define that an applicative D-structure is a model when primarily all conditional equations are valid in the structure. Finally, we define that data expressions are considered equal if they are equal in all models.

Definition 7.1.16 (Applicative $\mathcal{D}$-structure). Let $\mathcal{D} = (\Sigma, E)$ be a data specification. A collection of nonempty sets $\{M_D | D \in \mathcal{S}_\mathcal{S}\}$ is called an applicative $\mathcal{D}$-*structure* iff

- $D_\mathbb{B}$ is a set with two elements, denoted by $\mathbf{true}$ and $\mathbf{false}$.
- If $D \in \mathcal{S}$ and D is not a function sort, then M_D is a non empty set.
- If $D = D_1 \times \cdots \times D_n \to D'$, then M_D is the set of all functions from $M_{D_1} \times \cdots \times M_{D_n}$ to $M_{D'}$.

A function $[\![\cdot]\!]$ is called a $\mathcal{D}$-*interpretation* into an applicative $\mathcal{D}$-structure $\{M_D | D \in \mathcal{S}_\mathcal{S}\}$ iff for all $f \in \mathcal{C}_\mathcal{S} \cup \mathcal{F}_\mathcal{S}$ of sort D it holds that $[\![f]\!] \in M_D$.

We call $\sigma{:}\mathcal{X}_\mathcal{S} \to \bigcup_{D \in \mathcal{S}_\mathcal{S}} M_D$ a *valuation* if it holds that $\sigma(x) \in M_D$ for all $x{:}\mathcal{X}_\mathcal{D}$. We write $\sigma[d/x]$ for a valuation that maps variables according to σ except that it maps x to d. We write $\sigma[d_i/x_i]_{1 \le i \le n}$ for the valuation σ except that it maps each variable x_i to d_i.

The interpretation function $[\![\cdot]\!]^\sigma$ is extended to expressions as follows

- $[\![x]\!]^\sigma = \sigma(x)$ for every variable $x \in \mathcal{X}_D$.
- $[\![f]\!]^\sigma = [\![f]\!]$ for every function symbol $f \in \mathcal{C}_\mathcal{S} \cup \mathcal{F}_\mathcal{S}$.
- $[\![p\,(p_1, \ldots, p_n)]\!]^\sigma = [\![p_1]\!]^\sigma([\![p_1]\!]^\sigma, \ldots, [\![p_n]\!]^\sigma)$.
- $[\![\lambda x_1{:}D_1, \ldots, x_n{:}D_n.p]\!]^\sigma = f$ where $f : M_{D_1} \times \cdots \times M_{D_n} \to D$ is the function satisfying $f(d_1, \ldots, d_n) = [\![p]\!]^{\sigma[d_i/x_i]_{1 \le i \le n}}$ for all $d{:}M_D$.
- $[\![\forall x{:}D.p]\!]^\sigma = \mathbf{true}$ iff for all $d \in M_D$ it holds that $[\![p]\!]^{\sigma[d/x]} = \mathbf{true}$.
- $[\![\exists x{:}D.p]\!]^\sigma = \mathbf{true}$ iff for some $d \in M_D$ it holds that $[\![p]\!]^{\sigma[d/x]} = \mathbf{true}$.
- $[\![p\ \mathbf{whr}\ x_1{=}p_1, \ldots, x_n{=}p_n\ \mathbf{end}]\!]^\sigma = [\![p]\!]^{\sigma[[\![p_i]\!]^\sigma/x_i]_{1 \le i \le n}}$.

Definition 7.1.17 ($\mathcal{D}$-model). Let $\mathcal{D} = (\Sigma, E)$ be a data specification. An applicative $\mathcal{D}$-structure together with a $\mathcal{D}$-interpretation $[\![\cdot]\!]$ is called a $\mathcal{D}$-*model* iff

- for every equation $c \to p_1{=}p_2 \in E_\mathcal{S}$ is holds that if $[\![c]\!]^\sigma = \mathbf{true}$ then $[\![p_1]\!]^\sigma = [\![p_2]\!]^\sigma$ for every valuation σ.
- $[\![true]\!]^\sigma = \mathbf{true}$ and $[\![false]\!]^\sigma = \mathbf{false}$ for every valuation σ.
- If a basic sort D is a constructor sort (i.e. there is a constructor $f \in \mathcal{C}_\mathcal{S}$ of sort $D_1 \times \cdots \times D_n \to D$), then every element $d \in M_D$ is a constructor element, where a constructor element is inductively defined as follows:
  - Every element $d \in M_D$ is a constructor element, if D is a constructor sort and a constructor function $f \in \mathcal{C}_\mathcal{S}$ of sort $D_1 \times \cdots \times D_n \to D$ exists such that $d = [\![f]\!](e_1, \ldots, e_n)$ where e_i is either a constructor element of sort D_i, or sort D_i is not a constructor sort.

Definition 7.1.18 (Equality and validity). Let $\mathcal{D} = (\Sigma, E)$ be a data specification. We say that two data expressions p_1 and p_2 of sort D are *equal*, notation $\models p_1 = p_2$, iff for all $\mathcal{D}$-models consisting of an applicative $\mathcal{D}$-structure $\{M_D | D{\in}\mathcal{S}\}$ and a $\mathcal{D}$-interpretation $[\![\cdot]\!]$ and all valuations σ it holds that $[\![p_1]\!]^\sigma = [\![p_2]\!]^\sigma$. If p is a data expression of sort $\mathbb{B}$, we say that p is *valid*, notation $\models p$ iff $\models p = true$.

7.2 Operational semantics of the mCRL2 processes

Given that we know now what kind of elements are denotated by data expressions, we define how a process specification gives rise to a timed transition system. First we define what a process specification is.

7.2.1 Well-typed processes, action declarations and process equations

In chapter 4 the syntax of multi-actions have been introduced. The syntax of *process expressions* has only been sketched. Here we provide a more concise definition of process expressions and define when they are well-typed.

Definition 7.2.1 (Action declaration). Let $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$ be a signature. An *action declaration* is an expression of the form $a : D_1 \times \cdots \times D_n$ where $n \geq 0$ and all are sorts D_i are taken from $\mathcal{S}$.

Definition 7.2.2 (Process expression). Let $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$ be a signature. Process expressions are expressions with the following syntax.

$$\begin{aligned} p ::= \alpha \mid p + p \mid p \cdot p \mid \delta \mid c \rightarrow p \diamond p \mid \sum_{d:D} p \mid p \triangleleft t \mid t \gg p \mid p \| p \mid p \between p \mid \\ p \ll p \mid p | p \mid \Gamma_C(p) \mid \nabla_V(p) \mid \partial_B(p) \mid \rho_R(p) \mid \tau_I(p) \mid \Upsilon_U(p) \mid \\ X \mid X(u_1, \ldots, u_n) \mid X() \mid X(d_1 = u_1, \ldots, d_n = u_n). \end{aligned}$$

Here, α is a multi-action, c is a boolean data-expression, d, $d_1, \ldots, d_n$ $(n > 0)$ are variables, t is a data-expression of sort $\mathbb{R}$, D is a sort, C a set of communications, V a set of multi-action labels, I, U and B are sets of action labels, R is a set of renamings and $u_1, \ldots, u_n$ are data expressions.

We manipulate with process expressions in the same way as we do with data expressions. We use the special sort $\mathbb{P}$ to contain processes which is not a data sort. So, in data expressions we cannot make reference to processes. In the meta theory, we quite generally use functions from data to processes, e.g., $f : \mathbb{N} \to \mathbb{P}$ and sometimes even functions from processes to datat as in $g : \mathbb{P} \to \mathbb{N}$. But note that these are meta notations, and not formally part of mCRL2.

Definition 7.2.3 (Process equation). Let $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$ be a signature. An *process equation* is an expression of the form $P(d_1{:}D_1, \ldots, d_n{:}D_n) = p$ where $n \leq 0$ and d_i are variables and D_i are sorts from $\mathcal{S}$.

A process specification consists of a data specification, a set of process equations, an initial process and a set of global variables.

Definition 7.2.4. A *process specification* is a five tuple $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ where

- $\mathcal{D}$ is a data specification,
- AD is an action declaration,
- PE is a set of process equations,
- p is a process expression, and
- $\mathcal{X}$ is a set of global variables.

Definition 7.2.5. Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification and $\mathcal{X}_\mathcal{S}$ a set of variables such that $\mathcal{X} \subseteq \mathcal{X}_\mathcal{S}$. We say that a multi-action α is *well-typed* iff

1. The multi-action α equals τ.
2. The multi-action α is equal to single multi-action $a(t_1, \ldots, t_n)$ and there are sorts D_i and $E_i \subseteq D_i$ for $1 \leq i \leq n$ such that

- There is an action declaration in $a : D_1 \times \cdots \times D_n \in AD$ over $\mathcal{X}_\mathcal{S}$ and t_i are well-typed expressions of sort E_i over $\mathcal{X}_\mathcal{S}$, and
- if there are sorts D'_i and E'_i ($1 \leq i \leq n$) such that $a : D'_1 \times \cdots \times D'_n \in AD$ over $\mathcal{X}_\mathcal{S}$ and t_i are well-typed expressions of sort $E'_i \subseteq D'_i$ over $\mathcal{X}_\mathcal{S}$ then for all $1 \leq j \leq n$ it holds that $D_j \subseteq D'_j$ and $E_j \subseteq E'_j$.

3. If α is equal to $\alpha_1|\alpha_2$ and both α_1 and α_2 are well-typed over $\mathcal{X}_\mathcal{S}$.

Definition 7.2.6. Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification and let $\mathcal{X}_\mathcal{S}$ be a set of variables such that $\mathcal{X} \subseteq \mathcal{X}_\mathcal{S}$. We say that a process expression p is *well-typed* over $\mathcal{X}_\mathcal{S}$ iff one of the following conditions apply.

1. If p equals a multi-action α, then α must be well-typed over $\mathcal{X}_\mathcal{S}$.
2. If p equals $p_1 + p_2$, $p_1 \cdot p_2$, $p_1 \parallel p_2$, $p_1 \between p_2$, $p_1 \ll p_2$ or $p_1|p_2$, then p_1 and p_2 must be well-typed over $\mathcal{X}_\mathcal{S}$.
3. The process expression p equals δ.
4. If p is equal to $c \to p_1 \diamond p_2$, then c is a well-typed expression of sort $\mathbb{B}$ over $\mathcal{X}_\mathcal{S}$ and both p_1 and p_2 are well-typed process expressions over $\mathcal{X}_\mathcal{S}$.
5. If p is equal to $\sum_{d:D} p_1$, then D is a sort in $\mathcal{S}$ and p_1 is a well-typed process expression over $\mathcal{X}_\mathcal{S} \cup \{d{:}D\}$.
6. If p equals $p_1 \triangleleft t$ or $t \gg p_1$, then t is a well-typed expression of sort D over $\mathcal{X}_\mathcal{S}$ where $D \in \{\mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ and for all $D' \in \{\mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ such that t is a well-typed expression of sort D' over $\mathcal{X}_\mathcal{S}$, it holds that $D' = D$.
7. If p equals $\Gamma_C(p_1)$ then p_1 is well-typed over $\mathcal{X}_\mathcal{S}$ and for each communication $a_1|\cdots|a_n \to a \in C$ it holds that
   (a) all actions $b \in \{a_1, \ldots, a_n, a\}$ are declared. For some $m \geq 0$ and sorts $D_1, \ldots, D_m$ it holds that $b : D_1 \times \cdots \times D_n \in AD$.
   (b) $a : D_1 \times \cdots \times D_m \in AD$ iff for any $1 \leq i \leq n$ it holds that $a_i : D_1 \times \cdots \times D_m \in AD$.
8. If p equals $\nabla_V(p_1)$ then p_1 is well-typed over $\mathcal{X}_\mathcal{S}$ and for any multi-action $a_1|\cdots|a_n \in AD$ it holds that each action a_i ($1 \leq i \leq n$) is declared, i.e., $a_i : D_1 \times \cdots \times D_n \in AD$ for some sorts $D_1, \ldots D_n$.
9. If p equals $\partial_B(p_1)$, $\tau_B(p_1)$ or $\Upsilon_B(p_1)$, then p_1 is well-typed over $\mathcal{X}_\mathcal{S}$ and for any action $a \in B$, it holds that a is declared, i.e., $a : D_1 \times \cdots \times D_n \in AD$ for some sorts $D_1, \ldots D_n$.
10. If p equals $\rho_R(p_1)$ then p_1 is well-typed over $\mathcal{X}_\mathcal{S}$ and for any renaming $a \to b \in R$ it holds that
    (a) $a : D_1 \times \cdots \times D_n \in AD$ for some $n \geq 0$ and $D_1 \times \cdots \times D_n \in AD$.
    (b) $a : D_1 \times \cdots \times D_n \in AD$ iff $a : D_1 \times \cdots \times D_n \in AD$ for all $n \geq 0$ and sorts $D_1, \ldots, D_n$.
11. If p equals X, then $X = q$ is an equation in PE.
12. If p equals $X(u_1, \ldots, u_n)$,
    (a) $X(d_1{:}D_1, \ldots, d_n{:}D_n) = q$ is a process equation in PE and for all $1 \leq i \leq n$ it holds that u_i is a well-typed expression of sort $E_i \subseteq D_i$ over $\mathcal{X}_\mathcal{S}$, and
    (b) if there are sorts D'_i and E'_i ($1 \leq i \leq n$) such that $X(d'_1{:}D'_1, \ldots, d'_n{:}D'_n) = q'$ is a process equation in PE and u_i are well-typed expressions over $\mathcal{X}_\mathcal{S}$ of sort E'_i, it holds that $D_j \subseteq D'_j$ and $E_j \subseteq E'_j$ for all $1 \leq j \leq n$.
13. If p equals $X()$ then there is only one equation $X(d_1{:}D_1, \ldots, d_n{:}D_n) = q \in PE$ for some variables d_i, sorts D_n and process expression q.

14. If p equals $X(d_1 = u_1, \ldots, d_n = u_n)$, there is only one equation $X(d'_1{:}D'_1, \ldots, d'_m{:}D'_m) = q \in PE$ for some variables d_i, sorts D_n and process expression q. Moreover, for each $1{\leq}i{\leq}n$ it holds that d_i is equal to d'_j for some $1{\leq}j{\leq}m$ and u_i is a well-typed expressions of sort $D \subseteq D'_j$ over $\mathcal{X}_{\mathcal{S}}$.

Definition 7.2.7. We say that a process specification $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ where $\mathcal{D} = (\Sigma, E)$ and $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$, is *well-typed* iff all of the conditions below are fullfilled

1. $\mathcal{D}$ is well-typed.
2. For any action declaration $a : D_1 \times \cdots \times D_n \in AD$ no process declaration $a(d_1{:}D_1, \ldots, d_n{:}D_n) = p \in PE$ exists. Furthermore, for all $1{\leq}i{\leq}n$ it holds that $D_i \in \mathcal{S}$.
3. For any process definition $X(d_1{:}D_1, \ldots, d_n{:}D_n) = p \in PE$ it holds that there is no action declaration $X : D_1 \times \cdots \times D_n \in AD$. The variables d_i do not occur in $\mathcal{C}$ or $\mathcal{M}$. The sorts D_i all occur in $\mathcal{S}$. Furthermore, p is a well-typed process expression over $\{d_1{:}D_1, \ldots, d_n{:}D_n\} \cup \mathcal{X}$.
4. The process expression p is well-typed over $\mathcal{X}$.
5. If a variable $d{:}D$ occurs in $\mathcal{X}$, then $D \in \mathcal{S}$ and there is no function symbol d that occurs in $\mathcal{C} \cup \mathcal{M}$.

$\dfrac{}{(\alpha,\sigma) \xrightarrow{[\![\alpha]\!]^\sigma}_t \checkmark}\ t{>}0$	$\dfrac{}{(\alpha,\sigma) \rightsquigarrow_t}$	$\dfrac{}{(\delta,\sigma) \rightsquigarrow_t}$
$\dfrac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(p+q,\sigma) \xrightarrow{m}_t \checkmark}$	$\dfrac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(p+q,\sigma) \xrightarrow{m}_t (p',\sigma')}$	$\dfrac{(p,\sigma) \rightsquigarrow_t}{(p+q,\sigma) \rightsquigarrow_t}$
$\dfrac{(q,\sigma) \xrightarrow{m}_t \checkmark}{(p+q,\sigma) \xrightarrow{m}_t \checkmark}$	$\dfrac{(q,\sigma) \xrightarrow{m}_t (q',\sigma')}{(p+q,\sigma) \xrightarrow{m}_t (q',\sigma')}$	$\dfrac{(q,\sigma) \rightsquigarrow_t}{(p+q,\sigma') \rightsquigarrow_t}$
$\dfrac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(p{\cdot}q,\sigma) \xrightarrow{m}_t (t{\gg}q,\sigma)}$	$\dfrac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(p{\cdot}q,\sigma) \xrightarrow{m}_t (p'{\cdot}q,\sigma')}$	$\dfrac{(p,\sigma) \rightsquigarrow_t}{(p{\cdot}q,\sigma) \rightsquigarrow_t}$
$\dfrac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(b{\to}p,\sigma) \xrightarrow{m}_t \checkmark}\ [\![b]\!]^\sigma{=}\mathbf{true}$	$\dfrac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(b{\to}p,\sigma) \xrightarrow{m}_t (p',\sigma')}\ [\![b]\!]^\sigma{=}\mathbf{true}$	$\dfrac{(p,\sigma) \rightsquigarrow_t}{(b{\to}p,\sigma) \rightsquigarrow_t}\ [\![b]\!]^\sigma{=}\mathbf{true}$
$\dfrac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(b{\to}p{\diamond}q,\sigma) \xrightarrow{m}_t \checkmark}\ [\![b]\!]^\sigma{=}\mathbf{true}$	$\dfrac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(b{\to}p{\diamond}q,\sigma) \xrightarrow{m}_t (p',\sigma')}\ [\![b]\!]^\sigma{=}\mathbf{true}$	$\dfrac{(p,\sigma) \rightsquigarrow_t}{(b{\to}p{\diamond}q,\sigma) \rightsquigarrow_t}\ [\![b]\!]^\sigma{=}\mathbf{true}$
$\dfrac{(q,\sigma) \xrightarrow{m}_t \checkmark}{(b{\to}p{\diamond}q,\sigma) \xrightarrow{m}_t \checkmark}\ [\![b]\!]^\sigma{=}\mathbf{false}$	$\dfrac{(q,\sigma) \xrightarrow{m}_t (q',\sigma')}{(b{\to}p{\diamond}q,\sigma) \xrightarrow{m}_t (q',\sigma')}\ [\![b]\!]^\sigma{=}\mathbf{false}$	$\dfrac{(q,\sigma) \rightsquigarrow_t}{(b{\to}p{\diamond}q,\sigma) \rightsquigarrow_t}\ [\![b]\!]^\sigma{=}\mathbf{false}$

Table 7.1: Operational rules for the basic operators

7.2.2 Semantical multi-actions

In this section we define semantical multi-actions as multi-actions that have model elements as arguments. We want to define the semantics of a process expression as a timed transition system. In these transition systems transition labels play an important role. Of course these labels are derived from the (multi-)actions of a process. But we cannot take data expressions as the arguments of these actions. Consider for instance a sort D. The process $\sum_{d:D} a(d)$ can perform all kinds of actions $a(d_v)$ where d_v is an element from the domain M_D. The elements in M_D may not be denotable by data expressions for instance when D is not

$$\frac{(q,\sigma[\overrightarrow{d}:=[\![\overrightarrow{e}]\!]^\sigma]) \xrightarrow{m}_t \checkmark}{(P(\overrightarrow{e}),\sigma) \xrightarrow{m}_t \checkmark} \qquad \frac{(q[\overrightarrow{d}:=\overrightarrow{d'}],\sigma[\overrightarrow{d'}:=[\![\overrightarrow{e}]\!]^\sigma]) \xrightarrow{m}_t (q',\sigma')}{(P(\overrightarrow{e}),\sigma) \xrightarrow{m}_t (q',\sigma')} \qquad \frac{(q,\sigma[\overrightarrow{d}:=\overrightarrow{[\![e]\!]^\sigma}]) \rightsquigarrow_t}{(P(\overrightarrow{e}),\sigma) \rightsquigarrow_t}$$

where $P(\overrightarrow{d}:\overrightarrow{D}) = q \in PE$ and $\overrightarrow{d'}$ are fresh variables of sort $\overrightarrow{D}$.

Table 7.2: Operational rules for recursion

$$\frac{(p,\sigma[d{:=}e]) \xrightarrow{m}_t \checkmark}{(\sum_{d:D} p,\sigma) \xrightarrow{m}_t \checkmark}\ e{\in}\mathcal{M}_D \qquad \frac{(p,\sigma[d{:=}e]) \xrightarrow{m}_t (p',\sigma')}{(\sum_{d:D} p,\sigma) \xrightarrow{m}_t (p',\sigma')}\ e{\in}\mathcal{M}_D \qquad \frac{(p,\sigma[d{:=}e]) \rightsquigarrow_t}{(\sum_{d:D} p,\sigma) \rightsquigarrow_t}\ e{\in}\mathcal{M}_D$$

$$\frac{(p,\sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} \checkmark}{(p \lhd u,\sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} (p',\sigma')}{(p \lhd u,\sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(p \lhd u,\sigma) \rightsquigarrow_t}\ t{<}[\![u]\!]^\sigma$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(u \gg p,\sigma) \xrightarrow{m}_t \checkmark}\ [\![u]\!]^\sigma{<}t \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(u \gg p,\sigma) \xrightarrow{m}_t (p',\sigma')}\ [\![u]\!]^\sigma{<}t$$

$$\frac{(p,\sigma) \rightsquigarrow_t}{(u \gg p,\sigma) \rightsquigarrow_t} \qquad \frac{}{(u \gg p,\sigma) \rightsquigarrow_t}\ t{<}[\![u]\!]^\sigma$$

Table 7.3: Operational rules for the sum, time and the bounded initialisation operators

a constructor sort. The most direct way to denote these domain elements d_v is by using themselves. This leads us to semantical multi-actions where the arguments are model elements.

Definition 7.2.8 (Interpretation of a multi-action). Let $\mathcal{D} = (\Sigma, E)$ be a data specification and $[\![\cdot]\!]^\sigma$ a $\mathcal{D}$-model. We inductively define the interpretation of a multi-action α for any data-valuation σ as follows:

- $[\![\tau]\!]^\sigma = \tau$.
- $[\![a(t_1,\ldots,t_n)]\!]^\sigma = a([\![t_1]\!]^\sigma,\ldots,[\![t_n]\!]^\sigma)$.
- $[\![\alpha|\beta]\!]^\sigma = [\![\alpha]\!]^\sigma|[\![\beta]\!]^\sigma$.

Here $|$ is a new operator on semantical multi-actions that is associative and commutative. In analogy with syntactical multi-actions, we write τ for the empty multi-action. The empty (semantical) multi-action is a unit for $|$. I.e., $\tau|\alpha = \alpha|\tau = \alpha$.

The semantics of the processes are defined using so-called inference rules. In these rules we use several notations on semantical multi-actions. These are provided in the following definition.

Definition 7.2.9. Let α be a semantical multi-action. We define:

- $\alpha_{\{\}}$ is the set of all actions occurring in α. In particular $a_{\{\}} = \{a\}$ and $(\alpha|\beta)_{\{\}} = \alpha_{\{\}} \cup \beta_{\{\}}$.
- Let R be a set of renamings. $R \bullet \alpha$ is the semantical multi-action where the labels have been renamed according to R. More precisely: $R \bullet a(d_1,\ldots,d_n) = b(d_1,\ldots,d_n)$ if $a{\to}b \in R$. Otherwise, the result is just $a(d_1,\ldots,d_n)$. This operator distributes over $|$. A particular renaming operators is $\eta_I(\alpha)$ which renames all actions labels in I to int. Furthermore, we use $\theta_I(\alpha)$ which removes all actions with labels in that occur in I. Strictly spoken, $\theta_I(\alpha)$ is not a renaming.
- We write $\underline{\alpha}$ for the multi-action α where all data has been removed. In particular $\underline{a(t_1,\ldots,t_n)} = a$.

- Communication is defined using γ_C. It says that a communication within different actions in a multi-action takes place, exactly when this communication occurs in C and the arguments of these actions have the same data arguments. By defining $\gamma_{C_1 \cup C_2}(\alpha) = \gamma_{C_1}(\gamma_{C_2}(\alpha))$ we can define apply each renaming on its own:

$$\gamma_{\{a_0|\ldots|a_n \to b\}}(\alpha) = \begin{cases} b(\overrightarrow{d})|\gamma_{\{a_0|\ldots|a_n \to b\}}(\beta) & \text{if actions } a_i(\overrightarrow{d}) \text{ occur in } \alpha \text{ for all } 1 \leq i \leq n \\ \alpha & \text{otherwise} \end{cases}$$

where $\beta = \alpha \setminus (a_1(\overrightarrow{d})|\ldots|a_n(\overrightarrow{d}))$, i.e., the multi-action α from which actions $a_i(\overrightarrow{d})$ are removed.

7.2.3 Operational semantics

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \rightsquigarrow_t}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg q,\sigma)} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \rightsquigarrow_t}{(p\|q,\sigma) \xrightarrow{m}_t (p'\|t \gg q,\sigma')}$$

$$\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \xrightarrow{m}_t \checkmark}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg p,\sigma)} \qquad \frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \xrightarrow{m}_t (q',\sigma')}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg p\|q',\sigma')}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p\|q,\sigma) \xrightarrow{m|n}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p\|q,\sigma) \xrightarrow{m|n}_t (p',\sigma')}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t (q',\sigma'')}{(p\|q,\sigma) \xrightarrow{m|n}_t (p'\|q',\sigma' \cup \sigma'')} \qquad \frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p\|q,\sigma) \rightsquigarrow_t}$$

Table 7.4: Structured operational semantics for the parallel operator

Given a data specification and a process expression, we relate a timed transition system to the process expression as defined below. The timed transition system is the semantics of a process expression. It allows us to think of a process as having behaviour. The transitions and idle relations are defined using inference rules in a style commonly to as *operational semantics*. The inference rules are a particular style of inductive definitions which is especially suited to define the transition and idle relations on some given syntax.

Definition 7.2.10 (Semantics of a process). Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification. Let $\mathcal{A} = \{M_D | D \in \mathcal{S}\}$ be a $\mathcal{D}$-model, $[\![\cdot]\!]$ a $\mathcal{D}$-interpretation transition system and σ a closed validation. We define the semantics of PS given $\mathcal{A}$ and σ as a timed transition system $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ as follows:

- The states S contain all pairs (p', σ') for process expressions p' and valuations σ'. There is one special termination state, denoted by $\checkmark$.
- The labels are pairs of semantical multi-actions and time values from $\mathbb{R}^{>0}$.
- The transitions are inductively defined by the operational rules in tables 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6. The transition relation is generally denoted by $(p', \sigma) \xrightarrow{m}_t (p'', \sigma')$ or $(p', \sigma) \xrightarrow{m}_t \checkmark$.
- There is an idle relation in each state that expresses that in this state the process can idle up till and including time $t \in \mathbb{R}^{>0}$. It is denoted by $p \rightsquigarrow_t$ and it is also defined by the operational rules in the tables.
- The initial state is (p, σ_0) where σ_0 is the identity substitution, i.e., $\sigma(x) = x$ for all variables x.

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p|q,\sigma) \xrightarrow{m|n}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p|q,\sigma) \xrightarrow{m|n}_t (p',\sigma')}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t (q',\sigma'')}{(p|q,\sigma) \xrightarrow{m|n}_t (p'\|q',\sigma' \cup \sigma'')} \qquad \frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p|q,\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(q|p,\sigma) \xrightarrow{m|n}_t (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(q|p,\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \rightsquigarrow_t}{(p \between q,\sigma) \xrightarrow{m}_t (t \gg q,\sigma)} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \rightsquigarrow_t}{(p \between q,\sigma) \xrightarrow{m}_t (p'\|t \gg q,\sigma')}$$

$$\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p \between q,\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \rightsquigarrow_t}{(p \between q,\sigma) \xrightarrow{m}_t (t \gg q,\sigma)} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \rightsquigarrow_t}{(p \between q,\sigma) \xrightarrow{m}_t (p'\|t \gg q,\sigma')}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \xrightarrow{m}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \xrightarrow{m}_t (p',\sigma')}$$

$$\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \rightsquigarrow_t}$$

Table 7.5: Structured operational semantics for the auxiliary parallel operators

Example 7.2.11. The process $a^{\frown}1$ has the transition system given in figure 7.1 Only two states are drawn as these are relevant in this example. Formally any pair $\langle p, \sigma \rangle$ is a state, but all but two are not reachable. The two remaining states are $\langle a^{\frown}1, \sigma_0 \rangle$ and $\checkmark$. The first one is the initial state and the second one is a terminal state. There is a transition $\langle a^{\frown}1, \sigma_0 \rangle \xrightarrow{a}_{\mathbf{1}} \checkmark$. Here the boldface $\mathbf{0}$ and $\mathbf{1}$ are the semantical interpretations of the syntactical numbers 0 and 1. More precisely $\mathbf{0} = [\![0]\!]^{\sigma_0}$ and $\mathbf{1} = [\![1]\!]^{\sigma_0}$. We only write the boldface numbers here to stress the subtle difference between the syntactical and semantical objects. For standard data types we generally adhere to the standard notation to denote elements in both domains, which might sometimes be confusing. The transition can be derived using the inference rules, as shown below

$$\cfrac{\cfrac{}{\langle a, \sigma_0 \rangle \xrightarrow{a}_{[\![1]\!]^{\sigma_0}} \checkmark}\ \text{Rule 1/7 of table 7.1.}}{\langle a^{\frown}1, \sigma_0 \rangle \xrightarrow{a}_{[\![1]\!]^{\sigma_0}} \checkmark}\ \text{Rule 1/3 of table 7.3.}$$

We refer to a particular inference rule by x/y where x is the column and y the row counting from below. So, rule $1/1$ is the leftmost rule at the bottom in a table. Note that this is the only derivation that can be constructed for the process $a^{\frown}1$. This means that there are no other actions possible.

In a similar way we can derive idle transitions $\langle a^{\frown}1, \sigma_0 \rangle \rightsquigarrow_t$ for any $t \in \langle \mathbf{0}, \mathbf{1} \rangle$.

$$\cfrac{\cfrac{}{\langle a, \sigma_0 \rangle \rightsquigarrow_t}\ \text{Rule 2/7 of table 7.1 as } t > \mathbf{0}.}{\langle a^{\frown}1, \sigma_0 \rangle \rightsquigarrow_t}\ \text{Rule 3/4 of table 7.3 as } t < \mathbf{1}.$$

Exercise 7.2.12. Derive the timed transition systems for $\partial_{\{b\}}(a^{\frown}1 + b^{\frown}2)$ and $a^{\frown}1 \parallel b^{\frown}2$.

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\nabla_V(p),\sigma) \xrightarrow{m}_t \checkmark}\ \underline{m}\in V \cup \{\tau\} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\nabla_V(p),\sigma) \xrightarrow{m}_t (p',\sigma')}\ \underline{m}\in V \cup \{\tau\} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\nabla_V(p),\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\partial_B(p),\sigma) \xrightarrow{m}_t \checkmark}\ \underline{m}_{\{\}} \cap B = \emptyset \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\partial_B(p),\sigma) \xrightarrow{m}_t (p',\sigma')}\ \underline{m}_{\{\}} \cap B = \emptyset \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\partial_B(p),\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\rho_R(p),\sigma) \xrightarrow{R \bullet m}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\rho_R(p),\sigma) \xrightarrow{R \bullet m}_t (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\rho_R(p),\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)}_t (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\Gamma_C(p),\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)}_t (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\tau_I(p),\sigma) \rightsquigarrow_t}$$

$$\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)}_t \checkmark} \qquad \frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)}_t (p',\sigma')} \qquad \frac{(p,\sigma) \rightsquigarrow_t}{(\Upsilon_U(p),\sigma) \rightsquigarrow_t}$$

Table 7.6: Operational semantics for auxiliary operators

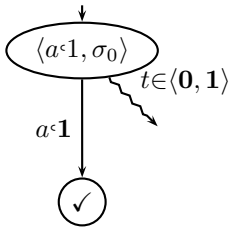


Figure 7.1: The semantics of the process $a^{\frown}1$

7.3 Validity of modal μ-calculus formulas

Given a transition system we can define in which states a formula holds. From this we know that a formula holds in the initial state, and we can even determine whether a formula holds for the initial state of the timed transition system that forms its semantics. First, we define the semantics of an action formula by defining which set of timed semantical multi-actions is associated with it.

Definition 7.3.1 (Semantics of action formulas). Let $\Sigma = (\mathcal{S}, \mathcal{C}_\mathcal{S}, \mathcal{M}_\mathcal{S})$ be a signature, $\mathcal{D} = (\Sigma, E_S)$ be a data specification, $\mathcal{A} = \{M_D | D \in \mathcal{S}_S\}$ a $\mathcal{D}$-structure, $[\![\cdot]\!]$ a $\mathcal{D}$-model, $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ be a timed transition system where Act consists of pairs of semantical multi-actions and time values from $\mathbb{R}^{>0}$.

Let af be an action expression. We define the interpretation of af, notation $[\![af]\!]^\sigma$ where σ is a validation, as a set of semantical actions, inductively by

- $[\![true]\!]^\sigma = Act$.
- $[\![false]\!]^\sigma = \emptyset$.
- $[\![t]\!]^\sigma = \begin{cases} Act & \text{if } [\![t]\!]^\sigma = true, \\ \emptyset & \text{otherwise.} \end{cases}$.
- $[\![\alpha]\!]^\sigma = \{\langle [\![\alpha]\!]^\sigma, t \rangle \mid t \in \mathbb{R}^{>0}\}$.

- $[\![\overline{af}]\!]^\sigma = Act \setminus [\![af]\!]^\sigma$.
- $[\![af \cap af']\!]^\sigma = [\![af]\!]^\sigma \cap [\![af']\!]^\sigma$.
- $[\![af \cup af']\!]^\sigma = [\![af]\!]^\sigma \cup [\![af']\!]^\sigma$.
- $[\![\forall d{:}D.af]\!]^\sigma = \bigcap_{d \in M_D} [\![af]\!]^{\sigma[x:=d]}$.
- $[\![\exists d{:}D.af]\!]^\sigma = \bigcup_{d \in M_D} [\![af]\!]^{\sigma[x:=d]}$.
- $[\![af \triangleleft u]\!]^\sigma = \{\langle m, t\rangle \mid \langle m, t\rangle \in [\![af]\!]^\sigma,\ t = [\![u]\!]^\sigma\}$.

Example 7.3.2. Consider actions a and b that have one natural number as argument. A few typical instances of the semantics of action formulas are given below.

$[\![a(3) \triangleleft 5]\!]^\sigma = \{\langle a(\mathbf{3}), \mathbf{5}\rangle\}$.
$[\![a(3)|b(5)]\!]^\sigma = \{\langle a(\mathbf{3})|b(\mathbf{5}), t\rangle | t \in \mathbb{R}^{>0}\}$.
$[\![\exists x{:}\mathbb{N}.a(x) \triangleleft x]\!]^\sigma = \{\langle a(n), n\rangle | n \in \mathbb{N}\}$.
$[\![\forall x{:}\mathbb{N}.a(x) \triangleleft x]\!]^\sigma = \emptyset$.

Below we define in which states of a timed transition system a formula holds given a model and an interpretation of the signature in this model. Subsequently, we define that a formula is valid for a process independent of a particular model. We generalise this even further by defining when two formulas are equivalent independent of a model and a particular process.

Definition 7.3.3 (Semantics of modal formulas). Let $\mathcal{D} = (\Sigma, E_S)$ be a data specification where $\Sigma = (\mathcal{S}, \mathcal{C}, \mathcal{M})$, $\mathcal{A} = \{M_D | D \in \mathcal{S}\}$ a $\mathcal{D}$-structure, $[\![\cdot]\!]$ a $\mathcal{D}$-interpretation, $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ be a timed transition system where Act consists of pairs of semantical multi-actions and time values from $\mathbb{R}^{>0}$.

Let ϕ be a modal formula. We inductively define the interpretation of ϕ, notation $[\![\phi]\!]^{\sigma,\rho}$, where σ is a valuation and ρ is a logical variable valuation, as a set of states where ϕ is valid, by

- $[\![true]\!]^{\sigma,\rho} = S$.
- $[\![false]\!]^{\sigma,\rho} = \emptyset$.
- $[\![t]\!]^{\sigma,\rho} = \begin{cases} S & \text{if } [\![t]\!]^{\sigma,\rho} = \mathbf{true}, \\ \emptyset & \text{if } [\![t]\!]^{\sigma,\rho} = \mathbf{false}. \end{cases}$
- $[\![\neg\phi]\!]^{\sigma,\rho} = S \setminus [\![\phi]\!]^{\sigma,\rho}$.
- $[\![\phi_1 \wedge \phi_2]\!]^{\sigma,\rho} = [\![\phi_1]\!]^{\sigma,\rho} \cap [\![\phi_2]\!]^{\sigma,\rho}$.
- $[\![\phi_1 \vee \phi_2]\!]^{\sigma,\rho} = [\![\phi_1]\!]^{\sigma,\rho} \cup [\![\phi_2]\!]^{\sigma,\rho}$.
- $[\![\forall x{:}D.\phi]\!]^{\sigma,\rho} = \bigcap_{d \in \mathcal{M}_D} [\![\phi]\!]^{\sigma[x:=d],\rho}$.
- $[\![\exists x{:}D.\phi]\!]^{\sigma,\rho} = \bigcup_{d \in \mathcal{M}_D} [\![\phi]\!]^{\sigma[x:=d],\rho}$.
- $[\![\langle\alpha\rangle\phi]\!]^{\sigma,\rho} = \{s \in S \mid \text{there are } m, t \text{ and } s' \text{ such that } s \xrightarrow{m}_t s',\ s' \in [\![\phi]\!]^{\sigma,\rho} \text{ and } \langle m, t\rangle \in [\![\alpha]\!]^\sigma\}$.
- $[\![[\alpha]\phi]\!]^{\sigma,\rho} = \{s \in S \mid \text{for all } m, t \text{ and } s' \text{ if } s \xrightarrow{m}_t s' \text{ and } \langle m, t\rangle \in [\![\alpha]\!]^\sigma \text{ then } s' \in [\![\phi]\!]^{\sigma,\rho}\}$.
- $[\![\Delta]\!]^{\sigma,\rho} = \{s \mid s \rightsquigarrow_t \text{ for all } t \in \mathbb{R}^{>0}\}$.
- $[\![\Delta \triangleleft u]\!]^{\sigma,\rho} = \{s \mid s \rightsquigarrow_{[\![u]\!]^\sigma}\}$.
- $[\![\nabla]\!]^{\sigma,\rho} = \{s \mid \text{not } s \rightsquigarrow_t \text{ for some } t \in \mathbb{R}^{>0}\}$.
- $[\![\nabla \triangleleft u]\!]^{\sigma,\rho} = \{s \mid \text{not } s \rightsquigarrow_{[\![u]\!]^\sigma}\}$.

- $[\![\mu X(x_1{:}D_1{:=}t_1,\ldots,x_n{:}D_n{:=}t_n).\phi]\!]^{\sigma,\rho} =$
$\bigcap_{f:\mathcal{M}_{D_1}\times\cdots\times\mathcal{M}_{D_n}\to 2^S}\{f([\![t_1]\!]^\sigma,\ldots,[\![t_n]\!]^\sigma) \mid \forall d_1{:}\mathcal{M}_{D_1},\ldots,d_n{:}\mathcal{M}_{D_n}.$
$f(d_1,\ldots,d_n) = [\![\phi]\!]^{\sigma[x_1:=[\![t_1]\!]^\sigma,\ldots,x_n:=[\![t_n]\!]^\sigma],\rho[X:=f]}\}.$
- $[\![\nu X(x_1{:}D_1{:=}t_1,\ldots,x_n{:}D_n{:=}t_n).\phi]\!]^{\sigma,\rho} =$
$\bigcup_{f:\mathcal{M}_{D_1}\times\cdots\times\mathcal{M}_{D_n}\to 2^S}\{f([\![t_1]\!]^\sigma,\ldots,[\![t_n]\!]^\sigma) \mid \forall d_1{:}\mathcal{M}_{D_1},\ldots,d_n{:}\mathcal{M}_{D_n}.$
$f(d_1,\ldots,d_n) = [\![\phi]\!]^{\sigma[x_1:=[\![t_1]\!]^\sigma,\ldots,x_n:=[\![t_n]\!]^\sigma],\rho[X:=f]}\}.$
- $[\![X(t_1,\ldots,t_n)]\!]^{\sigma,\rho} = \rho(X)([\![t_1]\!]^\sigma,\ldots,[\![t_n]\!]^\sigma)$.

We say that ϕ holds in A iff $s_0 \in [\![\phi]\!]^{\sigma,\rho_0}$ for any ρ, σ and $[\![\cdot]\!]$. argument to empty sets.

Definition 7.3.4 (Validity of a modal formula for process specification). Let $PS = (\mathcal{D}, AD, PE, p, \mathcal{X})$ be a process specification and let ϕ be a modal formula. We say that ϕ is valid in PS iff for any $\mathcal{D}$-structure $\mathcal{A}$, $\mathcal{D}$-interpretation $[\![\cdot]\!]$ the timed transition system $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$ that is the semantics of PS given $\mathcal{A}$ and σ, ϕ holds in A. We say that two modal formulas ϕ and ψ are equivalent iff for all process specifications ϕ is valid iff ψ is valid.

7.4 Soundness and completeness

We say that two process expressions p and q are equal, iff for all models the roots of their timed transitions systems are strongly bisimilar. The axioms in tables 4.2, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 and 4.22 all respect strong bisimulation unless it is explicitly stated that they are applicable to other equivalence, in which case these are respected. This means that if two expressions are transformed into each other by applying one of the axioms in the table the timed transition systems associated to these process expressions are equivalent. We say that the axioms are *sound* with respect to the equivalence. All the axioms in the tables are sound with respect to the indicated equivalence, which is strong bisimulation by default.

We say that a set of axioms are *complete* if two process expressions that have equivalent timed transition systems can be transformed to each other by applying axioms and inference rules. For finite processes, which do not have recursion, the axioms are complete. For infinite processes, the axioms and inference rules are complete for most, but not all, circumstances.

The axioms in table 5.1 are sound equivalence of modal formulas. This means that if the axioms are applied to transform a modal formula, then the original and the result are still valid for exactly the same transition systems. As it stands little is known regarding the completeness of these axioms.

Part II

Model transformations

Chapter 8

Basic manipulation of processes

8.1 Introduction

In this chapter we show how the axioms given in chapter 4 can be used to prove that different process expressions have the same behaviour. Due to the existence of data, some of the variables in the axioms range over functions from data to processes. In order to denote such functions, we introduce the lambda calculus here as a means to denote meta objects. In chapter 4 lambda abstraction was already introduced as way to construct functions within our specification language. Subsequently, we make very precise what we mean by a derivation in equational logic. Equational reasoning is very common, and therefore we have already applied it extensively in previous chapters.

There are three principles, not treated yet, necessary to prove processes equal. We discuss all these in this chapter. The first one is induction on data types. The second one is the Recursive Specification Principle (RSP) to equate infinite processes. The last one is Koomen's Fair Abstraction Rule, which we need to handle fair behaviour. Finally, we show the parallel expansion technique, which within the contexts of process algebras has long been the primary technique to prove parallel behaviours equal. This technique has severe limitations if it comes to applicability, which we overcome in later chapters.

8.2 Derivation rules for equations

In table 8.1 we give rules that say how we can prove that two terms are equal. These rules may look like overkill, especially, because almost anywhere in this book we will use a less logical, more mathematical verbose style to prove equations. The reason for this is that proofs that strictly adhere to the rules in table 8.1 tend to become unreadable, and have no room for intuitive annotation.

But any of the proofs of equalities are ultimately based on the rules in table 8.1. That means, with some work, they can be completely translated into a derivation sequence consisting of these rules (and some others that we provide later). This is a fallback in those cases when it is unclear whether a verbose proof is actually valid. If the verbose proof can be translated to a logical proof tree, it is valid.

The rules in table 8.1 are a schemas. The letters p, q and r stand for expressions, x for a variable symbol, D is a sort symbol and σ is a substitution. The letter Γ stands for a context, which is a set of equations. This means that for each context, expression, variable symbol, sort and substitution, there is an instance of a rule. The substitutions in the rules are used to define the rules. They are not themselves part of any concrete instance of a rule.

Each rule must be read as follows. If the equation(s) above the line can be proven from the context, then we can prove the equation below the line in this context. In case the set of premisses above the line is empty, we have no proof obligation and can consider the conclusion proven straight away.

The context contains the sets of axioms and assumptions upon which we base our proof. In our case the axioms for strong bisimulation are always assumed to be part of the context, as well as the identities valid for the data types. Besides that we can have additional axioms valid for branching or trace equivalence, but this is always made explicitly clear.

$\dfrac{}{\Gamma \cup \{p = q\} \vdash p = q}$	start
$\dfrac{}{\Gamma \vdash p = p}$	reflexivity
$\dfrac{\Gamma \vdash p = q}{\Gamma \vdash q = p}$	symmetry
$\dfrac{\Gamma \vdash p = q \quad \Gamma \vdash q = r}{\Gamma \vdash p = r}$	transitivity
$\dfrac{\Gamma \vdash p_1 = q_1 \quad \Gamma \vdash p_2 = q_2}{\Gamma \vdash p_1\, p_2 = q_1\, q_2}$	congruence
$\dfrac{\Gamma \vdash p\, x = q\, x}{\Gamma \vdash p = q},\ x \notin \Gamma$	extensionality
$\dfrac{\Gamma \vdash p = q}{\Gamma \vdash \lambda x{:}D.p = \lambda x{:}D.q},\ x \notin \Gamma$	abstraction
$\dfrac{\Gamma \vdash p = q}{\Gamma \vdash \sigma(p) = \sigma(q)}$	substitution
$\dfrac{}{\Gamma \vdash (\lambda x{:}D.p)\, q = p[x := q]}$	beta conversion

Table 8.1: Derivation rules for equations

The assumptions can contain additional information, such as, for instance the definition of certain processes. E.g. we put an equation $X = a{\cdot}X$ in the context, effectively defining the process X. From this we may conclude that $X = a{\cdot}a{\cdot}X$ is provable, formally showing that

$$\{X = a{\cdot}X\} \vdash X = a{\cdot}a{\cdot}X.$$

In the next paragraphs we give the exact derivation of this.

The proof rules we provide are a mixture of equational logic and rules taken from typed lambda calculus [4]. Below we define exactly what it means for an expression $\Gamma \vdash p = q$ to be provable.

Definition 8.2.1. Let S be a set of basic sorts. Let $\mathcal{F}_S$ be a set of $\mathcal{S}_S$-typed function symbols and $\mathcal{X}_S$ be a set of $\mathcal{S}_S$-typed variable symbols. Let p, q be expressions and Γ be a context over $\mathcal{F}_S$ and $\mathcal{X}_S$. We inductively define that $\Gamma \vdash p = q$ is provable, iff $\Gamma \vdash p = q$ is the conclusion of some rule in table 8.1 and all premisses of this rule are provable. Generally, we write a proof as a sequence of formulas, such that each formula is provable by a rule of which the premises occur in the sequence earlier.

Example 8.2.2. As an example we show that $\{X = a{\cdot}X\} \vdash X = a{\cdot}a{\cdot}X$ is provable. We give the proof as a numbered list. First we provide the formula that is proven, and subsequently the name of the employed rule, followed by the indices of the premises that were used.

1. $\{X = a{\cdot}X\} \vdash X = a{\cdot}X$ by start;
2. $\{X = a{\cdot}X\} \vdash (\lambda y{:}\mathbb{P}.a{\cdot}y) = (\lambda y{:}\mathbb{P}.a{\cdot}y)$ by reflexivity;
3. $\{X = a{\cdot}X\} \vdash (\lambda y{:}\mathbb{P}.a{\cdot}y)\,(a{\cdot}X) = a{\cdot}a{\cdot}X$ by beta conversion;
4. $\{X = a{\cdot}X\} \vdash (\lambda y{:}\mathbb{P}.a{\cdot}y)\,X = (\lambda y{:}\mathbb{P}.a{\cdot}y)\,(a{\cdot}X)$ by congruence using 2 and 1;
5. $\{X = a{\cdot}X\} \vdash (\lambda y{:}\mathbb{P}.a{\cdot}y)\,X = a{\cdot}a{\cdot}X$ by transitivity using 4 and 3;
6. $\{X = a{\cdot}X\} \vdash (\lambda y{:}\mathbb{P}.a{\cdot}y)\,X = a{\cdot}X$ by beta conversion;
7. $\{X = a{\cdot}X\} \vdash a{\cdot}X = (\lambda y{:}\mathbb{P}.a{\cdot}y)\,X$ by symmetry using 6;
8. $\{X = a{\cdot}X\} \vdash a{\cdot}X = a{\cdot}a{\cdot}X$ by transitivity using 7 and 5;
9. $\{X = a{\cdot}X\} \vdash X = a{\cdot}a{\cdot}X$ by transitivity using 1 and 8.

Generally, a linear style for proofs is much more convenient. So, suppose we want to prove that $p_1 = q_n$ for expressions p and q ($n \geq 0$) in some context Γ. As Γ is generally quite stable, we do not mention it explicitly in the proof, but only indicate what is in Γ. We prove $p = q$ using a sequence of the form:

$$p_1 = p_2 = p_3 = \cdots = p_{n-1} = p_n.$$

Each equation $p_i = p_{i+1}$ consists of the application of one axiom or assumption in Γ. More concretely, let $p = q$ or $q = p$ be this axiom or assumption, then p_i has the form $r[x := \sigma(p)]$ and p_{i+1} has the form $r[x := \sigma(q)]$. We can derive that $p_i = p_{i+1}$ as follows where we assume $q = p$ is in Γ. The case where $p = q$ is in Γ is one step simpler.

1. $\Gamma \vdash (\lambda x{:}D).r\,\sigma(q) = r[x := \sigma(q)]$ by beta conversion;
2. $\Gamma \vdash q = p$ by start;
3. $\Gamma \vdash p = q$ by symmetry using 2;
4. $\Gamma \vdash \sigma(p) = \sigma(q)$ by substitution using 3;
5. $\Gamma \vdash (\lambda x{:}D).r = (\lambda x{:}D).r$ by reflexivity;
6. $\Gamma \vdash (\lambda x{:}D).r\,\sigma(p) = (\lambda x{:}D).r\,\sigma(q)$ by congruence using 5 and 4;

7. $\Gamma \vdash (\lambda x{:}D).r\,\sigma(p) = r[x := \sigma(q)]$ by transitivity using 6 and 1;
8. $\Gamma \vdash (\lambda x{:}D).r\,\sigma(p) = r[x := \sigma(p)]$ by beta reduction;
9. $\Gamma \vdash r[x := \sigma(p)] = (\lambda x{:}D).r\,\sigma(p)$ by symmetry using 8;
10. $\Gamma \vdash r[x := \sigma(p)] = r[x := \sigma(q)]$ by transitivity using 9 and 7.

As we know how to translate such a linear proof to a formal proof derivation, it suffices to give the linear proof. The essential ingredient is the axiom or assumption that has been used. As a service to those that check the proof, a hint is often written above the equation sign. This can be the name of the axiom, the indication that an assumption is used (e.g. using 'ass.') or the essential equation itself. Possible forms for the equation above would be:

$$p_i \overset{p=q}{=} p_{i+1} \qquad p_i \overset{\text{ass.}}{=} p_{i+1} \qquad p_i \overset{\text{AX}}{=} p_{i+1}$$

assuming AX is the name of the axiom $p = q$.

Example 8.2.3. We give a linear proof as a replacement of the proof of example 8.2.2. We assume that $X = a{\cdot}X$ (now it is in Γ):

$$X \overset{\text{ass.}}{=} a{\cdot}X \overset{\text{ass.}}{=} a{\cdot}a{\cdot}X.$$

Common mathematical style is to derive auxiliary theorems (also called lemmas) from some context. Subsequently, these theorems are added to the context as a help to prove subsequent theorems. This does not yield one proof, but two. The question is can we conclude that the subsequent theorems can be considered proven, using the original context. In our setting:

$$\left.\begin{array}{l}\Gamma \vdash p_1 = q_1 \text{ is provable} \\ \Gamma \cup \{p_1 = q_1\} \vdash p_2 = q_2 \text{ is provable}\end{array}\right\} \implies \Gamma \vdash p_2 = q_2 \text{ is provable.}$$

This is called the *cut theorem*, because it allows to cut proofs in smaller parts. We can easily see that the cut theorem is valid. Namely, because $\Gamma \vdash p_1 = q_1$ and $\Gamma \cup \{p_1 = q_1\} \vdash p_2 = q_2$ are provable, there are proof sequences for both. Now concatenate both proof sequences with the one for $\Gamma \vdash p_1 = q_1$ occurring first. Moreover, remove all occurrences of $\{p_1 = q_1\}$ from the contexts in the second proof sequence. Now the conclusion of the sequence is $\Gamma \vdash p_2 = q_2$ as we require. The total sequence is still a proof sequence except for applications of the start rule in the second sequence where $\Gamma \vdash p_1 = q_1$ should be proven. But as $p_1 = q_1$ is removed from the context this is not a valid application of start. Fortunately, $p_1 = q_1$ is already proven in the first sequence, and hence this application of the start rule in the proof can be omitted. So, we have a valid proof with as conclusion $\Gamma \vdash p_2 = q_2$.

Another useful mechanism is the *weakening lemma*. Suppose we can prove $\Gamma \vdash p = q$. Then we can also prove $\Gamma \cup \Delta \vdash p = q$. This is useful, because it allows to derive facts in their minimal context, and apply them in a much wider context. The construction of a proof sequence for $\Gamma \cup \Delta \vdash p = q$ out of $\Gamma \vdash p = q$ is straightforward. Just replace each context Γ by the context $\Gamma \cup \Delta$ and verify that application of each rule in table 8.1 is still valid. The only complexity is the extensionality rule as x may occur in Δ. But in this case we can replace x by a fresh variable y not occurring in the whole proof and copy the proof of $p\,x = q\,x$ to a proof of $p\,y = p\,y$ in which all occurrences of x are replaced by y. This new proof is put in front of the old proof sequence.

So, the cut theorem allows to formulate useful general lemmas and the weakening lemma allows to prove these in a minimal setting without hampering its applicability.

Two common rules from the lambda calculus are alpha-conversion and eta-conversion. All are derivable in the current setting. We start out with eta conversion. It says that $(\lambda x{:}D.px) = p$, namely the lambda abstraction to the left is superfluous. We can prove it in the empty context (so it can be applied everywhere).

1. $\emptyset \vdash (\lambda x{:}D.p\,x)\,y = p\,y$ by beta conversion;
2. $\emptyset \vdash (\lambda x{:}D.p\,x) = p$ by extensionality using 1;

Alpha conversion says that the name of a variable in a lambda abstraction can be renamed to a fresh one. I.e.:

$$\emptyset \vdash \lambda x{:}D.p = \lambda y{:}D.p[x := y].$$

The proof goes as follows. By eta conversion and the abstraction rule we know that $\lambda z{:}D_1{\to}D_2.z = \lambda z{:}D_1{\to}D_2.\lambda y{:}D_1.z\,y$. Using the congruence rule, we can apply both expressions to $\lambda x{:}D_1.p$. Using beta conversion we derive $\lambda x{:}D_1.p = \lambda y{:}D_1.p[x := y]$, provided y does not occur in p.

With the lambda notation available, it is possible to write the sum operator Σ using lambda notation. The sum operator is just an operator that works on functions from data to a process, and yields a process. I.e., $\sum_{d:D} p$ is a shorthand for $\sum \lambda d{:}D.p$.

So, in order to prove $\sum_{d:D} p = \sum_{d:D} q$ given that $p = q$ is proven, formally requires to prove $\lambda d{:}D.p = \lambda d{:}D.q$ using abstraction (so d must not occur in the context). Subsequently, a straightforward application of the congruence rule helps to prove $\sum_{d:D} p = \sum_{d:D} q$.

The following lemma provides a very useful fact. Even more importantly, the proof shows how to use the axioms to derive such facts.

Lemma 8.2.4 (Exchange of sum operators).

$$\sum_{d:D}\sum_{e:E} X(d,e) = \sum_{e:E}\sum_{d:D} X(d,e).$$

Proof. It is sufficient to prove this theorem to show that

$$\sum_{d:D}\sum_{e:E} X(d,e) \subseteq \sum_{e:E}\sum_{d:D} X(d,e) \tag{8.1}$$

because we can substitute $\lambda d{:}D, e{:}E.X(e,d)$ for X in (8.1). We obtain $\sum_{d:D}\sum_{e:E} X(e,d) \subseteq \sum_{e:E}\sum_{d:D} X(e,d)$. As we prove this for any sorts E and D, we can exchange D and E:

$$\sum_{e:E}\sum_{d:D} X(d,e) \subseteq \sum_{d:D}\sum_{e:E} X(d,e).$$

Together, with (8.1) this implies the theorem.

So, we must prove equation (8.1). Using SUM3 twice we derive:

$$\sum_{d:D}\sum_{e:E} X(d,e) \supseteq \sum_{e:E} X(d,e) \supseteq X(d,e).$$

Using abstraction and congruence for Σ we derive:

$$\sum_{e:E}\sum_{d:D}\sum_{d:D}\sum_{e:E} X(d,e) \supseteq \sum_{e:E}\sum_{d:D} X(d,e). \tag{8.2}$$

Using SUM1 we can derive

$$\sum_{e:E}\sum_{d:D}\sum_{d:D}\sum_{e:E} X(d,e) = \sum_{d:D}\sum_{e:E} X(d,e).$$

Together with (8.2), this proves (8.1) as desired. □

Exercise 8.2.5. Prove that $\sum_{d:D} X(d) = \sum_{e:E} X(e)$. This is alpha-conversion for the sum operator. In a setting without the lambda calculus, this equality is generally added to table 4.2 as an axiom under the name SUM2.

Exercise 8.2.6. The normal rule for congruence is the following. If $p_1 = q_1, \ldots, p_n = q_n$, then $f(p_1,\ldots,p_n) = f(q_1,\ldots,q_n)$ for some function symbol f. Show that this can be proven using the congruence rule in table 8.1.

Exercise 8.2.7. Derive $\sum_{e:E}\sum_{d:D}\sum_{d:D}\sum_{e:E} X(d,e) = \sum_{d:D}\sum_{e:E} X(d,e)$ using the rules of table 8.1.

8.3 Derivation rules for formulas

It is not sufficient to only derive equations between data and processes. Sometimes it is necessary to conclude that expressions are not equal, or that they are only equal under certain conditions. Therefore, we introduce *formulas* using which this can be expressed.

Definition 8.3.1. Let S be a set of basic sorts. We inductively define formulas (over S) as follows:

- If p and q are expressions of the same sort, then $p = q$ is a formula.;
- If ϕ and ψ are formulas, then $\phi \Rightarrow \psi$ is a formula;
- If ϕ is a formula and x is a variable symbol of sort D, then $\forall x{:}D.\phi$ is a formula.

We use the following abbreviations that allow us to use other connectives in formulas.

- $\neg\phi$ stands for $\phi \Rightarrow true = false$;
- $\phi \vee \psi$ stands for $\neg\phi \Rightarrow \psi$;
- $\phi \wedge \psi$ stands for $\neg(\phi \Rightarrow \neg\psi)$;
- $\exists x{:}D.\phi$ stands for $\neg\forall x{:}D.\neg\phi$.

The notion of substitution must also be extended to formulas.

Definition 8.3.2. We extend the definition of substitution σ to a formula ϕ as follows:

- $\sigma(\phi)$ equals $\sigma(p) = \sigma(q)$ if ϕ is $p = q$;
- $\sigma(\phi)$ equals $\sigma(\psi) \Rightarrow \sigma(\chi)$ if ϕ is $\psi \Rightarrow \chi$;
- $\sigma(\phi)$ equals $\forall y{:}D.\sigma(\phi[x{:=}y])$ if ϕ equals $\forall x{:}D.\phi$ and y is a fresh variable of sort D.

The deriviation rules for formulas are given in table 8.2.

8.4 Induction for constructor sorts

If D is a constructor sort, i.e., there is at least one declared constructor $f{:}D_1 \times \cdots \times D_n \to D$, then elements of D can all be written as an application of a constructor function to smaller elements. This yields a well known proof principle, namely induction, also referred to as term induction. The idea is that a formula ϕ can be proven for all elements of a constructor sort D if it can be proven for all expressions of the form $c_i(p_1, \ldots, p_{n_i})$ where c_i is a constructor and n_i the arity of the constructor. Moreover, as those expressions p_i of sort D are smaller than $c_i(p_1, \ldots, p_{n_i})$, the formula ϕ can be used with p_i to prove ϕ with $c_i(p_1, \ldots, p_{n_i})$. The following induction rule summarises this, where variables x_i are taken, instead of expressions p_i. Here, D_i is the sort of variable x_i.

$$\frac{\{\Gamma \vdash \{ \bigwedge_{D_i = D} \phi[x := x_i] \Rightarrow \phi[x{:=}c_i(x_1, \ldots, x_{n_i})] \mid c_i \text{ is a constructor of } D\}}{\Gamma \vdash \phi} \text{ induction}$$

As an example we show how this general rule looks like if the constructor sort is $\mathbb{B}$. There are two constructors $true$ and $false$, which do not have arguments. The induction rule looks like:

$$\frac{\Gamma \vdash \phi[x{:=}true] \quad \Gamma \vdash \phi[x{:=}false]}{\Gamma \vdash \phi} \text{ induction on } \mathbb{B}$$

$\dfrac{}{\Gamma \cup \{\phi\} \vdash \phi}$	generalised start
$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \sigma(\phi)}$	generalised substitution
$\dfrac{\Gamma \vdash true = false}{\Gamma \vdash \phi}$	ex falso sequitur quod libet
$\dfrac{\Gamma \cup \{\phi\} \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$	$\Rightarrow$ introduction
$\dfrac{\Gamma \vdash \phi \Rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$	$\Rightarrow$ elimination, or modus ponens
$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \forall x{:}D.\phi}, x \notin \Gamma$	$\forall$ introduction
$\dfrac{\Gamma \vdash \forall x{:}D.\phi}{\Gamma \vdash \phi[x := p]}$	$\forall$ elimination

Table 8.2: Derivation rules for formulas

Using this rule we can prove $c \to p \diamond p = p$ with induction on c. Induction on booleans says that we must prove $true \to p \diamond p = p$ and $false \to p \diamond p = p$. By applying process axioms Cond1 and Cond2 we can easily show both equations to hold.

A useful process identity is $x{\approx}y{\to}Z(x) = x{\approx}y{\to}Z(y)$. Here, x and y are variables over some sort D and Z is a variable ranging over functions from $D{\to}\mathbb{P}$.

In order to prove this, we first prove the auxiliary

$$(c = true \Rightarrow x = y) \Rightarrow c{\to}x = c{\to}y. \tag{8.3}$$

Note that x and y are of sort $\mathbb{P}$ and c is of sort $\mathbb{B}$. The equation is proven with induction on $\mathbb{B}$. So, we need to show:

1. $(true = true \Rightarrow x = y) \Rightarrow true{\to}x = true{\to}y$, and
2. $(false = true \Rightarrow x = y) \Rightarrow false{\to}x = false{\to}y$, and

In the first case, using axiom Cond1 we must prove $x = y$ from $true = true{\Rightarrow}x = y$. As $true = true$ is always valid (by reflexivity), this is trivial. In the second case, using axiom T2 and Cond2 it is necessary to show $\delta{\cdot}0 = \delta{\cdot}0$, which also follows directly from reflexivity.

From equation 8.3 we derive the following by taking $x{\approx}y$ for c and $Z(x)$ and $Z(y)$ for x and y.

$$(x{\approx}y = true \Rightarrow Z(x) = Z(y)) \Rightarrow x{\approx}y{\to}Z(x) = x{\approx}y{\to}Z(y). \tag{8.4}$$

Now, using Bergstra's axiom, $x{\approx}y = true$ implies that $x = y$. So, using congruence, it follows that $Z(x) = Z(y)$. Hence, using (8.4) we can conclude $x{\approx}y{\to}Z(x) = x{\approx}y{\to}Z(y)$.

The constructors for lists are the empty list $[]$ and list prefix $d{\triangleright}l$. (see appendix A). Using the induction principle this immediately gives the following concrete induction rule:

$$\frac{\Gamma \vdash \phi[x{:=}[]] \quad \Gamma \vdash \phi[x{:=}l] \;\Rightarrow\; \phi[x{:=}d \triangleright l]}{\Gamma \vdash \phi} \quad \text{induction on } List(D)$$

Note that the reals, sets and bags do not have constructors and therefore do not have an induction principle.

Exercise 8.4.1. Give a precise formal derivation of the following formula, which is needed for the proof of formula (8.3).

$$\Gamma \vdash (true = true \Rightarrow x = y) \Rightarrow true{\rightarrow}x = true{\rightarrow}y.$$

Exercise 8.4.2. Prove that

- $c_1{\rightarrow}c_2{\rightarrow}x = (c_1 \wedge c_2){\rightarrow}x$.

Exercise 8.4.3. Formulate the induction principles for positive numbers, natural numbers and integers, based on the definition in appendix A.

Exercise 8.4.4. Standard induction on the positive numbers is given by the rule:

$$\frac{\Gamma \vdash \phi(1) \quad \Gamma \vdash \phi(n) \Rightarrow \phi(n{+}1)}{\Gamma \vdash \phi(n)}.$$

The constructors in appendix A suggests the induction principle

$$\frac{\Gamma \vdash \phi(1) \quad \Gamma \vdash \phi(n) \Rightarrow \phi(2{*}n) \quad \Gamma \vdash \phi(n) \Rightarrow \phi(2{*}n{+}1)}{\Gamma \vdash \phi(n)}.$$

Show that both induction rules are equivalent.

8.5 The sum elimination lemma

Using the material provided up till, we can prove the following important sum elimination lemma. This lemma is not only applied very frequently when calculating with parallel processes that exchange data, but its proof is also typical for many proofs involving the sum operator.

Lemma 8.5.1 (Sum elimination).

$$\sum_{d:D} d{\approx}e{\rightarrow}X(d) = X(e).$$

Proof. We split the proof in two cases:

$\supseteq$. We must show that $X(e) \subseteq \sum_{d:D} d{\approx}e{\rightarrow}X(d)$. This follows by a direct application of SUM3, which yields

$$e{\approx}e{\rightarrow}X(e) \subseteq \sum_{d:D} d{\approx}e{\rightarrow}X(d).$$

Using that $e{\approx}e = true$ and axiom Cond1 yields the result.

$\subseteq$. We must show that $\sum_{d:D} d{\approx}e{\rightarrow}X(d) \subseteq X(e)$. So, $X(e) = d{\approx}e{\rightarrow}X(e) + X(e)$. By using the equation derived above, we can show this equal to $X(e) = d{\approx}e{\rightarrow}X(d) + X(e)$. Using the rules for abstraction and congruence it follows that $\sum_{d:D} X(e) = \sum_{d:D}(d{\approx}e{\rightarrow}X(d) + X(e))$. Note that this is allowed, because there are no assumptions on d; d does not occur in the context. By applying SUM1 (twice) and SUM4 we get:

$$X(e) = \sum_{d:D} d{\approx}e{\rightarrow}X(d) + X(e).$$

This is an alternative formulation of $\sum_{d:D} d{\approx}e{\rightarrow}X(d) \subseteq X(e)$ which we had to prove.

□

Exercise 8.5.2. Prove $x = \sum_{t:\mathbb{R}}(t > 0) \to x \lhd t$.

Exercise 8.5.3. Prove that

$$\sum_{b:\mathbb{B}} b \to x \diamond y = x + y.$$

Exercise 8.5.4. Show for arbitrary sort D, variables $c : D \to \mathbb{B}$ and $x : \mathbb{P}$.

$$(\exists d{:}D.c(d) = \mathit{true}) \;\Rightarrow\; x = \sum_{d:D} c(d) \to x.$$

Exercise 8.5.5. Prove that

$$\sum_{n:\mathbb{N}} n \leq 2 \to X(n) = X(0) + X(1) + X(2).$$

8.6 Recursive specification principle

In this section we explain the recursive specification principle (RSP), which allows to prove that recursive processes are equal. Before doing so, we first introduce process operators.

A process operator maps a process to a process. We typically use letters Φ and Ψ for process operators. So, if we define $\Psi = \lambda X{:}\mathbb{P}.a \cdot X$, then $\Psi\,(b + c \cdot d)$ is the process $a \cdot (b + c \cdot d)$.

Process operators can also operate on processes with data. E.g. the process operator $\Psi' = \lambda X{:}\mathbb{N}\to\mathbb{P}.\lambda n{:}\mathbb{N}.a \cdot X(n{+}1)$ maps a process X which depends on a natural number to another process that depends on such a number. The type of Ψ' is $(\mathbb{N}\to\mathbb{P})\to(\mathbb{N}\to\mathbb{P})$.

Process operators can even be applied to processes with more than one data parameter. E.g. the following operator works on processes with a parameter of sort $\mathbb{B}$ and one of sort $\mathbb{N}$: $\Psi'' = \lambda X{:}\mathbb{B} \times \mathbb{N}\to\mathbb{P}.\lambda b{:}\mathbb{B}, n{:}\mathbb{N}.b \to \mathit{up} \cdot X(\mathit{true}, n{+}1) \diamond \mathit{twice} \cdot X(\mathit{false}, 2{*}n)$ where up and twice are actions. The type of Ψ'' is $(\mathbb{B}\to\mathbb{N}\to\mathbb{P}) \to (\mathbb{B}\to\mathbb{N}\to\mathbb{P})$.

Process operators have a direct link with equations. If Φ is a process operator, then $X = \Phi\, X$ is the equation associated to it. And vice versa, if $X(d_1{:}D_1, \ldots, d_n{:}D_n) = p$ is an equation, then the related process operator is $\lambda X{:}D_1 \times \cdots \times D_n, d_1{:}D_1, \ldots, d_n{:}D_n.p$ is the associated operator. So, for the operators Ψ, Ψ' and Ψ'' above the associated equations are $X = a \cdot X$, $X(n{:}\mathbb{N}) = a \cdot X(n{+}1)$ and $X(b{:}\mathbb{B}, n{:}\mathbb{N}) = b \to \mathit{up} \cdot X(\mathit{true}, n{+}1) \diamond \mathit{twice} \cdot X(\mathit{false}, 2{*}n)$.

In chapter 4 we indicated that guarded recursive equations, such as $X = a \cdot X$, define a process because there is only one process that is a solution of this equation, namely the process that can do a actions infinitely. The essential ingredient of this equation is that at the left hand X only occurs once and at the right hand side X only occurs in a guarded position. We make this last notion precise. In the definition below, we assume X has one argument of sort D, in line with our habit to provide formal definitions only for processes with one argument.

Definition 8.6.1 (Guardedness). We inductively define that a process variable $X{:}D\to\mathbb{P}$ is *guarded* in a process p whenever

- if p equals δ or a non-empty multi-action, then X occurs guarded in p;
- if $p = p_1 + p_2$, $p = p_1 \parallel p_2$, $p = p_1 \mid p_2$, $p = c \to p_1 \diamond p_2$ $p = p_1 \ll p_2$ then X occurs guarded in p if X occurs guarded in p_1 and p_2;
- if $p = \sum_{d:D} p_1$, $p = p_1 \lhd t$, $c \to p_1$, $t \gg p_1$, $p = \Gamma_C(p_1)$, $\nabla_V(p_1)$, $\partial_B(p_1)$ or $\rho_R(p_1)$, then X occurs guarded in p iff it occurs guarded in p_1;
- if $p = p_1 \cdot p_2$ or $p = p_1 \between p_2$ then X occurs guarded in p if X occurs guarded in p_1.

The recursive specification principle says that in any guarded recursive equation, i.e. an equation of the form $X = p$ where X occurs guarded in p, there is at most one process that is a solution for X. In terms of a process operator, if $\Phi = \lambda X{:}D{\to}\mathbb{P}, d{:}D.p$ and X occurs guarded in p, then we say that Φ is a guarded process operator and Φ has only one fixed point. So, there is only one process Y satisfying $Y = \Phi Y$.

The following derivation rule compactly characterises the uniqueness of solutions:

$$\frac{\Gamma \vdash X = \Psi X \quad \Gamma \vdash Y = \Psi Y}{\Gamma \vdash X = Y} \quad \Psi \text{ guarded process operator } (RSP)$$

Example 8.6.2. As a simple example, consider the following definitions for X and Y.

proc $X = a{\cdot}X;$
$Y = a{\cdot}a{\cdot}Y;$

Both X and Y are fixed points of the guarded process operator $\Phi = \lambda Z{:}\mathbb{P}.a{\cdot}a{\cdot}Z$. In order to prove this, we must show $X = \Phi\, X$ and $Y = \Phi Y$. Or in other words, $X = a{\cdot}a{\cdot}X$ and $Y = a{\cdot}a{\cdot}Y$. It is easy to derive that $X = a{\cdot}X = a{\cdot}a{\cdot}X$ using two applications of the defining equation for X. The defining equation for Y already says that $Y = a{\cdot}a{\cdot}Y$. So, we can apply the rule RSP, and conclude that X and Y are equal, i.e. $X = Y$.

Example 8.6.3. A slightly more elaborate example is the following. Assume that the following equations define X and Y.

proc $X(b{:}\mathbb{B}) = a(b){\cdot}X(\neg b);$
$Y(n{:}\mathbb{N}) = a(n|_2{\approx}0){\cdot}Y(n{+}1);$

The first process clearly has behaviour $\ldots a(\mathit{true})\, a(\mathit{false})\, a(\mathit{true})\, a(\mathit{false}) \ldots$. The second process has the same behaviour. It indicates whether n is even, by calculating whether n modulo 2 equals 0. Subsequently, n is increased by one. We show that $X(\mathit{true}) = Y(0)$.

In order to show that $X(\mathit{true}) = Y(0)$ we must show a more general equation from which it follows directly. Finding such a more general statement is often one of the more difficult steps in a proof and is called *property lifting*. In this concrete case we show that $X(n|_2{\approx}0) = Y(n)$ for $n{:}\mathbb{N}$. By taking $n = 0$ the desired result follows.

We prove $X(n|_2{\approx}0) = Y(n)$ by showing that both sides of the equation are a fixed point of the guarded process operator

$$\Psi = \lambda Z{:}\mathbb{N} \to \mathbb{P}, n{:}\mathbb{N}.a(n|_2{\approx}0){\cdot}Z(n{+}1).$$

A problem is that both sides of the equation are not yet of the right type to apply the operator Ψ to it. Therefore we apply lambda abstraction to both sides and get $\lambda n{:}\mathbb{N}.X(n|_2{\approx}0) = Y$. To prove that both sides are a fixed point of the operator we must show (we apply both sides to n, to get more decent equations):

1. For the left hand side we get:

   $$\lambda n{:}\mathbb{N}.X(n|_2{\approx}0)(n) = (\lambda Z{:}\mathbb{N} \to \mathbb{P}, m{:}\mathbb{N}.a(m|_2{\approx}0){\cdot}Z(m{+}1))(\lambda n{:}\mathbb{N}.X(n|_2{\approx}0))(n).$$

   This reduces by beta conversion to

   $$X(n|_2{\approx}0) = a(n|_2{\approx}0){\cdot}X(n{+}1|_2{\approx}0).$$

   Using that $n{+}1|_2{\approx}0 = \neg(n|_2{\approx}0)$ we must show that

   $$X(n|_2{\approx}0) = a(n|_2{\approx}0){\cdot}X(\neg(n|_2{\approx}0)).$$

   If we now look at the definition for the process X, we see that $X(b{:}\mathbb{B}) = a(b){\cdot}X(\neg b)$ for any boolean b. So, in particular for $b = n|_2{\approx}0$. This yields exactly the last equation that we had to prove.

2. For the right hand side we must show that

$$Y(n) = (\lambda Z{:}\mathbb{N} \to \mathbb{P}, m{:}\mathbb{N}.a(m|_2{\approx}0){\cdot}Z(m{+}1))\, Y\, n.$$

By beta conversion this is equivalent to

$$Y(n) = a(n|_2){\cdot}Y(n{+}1).$$

This is exactly equal to the equation defining Y. So, this equation holds also.

So, we have shown using RSP that $X(n|_2{\approx}0) = Y(n)$, as required.

It is not always straightforward to find the right guarded process operator. A good guideline is to take the process with the largest state space and transform that one into the operator.

There is another principle which is called the recursive definition principle (RDP). It says that any recursive equation has a solution. We use this principle implicitly, by acting as if X refers to an existing process if it is defined by an equation of the form $X = p$ where X can occur in p. We do not pay further attention to this.

A derived rule is RSP for a set of guarded recursive equations, also called a guarded recursive specification. So, consider the sequence of process operators $\Phi_1, \ldots, \Phi_n$. Assume that $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ are both fixed points of Φ_i, all with type $D{\to}\mathbb{P}$. If the types of the variables are not equal, it is a simple operation to generalise the the types of the variables to harmonize them. Then we can conclude that $X_1 = Y_1, \ldots, X_n = Y_n$. As an inference rule:

$$\frac{\begin{array}{c}\Gamma \vdash X_1 = \Phi_1 X_1,\ \ldots,\ \Gamma \vdash X_n = \Phi_n X_n \\ \Gamma \vdash Y_1 = \Phi_1 Y_1,\ \ldots,\ \Gamma \vdash Y_n = \Phi_n Y_n\end{array}}{\Gamma \vdash X_1 = Y_1, \ldots, \Gamma \vdash X_n = Y_n}.$$

We prove this rule from RSP as follows. Define the operator Ψ by

$$\Psi = \lambda Z{:}D \to \mathbb{N}, d{:}D, i{:}\mathbb{N}.(i{\approx}1){\to}\Phi(d,1) \diamond (i{\approx}2){\to}\Phi(d,2) \diamond \ldots (i{\approx}n{-}1)P{\to}\Phi(d,n{-}1) \diamond \phi(d,n).$$

Now define the two processes

$$X(d{:}D, i{:}\mathbb{N}) = (i{\approx}1) \to X_1(d) \diamond \cdots (i{\approx}n{-}1){\to}X_{n-1}1(n) \diamond X_n(d) \text{ and}$$
$$Y(d{:}D, i{:}\mathbb{N}) = (i{\approx}1) \to Y_1(d) \diamond \cdots (i{\approx}n{-}1){\to}Y_{n-1}1(n) \diamond Y_n(d).$$

It is now easy to show that X and Y are fixed points of Ψ. So, $X(d,i) = Y(d,i)$. In particular $X_i(d) = X(d,i) = Y(d,i) = Y_i(d)$, which we had to prove.

Exercise 8.6.4. Prove that $X = Y$, where X and Y are given by $X = a{\cdot}X$ and $Y = a{\cdot}a{\cdot}a{\cdot}Y + a{\cdot}a{\cdot}Y$.

Exercise 8.6.5. Prove using RSP that the processes X and Y defined by

proc $X = a{\cdot}X;$
$Y = a{\cdot}Y{\cdot}\delta;$

are equal.

Exercise 8.6.6. Show that $X(0) = Y([])$ for the following two processes

proc $X(n{:}\mathbb{N}) = a(n){\cdot}X(n{+}1);$
$Y(l{:}List(\mathbb{N})) = \sum_{d:\mathbb{N}} a(\#l){\cdot}Y(d{\triangleright}l)$

Exercise 8.6.7. Show that $X(0) = Y(0)$ where $N{:}\mathbb{N}^+$ for the following two processes

proc $X(n{:}\mathbb{N}) = a(n){\cdot}X((n{+}1)|_N);$
$Y(m{:}\mathbb{N}) = a(m|_N){\cdot}Y(m{+}1);$

Exercise 8.6.8. Consider the process $X = a{\cdot}b{\cdot}c{\cdot}X$ and the process $Y(n{:}\mathbb{N}^+) = n{\approx}1{\to}a{\cdot}Y(2) + n{\approx}2{\to}b{\cdot}Y(3) + n{\approx}3{\to}c{\cdot}Y(1)$. Prove using RSP that $X = Y(1)$.

Exercise 8.6.9. Consider the following processes:

proc $X = a{\cdot}\nabla_{\{a,b\}}(b\|X);$
$Y(n{:}\mathbb{N}) = a{\cdot}Y(n{+}1) + n{>}0{\to}b{\cdot}Y(n{-}1);$

Show that $X = Y(0)$ using RSP.

8.7 Koomen's fair abstraction rule

Most processes can perform many different interactions simultaneously, and we only want to study some of them. For instance, a buffer can forward data from gate 1 to 2, or vice versa, from gate 2 to gate 1. In a picture and as a process:

proc $Buffer = read_1 \cdot deliver_2 \cdot Buffer + read_2 \cdot deliver_1 \cdot Buffer;$

We may be interested in studying the data transfer from gate 1 to gate 2, hiding the transfer of data from 2 to 1, under that assumption that the amount of data being transferred from 2 to 1 is not so large that it prevents data to be transferred in the other direction. In other words we assume that data transfer from 2 to 1 *fair*, in the sense that it allows other events to happen also.

So, we expect that $\tau_{\{deliver_1, read_2\}}(Buffer)$ to satisfy the equation

$$\tau_{\{deliver_1, read_2\}}(Buffer) = read_1 \cdot deliver_2 \cdot \tau_{\{deliver_1, read_2\}}(Buffer).$$

What we can prove however, by directly applying the axioms for hiding is:

$$\tau_{\{deliver_1, read_2\}}(Buffer) = read_1 \cdot deliver_2 \cdot \tau_{\{deliver_1, read_2\}}(Buffer) + \tau \cdot \tau \cdot \tau_{\{deliver_1, read_2\}}(Buffer).$$

With the rules given hitherto, we will not be able to prove both equations equal. We first introduce a rule that can do the job, and will come back to this example after that.

The principle *Koomens fair abstraction rule* is suited for the job. It is valid in branching bisimulation, not in strong bisimulation, and it has the following form:

$$\frac{\Gamma \vdash X = i \cdot X + Y}{\Gamma \vdash \tau \cdot \tau_{\{i\}}(X) = \tau \cdot \tau_{\{i\}}(Y)} \quad \text{KFAR (branching bisimulation)}$$

It says that in a process X, where an action i can be done infinitely often, it will not block behaviour in Y. In other words, i is a fair action. So, if i is hidden, the behaviour of Y is still fully visible, except of course that the action i is also hidden in Y. The KFAR rule valid in weak bisimulation is slightly simpler. It does not have a τ at the left hand side of the conclusion of the rule. So its conclusion has the form $\Gamma \vdash \tau_{\{i\}}(X) = \tau \cdot \tau_{\{i\}}(Y)$.

The formulation of Koomen's fair abstraction rule looks somewhat complex, and tends to stimulate to investigate the following simplified formulation, which is wrong:

$$\frac{\Gamma \vdash X = \tau \cdot X + Y}{\Gamma \vdash X = \tau \cdot Y}$$

This can be seen as follows. Consider the process $x = \tau \cdot y$ where y can be any process. Using axioms T1 and B1 it follows that $x = \tau \cdot x + \delta \triangleleft 0$. Using the faulty version of KFAR, we can conclude that $x = \tau \cdot \delta \triangleleft 0$. Or in other words, any process of the form $\tau \cdot y$ equals $\tau \cdot \delta \triangleleft 0$. This is clearly undesirable.

If we consider the example we can prove easily using the axioms for hiding and the first tau law that:

$$\tau_{\{deliver_1\}}(Buffer) = read_1 \cdot deliver_2 \cdot \tau_{\{deliver_1\}}(Buffer) + read_2 \cdot \tau_{\{deliver_1\}}(Buffer).$$

If we let $deliver_1$ match i and $\tau_{\{deliver_1\}}(Buffer)$ match X in the KFAR rule, then we can conclude from it that:

$$\tau \cdot \tau_{\{read_2, deliver_1\}}(Buffer) = \tau \cdot read_1 \cdot deliver_2 \cdot \tau_{\{deliver_1, read_2\}}(Buffer).$$

Except for initial internal steps this is what we would like to prove. It is not possible to remove the initial τ's, as they reflect that initially some hidden messages travel from gate 2 to 1.

For completeness, we also formulate the Koomen's Fair Abstraction Rule for weak bisimulation. This differs in the sense that there is no initial τ at the left hand side of the conclusion.

$$\frac{\Gamma \vdash X = i{\cdot}X + Y}{\Gamma \vdash \tau_{\{i\}}(X) = \tau{\cdot}\tau_{\{i\}}(Y)} \qquad \text{KFAR (weak bisimulation)}$$

Exercise 8.7.1. Define $X = head{\cdot}X + tail$. Prove using KFAR that $\tau{\cdot}\tau_{\{head\}}(X) = \tau{\cdot}tail$.

Exercise 8.7.2. Consider the process $X_1 = i{\cdot}X_2 + a_1$, $X_2 = i{\cdot}X_1 + a_2$. Prove using KFAR for weak bisimulation that $\tau\{i\}(X_1) = \tau{\cdot}(a_1{+}a_2)$. Also show that this cannot be proven using KFAR for branching bisimulation. This is a difficult exercise.

8.8 Parallel expansion

Up till now we have not really discussed how to calculate with parallel processes. But real difficulties with behaviour come with parallelism. There are so many actions that can take place in so many different orders, that it is impossible to understand what is happening by just imagining the traces.

The universal technique that we apply to understand parallel processes, is by eliminating the parallel operator and by translating it to sequential behaviour. In this section we explain how this can be done, directly using the axioms. This technique is very common in process algebra, and generally called *parallel expansion*. We also show how to prove the parallel processes equal to simplified counterparts, where internal actions are hidden.

Parallel expansion falls short if processes become more complex. Therefore, in the next chapter, we explain linearisation, as an advanced form of parallel expansion.

8.8.1 Basic parallel expansion

The description of a system S of n communicating parallel components $X_1, \ldots, X_n$ is generally as follows:

$$S = \nabla_V(\Gamma_\Gamma(X_1 \parallel \cdots \parallel X_n))$$

where the behaviour of the individual components X_i are given by a guarded recursive specification. By systematically applying the axioms in table 4.12 to eliminate the parallel operator, the axioms in table 4.13 to eliminate the communication operator and 4.14 to remove the allow operator, a set of simple process equations results. The axioms are such that they can strictly be used from left to right.

As an example consider the system $S = \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(X_1 \parallel X_2))$ where $X_1 = a{\cdot}X_1$ and $X_2 = b{\cdot}c{\cdot}X_2$. Applying the axioms yields:

$$\begin{array}{lll}
S & \overset{\text{def}}{=} & \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(X_1 \parallel X_2)) \\
& \overset{\text{def}}{=} & \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(a{\cdot}X_1 \parallel b{\cdot}c{\cdot}X_2)) \\
& \overset{\text{M}}{=} & \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(a{\cdot}X_1 \between b{\cdot}c{\cdot}X_2 + b{\cdot}c{\cdot}X_2 \between a{\cdot}X_1 + a{\cdot}X_1|b{\cdot}c{\cdot}X_2)) \\
& \overset{\text{LM3,S6}}{=} & \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}((a{\ll}b{\cdot}c{\cdot}X_2){\cdot}(X_1\|b{\cdot}c{\cdot}X_2){+}(b{\ll}a{\cdot}X_1){\cdot}(c{\cdot}X_2\|a{\cdot}X_1){+}(a|b){\cdot}(X_1\|c{\cdot}X_2))) \\
& \overset{\text{TB1,TB5}}{=} & \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(a{\cdot}(X_1 \parallel b{\cdot}c{\cdot}X_2){+}b{\cdot}(c{\cdot}X_2 \parallel a{\cdot}X_1){+}(a|b){\cdot}(X_1\|c{\cdot}X_2))) \\
& \overset{\text{C1,C3,C4,}}{\underset{\text{V1,V2,V4,V6}}{=}} & d{\cdot}\nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(X_1\|c{\cdot}X_2))).
\end{array}$$

At such a state it is handy to introduce a new name for the expressions after the initial actions. In this case, it is only one. So, we define:

$$S_1 = \nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(X_1\|c{\cdot}X_2))).$$

We can apply the same expansion technique and derive

$$S_1 = c{\cdot}\nabla_{\{c,d\}}(\Gamma_{\{a|b\to d\}}(X_1\|X_2)).$$

In other words we have shown that S satisfies the following recursive specification:

$$S = d{\cdot}S_1, \qquad S_1 = c{\cdot}S.$$

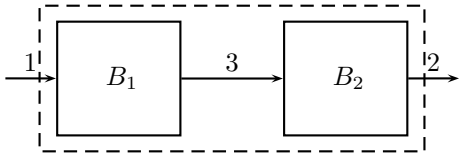
Exercise 8.8.1. Take X_1 and X_2 as above and calculate the parallel expansion for

$$\nabla_{\{a,c,d\}}(\Gamma_{\{a|b\to d\}}(X_1 \parallel X_2)).$$

8.8.2 Parallel expansion with data: two one-bit buffers

Two actions can only communicate if their data are the same. In this section we carry out a parallel expansion of two sequentially connected one place buffers. This shows how expansion of actions with data is carried out. In particular we see the sum elimination theorem in action.

So, we consider two buffers of capacity one that are put in sequence: buffer B_1 reads a datum from a channel 1 and sends this datum into channel 3, while buffer B_2 reads a datum from a channel 3 and sends this datum into channel 2. This system can be depicted as follows:



Let D denote a data domain. Action $r_i(d)$ represents reading datum d from channel i, while action $s_i(d)$ represents sending datum d into channel i. Moreover, action $c_3(d)$ denotes communication of datum d through channel 3. We let actions r_3 and s_3 communicate to c_3 and only allow actions r_1, s_2 and c_3. On top of this we hide c_3. So, action c_3 can happen, but we cannot observe it directly.

The buffers B_1 and B_2 are defined by the process declaration

proc $\quad B_1 = \sum_{d:D} r_1(d)\cdot s_3(d)\cdot B_1;$
$\qquad\quad B_2 = \sum_{d:D} r_3(d)\cdot s_2(d)\cdot B_2;$

The behaviour of the whole system is characterised by the process X consisting of the buffers B_1 and B_2 in parallel:

proc $\quad X = \tau_{\{c_3\}}(\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel B_2)));$

We show that X behaves as a queue of capacity two, which can read two data elements from channel 1 before sending them in the same order into channel 2. I.e., X satisfies the following equations:

$$\begin{array}{l} X = \sum_{d:D} r_1(d)\cdot Y(d); \\ Y(d{:}D) = \sum_{d':D} r_1(d')\cdot Z(d,d') + s_2(d)\cdot X; \\ Z(d,d'{:}D) = s_2(d)\cdot Y(d'). \end{array} \tag{8.5}$$

In state X, the queue of capacity two is empty, so that it can only read a datum d from channel 1 and proceed to the state $Y(d)$ where the queue contains d. In $Y(d)$, the queue can either read a second datum d' from channel 1 and proceed to the state $Z(d,d')$ where the queue contains d and d', or send datum d into channel 2 and proceed to the state X where the queue is empty. Finally, in state $Z(d,d')$ the queue is full, so that it can only send datum d into channel 2 and proceed to the state $Y(d')$ where it contains d'.

Below we provide the derivation in substantial detail. In each derivation step, the subexpresions that

are reduced are underlined. First we expand $B_1 \parallel B_2$:

$$\begin{array}{lll}
\underline{B_1 \parallel B_2} & \overset{\text{M}}{=} & \underline{B_1 \between B_2} + \underline{B_2 \between B_1} + \underline{B_1 | B_2} \\
& \overset{\text{def}}{=} & \underline{(\sum_{d:D} r_1(d)\cdot s_3(d)\cdot B_1) \between B_2} \\
& + & \underline{(\sum_{d:D} r_3(d)\cdot s_2(d)\cdot B_2) \between B_1} \\
& + & \underline{(\sum_{d:D} r_1(d)\cdot s_3(d)\cdot B_1) | (\sum_{d':D} r_3(d')\cdot s_2(d')\cdot B_2)} \\
& \overset{\text{LM5,S8}}{=} & \sum_{d:D} \underline{(r_1(d)\cdot s_3(d)\cdot B_1) \between B_2} \\
& + & \sum_{d:D} \underline{(r_3(d)\cdot s_2(d)\cdot B_2) \between B_1} \\
& + & \sum_{d:D} \sum_{d':D} \underline{(r_1(d)\cdot s_3(d)\cdot B_1) | (r_3(d')\cdot s_2(d')\cdot B_2)} \\
& \overset{\text{LM3,S6}}{=} & \sum_{d:D} \underline{(r_1(d) \ll B_2)}\cdot((s_3(d)\cdot B_1) \parallel B_2) \\
& + & \sum_{d:D} \underline{(r_3(d) \ll B_1)}\cdot((s_2(d)\cdot B_2) \parallel B_1) \\
& + & \sum_{d:D} \sum_{d':D} (r_1(d) | r_3(d'))\cdot((s_3(d)\cdot B_1) \parallel (s_2(d')\cdot B_2)) \\
& \overset{\text{def,TB5,TB1}}{=} & \sum_{d:D} r_1(d)\cdot(s_3(d)\cdot B_1 \parallel B_2) \\
& + & \sum_{d:D} r_3(d)\cdot(s_2(d)\cdot B_2 \parallel B_1) \\
& + & \sum_{d:D} \sum_{d':D} (r_1(d) | r_3(d'))\cdot((s_3(d)\cdot B_1) \parallel (s_2(d')\cdot B_2)).
\end{array}$$

For convenience, we define the process $X_1 = \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(\underline{B_1 \parallel B_2}))$. We now apply the allow and communication operator:

$$\begin{array}{lll}
X_1 & = & \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel B_2)) \\
& = & \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(\underline{\sum_{d:D} r_1(d)\cdot(s_3(d)\cdot B_1) \parallel B_2} + \sum_{d:D} r_3(d)\cdot(s_2(d)\cdot B_2) \parallel B_1) + \\
& & \qquad + \underline{\sum_{d:D} \sum_{d':D} (r_1(d) | r_3(d'))\cdot(s_3(d)\cdot B_1 \parallel s_2(d')\cdot B_2)}) \\
& \overset{\text{C3,C5,V3,V6}}{=} & \sum_{d:D} \underline{\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(r_1(d)\cdot(s_3(d)\cdot B_1 \parallel B_2)))} \\
& + & \sum_{d:D} \underline{\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(r_3(d)\cdot(s_2(d)\cdot B_2 \parallel B_1)))} \\
& + & \sum_{d:D} \underline{\sum_{d':D} \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}((r_1(d) | r_3(d'))\cdot(s_3(d)\cdot B_1 \parallel s_2(d')\cdot B_2)))} \\
& \overset{\text{C1,C4,V1,V2,V5}}{=} & \sum_{d:D} r_1(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d)\cdot B_1 \parallel B_2)) \underline{+\delta + \delta} \\
& \overset{\text{A6}}{=} & \sum_{d:D} r_1(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d)\cdot B_1 \parallel B_2)).
\end{array}$$

Summarising, we have derived

$$X_1 = \sum_{d:D} r_1(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d)\cdot B_1 \parallel B_2)). \tag{8.6}$$

We define a new process

$$X_2(d{:}D) = \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d)\cdot B_1 \parallel B_2))$$

and we proceed to expand $X_1(d)$. At the first equation we apply the same expansion as above, and block the single s_3 and r_3 actions. Only the communication between s_3 and r_3 survives.

$$X_2(d) \quad = \quad \sum_{d':D} \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d) | r_3(d'))\cdot(B_1 \parallel s_2(d')\cdot B_2))$$

Now, the communication operator is only effective if $d = d'$. So, we split the condition, and see that the right hand side of the last equation is equal to:

$$\sum_{d':D} \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(d{\approx}d' \to (s_3(d) | r_3(d'))\cdot(B_1 \parallel s_2(d') \diamond (s_3(d) | r_3(d'))\cdot(B_1 \parallel s_2(d')\cdot B_2)))).$$

Distribution of the allow and communication over the condition and the action yields:

$$\sum_{d':D} d{\approx}d' \to c_3(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel s_2(d')\cdot B_2)) \diamond \delta.$$

To this expression we can apply the sum elimination lemma (lemma 8.5.1) and obtain

$$c_3(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel s_2(d)\cdot B_2)) \diamond \delta.$$

So, summarising, we get

$$X_2(d{:}D) = c_3(d)\cdot\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel s_2(d)\cdot B_2)) \diamond \delta.$$

We can now repeat this process, by iteratively introducing new definitions and expanding these. This is a very mechanical procedure that quite often terminates. But if it does not, other, more intricate ways need to be found to eliminate the parallel operators. For the two parallel buffers we need the following auxiliary definitions:

$$X_3(d{:}D) = \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(B_1 \parallel s_2(d)\cdot B_2));$$
$$X_4(d,d'{:}D) = \nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(s_3(d)\cdot B_1 \parallel s_2(d')\cdot B_2)).$$

We get the following set of equations, including the two derived above:

$$X_1 = \textstyle\sum_{d:D} r_1(d)\cdot X_2(d);$$
$$X_2(d) = c_3(d)\cdot X_3(d);$$
$$X_3(d) = \textstyle\sum_{d':D}(r_1(d')\cdot X_4(d',d) + s_2(d)\cdot X_1);$$
$$X_4(d,d') = s_2(d')\cdot X_2(d).$$

The only task that remains to be done is to show that $\tau_{\{c_3\}}(X)$ is a solution for X in (8.5). We actually prove that $\tau_{\{c_3\}}(X_1)$ is a solution for X, $\lambda d{:}D.\tau_{\{c_3\}}(X_3(d))$ is a solution for Y and $\lambda d,d'{:}D.\tau_{\{c_3\}}(X_4(d',d))$ is a solution for Z in (8.5). This boils down to:

$$\tau_{\{c_3\}}(X_1) = \textstyle\sum_{d:D} r_1(d)\cdot\tau_{\{c_3\}}(X_3(d));$$
$$\tau_{\{c_3\}}(X_3(d)) = \textstyle\sum_{d':D} r_1(d')\cdot\tau_{\{c_3\}}(X_4(d',d)) + s_2(d)\cdot\tau_{\{c_3\}}(X_1);$$
$$\tau_{\{c_3\}}(X_4(d',d)) = s_2(d)\cdot\tau_{\{c_3\}}(X_3(d')).$$

The first equation follows directly using the equation for X_1, X_2 and the axioms, among which B1:

$$\tau_{\{c_3\}}(X_1) = \tau_{\{c_3\}}(\textstyle\sum_{d:D} r_1(d)\cdot X_2(d)) = \sum_{d:D} r_1(d)\cdot\tau_{\{c_3\}}(X_2(d)) =$$
$$\textstyle\sum_{d:D} r_1(d)\cdot\tau_{\{c_3\}}(c_3(d)\cdot X_3(d)) = \sum_{d:D} r_1(d)\cdot\tau\cdot\tau_{\{c_3\}}(X_3(d)) \overset{\text{B1}}{=} \sum_{d:D} r_1(d)\cdot\tau_{\{c_3\}}(X_3(d)).$$

The second equation can be proven as follows:

$$\tau_{\{c_3\}}(X_3(d)) = \tau_{\{c_3\}}(\textstyle\sum_{d':D}(r_1(d')\cdot X_4(d',d)+s_2(d)\cdot X_1)) =$$
$$\textstyle\sum_{d':D} r_1(d')\cdot\tau_{\{c_3\}}(X_4(d',d))+s_2(d)\cdot\tau_{\{c_3\}}(X_1).$$

The third equation

$$\tau_{\{c_3\}}(X_4(d,d')) = \tau_{\{c_3\}}(s_2(d')\cdot X_2(d)) =$$
$$\tau_{\{c_3\}}(s_2(d')\cdot c_3(d)\cdot X_3(d)) = s_2(d')\cdot\tau\cdot\tau_{\{c_3\}}(X_3(d)) = s_2(d')\cdot\tau_{\{c_3\}}(X_3(d)).$$

Exercise 8.8.2. Consider the following system (already described on page 59). Expand the definition of S. Compare this to the transition system of figure 4.6.

proc $X = \sum_{d:D}(r_1(d) + r_3(d))\cdot(s_2(d) + s_3(d))\cdot X;$
$B = \sum_{d:D} r_3(d)\cdot s_3(d)\cdot B;$
$S = \tau_{\{c_3\}}(\nabla_{\{r_1,s_2,c_3\}}(\Gamma_{\{r_3|s_3\to c_3\}}(X\|B))).$

8.8.3 Parallel expansion with time

In this section we look at the parallel composition of two simple timed processes. One process X must perform an a action, every other second, whereas a second process Y must perform a b action, only half a second later. See figure 8.1 Now assume action a and b must synchronise, resulting in the action c. The moments when c must take place are indicated in figure 8.1. We show how the individual processes are specified and how the moments where c must take place are calculated.

The processes X and Y can be specified as follows. The process S is the parallel composition of X and Y, where a and b must synchronise to c:

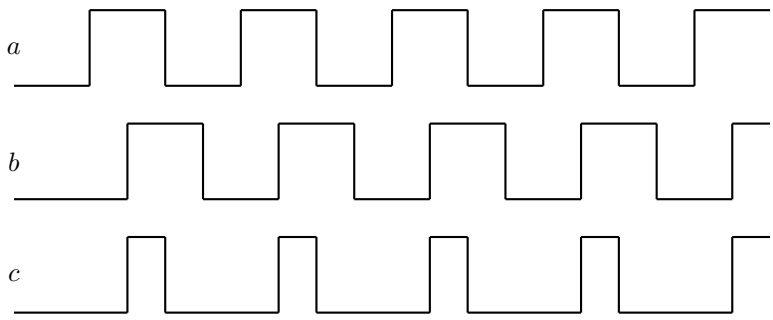


Figure 8.1: Two simple timed processes

proc $X(n{:}\mathbb{N}^+) = \sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}a{\triangleleft}t{\cdot}X(n{+}2);$
$Y(n{:}\mathbb{N}^+) = \sum_{u:\mathbb{R}}(n{+}0.5{\leq}u{\leq}n{+}1.5){\to}b{\triangleleft}u{\cdot}Y(n{+}2);$
$S = \nabla_{\{c\}}(\Gamma_{\{a|b\to c\}}(X(1)\|Y(1)));$

First, we define a generalised process $S(n{:}\mathbb{N}^+) = \nabla_{\{c\}}(\Gamma_{\{a|b\to c\}}(X(n)\|Y(n)))$. Clearly, $S = S(1)$. We expand the parallel operator in $X(n)\|Y(n)$ using the axioms.

$$X(n)\|Y(n) \stackrel{\text{M}}{=} X(n) \between Y(n) + Y(n) \between X(n) + X(n)|Y(n) \tag{8.7}$$

We treat the summands separately. We prove the first one in detail

$$\begin{array}{ll}
\underline{X(n) \between Y(n)} & \\
\stackrel{\text{def}}{=} & \underline{\sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}a{\triangleleft}t{\cdot}X(n{+}2) \between Y(n)} \\
\stackrel{\text{LM5}}{=} & \sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}(\underline{a{\triangleleft}t{\cdot}X(n{+}2) \between Y(n)}) \\
\stackrel{\text{TA4,LM6}}{=} & \sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}(\underline{a{\cdot}X(n{+}2) \between Y(n)}){\triangleleft}t \\
\stackrel{\text{LM3}}{=} & \sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}(a \ll \underline{Y(n)}){\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t \\
\stackrel{\text{def}}{=} & \sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}(\underline{a \ll \textstyle\sum_{u:\mathbb{R}}(n{+}0.5{\leq}u{\leq}n{+}1.5){\to}b{\triangleleft}u{\cdot}Y(n{+}2)}){\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t \\
\stackrel{\text{TB6}}{=} & \sum_{u,t:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(\underline{a \ll b{\triangleleft}u{\cdot}Y(n{+}2)}){\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t \\
\stackrel{\text{TB5}}{=} & \sum_{u,t:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(\underline{a \ll b{\triangleleft}u}){\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t \\
\stackrel{\text{TB3,TB1}}{=} & \sum_{u,t:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(\underline{\textstyle\sum_{v:\mathbb{R}} v < u{\to}(a{\triangleleft}v)}){\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t \\
= & \sum_{u,t,v:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5 \wedge v < u){\to}\underline{a{\triangleleft}v{\cdot}(X(n{+}2) \parallel Y(n)){\triangleleft}t} \\
\stackrel{\text{TA4}}{=} & \sum_{u,t,v:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5 \wedge v < u){\to}a{\triangleleft}t{\triangleleft}v{\cdot}(X(n{+}2) \parallel Y(n)).
\end{array}$$

In the same way we derive:

$$Y(n) \between X(n) \quad = \quad \sum_{t,u,v:\mathbb{R}}(n{+}0.5{\leq}u{\leq}n{+}1.5{\wedge}n{\leq}t{\leq}n{+}1 \wedge v < t){\to}b{\triangleleft}u{\triangleleft}v(X(n) \parallel Y(n{+}2)).$$

The communication summand can be expanded as follows:

$$\begin{array}{rcl}
\underline{X(n)|Y(n)} & \overset{\text{def}}{=} & \underline{(\sum_{t:\mathbb{R}}(n{\leq}t{\leq}n{+}1){\to}a{\triangleleft}t{\cdot}X(n{+}2))|(\sum_{u:\mathbb{R}}(n{+}0.5{\leq}u{\leq}n{+}1.5){\to}b{\triangleleft}u{\cdot}Y(n{+}2))} \\
& \overset{\text{S8}}{=} & \sum_{t,u:\mathbb{R}}(n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(\underline{a{\triangleleft}t{\cdot}X(n{+}2)|b{\triangleleft}u{\cdot}Y(n{+}2)}) \\
& \overset{\text{S9}}{=} & \sum_{t,u:\mathbb{R}}(n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(\underline{a{\cdot}X(n{+}2)|b{\cdot}Y(n{+}2)}){\triangleleft}t{\triangleleft}u \\
& \overset{\text{S6}}{=} & \sum_{t,u:\mathbb{R}}(n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}u{\leq}n{+}1.5){\to}\underline{((a|b){\cdot}(X(n{+}2)\|Y(n{+}2))){\triangleleft}t{\triangleleft}u} \\
& \overset{\text{TA4}}{=} & \sum_{t,u:\mathbb{R}}(n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}u{\leq}n{+}1.5){\to}(a|b)\underline{{\triangleleft}t{\triangleleft}u}{\cdot}(X(n{+}2)\|Y(n{+}2)) \\
& \overset{\text{TA1,sum elim}}{=} & \sum_{t:\mathbb{R}}(\underline{n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}t{\leq}n{+}1.5}){\to}(a|b){\triangleleft}t{\cdot}(X(n{+}2)\|Y(n{+}2)) \\
& + & \sum_{t,u:\mathbb{R}}(\underline{n{\leq}t{\leq}n{+}1{\wedge}n{+}0.5{\leq}u{\leq}n{+}1.5 \wedge t{\not\approx}u}){\to}\delta{\triangleleft}\min(t,u) \\
& \overset{\text{simplify}}{=} & \sum_{t:\mathbb{R}}(n{+}0.5{\leq}t{\leq}n{+}1){\to}(a|b){\triangleleft}t{\cdot}(X(n{+}2)\|Y(n{+}2)) \\
& + & \underline{\sum_{v:\mathbb{R}}(n{\leq}v{\leq}n{+}1){\to}\delta{\triangleleft}v} \\
& \overset{\text{delta incl.}}{=} & \sum_{t:\mathbb{R}}(n{+}0.5{\leq}t{\leq}n{+}1){\to}(a|b){\triangleleft}t{\cdot}(X(n{+}2)\|Y(n{+}2)).
\end{array}$$

Applying the communication and allow operator is now pretty standard. It yields:

$$\begin{array}{rcl}
S(n{:}\mathbb{N}^+) & = & \nabla_{\{c\}}(\Gamma_{\{a|b\to c\}}(X(n)\|Y(n))) \\
& = & \sum_{t,u,v:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5 \wedge v < u){\to}\delta{\triangleleft}\min(t,v) \\
& + & \sum_{t,u,v:\mathbb{R}}(n{+}0.5{\leq}u{\leq}n{+}1.5{\wedge}n{\leq}t{\leq}n{+}1 \wedge v < t){\to}\delta{\triangleleft}\min(u,v) \\
& + & \sum_{t:\mathbb{R}}(n{+}0.5{\leq}t{\leq}n{+}1){\to}c{\triangleleft}t{\cdot}\nabla_{\{c\}}(\Gamma_{\{a|b\to c\}}(X(n{+}2)\|Y(n{+}2))) \\
& = & \sum_{t:\mathbb{R}}(t \leq n+1){\to}\delta{\triangleleft}t \\
& + & \sum_{t:\mathbb{R}}(t \leq n+1){\to}\delta{\triangleleft}t \\
& + & \sum_{t:\mathbb{R}}(n{+}0.5{\leq}t{\leq}n{+}1){\to}c{\triangleleft}t{\cdot}S(n{+}2) \\
& = & \sum_{t:\mathbb{R}}(n{+}0.5{\leq}t{\leq}n{+}1){\to}c{\triangleleft}t{\cdot}S(n{+}2).
\end{array}$$

Note that c can exactly take place at those moments the third block diagram in figure 8.1 is high.

Exercise 8.8.3. Prove that

$$\sum_{t,u:\mathbb{R}}(n{\leq}t{\leq}n{+}1 \wedge n{+}0.5{\leq}u{\leq}n{+}1.5 \wedge t{\not\approx}u){\to}\delta{\triangleleft}\min(t,u) = \sum_{v:\mathbb{R}}(n \leq v \leq n{+}1){\to}\delta{\triangleleft}v.$$

Chapter 9

Linear Process Equations and Linearisation

The technique of parallel expansion in the previous chapter is cumbersome. Yet, the basic technique, namely, elimination of the parallel composition operator in favour of the alternative and sequential composition operator is the only effective technique if it comes to analysing parallel systems. By introducing the explicit notion of a *linear process equation*, we standardize the form of processes, and make it possible to formulate generic parallel expansion laws. We can also formulate certain rules, such as CL-RSP and the invariant rule, much easier on linear processes than on processes in general.

9.1 Linear process equations

We define a *linear processes* as a process of a restricted form. They come in three equivalent flavours, namely (general) linear process equations, clustered linear process equations and linear process operators. The two essential properties of linear processes are that each process can be transformed to linear form and that the linear form is so simple that it is relatively easy to manipulate with it.

9.1.1 General linear process equations

The most important characteristics of a *linear process equation* (*LPE*) are that one single process name is used in the right hand side and that there is precisely one action in front of the recursive invocation of the process variable at the right-hand side.

Definition 9.1.1 (Linear Process Equation). A linear process equation (LPE) is a process of the following form

$$\begin{array}{rcl} X(d{:}D) & = & \sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i)\to\alpha_i(d,e_i)\triangleleft t_i(d,e_i)\cdot X(g_i(d,e_i)) \\ & + & \sum_{j\in J}\sum_{e_j:E_j} c_j(d,e_j)\to\alpha_{\delta j}(d,e_j)\triangleleft t_j(d,e_j) \end{array}$$

where I and J are disjoint and finite index sets, and for $i \in I$ and $j \in J$:

- $c_i : D \times E_i \to \mathbb{B}$ and $c_j : D \times E_j \to \mathbb{B}$ are conditions,
- $\alpha_i(d, e_i)$ is a multi-action $a_i^1(f_i^1(d, e_i))|\cdots|a_i^{n_i}(f_i^{n_i}(d, e_i))$, where $f_i^k(d, e_i)$ (for $1 \le k \le n_i$) gives the parameters of action name a_i^k,
- $\alpha_{\delta j}(d, e_j)$ is either δ or a multi-action $a_j^1(f_j^1(d, e_j))|\cdots|a_j^{n_j}(f_j^{n_j}(d, e_j))$, where $f_j^k(d, e_j)$ (for $1 \le k \le n_j$) gives the parameters of action name a_j^k,
- $t_i : D \times E_i \to \mathbb{R}$ and $t_j : D \times E_j \to \mathbb{R}$ are the time stamps of multi-actions $\alpha_i(d, e_i)$ and $\alpha_{\delta j}(d, e_j)$,
- $g_i : D \times E_i \to D$ is the next state.

Note that the summands $\sum_{i\in I}$ and $\sum_{j\in J}$ are *meta-level* operations: $\sum_{i\in I} p_i$ is a shorthand for $p_1 + \cdots + p_n$, assuming $I = \{1, \ldots, n\}$. If the index set I is empty, $\sum_{i\in I} p_i$ is equal to $\delta \lhd 0$.

We call data parameter d the *state* parameter. Recall that we use only one state parameter d and one sum variable e_i per summand in the definitions to keep the formulas concise. In examples we generally use more than one (or sometimes 0) of such parameters.

The form as described above is sometimes described as the *condition-action-effect* rule. In a particular state d the multi-action α_i can be done at time t_i if condition c_i holds. The effect of the action is given by the function g_i.

There is an important theorem that says that any guarded recursive specification can be transformed to a linear process [49]. The proof of this theorem is rather involved, and we only show how to linearise certain subclasses of processes in this section. In many cases linearising a given process is so straightforward, that it can be done without the help of theorems or lemma's provided below.

A last general remark about linear processes is that the α_δ summands, in case α_δ is not equal to δ indicate the possibility of termination after doing an α_δ-action. In most cases that we encounter we find that α_δ is equal to δ. Therefore, we simplify the definition of a linear process below by assuming that α_δ always equals δ. This simplifies many of the definitions and theorems. If α_δ could be a multi-action we generally have to treat a few more case distinctions that do not provide new insight. If necessary, all what follows can be redone using the more general definition of a linear process.

Example 9.1.2. As a first example, consider the following simple process

proc $\quad X = a{\cdot}b{\cdot}c{\cdot}X;$

This is not a linear process because there is more than one action preceding variable X at the right hand side. We must encode the state of this process in an explicit data variable. For this purpose we can take a positive number $s : \mathbb{N}^+$. We let $s = 1$ equal the state where the a can be done, $s = 2$ equals the state where b is possible, and $s = 3$ represent the state where c can be done. The linearised version of this process then becomes:

$$Y(s{:}\mathbb{N}^+) = (s{\approx}1){\to}a{\cdot}Y(2){+}(s{\approx}2){\to}b{\cdot}Y(3){+}(s{\approx}3){\to}c{\cdot}Y(1); \tag{9.1}$$

Note that the linearised version of X is much less readable than the original version, which is a strong argument not to specify in linear equations, but to use the full potential of the language.

Formally, we can prove that $X = Y(1)$ using RSP. To do so, one can show that

$$\lambda n{:}\mathbb{N}^+.(n{\approx}1){\to}X + (n{\approx}2){\to}b{\cdot}c{\cdot}X + (n{\approx}3){\to}c{\cdot}X \tag{9.2}$$

is a solution for Y in equation (9.1). If the expression in (9.2) is indeed a solution, it is equal to Y, and it immediately follows that $X = Y(1)$. So, we substitute the expression in (9.2) for Y in (9.1) we obtain the following proof obligation (after beta reduction and simplifications of conditions):

$$(s{\approx}1){\to}X + (s{\approx}2){\to}b{\cdot}c{\cdot}X + (s{\approx}3){\to}c{\cdot}X = (s{\approx}1){\to}a{\cdot}b{\cdot}c{\cdot}X{+}(s{\approx}2){\to}b{\cdot}c{\cdot}X{+}(s{\approx}3){\to}c{\cdot}X,$$

which easily follows using the definition of X.

Example 9.1.3. Consider as another example a buffer that reads natural numbers via channel 1 and delivers via channel 2. It is described by:

proc $\quad X = \sum_{n:\mathbb{N}} r_1(n){\cdot}s_2(n){\cdot}X;$

We can use a similar technique as above, by numbering the states in the process. But note that between the action r_1 and the action s_2, the process must also recall the value of n. So, n must be added to the parameters of a linear process. Besides this, linearisation is straightforward and yields:

$$Y(n{:}\mathbb{N}, b{:}\mathbb{B}) = \sum_{m:\mathbb{N}} b{\to}r_1(m){\cdot}Y(m, \neg b) + \neg b{\to}s_2(n){\cdot}Y(n, \neg b).$$

It holds that $Y(n, true) = X$ for any $n{:}\mathbb{N}$. Note that sum operators over empty sequences of variables are simply ommitted.

Exercise 9.1.4. Prove that the linearisation of the buffer is indeed equal to the buffer.

Exercise 9.1.5. Linearise the process $X = a{\cdot}b{\cdot}X{+}b{\cdot}a{\cdot}Y, Y = a{\cdot}b{\cdot}X{+}a{\cdot}X$. Idem for $X = a{\cdot}b{\cdot}X{+}b{\cdot}a{\cdot}Y$, $Y = a{\cdot}b{\cdot}X + X$. Why is it not possible to linearise $X = a{\cdot}b{\cdot}X + b{\cdot}a{\cdot}Y, Y = a{\cdot}b{\cdot}X + Y$.

9.1.2 Clustered linear process equations

Sometimes it is useful to group the actions in a linear process by the label of an action. For convenience we avoid multi-actions here. The resulting linear process has a single summand for each action label. Such a linear process is called a clustered linear process equation.

Definition 9.1.6 (Clustered LPE). A *clustered linear process equation* is a process of the following form

$$\begin{array}{lcl} X(d{:}D) & = & \sum_{a\in Act}\sum_{e_a:E_a} c_a(d,e_a){\to}a(f_a(d,e_a)){\lhd}t_a(d,e_a){\cdot}X(g_a(d,e_a)) \\ & + & \sum_{e_\delta:E_\delta} c_\delta(d,e_\delta){\to}\delta{\lhd}t_\delta(d,e_\delta) \end{array}$$

where Act is a set of action labels possibly containing τ as a special element. The other elements in the definition are just as in definition 9.1.1.

Every linear process where the multi-actions consist of a single action or τ can straightforwardly be transformed to a clustered linear process as illustrated in the following example.

Example 9.1.7. Consider the linear process

$$\begin{array}{lcl} X(d{:}D) & = & \sum_{e_1:E_1} c_1(d,e_1){\to}a(f_1(d,e_1)){\cdot}X(g_1(d,e_1)) \\ & + & \sum_{e_2:E_2} c_2(d,e_2){\to}a(f_2(d,e_2)){\cdot}X(g_2(d,e_2)) \\ & + & \sum_{e_3:E_3} c_3(d,e_3){\to}a(f_3(d,e_3)){\cdot}X(g_3(d,e_3)). \end{array}$$

By introducing a new domain E with three elements this linear process can be translated to a clustered linear process. In addition we define a number of case functions C_S for some sort S where S is $\mathbb{B}$, F (the sort of the argument of action a) or D. The expression $C(e,t_1,t_2,t_3)$ equals t_i if e is equal to e_i.

$$\begin{array}{ll} \textbf{sort} & E = \mathbf{struct}\ enum_1 \mid enum_2 \mid enum_3; \\ \textbf{map} & C_S : E \times S \times S \times S; \\ \textbf{var} & t_1, t_2, t_3 : S; \\ \textbf{eqn} & C_S(enum_1, t_1, t_2, t_3) = t_1; \\ & C_S(enum_2, t_1, t_2, t_3) = t_2; \\ & C_S(enum_3, t_1, t_2, t_3) = t_3; \end{array}$$

The clustered linear process that is equivalent to X has the following shape:

$$\begin{array}{l} Y(d{:}D) = \sum_{e:E,e_1:E_1,e_2:E_2,e_3:E_3} C_{\mathbb{B}}(e,c_1(d,e_1),c_2(d,e_2),c_3(d,e_3)){\to} \\ \qquad a(C_F(e,f_1(d,e_1),f_2(d,e_2),f_3(d,e_3))){\cdot}Y(C_D(e,g_1(d,e_1),g_2(d,e_2),g_3(d,e_3))). \end{array}$$

Exercise 9.1.8. Give a clustered linear process for $X = a{\cdot}a{\cdot}b{\cdot}X$.

9.2 Linearisation

We provide several systematic ways to linearise processes. First we explain how processes without parallelism can be linearised. Then we explain how two linear processes can be put in parallel. Finally, we show how a process can be put in parallel with itself n times. In [49] it is explained how processes in general can be linearised.

9.2.1 Linearisation of sequential processes

Assume we have a guarded recursive specification with multi-actions, sum operators, conditions, time and the alternative and sequential composition operator. We call these sequential processes. We first observe that such a process can be written in *restricted Greibach normal form*. The term Greibach normal form comes from formal language theory. We use it here in adapted form, due to our slightly different, and much richer setting.

Definition 9.2.1 ((Restricted) Greibach normal form). A recursive equation is said to be in *Greibach Normal Form* (*GNF*) if it has the shape:

$$\begin{array}{rcl} X_i(d_i{:}D_i) & = & \sum_{j\in J_i}\sum_{e_{ij}:E_{ij}} c_{ij}(d_i,e_{ij}){\rightarrow}p^1_{ij}(d_i,e_{ij}){\cdot}p^2_{ij}(d_i,e_{ij}){\cdot}\cdots{\cdot}p^{n_{ij}}_{ij}(d_i,e_{ij}) \\ & + & \sum_{k\in K_i}\sum_{e_{ik}:E_{ik}} c_{ik}(d_i,e_{ik}){\rightarrow}\delta\lhd t_{ik}(d_i,e_{ik}). \end{array}$$

Here I, J_i and K_i are finite index sets. Each expression

$p^\ell_{ij}(d_i,e_{ij})$ is a process variable of the form $X_{m(i,j,\ell)}(g_{ij\ell}(d_i,e_{ij}))$ where $m(i,j,\ell)\in I$. The only exception is $p^1_{ij}(d_i,e_{ij})$ which can also be a (possibly timed) multi-action

$$(a^1_{ij\ell}(f^1_{ij\ell}(d_i,e_{ij}))|\cdots|a^{n_{ij\ell}}_{ij\ell}(f^{n_{ij\ell}}_{ij\ell}(d_i,e_{ij})))[\lhd t_{ij\ell}(d_i,e_{ij})].$$

If all $p^1_{ij}(d_i,e_{ij})$ are multi-actions, we say that the recursive equation is in *restricted Greibach normal form*.

If all equations in a recursive specification are in (restricted) Greibach normal form, the recursive specification in in (restricted) Greibach normal form. each equation $i\in I$ in

The translation to Greibach normal form can be done in linear place and time for every sequential recursive specification. The procedure is rather straightforward. Consider an equation $X(d:D)=p$ in a recursive specification. Then the right hand side can have such a form that the equation is in restricted Greibach normal form. Or, there is a strict sub-expression in p, say q, which violates the Greibach normal form. We can replace q by a fresh process variable $Y(d,e_i)$ and add a new equation $Y(d,e_i)=q$, to that part of the recursive specification that still has to be transformed.

Example 9.2.2.

proc $X = a{\cdot}(X+Y);$
$Y = b{\cdot}Y;$

The equation for Y is in Greibach normal form. But, the equation for X has the sub-expression $X+Y$. As a $+$ is only allowed as the outermost symbol of a GNF, this sub-expression violates the property of being in GNF. Therefore we replace $X+Y$ by Z and obtain

$X = a{\cdot}Z;$
$Y = b{\cdot}Y;$
$Z = X+Y;$

This set of equation is in Greibach normal form.

The translation to restricted GNF is more involved, as it can cause an exponential blow-up of the specification. Fortunately, in virtually all practical cases this potential blow-up does not appear to occur.

The only difference between a GNF and a restricted GNF is that in a restricted GNF the $p^1_{ij}(d_i,e_{ij})$ must always be an action. Consider the example above where we have the equation $Z=X+Y$ that is not in restricted GNF. However, the equations for X and Y are. So, by substituting the bodies of X and Y in the right hand side of Z we obtain

$$Z = a{\cdot}Z + b{\cdot}Y,$$

which is in restricted Greibach normal form. This is the general procedure. Find an equation that is not in restricted Greibach normal form of which all equations of the first process variables occurring in the right hand side are in restricted Greibach normal form. Substitute these and normalise the result. The equation for which we carried out the substitution is now also in restricted Greibach normal form.

An important property is that if the original recursive specification is guarded, the procedure above terminates and leads to a guarded recursive specification in restricted Greibach normal form.

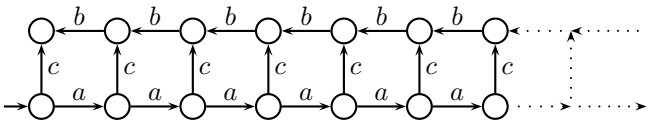


Figure 9.1: The behaviour of a stack or counter

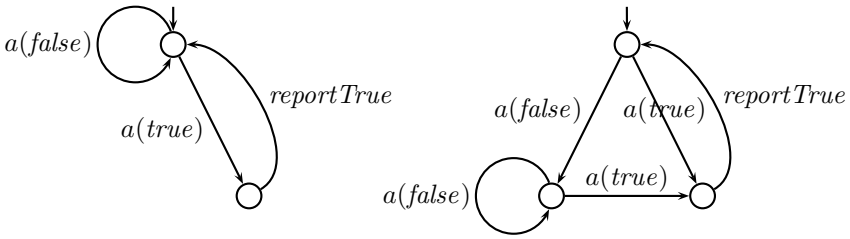


Figure 9.2: The transition system for two linearized processes

Example 9.2.3. As an example consider the following recursive specification, that models the behaviour of a counter. If a c-action is done after doing n a-actions, exactly n b-actions must be executed. See figure 9.1:

proc $X = a{\cdot}X{\cdot}b + c;$
$Y = X{\cdot}\delta;$

First we must bring this specification in Greibach normal form. The problems are the action b and the δ. So, we introduce two new variables B and D with as bodies b and δ. We get:

$X = a{\cdot}X{\cdot}B + c;$
$Y = X{\cdot}D;$
$B = b;$
$D = \delta.$

In order to bring this into restricted GNF, we must substitute the body of X for X in the body of Y. We obtain for Y:

$$Y = (a{\cdot}X{\cdot}B + c){\cdot}D.$$

Using the axiom A4 we distribute D over the choice operator and obtain:

$X = a{\cdot}X{\cdot}B + c;$
$Y = a{\cdot}X{\cdot}B{\cdot}D + c{\cdot}D;$
$B = b;$
$D = \delta.$

If data plays a role, the exact formulation of a process can influence the size of the state space generated via linearisation. The following three examples show this.

Example 9.2.4. Consider the following formulations of the same process:

proc $X = \sum_{b:\mathbb{B}} b{\to}a(b){\cdot}\mathit{reportTrue}{\cdot}X \diamond a(b){\cdot}X;$
$Y = \sum_{b:\mathbb{B}} a(b)(b{\to}\mathit{reportTrue}{\cdot}Y \diamond Y);$

The second process appears more attractive, as action $a(b)$ is only used once. However, when linearising, the first one turns out to have a smaller state space. We translate both to restricted Greibach normal form in a straightforward way:

proc $X = \sum_{b:\mathbb{B}} b{\to}a(b){\cdot}reportTrue{\cdot}X + \sum_{b:\mathbb{B}} \neg b{\to}a(b){\cdot}X;$
$Y = \sum_{b:\mathbb{B}} a(b){\cdot}Z(b);$
$Z(b{:}\mathbb{B}) = b{\to}reportTrue{\cdot}Y + \sum_{b:\mathbb{B}} \neg b{\to}a(b){\cdot}Z(b);$

The state spaces are drawn in figure 9.2. The figure to the left corresponds to X, and the figure to the right to Y. It is clear that the newly introduced $Z(b{:}\mathbb{B})$ gives rise to one extra state.

Example 9.2.5. The type of data passed on in processes can also have an influence of the number of states. Consider the following two processes:

proc $X = \sum_{n:\mathbb{N}} r(n){\cdot}((n{<}4){\to}reportOk{\cdot}X \diamond reportNotOk{\cdot}X);$
$Y = \sum_{n:\mathbb{N}}(n{<}4){\to}r(n){\cdot}reportOk{\cdot}Y \diamond reportNotOk{\cdot}Y;$

Transforming both processes to Greibach normal form yields the following result:

proc $X = \sum_{n:\mathbb{N}} r(n){\cdot}X'(n);$
$X'(n{:}\mathbb{N}) = (n{<}4){\to}reportOk{\cdot}X + (n{\geq}4){\to}reportNotOk{\cdot}X;$
$Y = \sum_{n:\mathbb{N}}(n{<}4){\to}r(n){\cdot}reportOk{\cdot}Y + \sum_{n:\mathbb{N}}(n{\geq}4){\to}r(n){\cdot}reportNotOk{\cdot}Y;$

The linearised process X has states $X(n)$ for each natural number n, whereas in Y only two states are available, namely one corresponding to all cases where $n < 4$ and one more where $n \geq 4$.

Example 9.2.6. Consider the following process that is in Greibach normal form.

proc $X_1 = X_2{\cdot}Y_1 + X_2$
$X_2 = X_3{\cdot}Y_2 + X_3$
$\ldots$
$X_n = X_{n+1}{\cdot}Y_n + X_{n+1}$
$X_{n+1} = a$

If we want to transform it to restricted Greibach normal form, we must substitute the variables upward. The result is that variable X_i has summands at the right hand side $a{\cdot}\sigma$ for any ordered sequence σ containing a subset of the actions $Y_{n-1} \ldots a_i$. So,

$$\begin{aligned}
&X_n = a{\cdot}Y_{n-1} + a,\\
&X_{n-1} = a{\cdot}Y_{n-1}{\cdot}Y_{n-2} + a{\cdot}Y_{n-2} + a{\cdot}Y_{n-1} + a,\\
&X_{n-2} = a{\cdot}Y_{n-1}{\cdot}Y_{n-2}{\cdot}Y_{n-3} + a{\cdot}Y_{n-2}{\cdot}Y_{n-3} + a{\cdot}Y_{n-1}{\cdot}Y_{n-3} + a{\cdot}Y_{n-3} +\\
&\qquad\qquad a{\cdot}Y_{n-1}{\cdot}Y_{n-2} + a{\cdot}Y_{n-2} + a{\cdot}Y_{n-1} + a,\\
&X_{n-4} = \ldots
\end{aligned}$$

So, X_i has 2^{n-i+1} summands. In particular, X_1 has 2^n summands. This shows that the transformation from restricted Greibach normal form to Greibach normal form can cause an exponential blow-up of the process. As said above, we hardly encounter this, when transforming models of behaviour. Therefore, in practice this blow-up does not turn out to be a problem.

Now we assume that we have a set of process equations in restricted GNF as indicated in definition 9.2.1. We must transform this set to a linear process equation. The essential observation is that the state of the process can be represented as a sequence of process variables. By inspecting the restricted GNF of the first variable, it is easy to determine what the initial actions are. The first variable is subsequently replaced by the sequence of process variables following this action.

First we introduce a data type *PA* to contain the individual process variables, together with their arguments. We base ourselves on definition 9.2.1 where we let $I = \{1, \ldots, n\}$.

sort $PA = \mathbf{struct}\ pr_1(data_1{:}D_1)?isPr_1 \mid pr_2(data_2{:}D_2)?isPr_2 \mid \cdots \mid pr_n(data_n{:}D_n)?isPr_n;$
$Stack = \mathbf{struct}\ empty?isEmpty \mid push(top{:}PA, pop{:}Stack);$

The translation of the process in 9.2.1 now becomes:

proc $X(s{:}Stack) =$
$\sum_{i\in I, j\in J_i} \sum_{e_{ij}:E_{ij}} isPr_i(top(s)) \wedge c_{ij}(data_i(top_i(s)), e_{ij}) \rightarrow$
$p^1_{ij}(data_i(top_i(s)), e_{ij}) \cdot$
$X(push(pr_{m(i,j,2)}(data_i(top_i(s)), e_{ij}), \ldots,$
$push(pr_{m(i,j,n_{ij})}(data_i(top_i(s)), e_{ij}), pop(s))))+$
$\sum_{i\in I, k\in K_i} \sum_{e_{ik}:E_{ik}} isPr_i(top(s)) \wedge c_{ik}(data_i(top_i(s)), e_{ik}) \rightarrow \delta \lhd t_{ik}(data_i(top_i(s)), e_{ik});$

Note that $p^1_{ij}(data_i(top_i(s)), e_{ij})$ is an action, and $m(i, j, \ell)$ is defined in definition 9.2.1.

Example 9.2.7. The restricted Greibach normal form in example 9.2.3 gives rise to the following

sort $PA = \mathbf{struct}\ prX?isX \mid prY?isY \mid prB?isB \mid prD?isD;$
$Stack = \mathbf{struct}\ empty?isEmpty \mid push(top{:}PA, pop{:}Stack);$
proc $X(s{:}Stack) =$
$isX(top(s)) \rightarrow a \cdot X(push(prX, push(prB, pop(s))))+$
$isX(top(s)) \rightarrow c \cdot X(pop(s))+$
$isY(top(s)) \rightarrow a \cdot X(push(prX, push(prB, push(prD, pop(s)))))+$
$isY(top(s)) \rightarrow c \cdot X(push(prD, pop(s)))+$
$isB(top(s)) \rightarrow b \cdot X(pop(s)).$

Note that there is no summand for D, because it cannot perform any action. If the initial state of the process is Y, then the initial state of the linear process is $X(push(prY, empty))$.

The use of a stack in linear processes has a substantial disadvantage. It is hard to investigate and use properties of individual data variables in the original process, because their values are stored somewhere in the complex stack structure. In [37] it has been shown that it is decidable for processes in restricted GNF whether the stack can maximally grow to a limited size (result is shown in a setting without data). In this case, each stack configuration can get an explicit index, which we can call the programme counter. Consider for example the following process:

proc $X = a \cdot Y \cdot Y \cdot Z + b \cdot Z \cdot Z;$
$Y = a;$
$Z = b \cdot Z + c \cdot Y \cdot Y \cdot X;$

First note that Z cannot terminate, so, $Z \cdot Z = Z$. Now a quick investigation learns that there are a finite number of stack frames which we can number as follows:

1:	X	4:	Z
2:	$Y \cdot Z$	5:	$Y \cdot X$
3:	$Y \cdot Y \cdot Z$	6:	$Y \cdot Y \cdot X$

The linearized process can be formulated using the program counter. The result looks like this.

proc $X(pc{:}\mathbb{N}^+) =$
$pc \approx 1 \rightarrow a \cdot X(3)+$
$pc \approx 1 \rightarrow b \cdot X(4)+$
$pc \approx 2 \rightarrow a \cdot X(4)+$
$pc \approx 3 \rightarrow a \cdot X(2)+$
$pc \approx 4 \rightarrow b \cdot X(4)+$
$pc \approx 4 \rightarrow c \cdot X(6)+$
$pc \approx 5 \rightarrow a \cdot X(1)+$
$pc \approx 6 \rightarrow a \cdot X(5);$

Example 9.2.8. The definition of a buffer is:

proc $X = \sum_{d:D} r(d) \cdot s(d) \cdot X;$

This process is already in restricted Greibach normal form and moreover, a quick inspection reveals that there are only two interesting stack frames. So, this process can be linearised using a stack pointer with two values. Besides the program counter, the value of d must also be recalled between a r and an s action.

proc $X(pc{:}\mathbb{N}^+, d{:}D) =$
$\sum_{d':D} pc{\approx}1{\to}r(d'){\cdot}X(2,d'){+}$
$pc{\approx}2{\to}s(d){\cdot}X(1,d);$

Note that the value of d in $X(1,d)$ in the last summand is irrelevant. To avoid unnecessary growth of the resulting transition system, it were better to set it to some default value.

Exercise 9.2.9. Give an LPE representing the following process:

$$X = (a+b){\cdot}d{\cdot}X$$

Exercise 9.2.10. Give an LPE representing the following process:

$$\begin{aligned} X &= a{\cdot}(a{\cdot}X + Y) \\ Y &= b{\cdot}(X+Y) \end{aligned}$$

Exercise 9.2.11. Linearise the following three processes:

proc $X_1 = \sum_{n:\mathbb{N}} a(n){\cdot}(X_1 + c(n)){\cdot}X_1;$
$X_2 = \sum_{n:\mathbb{N}} a(n){\cdot}X_2{\cdot}c(n){\cdot}X_2;$
$X_3 = \sum_{n:\mathbb{N}} a(n){\cdot}(X_3{\cdot}c(n) + c(n));$

Exercise 9.2.12. Linearisation of sequential timed processes is in essence not different from linearising sequential processes. Linearise:

proc $X = a{\triangleleft}3{\cdot}c{\triangleleft}4{\cdot}X;$
$Y = \sum_{n:\mathbb{N}} r(n){\cdot}((n{>}10) \to a{\triangleleft}n{\cdot}Y);$

9.2.2 Parallelization of linear processes

Assume that we have two linear processes X and Y that we want to put in parallel. This is a straightforward operation, where the result in general is linear in the size of the components. This is *the* huge advantage of linear processes, namely, that parallel composition does not blow up in general. This is contrary to calculating the parallel composition of two labelled transition systems, where in general the size of the result is proportional to the product of the sizes of the constituents. At the end of this section we indicate situations where the parallel composition of two linear processes grows more than linearly.

We first look at putting two untimed linear processes in parallel. So, consider processes X and Y given by the following linear process equations:

$$\begin{aligned} X(d{:}D) &= \textstyle\sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i){\to}\alpha_i(d,e_i){\cdot}X(g_i(d,e_i)) \\ Y(d'{:}D') &= \textstyle\sum_{j\in J}\sum_{e'_j:E'_j} c'_j(d',e'_j){\to}\alpha_j(d',e'_j){\cdot}Y(g'_j(d',e'_j)) \end{aligned}$$

The parallel composition consists of concatenating the parameters of X and Y and by listing all the actions of X and of Y and subsequently all actions of X and Y in combination. The result is the process XY characterized by the following equation:

$$\begin{aligned} XY(d{:}D, d'{:}D') &= \textstyle\sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i){\to}\alpha_i(d,e_i){\cdot}XY(g_i(d,e_i),d') \\ &+ \textstyle\sum_{j\in J}\sum_{e'_j:E'_j} c'_j(d',e'_j){\to}\alpha_j(d',e'_j){\cdot}XY(d,g'_j(d',e'_j)) \\ &+ \textstyle\sum_{i\in I}\sum_{j\in J}\sum_{e_i:E_i}\sum_{e'_j:E'_j} c_i(d,e_i)\wedge c'_j(d',e'_j){\to} \\ &\qquad \alpha_i(d,e_i)|\alpha_j(d',e'_j){\cdot}XY(g_i(d,e_i),g'_j(d',e'_j)) \end{aligned}$$

Note that when actions of X happen on their own, the parameter d' stays untouched. Similarly, when actions from Y happen without synchronising with those of X, the data of process X is not touched either.

With time, the linearisation is only slightly trickier. When actions synchronise, their time stamps must be equal. Furthermore, if actions of one process must happen before a certain time, the other process cannot

do any action at a later time. So, consider the following timed linear equations:

$$\begin{array}{rcl} X(d{:}D) & = & \sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i){\to}\alpha_i(d,e_i){\lhd}t_i(d,e_i){\cdot}X(g_i(d,e_i)) \\ & + & \sum_{j\in J}\sum_{e_j:E_j} c_j(d,e_j){\to}\delta_j(d,e_j){\lhd}t_j(d,e_j) \\ Y(d'{:}D') & = & \sum_{i\in I'}\sum_{e'_i:E'_i} c'_i(d',e'_i){\to}\alpha'_i(d',e'_i){\lhd}t'_i(d',e'_i){\cdot}Y(g'_i(d',e'_i)) \\ & + & \sum_{j\in J'}\sum_{e'_j:E'_j} c'_j(d',e'_j){\to}\delta{\lhd}t'_j(d',e'_j) \end{array}$$

The parallel composition of X and Y has the behaviour of the process XY given by the following equation

$$\begin{array}{l} XY(d{:}D,d'{:}D') \\ = \sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i){\wedge} \\ \qquad ((\bigvee_{i'\in I'}\exists e'_{i'}{:}E_{i'}.c'(d',e'_{i'}){\wedge}t_i(d,e_i)<t'_{i'}(d',e'_{i'})){\vee} \\ \qquad (\bigvee_{j'\in J'}\exists e'_{j'}{:}E_{j'}.c'(d',e'_{i'}){\wedge}t_i(d,e_i)<t'_{j'}(d',e'_{j'}))){\to} \\ \quad \alpha_i(d,e_i){\lhd}t_i(d,e_i){\cdot}XY(g_i(d,e_i),d') \\ + \sum_{i\in I'}\sum_{e_i:E'_i} c'_i(d',e_i){\wedge} \\ \qquad ((\bigvee_{i'\in I}\exists e'_{i'}{:}E_{i'}.c(d,e'_{i'}){\wedge}t'_i(d',e_i)<t_{i'}(d,e'_{i'})){\vee} \\ \qquad (\bigvee_{j'\in J}\exists e'_{j'}{:}E_{j'}.c(d,e'_{i'}){\wedge}t'_i(d',e_i)<t_{j'}(d,e'_{j'}))){\to} \\ \quad \alpha'_i(d',e_i){\lhd}t_i(d,e_i){\cdot}XY(d,g'_i(d',e_i)) \\ + \sum_{i\in I}\sum_{i\in I'}\sum_{e_i:E_i}\sum_{e'_i:E'_i} c_i(d,e_i)\wedge c'_{i'}(d',e'_{i'})\wedge t_i(d,e_i){\approx}t'_{i'}(d',e'_{i'}){\to} \\ \quad \alpha_i(d,e_i)|\alpha'_{i'}(d',e'_i){\lhd}t_i(d,e_i){\cdot}XY(g_i(d,e_i),g'_{i'}(d',e'_{i'})) \\ + \sum_{j\in J}\sum_{j'\in J'}\sum_{e_j:E_j}\sum_{e'_{j'}:E'_{j'}} c_j(d,e_j)\wedge c'_{j'}(d',e'_{j'})\wedge t_j(d,e_j){\approx}t'_{j'}(d',e'_{j'}){\to}\delta{\lhd}t_j(d,e_j) \end{array}$$

The first and second set of summands are the multi-actions of one of the processes that happen on their own. Expressions of the form

$$\begin{array}{l} ((\bigvee_{i'\in I'}\exists e'_{i'}{:}E_{i'}.c'(d',e'_{i'}){\wedge}t_i(d,e_i)<t'_{i'}(d',e'_{i'})){\vee} \\ \qquad (\bigvee_{j'\in J'}\exists e'_{j'}{:}E_{j'}.c'(d',e'_{i'}){\wedge}t_i(d,e_i)<t'_{j'}(d',e'_{j'}))) \end{array}$$

express that actions in one process must happen either before the latest action in the other process can occur, or before the other process encounters a timed deadlock.

The third set of summands are the synchronisations between the actions of both processes and the fourth set of summands record the combined timed deadlocks of both processes.

Exercise 9.2.13. Consider the example from section 8.8.2 where two buffers are put in parallel. Calculate a linear equation for a single buffer and one for the parallel combination.

Exercise 9.2.14. Calculate a linear process equations for the process S from section 8.8.3.

9.2.3 Linearisation of n parallel processes

When studying distributed algorithms, often n processes are put in parallel, for an arbitrary positive number n. E.g., we are interested in the behaviour of

$$\nabla_V(\Gamma_C(X(1,f(1)) \parallel \cdots \parallel X(n,f(n)))) \tag{9.3}$$

where X is given by the following linear process equation

$$X(id{:}\mathbb{N}^+,d{:}D) = \sum_{i\in I}\sum_{e_i:E_i} c_i(id,d,e_i){\to}a_i(f_i(id,d,e_i)){\cdot}X(g_i(id,d,e_i)).$$

and $f:\mathbb{N}^+\to D$ is a function that gives for each process i its initial state $f(i)$. So, $X(i,f(i))$ is process i with initial value $f(i)$.

For simplicity, we leave out time and let the multi-action consist of a single action and only allow two actions to communicate in Γ. Furthermore, we assume that the index set I is ordered. A precise way of describing the process in (9.3) is the following:

$$\begin{array}{l} Y(n{:}\mathbb{N}^+,f{:}\mathbb{N}^+{\to}D) = \nabla_V(\Gamma_C(Z(f,n))) \\ Z(n{:}\mathbb{N}^+,f{:}\mathbb{N}^+{\to}D) = (n{\approx}1)\to X(1,f(1))\diamond(Z(n{-}1,f)\parallel X(n,f(n))) \end{array}$$

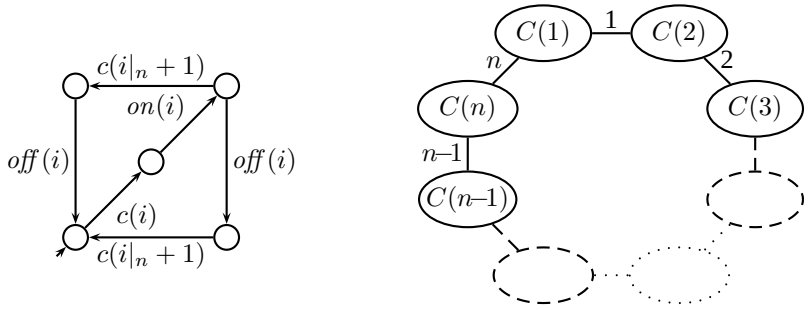


Figure 9.3: The behaviour of a cycler and the overall process structure in Milner's scheduler

Although, the equation for Z is unguarded, it has been proven that it defines a unique process. Moreover, it has been shown that the process Y also satisfies the following linear equation. This equation has a very regular structure, and therefore, it is very useful to eliminate the parallel operator from n parallel linear processes in concrete cases.

$$
\begin{aligned}
&Y(n{:}\mathbb{N}^+, f{:}\mathbb{N}^+{\to}D) = \\
&\quad \sum_{i\in I, a_i\in V} \sum_{k:\mathbb{N}^+, e_i:E_i} (c_i(k,f(k),e_i) \wedge k \le n){\to} \\
&\qquad a_i(f_i(k,f(k),e_i)){\cdot}Y(n, f[k := g_i(k,f(k),e_i)]) + \\
&\qquad \sum_{i,j\in I, i\le j, C(a_i|a_j)\in V} \sum_{k,\ell:\mathbb{N}^+, e_i:E_i, e_j:E_j} (c_i(k,f(k),e_i) \wedge c_j(\ell,f(\ell),e_j) \wedge \\
&\qquad\quad f_i(k,f(k),e_i) {\approx} f_j(\ell,f(\ell),e_j) \wedge k {\not\approx} \ell \wedge k \le n \wedge \ell \le n){\to} \\
&\qquad\quad C(a_i|a_j)(f_k(k,f(k),e_i)){\cdot}Y(n, f[k := g_i(k,f(k),e_i)][\ell := f(\ell,f(\ell),e_j])
\end{aligned}
$$

Example 9.2.15. As an example we consider the well known *Milner's scheduler*. This process was used as an example in [41]. The scheduler controls $n > 1$ production machines. Each machine i can be switched on (using an action $on(i)$) and off (using $off(i)$). The requirements are that each machine must alternatingly be switched on and off, and all machines must be started in sequence. Milner defined one control process per machine as follows:

$$C(id{:}\mathbb{N}^+) = c(id){\cdot}on(id)(off(id) \parallel c(id|_n + 1)){\cdot}C(id).$$

The actions $c(id)$ are used to signal the next process that it can start the machine it controls. The process $C(1)$ must initially skip the initial $c(1)$ action, as otherwise the whole system will not start up.

Linearising the behaviour of a cycler is straightforward. The process $C(id)$ is equal to $C(id, 1)$ where the behaviour of $C(id, 1)$ is given by the following equation

$$
\begin{aligned}
C(id, state{:}\mathbb{N}^+) \;=\; & (state{\approx}1){\to}c(id){\cdot}C(id,2) \\
+\; & (state{\approx}2){\to}on(id){\cdot}C(id,3) \\
+\; & (state{\approx}3){\to}c(id|_n{+}1){\cdot}C(id,4) \\
+\; & (state{\approx}3){\to}off(id){\cdot}C(id,5) \\
+\; & (state{\approx}4){\to}off(id){\cdot}C(id,1) \\
+\; & (state{\approx}5){\to}c(id|_n{+}1){\cdot}C(id,1)
\end{aligned}
$$

The behaviour of the scheduler is defined by

$$
\begin{aligned}
&Scheduler(n{:}\mathbb{N}^+) = \nabla_{\{on,off,d\}}(\Gamma_{\{c|c\to d\}}(Sched(n))) \\
&Sched(n{:}\mathbb{N}^+) = (n{\approx}1){\to}C(n,2) \diamond (Sched(n{-}1) \parallel C(n,1))
\end{aligned}
$$

Note that we do not use the function f in this definition explicitly to indicate the initial values of each process. Using the linearisation result above as a guideline, it can directly be seen that the scheduler $Scheduler(n)$ is equal to $Scheduler(n, f_{init})$ of which the defining equation is given below. The function $f_{init} : \mathbb{N}^+ \to \mathbb{N}^+$ is defined as $f_{init} = \lambda n{:}\mathbb{N}^+.if(n{\approx}1, 2, 1)$.

$$\begin{array}{l} Scheduler(n{:}\mathbb{N}^+, f{:}\mathbb{N}^+{\to}\mathbb{N}^+) = \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}2){\to}on(k){\cdot}Scheduler(k, f[k := 3]) + \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}3){\to}off(k){\cdot}Scheduler(k, f[k := 5]) + \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}4){\to}off(k){\cdot}Scheduler(k, f[k := 1]) + \\ \quad \sum_{\ell:\mathbb{N}^+}(f(\ell|_n{+}1){\approx}1) \wedge f(\ell){\approx}3 \wedge \ell{\leq}n){\to}d(\ell|_n{+}1){\cdot}Scheduler(n, f[\ell|_n{+}1 := 2][\ell := 4]) + \\ \quad \sum_{\ell:\mathbb{N}^+}(f(\ell|_n{+}1){\approx}1) \wedge f(\ell){\approx}5 \wedge \ell{\leq}n){\to}d(\ell|_n{+}1){\cdot}Scheduler(n, f[\ell|_n{+}1 := 2][\ell := 1]) \end{array} \tag{9.4}$$

In section 11.2.2 we show how to reduce this equation further.

Exercise 9.2.16. Derive the equation in (9.4) in detail. Note that in order to do this it is required that $n > 1$. What is the behaviour of the scheduler if $n = 1$?

9.3 Proof rules for linear processes

In this section we introduce two proof rules, namely the rule CL-RSP (Convergent Linear Recursive Specification Principle) and CL-RSP with invariants. Both rules are derivable from RSP, and as such do not add proof strength. However, they are more convenient if it comes to proving the correctness of concrete processes.

9.3.1 τ-convergence

We first introduce τ-convergence that says from no state of a linear process an infinite number of τs can be performed.

Definition 9.3.1 (τ-convergent LPE). An LPE is *τ-convergent* if it cannot exhibit an infinite sequence of τ-transitions from any state.

An alternative formulation uses well-founded orderings. This formulation allows a straightforward proof technique to show that there are no infinite tau-sequences.

Lemma 9.3.2. An LPE written as in Definition 9.1.1 is τ-convergent if for all $i \in I$ such that $a_i = \tau$, there is a well-founded ordering $<$ on D such that for all $e_i{:}E_i$ and $d{:}D$, $c_i(d, e_i)$ implies $g_i(d, e_i) < d$.

A well known well-founded ordering is the smaller than ('$<$') relation on natural numbers. For a wealth of other well founded relations we refer to chapter 6 of [48].

Example 9.3.3. Consider the following linear process, where first an arbitrary number n is selected, and then n τ-steps can be performed.

$$\begin{array}{rcl} X(n{:}\mathbb{N}) & = & \sum_{m:\mathbb{N}}(n{\approx}0) \to select(m){\cdot}X(m) \\ & + & (n > 0){\to}\tau{\cdot}X(n{-}1) \end{array}$$

Observe that with each τ step n decreases according to the well founded ordering relation $<$ on natural numbers. More precisely, we find for the second summand of the linear process that:

$$(n > 0) \;\; implies \;\; n - 1 < n$$

which is undeniably true. So, we can conclude that this linear process is τ-convergent. Note that this linear process illustrates one important point, namely that for a τ-convergent linear process there is no upperbound on the number of consecutive τ-steps that can be performed. In the first summand m can be set to any (finite) number.

Example 9.3.4. An alternative technique to show τ-convergence is to map the data state to some domain (generally the natural numbers), and show that in this domain τ-steps are decreasing according to some well founded relation.

Consider for instance two unbounded queues (see figure 9.4). The queues Q_1 and Q_2 contain data

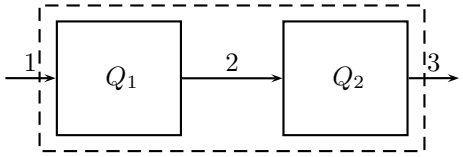


Figure 9.4: Two connected unbounded queues

elements of some sort D and are defined as follows:

$$Q_1(q_1{:}List(D)) = \sum_{d:D} r_1(d){\cdot}Q_1(d \triangleright q_1) + (q_1 \not\approx []) \rightarrow s_2(rhead(q_1)){\cdot}Q_1(rtail(q_1))$$
$$Q_2(q_2{:}List(D)) = \sum_{d:D} r_2(d){\cdot}Q_2(d \triangleright q_2) + (q_2 \not\approx []) \rightarrow s_3(rhead(q_2)){\cdot}Q_2(rtail(q_2))$$
$$System = \tau_{\{c_2\}}(\nabla_{\{r_1,c_2,s_3\}}(\Gamma_{\{s_2|r_2\rightarrow c_2\}}(Q_1([]) \parallel Q_2([]))))$$

Linearizing these equations and applying sum elimination yields the following linear process where $System$ is equal to the process $X([],[])$.

$$\begin{array}{lcl} X(q_1,q_2{:}List(D)) & = & \sum_{d:D} r_1(d){\cdot}X(d \triangleright q_1, q_2) \\ & + & (q_2 \not\approx []) \rightarrow s_3(rhead(q_2)){\cdot}X(q_1, rtail(q_2)) \\ & + & (q_1 \not\approx []) \rightarrow \tau{\cdot}X(rtail(q_1), rhead(q_1) \triangleright q_2) \end{array} \tag{9.5}$$

In order to show that this is a τ-convergent linear process equation we show that the when doing τ steps the length of the first queue decreases. Formally we define a *variant function* f from the state of the linear process to the natural numbers as follows:

$$f(q_1, q_2) = \#q_1.$$

Now expressing that the variant function decreases with every τ-step is expressed by the following condition:

$$(q_1 \not\approx []) \rightarrow f(rtail(q_1), rhead(q_1) \triangleright q_2) < f(q_1, q_2).$$

Applying the definition of f leads to the following that obviously holds:

$$(q_1 \not\approx []) \rightarrow \#rtail(q_1) < \#q_1.$$

From this we can immediately conclude that the linear process equation in (9.5) is τ-convergent.

The notion of a variant function is coined in [13] but as it is commonly used, other names for it also occur in the literature (e.g., bounded functions). It generally suffices to let variant functions map to the natural numbers to prove τ-convergence. However, sometimes it is convenient or even necessary to consider richer well founded domains.

9.3.2 Convergent Linear Recursive Specification Principle (CL-RSP)

The rule CL-RSP (Convergent Linear Recursive Specification Principle) is essentially the same as RSP, except that the guardedness condition is replaced by τ-convergence [7]. Hennessy and Lin [27] introduced a similar derivation rule called UFI-O.

We define CL-RSP as follows:

Definition 9.3.5 (CL-RSP).

$$\frac{\Gamma \vdash X = \Psi X \quad \Gamma \vdash Y = \Psi Y}{\Gamma \vdash X = Y} \quad \Psi \text{ is a } \tau\text{-convergent process operator} \quad \text{(CL-RSP)}$$

Note that when τ-actions occur in a linear process, the process is not guarded anymore. Actually, CL-RSP is stronger than RSP, in the sense that RSP is only applicable to processes with a bounded number of τ's. As we have seen in example 9.3.3, τ-convergent linear processes do not allow such an a priori bound.

Example 9.3.6. The process $X(n)$ of example 9.3.3 is equal to the process $Y(n > 0)$ defined by

$$Y(b{:}\mathbb{B}) = \sum_{m:\mathbb{N}} b \to select(m){\cdot}Y(true) + \neg b \to \tau{\cdot}Y(true).$$

Note that contrary to X, the process Y has no unbounded sequences of τ's. We show that $\lambda n{:}\mathbb{N}.Y(n{\approx}0)$ is a fixed point for the linear process operator induced by the equation for X. Hence, we must show:

$$\begin{array}{rcl} Y(n{\approx}0) & = & \sum_{m:\mathbb{N}}(n{\approx}0) \to select(m){\cdot}Y(m{\approx}0) \\ & + & \neg(n{\approx}0) \to \tau{\cdot}Y(n{-}1{\approx}0) \end{array} \tag{9.6}$$

In order to prove this, we observe that $a{\cdot}Y(false) = a{\cdot}Y(true)$ for any multi-action a. The proof relies on the use of the first τ-law B1:

$$a{\cdot}Y(false) \;=\; a{\cdot}\tau{\cdot}Y(true) \;\stackrel{\text{B1}}{=}\; a{\cdot}Y(true).$$

So, using this observation, the equation (9.6) can be transformed to the equivalent:

$$\begin{array}{rcl} Y(n{\approx}0) & = & \sum_{m:\mathbb{N}}(n{\approx}0) \to select(m){\cdot}Y(true) \\ & + & \neg(n{\approx}0) \to \tau{\cdot}Y(true) \end{array}$$

But this is exactly the defining equation for Y where $n{\approx}0$ has been substituted for b. Therefore, it holds, and so, we have shown that $\lambda n{:}\mathbb{N}.Y(n{\approx}0)$ and X are fixed points of a τ-convergent linear process operator. Therefore, we conclude that $Y(n{\approx}0) = X(0)$.

9.3.3 CL-RSP with invariants

An invariant for a linear process is a property, i.e., a function from the state variable D to booleans, which once it is valid can never be invalidated by any step of the process. In particular, if the invariant holds in the initial state, it must also hold for all reachable states.

Definition 9.3.7 (Invariant). A mapping $\mathcal{I} : D \to \mathbb{B}$ is an *invariant* for an LPE as in definition 9.1.1 iff, for all $i \in I$, $d{:}D$ and $e{:}E$,

$$\mathcal{I}(d) \wedge c_i(d, e_i) \;\Rightarrow\; \mathcal{I}(g_i(d, e_i)) \tag{9.7}$$

Example 9.3.8. Consider the LPE $X(n{:}\mathbb{N}) \;=\; a(n){\cdot}X(n{+}2)$. Two invariants for this LPE are

$$\mathcal{I}_1(n) \;=\; \begin{cases} true & \text{if } n \text{ is even} \\ false & \text{if } n \text{ is odd} \end{cases} \qquad \mathcal{I}_2(n) \;=\; \begin{cases} false & \text{if } n \text{ is even} \\ true & \text{if } n \text{ is odd} \end{cases}$$

The relation between the total state space, an invariant and the reachable state space can neatly be depicted as in figure 9.5. At the bottom there is an initial state. The reachable state space is characterised by the grey area. It turns out that for many reactive systems this reachable state space is very irregular and as such hard to understand and model. Often it is possible to find a nice characterisation of an outer boundary that includes the reachable state space and that satisfies the invariant property. This property is generally chosen as the invariant. The outer square characterises the total state space, of which, as suggested by the picture, we found that most states are not reachable from the initial state.

The following useful lemma gives some calculation rules about invariants that allow to prove invariants in parts.

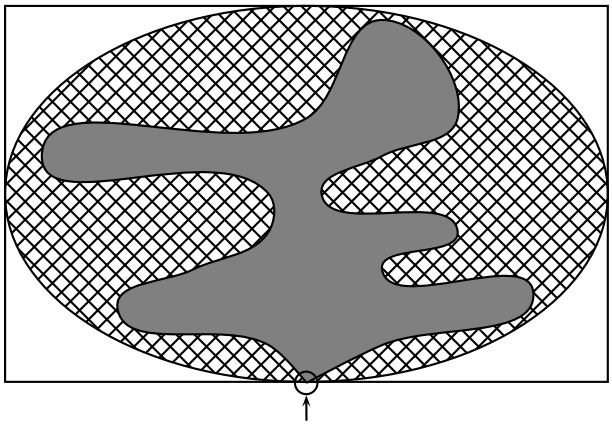


Figure 9.5: An invariant, the total and reachable state space

Lemma 9.3.9. Consider a linear process operator Ψ.

- Let, for all $1 \leq k \leq n$, $\mathcal{I}_k$ be invariants of Ψ. Then $\lambda d{:}D.\bigwedge_{k=1}^{n} \mathcal{I}_k(d)$ is also an invariant of Ψ.
- If for all $1 \leq k \leq n$, $i{\in}I$ $d{:}D$ and $e_i{:}E_i$, it holds that $(\bigwedge_{\ell=1}^{n} \mathcal{I}_\ell(d)) \wedge c_i(d,e_i) \Rightarrow \mathcal{I}_k(g_i(d,e_i))$, then $\lambda d{:}D.\bigwedge_{k=1}^{n} \mathcal{I}_k(d)$ is an invariant of Ψ.

Note that the disjunction of two invariants is also an invariant, but the negation of an invariant is not necessarily an invariant itself.

Invariants tend to play a crucial role in algebraic verifications of system behaviour that involve data. The reason for this is that it allows separation of concerns. Before commencing with the proof that two processes behave the same, invariants can be proven that can be employed in the verification later on. This does not mean that finding invariants is easy. On the contrary finding proper and useful invariants is one of the harder jobs in any verification.

The following adaptation of CL-RSP allows to use invariants when proving two processes equivalent:

Definition 9.3.10 (CL-RSP with invariants). Let Ψ be a τ-convergent linear process operator and $\mathcal{I} : D \to \mathbb{B}$ be an invariant of Ψ.

$$\frac{\Gamma \vdash \forall d{:}D.(\mathcal{I}(d) \Rightarrow Xd = \Psi Xd) \quad \Gamma \vdash \forall d{:}D.(\mathcal{I}(d) \Rightarrow Yd = \Psi Yd)}{\Gamma \vdash \forall d{:}D.(\mathcal{I}(d) \Rightarrow Xd = Yd)}$$ CL-RSP with invariants

The following theorem says that CL-RSP with invariants is not essential. Any proof that uses CL-RSP with invariants, can be transformed to a proof without the explicit use of invariants.

Theorem 9.3.11. Every proof using CL-RSP with invariants can be replaced by a proof using CL-RSP.

Proof. Consider a proof in which CL-RSP with invariants is used. We replace each occurrence of CL-RSP one by one. So, consider a single application of CL-RSP with invariants. This means there is a τ-convergent linear process operator Ψ with invariant $\mathcal{I}$ and processes X and Y such that

$$\Gamma \vdash \forall d{:}D.\mathcal{I}(d) \Rightarrow Xd = \Psi Xd$$
$$\Gamma \vdash \forall d{:}D.\mathcal{I}(d) \Rightarrow Yd = \Psi Yd$$

are proven. Now consider process operator $\Phi = \lambda Z{:}D \to \mathbb{P}, d{:}D.\mathcal{I}(d) \to \Psi Z d$. We show using CL-RSP the processes $\lambda d{:}D.\mathcal{I}(d) \to X(d)$ and $\lambda d{:}D.\mathcal{I}(d) \to Y(d)$ are solutions of Φ. As both cases are symmetric, we only show the first. So, we must prove that

$$\Gamma \vdash \lambda d{:}D.\mathcal{I}(d) \to X(d) = \Phi(\lambda d{:}D.\mathcal{I}(d) \to X(d)).$$

Expanding the definition of Φ and applying beta reduction yields:

$$\Gamma \vdash \lambda d{:}D.\mathcal{I}(d) \to X(d) = \lambda d{:}D.\mathcal{I}(d) \to \Psi(\lambda d{:}D.\mathcal{I}(d) \to X(d)).$$

Now expand the definition of Ψ.

$$\Gamma \vdash \lambda d{:}D.\mathcal{I}(d) \to X(d) = \lambda d{:}D.\mathcal{I}(d) \to (\sum_{i:I} \sum_{e_i:E_i} c_i(d,e_i) \to a_i(f_i(d,e_i)) \cdot (\mathcal{I}(g_i(d,e_i)) \to X(g_i(d,e_i))). \tag{9.8}$$

As $\mathcal{I}(d) \wedge c_i(d,e_i)$ and $\mathcal{I}$ is an invariant, we can show that $\mathcal{I}(g_i(d,e_i)) = true$. So, equation (9.8) becomes:

$$\Gamma \vdash \lambda d{:}D.\mathcal{I}(d) \to X(d) = \lambda d{:}D.\mathcal{I}(d) \to (\sum_{i:I} \sum_{e_i:E_i} c_i(d,e_i) \to a_i(f_i(d,e_i)) \cdot X(g_i(d,e_i)).$$

Now, as X is a solution for Ψ, this in turn reduces to

$$\Gamma \vdash \lambda d{:}D.\mathcal{I}(d) \to X(d) = \lambda d{:}D.\mathcal{I}(d) \to X(d),$$

which is obviously true. □

Example 9.3.12. Let $even{:}\mathbb{N} \to \mathbb{B}$ map even numbers to $true$ and odd numbers to $false$. Consider the following two LPEs:

$$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & a(even(n)) \cdot X(n{+}2) \\ Y & = & a(true) \cdot Y \end{array}$$

Substituting Y for $X(n)$ for even numbers n in the first LPE yields $Y = a(true) \cdot Y$, which follows from the second LPE. Since

$$\mathcal{I}(n) = \begin{cases} true & \text{if } n \text{ is even} \\ false & \text{if } n \text{ is odd} \end{cases}$$

is an invariant for the first LPE (cf. Example 9.3.8), and this LPE is τ-convergent, by definition 9.3.10

$$X(n) = Y$$

for even n.

Exercise 9.3.13. Show that the mappings $\mathcal{I}_1(d) = true$ for all $d{:}D$ and $\mathcal{I}_2(d) = false$ for all $d{:}D$ are invariants for all LPEs.

Exercise 9.3.14. Consider the process declarations

$$X(b_1, b_2{:}\mathbb{N}) = a(b_1 \vee b_2) \cdot X(b_2, b_1)$$

and

$$Y = a(true) \cdot Y.$$

Prove, using CL-RSP with invariants, that $X(true, false) = Y$.

Chapter 10

Confluence and τ-prioritisation

When studying the state spaces of parallel systems, the pattern seen in figure 10.1 may catch your eyes. Here we depicted the behaviour of the process $\tau_{\{b,c\}}(\nabla_{\{a,b,c,d\}}(a{\cdot}b \parallel c{\cdot}d))$, i.e., the parallel composition of the processes $a{\cdot}b$ and $c{\cdot}d$ where neither communication nor multi-actions are allowed. The reason for these patterns is that actions in parallel components act independently, meaning that executing the two actions consecutively leads to the same state, irrespective of the fact which of the two transitions is executed first. This phenomenon is generally referred to as *confluence*.

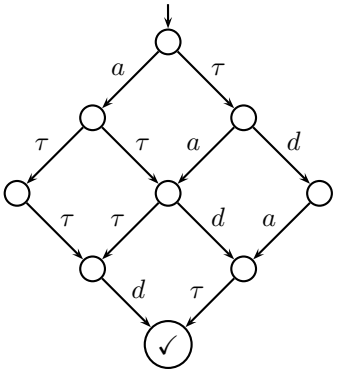


Figure 10.1: Typical confluent behavioural pattern

The independence of actions that happen in parallel is the cause of the state space explosion problem. But by employing confluence we can sometimes circumvent this problem, by applying τ-prioritisation. This means that in any state a confluent τ can be chosen and other actions can be ignored.

The importance of confluence was already indicated in [41] but the approach we follow is the one of [25, 22]. As it is an important notion, there are many alternative names for roughly the same phenomenon, such as partial orders and stubborn sets.

10.1 τ-confluence on labelled transition systems

First we define confluence on labelled transition systems. Assume a state space. We consider a subsets U of τ transitions to be τ-confluent (or confluent for short) as depicted in figure 10.2. For every state s, which is the state on top in figure 10.2, with outgoing τ transition in U and outgoing a transition, it is possible to finish the diamond shape. The diagram must be interpreted in such a way that if the solid arrows are present, the dotted arrows must be shown to exist. The definition below makes this precise. Note that it is required that the dotted τ-transition is a member of U.

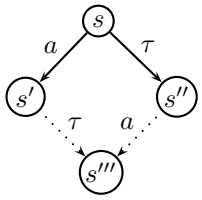


Figure 10.2: The confluence pattern

Definition 10.1.1 (Confluence). Let $A = (S, Act, \rightarrow, s_0, T)$ be a labelled transition system. A set U of τ-transitions is called τ-*confluent* if for all transitions $s \xrightarrow{a} s'$ in $\rightarrow$ and $s \xrightarrow{\tau} s'' \in U$:

(1) either $s' \xrightarrow{\tau} s''' \in U$ and $s'' \xrightarrow{a} s''' \in \rightarrow$, for some state $s''' \in S$;

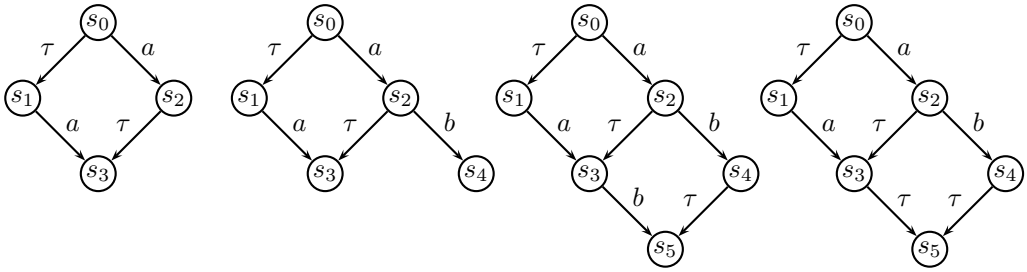
(2) or $a = \tau$ and $s' = s''$.

The union of confluent sets of τ-transitions is again confluent, so there is a maximal confluent set of τ-transitions.

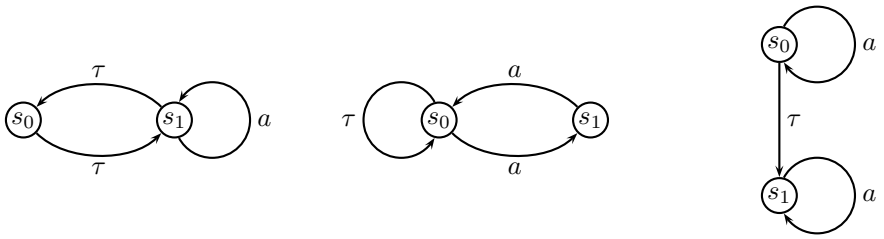
There are many variations of τ-confluence. For instance, it is possible to require that s' goes to s''' with zero or more τ-transitions (all in U). It can also be required that from s'' the state s''' can be reached via a sequence of τ's, an a and again a number of τ's. It can even be required that $s' \xrightarrow{\tau} s'''$ and $s'' \xrightarrow{a} s''''$ where s''' and s'''' are related via for instance branching bisimulation. We do not consider these extended versions here. See [25] for the treatment of some of these cases.

Example 10.1.2. Note that in figure 10.1 the maximal set of confluent τ transitions is the set of all τ transitions. However, by removing a single τ transition the confluence property can be destroyed. For instance the removal of the τ-transition to the terminating state, implies that all other τ-transitions that originate from hiding the action b, violate the confluence property.

Exercise 10.1.3. Give the maximal τ-confluent sets of τ-transitions of the following four state spaces:



Exercise 10.1.4. Give the maximal τ-confluent sets for the following three state spaces:



Exercise 10.1.5. Consider a finite transition system A with a finite number of states. Let U and U' be τ-confluent sets of transition systems. Show that $U \cup U'$ is also a τ-confluent transition system. Argue that this shows the existence of a unique maximal τ-confluent set of τ-transitions for A.

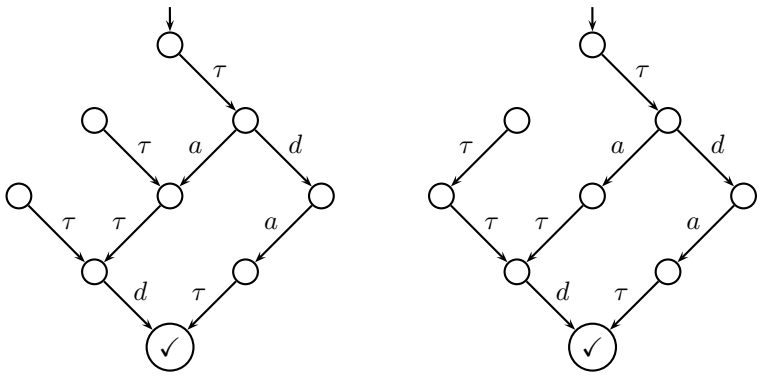


Figure 10.3: The state space of figure 10.1 after prioritisation

In order to prove the same property for an infinite state transition system it is necessary to consider an infinite set U_i ($i \in I$) of τ-confluent sets. It must be shown that U, defined by

$$U = \bigcup_{i \in I} U_i,$$

is τ-confluent.

Exercise 10.1.6. Prove that in a transition system A with a τ-confluent set of transitions U, every transition $s \xrightarrow{\tau} s'$ in U is *inert*, i.e. s and s' are branching bisimilar. Note that we do not need τ-convergence to prove this.

10.2 τ-prioritisation labelled transition systems

If a transition system is τ-confluent and τ-convergent, then τ-prioritisation preserves branching bisimulation. The τ-prioritisation means that in any state with an outgoing τ that occurs in a τ-confluent set of transitions, this transition can be taken, and the others can be omitted. For the transition system in figure 10.1 there are two ways to prioritise. Both are depicted in figure 10.3. Note that in both cases the reachable state space is the same. After applying the first τ law, the state space reduces to that of $\tau{\cdot}(a \parallel d)$.

The following definition formulates that τ-prioritisation is not an operation on state spaces, but rather a relation between them.

Definition 10.2.1. Let $A_1 = (S, Act, \rightarrow_1, s_0, T)$ and $A_2 = (S, Act, \rightarrow_2, s_0, T)$ be two transition systems. Let U be a set of τ-transitions. We call A_2 a τ-prioritisation of A_1 with respect to $U \subseteq \rightarrow$ iff for all states $s, s' \in S$

- if $s \xrightarrow{a}_2 s'$ then $s \xrightarrow{a}_1 s'$, and
- if $s \xrightarrow{a}_1 s'$, then $s \xrightarrow{a}_2 s'$ or $s \xrightarrow{\tau}_2 s'' \in U$ for some state $s'' \in S$.

Theorem 10.2.2. Let $A_1 = (S, Act, \rightarrow_1, s_0, T)$ and $A_2 = (S, Act, \rightarrow_2, s_0, T)$ be transition systems. Let $U \subseteq \longrightarrow_1$ be a τ-convergent set of τ-transitions, A_2 a τ-prioritisation of A_1 with respect to U and let A_2 be τ-convergent. Then A_1 and A_2 are branching bisimilar.

Proof. Consider the identity $R \subseteq S \times S$ defined by

$$R = \{\langle s_1, s_n \rangle \in S \times S \mid s_i \xrightarrow{\tau}_1 s_{i+1} \text{ is a transition in } U \text{ for all } 1 \leq i < n\}.$$

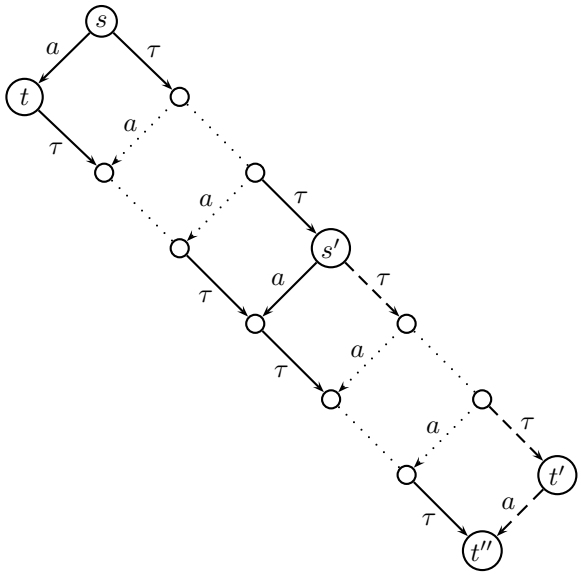
We show that R is a branching bisimulation relation between the states of A_1 and A_2. As A_2 is τ-convergent, the states of A_2 can be ordered on the basis of the number of τ's that still can be performed. We prove with induction on this ordering that R is a branching bisimulation relation.

Consider a pair of states $\langle s, s' \rangle \in R$. By definition of R it holds that $s = s_1 \xrightarrow{\tau}_1 s_2 \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s_n = s'$ and all these τ-transitions occur in U.

First suppose that $s' \xrightarrow{a}_2 t'$. Hence, this step is mimicked in A_1 by the sequence $s \xrightarrow{\tau}_1 s'' \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s' \xrightarrow{a}_1 t'$. That the conditions for branching bisimulation hold can be seen as follows. For all s_i it holds that $\langle s_i, s' \rangle \in R$ and moreover $\langle t', t' \rangle \in R$.

Now suppose $s \xrightarrow{a}_1 t$. If $a = \tau$ and $t = s''$ we mimic this step by staying in s'. As $s'' \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 s'$ we conclude that $\langle t, s' \rangle \in R$.

If $a \neq \tau$ or $t \neq s''$, then we get the following situation. The solid arrows are those occurring in A_1 and may not occur in A_2. The dashed arrows also occur in A_2.



First observe that as A_2 is τ-convergent from s' there is a finite sequence of τ-transitions to a state t' which has no outgoing τ-transition. As $s \xrightarrow{a}_1 t$ and $s \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 t'$, it follows using τ-confluence that $t \xrightarrow{\tau}_1 \cdots \xrightarrow{\tau}_1 t''$ and $t' \xrightarrow{a}_1 t''$. As t' has no outgoing τ transition, it follows via τ-prioritisation that $t' \xrightarrow{a}_2 t''$. So, we observe that if $s \xrightarrow{a}_1 t$, then $s' \xrightarrow{\tau}_2 \cdots \xrightarrow{\tau}_2 t' \xrightarrow{a}_2 t''$. Moreover, for all states s'' on the τ-path from s' to t' in A_2 it holds that $\langle s, s'' \rangle \in R$. Furthermore, $\langle t, t'' \rangle \in R$ as there is a τ-path between t and t'' in A_1.

So, we can conclude that R is a branching bisimulation relation. In particular, as the initial states $\langle s_0, s_0 \rangle \in R$ both transition systems A_1 and A_2 bisimilar. □

The next example shows that τ-convergence is essential for the soundness of prioritisation of confluent τ-transitions.

Example 10.2.3. This example shows that τ-convergence is essential for the soundness of prioritisation of confluent τ-transitions.

Consider the state space defined by the process declaration $X = (\tau + a) \cdot X$; note that it contains a τ-loop. The τ-transition in this state space is confluent. If the a-transition is eliminated from this state space, then the resulting state space is defined by the process declaration $Y = \tau \cdot Y$. Clearly the state belonging to X and the state belonging to Y are not branching bisimilar.

The next example shows that in general the maximal confluent set of τ-transitions may be a proper subset of the set of inert τ-transitions of a state space.

Example 10.2.4. Consider the state space

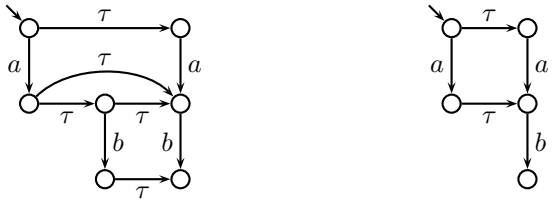
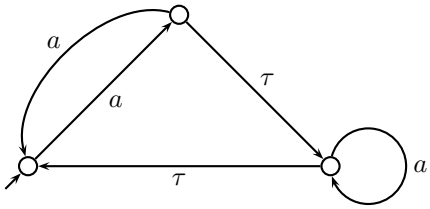


Figure 10.4: Repeated τ-prioritisation makes sense



This state space is minimal modulo bisimulation equivalence, and its maximal confluent set of τ-transitions is empty. However, the two τ-transitions are both inert.

Example 10.2.5. Finally, we note that after compression of a state space on the basis of its maximal confluent set of τ-transitions, the resulting state space may again contain confluent τ-transitions. Hence, it makes sense to iterate τ-prioritisation until the maximal confluent set of τ-transitions in the resulting state space has become empty.

In figure 10.4 we provide a state space before and after compression with respect to the confluent τ-transitions. Compression with respect to the confluent τ-transitions in the latter state space produces the state space belonging to $a{\cdot}b{\cdot}\delta$.

Exercise 10.2.6. Apply τ-prioritisation to all transition systems in examples 10.1.3 and 10.1.4. Which prioritised transition systems are branching bisimilar to to their original. Is this in accordance with the theory?

10.3 Confluence and linear processes

The application of τ-confluence and τ-prioritisation can give rise to tremendous reduction of a transition system while maintaining branching bisimulation. However, the techniques as presented up till now can only be applied if the transition system has been obtained first.

In this section we show how to define and prove τ-confluence on linear processes. When generating a state space from a linear process, τ-prioritisation can be applied on the fly and consequently the generated state space can be much smaller than that generated without the use of confluence. We first define τ-confluence in terms of linear processes that do not contain time. In figure 10.5 the situation is depicted.

Definition 10.3.1 (τ-confluence for LPEs). Assume a clustered LPE as defined in definition 9.1.6. We call this LPE τ-confluent iff for all actions $a \in Act$, for all $d{:}D$, $e_a{:}E_a$, $e_\tau{:}E_\tau$,
if $c_a(d, e_a) \wedge c_\tau(d, e_\tau)$ then

- either $a = \tau$ and $g_a(d, e_a) = g_\tau(d, e_\tau)$,
- or

$$c_a(g_\tau(d, e_\tau), e_a) \ \wedge \ c_\tau(g_a(d, e_a), e_\tau) \wedge$$
$$f_a(d, e_a) = f_a(g_\tau(d, e_\tau), e_a) \ \wedge \ g_a(g_\tau(d, e_\tau), e_a) = g_\tau(g_a(d, e_a), e_\tau).$$

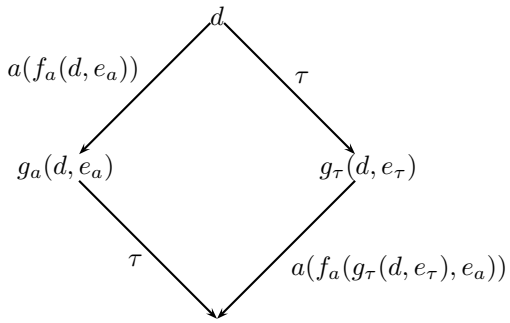


Figure 10.5: τ-confluence for a linear process

Example 10.3.2. We consider two unbounded queues in sequence of example 9.3.4. In this example the following process equation has been derived:

$$\begin{array}{lcl} X(q_1,q_2{:}List(D)) & = & \sum_{d:D} r_1(d)\cdot X(d \triangleright q_1, q_2) \\ & + & (q_2 \not\approx []) \rightarrow s_3(rhead(q_2))\cdot X(q_1, rtail(q_2)) \\ & + & (q_1 \not\approx []) \rightarrow \tau\cdot X(rtail(q_1), rhead(q_1) \triangleright q_2). \end{array}$$

We compute the confluence formulas.

1. Confluence for $a = \tau$. In this case we only write down the first part of the confluence formula, namely where $a = \tau$. Note that because this process has state variables q_1 and q_2, the conditions for the nextstate function g_a are split in two cases.

$$q_1 \not\approx [] \wedge q_1 \not\approx [] \Rightarrow ((rtail(q_1) = rtail(q_1) \wedge rhead(q_1) \triangleright q_2 = rhead(q_1) \triangleright q_2).$$

2. Confluence for $r_1(d)$:

$$\forall d{:}D.(q_1 \not\approx [] \Rightarrow (d \triangleright q_1 \not\approx [] \wedge d{=}d \wedge d \triangleright rtail(q_1){=}rtail(d \triangleright q_1) \wedge rhead(q_1) \triangleright q_2{=}rhead(d \triangleright q_1) \triangleright q_2)).$$

3. Confluence for $s_3(rhead(q_2))$ yields the following formula:

$$\begin{array}{c} q_1 \not\approx [] \wedge q_2 \not\approx [] \Rightarrow rhead(q_1) \triangleright q_2 \not\approx [] \wedge q_1 \not\approx [] \wedge rhead(q_2){=}rhead(rhead(q_1) \triangleright q_2) \wedge \\ rtail(q_1){=}rtail(q_1) \wedge rtail(rhead(q_1) \triangleright q_2){=}rhead(q_1) \triangleright rtail(q_2). \end{array}$$

Using the definitions of the standard operators on lists these formulas can straightforwardly be proven. In example 10.4.1 we show how we can employ the confluence of these concatenated queues.

Exercise 10.3.3. Consider the following LPEs. In each case, show whether or not the confluence formula for the τ-summand is true. If not, show a part of the generated transition system that is not τ-confluent.

(1) $$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & even(n) \rightarrow \tau\cdot X(n) \\ & + & true \rightarrow a(n)\cdot X(n{+}2) \end{array}$$

(2) $$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & even(n) \rightarrow \tau\cdot X(n{+}1) \\ & + & true \rightarrow a(n)\cdot X(n{+}2) \end{array}$$

(3) $$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & true \rightarrow \tau\cdot X(n{+}1) \\ & + & even(n) \rightarrow a\cdot X(n{+}2) \end{array}$$

10.4 τ-prioritisation for linear processes

If a linear process is τ-confluent, there are several ways to employ this information. The first one is by adding conditions to the linear process such that if a τ action is possible, other actions are blocked. Concretely consider a clustered linear process without time.

$$X(d{:}D) = \sum_{a\in Act}\sum_{e_a:E_a} c_a(d,e_a)\to a(d,e_a)\cdot X(g_a(d,e_a))$$

We apply τ-prioritisation to this process by adding the negation of $c_\tau(d,e_\tau)$ to every non τ-summand:

$$\begin{array}{lcl} X(d{:}D) & = & \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} c_a(d,e_a)\wedge\neg\exists e_\tau{:}E_\tau.c_\tau(d,e_\tau)\to a(d,e_a)\cdot X(g_a(d,e_a)) \\ & + & \sum_{e_\tau:E_\tau} c_\tau(d,e_\tau)\to\tau\cdot X(g_\tau(d,e_a)). \end{array}$$

Just as with τ-prioritisation for transition systems, this transformation of processes maintains branching bisimulation if the resulting transition system is τ-convergent.

Example 10.4.1. Consider the linear process from example 10.3.2 which was shown to be τ-confluent. Also note that this process is τ-convergent because the number of consecutive τ's that can be done is bounded by the size of q_1.

$$\begin{array}{lcl} X(q_1,q_2{:}List(D)) & = & \sum_{d:D} r_1(d)\cdot X(d\triangleright q_1,q_2) \\ & + & (q_2\not\approx[])\to s_3(rhead(q_2))\cdot X(q_1,rtail(q_2)) \\ & + & (q_1\not\approx[])\to\tau\cdot X(rtail(q_1),rhead(q_1)\triangleright q_2) \end{array}$$

If we add conditions to apply τ-prioritisation we obtain:

$$\begin{array}{lcl} X(q_1,q_2{:}List(D)) & = & (q_1\approx[])\to\sum_{d:D} r_1(d)\cdot X(d\triangleright q_1,q_2) \\ & + & (q_2\not\approx[]\wedge q_1\approx[])\to s_3(rhead(q_2))\cdot X(q_1,rtail(q_2)) \\ & + & (q_1\not\approx[])\to\tau\cdot X(rtail(q_1),rhead(q_1)\triangleright q_2) \end{array} \tag{10.1}$$

In other words, if q_1 is not empty, it is only possible to transfer data from the first queue to the second. So, if data is received via r_1, it must immediately be forwarded using an internal action to the second queue. More concretely, the expression in equation (10.1):

$$(q_1\approx[])\to\sum_{d:D} r_1(d)\cdot X(d\triangleright q_1,q_2)$$

is equal to (using equation (10.1)):

$$\begin{array}{ll} (q_1\approx[])\to\sum_{d:D} r_1(d)\cdot & (d\triangleright q_1\approx[])\to\sum_{d':D} r_1(d)\cdot X(d'\triangleright d\triangleright q_1,q_2) \\ & +(q_2\not\approx[]\wedge d\triangleright q_1\approx[])\to s_3(rhead(q_2))\cdot X(d\triangleright q_1,rtail(q_2)) \\ & +(d\triangleright q_1\not\approx[])\to\tau\cdot X(rtail(d\triangleright q_1),rhead(d\triangleright q_1)\triangleright q_2). \end{array}$$

This simplifies to:

$$(q_1\approx[])\to\sum_{d:D} r_1(d)\cdot\tau\cdot X(rtail(d\triangleright q_1),rhead(d\triangleright q_1)\triangleright q_2).$$

We can apply the first τ law to this equation and substitute this back in (10.1) yields:

$$\begin{array}{lcl} X(q_1,q_2{:}List(D)) & = & (q_1\approx[])\to\sum_{d:D} r_1(d)\cdot X(rtail(d\triangleright q_1),rhead(d\triangleright q_1)\triangleright q_2) \\ & + & (q_2\not\approx[]\wedge q_1\approx[])\to s_3(rhead(q_2))\cdot X(q_1,rtail(q_2)) \\ & + & (q_1\not\approx[])\to\tau\cdot X(rtail(q_1),rhead(q_1)\triangleright q_2) \end{array} \tag{10.2}$$

So, modulo branching bisimulation, the behaviour of the process X in equation (10.1) is also characterised by the previous equation. Now note that for this equation the simple invariant $q_1 = []$ holds. Using this invariant we can show using CL-RSP with invariants that $\lambda q_1,q_2{:}List(D).Y(q_1{+\!\!+}q_2)$ is a solution for X in 10.2 where Y is defined by

$$\begin{array}{lcl} Y(q{:}List(D)) & = & \sum_{d:D} r_1(d)\cdot Y(d\triangleright q) \\ & + & (q\not\approx[])\to s_3(rhead(q))\cdot Y(rtail(q)) \end{array}$$

So, we can conclude $Y([]) = X([],[])$.

10.4.1 Using confluence for state space generation

An effective way to use τ-confluence is to establish that a linear process is τ-confluent. Following an algorithm by Blom [8], it is then possible to generate a τ-prioritised state space, implicitly checking τ-convergence. If the linear process is not τ-convergent but has a finite state space, there is a τ-loop on which all the states are branching bisimilar. While generating the state space, all those states are taken together.

The algorithm of Blom applies the following three reductions when generating a state space.

1. It prioritises confluent tau's. So if a state s has confluent outgoing tau's it selects one of them and ignores all other outgoing transitions.

2. The target state s' of the confluent transition found above is substituted for s (except if s is the initial state), which is allowed by Milner's first tau law ($a \cdot \tau \cdot x = a \cdot x$). Note the τ transition is removed alltogether in this operation. In order to detect the outgoing transitions of s, the outgoing transitions of s' are immediately investigated.

3. Note that the step above can be repeated indefinitely, if s' has an outgoing confluent tau transition to a state with an outgoing confluent tau transition. If the state space is finite, this must ultimately lead to a loop of tau transitions. All the states on this loop are branching bisimilar, and hence can be taken together to one single state.

If this state space generation algorithm terminates, it yields a reduced state space which is branching bisimilar to the state space that would directly be generated from the linear process.

In order to get an idea about the effect of confluence reductions, the table below shows the number of states and transitions for the state spaces generated with and without taking confluence into account. The protocols chosen are the Alternating Bit Protocol (ABP), the Bounded Retransmission Protocol (BRP) both with a data domain of two elements, and the Tree Identify Protocol (TIP) with networks of 10, 12 and 14 nodes.

	standard state space		reduced state space	
system	states	transitions	states	transitions
ABP(2)	97	122	29	54
BRP(2)	1,952	2,387	1,420	1,855
TIP(10)	72,020	389,460	6,171	22,668
TIP(12)	446,648	2,853,960	27,219	123,888
TIP(14)	2,416,632	17,605,592	105,122	544,483

Part III

Checking conformance between specification and implementation

Chapter 11

Cones and foci

In this chapter we explain the cones and foci technique to prove an implementation of some system branching or weak bisimilar to its external behaviour. The main motivation behind the cones and foci technique is that it makes verification easier than the techniques that we have treated up till now. Establishing an explicit branching bisimulation relation between specification and implementation is not a very feasible technique if it comes to actual systems because it is far too hard to find and denote the required relation. In this chapter we only consider processes without time. Note that we follow in this chapter the verification technique by Fokkink and Pang [16] which is a more effective formulation than that in the original paper [26].

11.1 Cones and foci

The main idea of the cones and foci technique is that in most implementations hidden events progress inertly towards a state in which no hidden actions can be executed. We call such a state a *focus point*, as it is the focus towards which internal activity is directed. The *cone* of a focus point consists of the states from which this focus point can be reached by a sequence of hidden actions. Imagine the state space forming a cone or funnel pointing towards the focus. Figure 11.1 visualises the core idea underlying this method. Note that the external actions a, b, c and d are at the edge of the depicted cone. These are used to leave the cone and once they can take place, they can be executed in every subsequent state reachable by an internal step. In particular they can also be executed in the ultimate focus point; this is essential if one wants to apply the cones and foci technique, as otherwise the hidden actions in the cone would not be inert. For the cones and foci method, an implementation is viewed as a set of cones, as depicted in figure 11.2. All states in each cone are related to some state in the specification. One can move from cone to cone using externally visible actions, that directly match the actions in the external behaviour.

We assume that an implementation and a specification of some system are given by clustered linear process operators. The process X is the implementation given by the following linear process equation. The set Act of actions does not contain the special internal action int. The reason for taking int instead of the internal action τ is that a linear process with τ-loops is not guarded, and does not have a unique solution.

$$X(d{:}D) = \sum_{a \in Act \cup \{int\}} \sum_{e_a : E_a} c_a(d, e_a) \rightarrow a(f_a(d, e_a)) \cdot X(g_a(d, e_a)) \tag{11.1}$$

The following equation gives the specification. Note that this process does not contain any int's.

$$Y(d'{:}D') = \sum_{a \in Act} \sum_{e_a : E_a} c'_a(d', e_a) \rightarrow a(f'_a(d', e_a)) \cdot Y(g'_a(d', e_a)) \tag{11.2}$$

The process equations both have the same number of summands (except for the int-summand in the implementation), and in each summand the same sum operator is used. This is not a real restriction, as it is

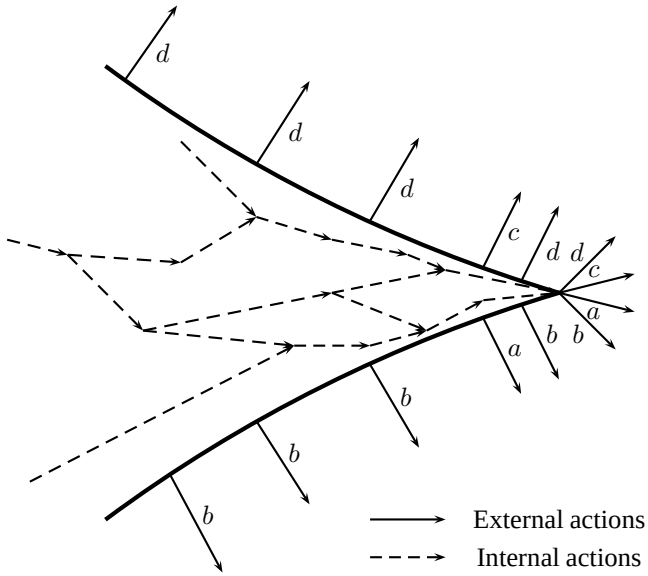


Figure 11.1: A cone with a focus point

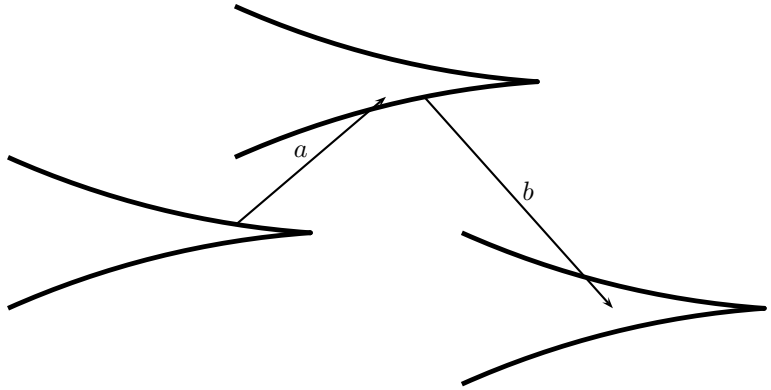


Figure 11.2: An implementation is a set of cones

always possible to align the implementation and specification in this respect, but in concrete cases, it may require some work to do so.

In order to relate the states of the implementation D to the states of the specification D' we introduce a *state mapping* $h : D \to D'$. It maps all the states in a cone to a corresponding (bisimilar) state in the specification. For concrete examples this state mapping is generally quite easy to define, because it reflects the intuition of how the implementation and specification are related. In order to show that the state mapping is indeed a proper branching or weak bisimulation relation, matching criteria must be proven, which are given below.

The endpoint of a cone is its focus point. We let a predicate $FC : D \to \mathbb{B}$ (*the focus condition*) indicate whether a state is a focus point or not. If there are no infinite sequences of internal actions, it suffices to let the focus points be exactly those states where no internal actions can be done. This means that $FC(d) = \forall e_{int}{:}E_{int}.\neg c_{int}(d, e_{int})$. If there are infinite internal sequences, the focus points must be chosen such that it can be shown that in every state that is not a focus point there is one internal transition that brings one closer to a focus point.

Generally, not all the states of the implementation are relevant, cf., figure 9.5. An invariant $\mathcal{I}$ can be used to restrict the statespace of the implementation to the relevant part. Contrary to formulating the

state mapping, finding the necessary invariants is often tedious and cumbersome. Fortunately, if the right invariants are found, it is straightforward to prove that the invariants satisfy the invariant property.

This state mapping h must satisfy a number of *matching criteria*, which ensure that the mapping establishes a branching bisimulation relation between the two LPEs in question, and moreover that all states in a cone of X are mapped to the same state in Y.

A state mapping $h : D \to D'$ satisfies the *matching criteria* if for all for $d \in D$:

1. If not in a focus point, there is at least one internal step such that the target state is closer to the focus point:

   $$\exists e_{int}{:}E_{int}.(\mathcal{I}(d) \wedge \neg FC(d)) \Rightarrow c_{int}(d, e_{int}) \wedge M(d) > M(g_{int}(d, e_{int}))$$

   where M is a well founded measure on D;

2. For every internal step, the state mapping h maps source and target state to the same state in the specification:

   $$\forall e_{int}{:}E_{int}.(\mathcal{I}(d) \wedge c_{int}(d, e_{int})) \Rightarrow h(d) = h(g_{int}(d, e_{int}));$$

3. Every visible action in the specification must be mimicked in the implementation in a focus point. For all $a \in \mathsf{Act}$

   $$\forall e_a{:}E_a.(\mathcal{I}(d) \wedge FC(d) \wedge c'_a(h(d), e_a) \Rightarrow c_a(d, e_a));$$

4. Every visible action in the implementation must be mimicked in the corresponding state in the specification. This means that for all actions $a \in \mathsf{Act}$ it must hold that

   $$\forall e_a{:}E_a.(\mathcal{I}(d) \wedge c_a(d, e_a)) \Rightarrow c'_a(h(d), e_a);$$

5. When a matching action in specification and implementation can be done, their parameters must be equal. For all actions $a \in \mathsf{Act}$ it is the case that

   $$\forall e_a{:}E_a.(\mathcal{I}(d) \wedge c_a(d, e_a)) \Rightarrow f_a(d, e_a) = f'_a(h(d), e_a);$$

6. For all matching actions in specification and implementation, their endpoints must be related. So, for all $a \in \mathsf{Act}$ we have

   $$\forall e_a{:}E_a.(\mathcal{I}(d) \wedge c_a(d, e_a)) \Rightarrow h(g_a(d, e_a)) = g'_a(h(d), e_a).$$

When the matching criteria are proven, the general equality theorem says that implementation and specification are the same in the sense of branching bisimulation and weak bisimulation.

Theorem 11.1.1 (General Equality Theorem). Let X and Y be defined by equations (11.1) and (11.2). Let $\mathcal{I}$ be an invariant for X. If $h : D \to D'$ satisfies the matching criteria defined above, then:

$$\mathcal{I}(d) \Rightarrow \ \tau{\cdot}\tau_{int}(X(d)) = \tau{\cdot}Y(h(d)).$$

Generally, this theorem is used for a concrete initial state d_0 for which the invariant is valid (i.e. $\mathcal{I}(d_0) = true$). In case there are no outgoing internal steps from d_0 in the implementation, the theorem can be slightly strengthened to $\tau_{int}(X(d_0)) = Y(h(d_0))$, i.e. the leading τ can be ommitted in that case.

Example 11.1.2. In figure 11.3 two processes are depicted, namely an implementation at the left and a specification at the right. In the implementation there are two cones ($\{1, 2\}$ and $\{3\}$) with respective focus points the states 2 and 3. There is a state mapping h that maps states 1 and 2 to state 4, and state 3 to 5. Note that all mapping criteria are valid, showing that both the specification and implementation are branching bisimilar.

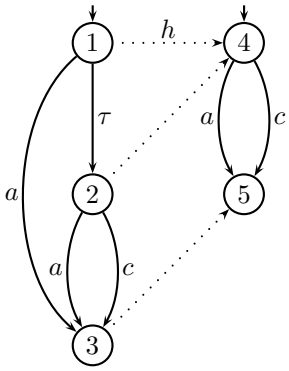


Figure 11.3: A simple pair of a specification and an implementation

Exercise 11.1.3. Let

$$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & (n|_3{\approx}0){\to}a{\cdot}X(n{+}1) + (n|_3{\approx}1){\to}b{\cdot}X(n{+}1) + (n|_3{\approx}2){\to}int{\cdot}X(n{+}1)) \\ Y(c{:}\mathbb{B}) & = & c \to a{\cdot}Y(false) + \neg c \to b{\cdot}Y(true) \end{array}$$

What is a suitable focus condition for X with an associated well-founded measure. Give a state mapping $h : \mathbb{N} \to \mathbb{B}$ such that the matching criteria are satisfied. Prove that the matching criteria are satisfied.

Exercise 11.1.4. Let

$$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & \sum_{m:\mathbb{N}}(n{\approx}3m){\to}a{\cdot}X(n{+}1) \\ & + & \sum_{m:\mathbb{N}}(n{\approx}3m{+}1){\to}b{\cdot}X(n{+}1) \\ & + & \sum_{m:\mathbb{N}}(n{\approx}3m{+}2){\to}int{\cdot}X(n{+}1)) \\ Y(c{:}\mathbb{B}) & = & c \to a{\cdot}Y(false) + \neg c \to b{\cdot}Y(true) \end{array}$$

Transform the first process such that the cones and foci theorem can be applied directly.

Exercise 11.1.5. Let

$$\begin{array}{lcl} X(n{:}\mathbb{N}) & = & \sum_{m:\mathbb{N}}(n{\approx}3m) \to a{\cdot}X(n+2) \\ & + & \sum_{m:\mathbb{N}}(n{\approx}3m{+}1) \to d{\cdot}X(n+1) \\ & + & \sum_{m:\mathbb{N}}(n{\approx}3m{+}2) \to int{\cdot}X(n+1) \\ Y & = & a{\cdot}Y \end{array}$$

Transform the first process such that the cones and foci theorem can be applied. Give a state mapping $h : \mathbb{N} \to Nil$ where Nil is a sort containing only one element. Formulate an adequate invariant $\mathcal{I} : \mathbb{N} \to \mathbb{B}$, valid for $n = 0$ and use it to show that the matching criteria hold.

11.2 Protocol verifications using the cones and foci proof technique

In this section we present the verifications of three protocols, using the cones and foci technique, namely two unbounded queues, Milner's scheduler and the alternating bit protocol.

11.2.1 Two unbounded queues form a queue

Consider the two bounded buffers of example 9.3.4. The two connected queues have the behaviour defined by the following linear equation, where the communication action c_2 is pre-hidden to int. We consider this to be the implementation.

$$\begin{array}{lcl} X(q_1, q_2{:}List(D)) & = & \sum_{d:D} r_1(d){\cdot}X(d \triangleright q_1, q_2) \\ & + & (q_2 {\not\approx} []){\to}s_3(rhead(q_2)){\cdot}X(q_1, rtail(q_2)) \\ & + & (q_1 {\not\approx} []){\to}int{\cdot}X(rtail(q_1), rhead(q_1) \triangleright q_2) \end{array}$$

The claim is that the behaviour of this process is equal to that of a single queue. So, the specification is characterised by

$$\begin{array}{lcl} Y(q{:}List(D)) & = & \sum_{d:D} r_1(d){\cdot}Y(d \triangleright q) \\ & + & (q \not\approx []) \rightarrow s_3(rhead(q)){\cdot}Y(rtail(q)) \end{array}$$

In order to prove the two queues equal, we need to indicate a state mapping between the two queues of X and the single queue in Y. The queue q in a sense consists of the concatenation of the queues q_1 and q_2. This is formalized by introducing the state mapping $h : List(D) \times List(D) \rightarrow List(D)$ defined by

$$h(q_1, q_2) = q_1 {+\!\!+} q_2.$$

As the implementation X cannot exhibit an infinite number of int steps, we can take the focus points X to be those states where no int action is possible. I.e.,

$$FC(q_1, q_2) = q_1 \approx [].$$

It turns out that an invariant is not necessary for this verification. The matching criteria now say that we must show that:

1. The process equation X is int-well-founded. This has been shown in 9.3.4.

2. The state mapping is preserved under internal steps:

$$q_1 {+\!\!+} q_2 = rtail(q_1) {+\!\!+} (rhead(q_1) \triangleright q_2).$$

3. Every visible action Y must be mimicked in a focus point of X:

r_1: $\forall d{:}D.q_1 \approx [] \wedge true \Rightarrow true$.
s_3: $q_1 \approx [] \wedge q_1 {+\!\!+} q_2 \not\approx [] \Rightarrow q_2 \not\approx []$.

4. Every visible action in the implementation must be possible in the related state in the specification

r_1: $\forall d{:}D.true \Rightarrow true$.
s_3: $q_2 \not\approx [] \Rightarrow (q_1 {+\!\!+} q_2) \not\approx []$.

5. The parameters of actions in implementation and specification must match:

r_1: $\forall d{:}D.d = d$.
s_3: $q_2 \not\approx [] \Rightarrow rhead(q_2) = rhead(q_1 {+\!\!+} q_2)$.

6. The endpoints of actions must be related.

r_1: $\forall d{:}D.(d \triangleright q_1) {+\!\!+} q_2 = d \triangleright (q_1 {+\!\!+} q_2)$.
s_3: $q_2 \not\approx [] \Rightarrow q_1 {+\!\!+} rtail(q_2) = rtail(q_1 {+\!\!+} q_2)$.

Note that most of the properties are trivial. Others must be proven using induction on the lengths of queues. From the general equality theorem we draw the conclusion that

$$\tau{\cdot}\tau_{\{int\}}(X(q_1, q_2)) \;=\; \tau{\cdot}Y(q_1 {+\!\!+} q_2).$$

As there in no internal step possible in the implementation in the initial state (with $q_0 = []$ and $q_1 = []$), we can even conclude:

$$\tau_{\{int\}}(X([], [])) = Y([]).$$

11.2.2 Milner's scheduler

In section 9.2.15 the behaviour of a scheduler was calculated for n individual cyclers. Here we show one of the two main properties of the scheduler, namely that the scheduler performs subsequent $on(k)$ actions for increasing k. This behaviour is described by

proc $\quad X(k,n{:}\mathbb{N}^+) = on(k){\cdot}X(k|_n{+}1,n);$

So, the theorem that we want to prove in branching bisimulation is

$$X(1,n) = \tau_{\{d,off\}}(Scheduler(n,\lambda m{:}\mathbb{N}.if(m{\approx}1,2,1)))$$

The scheduler calculated in section 9.2.15 is

$$\begin{array}{l} Scheduler(n{:}\mathbb{N}^+, f{:}\mathbb{N}^+{\to}\mathbb{N}^+) = \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}2){\to}on(k){\cdot}Scheduler(k, f[k:=3]) + \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}3){\to}off(k){\cdot}Scheduler(k, f[k:=5]) + \\ \quad \sum_{k:\mathbb{N}^+}(f(k){\approx}4){\to}off(k){\cdot}Scheduler(k, f[k:=1]) + \\ \quad \sum_{\ell:\mathbb{N}^+}(f(\ell|_n{+}1){\approx}1) \wedge f(\ell){\approx}3 \wedge \ell{\leq}n){\to}d(\ell){\cdot}Scheduler(n, f[\ell|_n{+}1:=2][\ell:=4]) + \\ \quad \sum_{\ell:\mathbb{N}^+}(f(\ell|_n{+}1){\approx}1) \wedge f(\ell){\approx}5 \wedge \ell{\leq}n){\to}d(\ell){\cdot}Scheduler(n, f[\ell|_n{+}1:=2][\ell:=1]) \end{array} \tag{11.3}$$

In order to apply the cones and foci theorem, we must make the sum operators in front of the visible actions equal. Therefore, we rewrite the linear equation for X to the equivalent:

proc $\quad X(k',n{:}\mathbb{N}^+) = \sum_{k:\mathbb{N}^+}(k{\approx}k'){\to}on(k){\cdot}X(k|_n{+}1,n);$

First we observe that there are two relevant invariants for the scheduler. The first one is that for all $j > n$, it is the case that $f(j) = 1$. The second one is that there is exactly one unique $j \leq n$ such that $f(j) \in \{2,3,5\}$. Using this latter invariant we define $N(f)$ as follows:

$$N(f) = \begin{cases} k & \text{if } f(k) = 2 \text{ for some } k, \\ k|_n{+}1 & \text{if } f(k) \in \{3,5\} \text{ for some } k. \end{cases}$$

The state mapping $h(n,f) = \langle N(f), n\rangle$ and the focus condition is $FC(n,f) = \forall j{:}\mathbb{N}^+.(j{\approx}N(f) \vee f(j){\approx}1)$. The matching criteria become:

1. All cyclers are forced to do an on step within a finite number of steps. So, there cannot be a loop of internal steps in the scheduler.
2. This case deals with preservation of the state mapping by internal steps. It can be split in a number of cases for all internal summands:
   - $\forall k{:}\mathbb{N}^+.(f(k){\approx}3 \to N(f) = N(f[k:=5]))$.
   - $\forall k{:}\mathbb{N}^+.(f(k){\approx}4 \to N(f) = N(f[k:=1]))$.
   - $\forall \ell{:}\mathbb{N}^+.(f(\ell|_n{+}1{\approx}1) \wedge f(\ell){\approx}3 \wedge \ell{\leq}n \Rightarrow N(f) = N(f[\ell|_n{+}1{:=}2][\ell{:=}4]))$.
   - $\forall \ell{:}\mathbb{N}^+.(f(\ell|_n{+}1{\approx}1) \wedge f(\ell){\approx}5 \wedge \ell{\leq}n \Rightarrow N(f) = N(f[\ell|_n{+}1{:=}2][\ell{:=}1]))$.
3. $\forall k{:}\mathbb{N}^+.(N(f) = k \wedge FC(n,f) \Rightarrow f(k){\approx}2)$.
4. $\forall k{:}\mathbb{N}^+.(f(k){\approx}2 \Rightarrow N(f) = k)$.
5. $\forall k{:}\mathbb{N}^+.(f(k){\approx}2 \Rightarrow k = k)$.
6. $\forall k{:}\mathbb{N}^+.(f(k){\approx}2 \Rightarrow N(f)|_n{+}1 = N(f[k{:=}3]) \wedge n = n)$.

These matching criteria can straightforwardly be checked. Note that the invariants are made valid by the initial state. Also note that as the focus condition is valid in the initial state, and there are no internal steps possible for states where the focus condition holds, the initial τ's that come with the cones and foci theorem can be ommitted.

11.2.3 The alternating bit protocol

We consider the specification given in section 6.1 of the alternating bit protocol. This sections ends with equation (6.1) saying that the behaviour of the alternating bit protocol ABP is equal to that of a simple buffer B. Here we prove this equation using the cones and foci technique.

The first step is to linearise the behaviour of $\Upsilon_{\{c_2,c_3,c_5,c_6,i\}}(ABP)$, i.e. the behaviour of the alternating bit protocol where the internal actions are renamed to the visible internal action int. The result of linearisation is the following process equation:

$$
\begin{aligned}
&X(s_S{:}\mathbb{N}^+, d_S{:}D, b_S{:}\mathbb{B}, s_R{:}\mathbb{N}^+, d_R{:}D, b_R{:}\mathbb{B}, s_K{:}\mathbb{N}^+, d_K{:}D, b_K{:}\mathbb{B}, s_L{:}\mathbb{N}^+, b_L{:}\mathbb{B}) = \\
&\quad \textstyle\sum_{d:D}(s_S{\approx}1) \to r_1(d){\cdot}X(s_S := 2, d_S := d) + \\
&\quad (s_S{\approx}2 \wedge s_K{\approx}1) \to int{\cdot}X(s_S := 3, s_K := 2, d_K := d_S, b_K := b_S) + \\
&\quad (s_K{\approx}2) \to int{\cdot}X(s_K := 3) + \\
&\quad (s_K{\approx}2) \to int{\cdot}X(s_K := 4) + \\
&\quad (s_R{\approx}1 \wedge s_K{\approx}3 \wedge b_R{\not\approx}b_K) \to int{\cdot}X(s_R := 4, s_K := 1) + \\
&\quad (s_R{\approx}1 \wedge s_K{\approx}3 \wedge b_R{\approx}b_K) \to int{\cdot}X(s_R := 2, d_R := d_K, s_K := 1) + \\
&\quad (s_R{\approx}1 \wedge s_K{\approx}4) \to int{\cdot}X(s_R := 4, s_K := 1) + \\
&\quad (s_R{\approx}2) \to s_4(d_R){\cdot}X(s_R := 3) + \\
&\quad (s_R{\approx}3 \wedge s_L{\approx}1) \to int{\cdot}X(s_R := 1, b_R := \neg b_R, s_L := 2, b_L := b_R) + \\
&\quad (s_R{\approx}4 \wedge s_L{\approx}1) \to int{\cdot}X(s_R := 1, s_L := 2, b_L := \neg b_R) + \\
&\quad (s_L{\approx}2) \to int{\cdot}X(s_L := 3) + \\
&\quad (s_L{\approx}2) \to int{\cdot}X(s_L := 4) + \\
&\quad (s_S{\approx}3 \wedge s_L{\approx}3 \wedge b_S{\approx}b_L) \to int{\cdot}X(s_S := 1, b_S := \neg b_S, s_L := 1) + \\
&\quad (s_S{\approx}3 \wedge s_L{\approx}3 \wedge b_S{\not\approx}b_L) \to int{\cdot}X(s_S := 2, s_L := 1) + \\
&\quad (s_S{\approx}3 \wedge s_L{\approx}4) \to int{\cdot}X(s_S := 2, s_L := 1)
\end{aligned}
$$

The process *ABP* and the linear process X are related by

$$\Upsilon_{\{c_2,c_3,c_5,c_6,i\}}(ABP) = X(s_S := 1, b_S := true, s_R := 1, b_R := true, s_K := 1, s_L := 1). \quad (11.4)$$

For this linear process we can establish the following invariant properties that can straightforwardly be proven.

1. Out of $s_S{\approx}3$, $s_R{\approx}1$, $s_K{\approx}1$, $s_L{\approx}1$ exactly one does not hold.
2. $(s_K{\approx}2 \vee s_K{\approx}3) \Rightarrow b_S{\approx}b_K \wedge d_S{\approx}d_K$;
3. $(s_L{\approx}2 \vee s_L{\approx}3) \Rightarrow (b_L{\not\approx}b_R)$;
4. $s_S{\approx}1 \Rightarrow (b_S{\approx}b_R)$;
5. $(s_R{\approx}2 \vee s_R{\approx}3) \Rightarrow (d_S{\approx}d_R \wedge b_S{\approx}b_R)$;

The buffer can be described by the linear equation

$$
\begin{aligned}
B(d_B{:}D, b_B{:}\mathbb{B}) \;&=\; \textstyle\sum_{d:D} b_B \to r_1(d){\cdot}B(d, false) \\
&+\; \neg b_B \to s_4(d_B){\cdot}B(d_B, true)
\end{aligned}
$$

Note that the sum operators of the visible actions match neatly, which means that we can directly apply the cones and foci method. The state mapping must map the sizeable state vector of X to that of B. We select the contents of the buffer to be that of the sender. So, the state mapping maps d_S to d_B. Invariant 2 guarantees when the data is delivered in the implementation $s_4(d_R)$, that $d_R = d_S$.

The boolean d_B is true when the buffer can read and false when it can write. In the implementation, reading is possible after data is delivered via $s_4(d_R)$, although some internal activity is needed before reading is actually possible. After the action $s_4(d_R)$, it is the case that $s_R{\approx}3$. At the next step of R, the variable s_R is set to 1 and the bit b_R is flipped (see the 9th summand of X). Using invariant 5 we see that from that point $b_S{\not\approx}b_R$. The bits are only made equal again when the reader receives an acknowledgement,

in summand 13 of X, setting s_S to 1, indicating that it is ready to read. So, the situation where the alternating bit protocol is (eventually) willing to read is characterised by:

$$s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1.$$

So, the state mapping becomes:

$$h(s_S, d_S, b_S, s_R, d_R, b_R, s_K, d_K, b_K, s_L, b_L) = \langle d_S, s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1\rangle.$$

The focus condition is the situation where external actions are done. In this case we define the focus condition to be:

$$FC(s_S, s_R) := s_S{\approx}1 \vee s_R{\approx}2$$

as these are the situations where the process can read and write.

It is now straightforward to write down the matching criteria. We omit stating the invariant in each criterion as this would make the criteria unreadable:

1. It is tricky to provide a decreasing measure on the state space. It is more effective to look at the transition system of the ABP in figure 11.4. There it can be seen that there is always an internal step bringing one closer to the focus point, i.e. the situation where a r_1 or a s_4 must be done.

2. In this case we must show that no internal step has any effect on the state mapping h. As there are 13 internal steps, we must check the following 13 cases. The first part of each case is to show that the first part of the state mapping, d_S does not change. This is trivial and therefore ommitted, as d_S is only changed in the first summand with visible action s_4. So, below we only concentrate on the second part. The cases where the right hand side of the implication is the identity, are omitted. So, we only deal with summands (2), (5), (6), (7), (9), (10), (13), (14), (15).

   $$s_S{\approx}2 \wedge s_K{\approx}1 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = s_R{\approx}3 \vee b_S{\not\approx}b_R \vee 3{\approx}1);$$
   $$s_R{\approx}1 \wedge s_K{\approx}3 \wedge b_R{\not\approx}b_K \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = 4{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1);$$
   $$s_R{\approx}1 \wedge s_K{\approx}3 \wedge b_R{\approx}b_K \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = 2{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1);$$
   $$s_R{\approx}1 \wedge s_K{\approx}4 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = 4{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1);$$
   $$s_R{\approx}3 \wedge s_L{\approx}1 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = 1{\approx}3 \vee b_S{\not\approx}{\neg}b_R \vee s_S{\approx}1);$$
   $$s_R{\approx}4 \wedge s_L{\approx}1 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = 1{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1);$$
   $$s_S{\approx}3 \wedge s_L{\approx}3 \wedge b_S{\approx}b_L \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = s_R{\approx}3 \vee {\neg}b_S{\not\approx}b_R \vee 1{\approx}1);$$
   $$s_S{\approx}3 \wedge s_L{\approx}3 \wedge b_S{\not\approx}b_L \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = s_R{\approx}3 \vee b_S{\not\approx}b_R \vee 2{\approx}1);$$
   $$s_S{\approx}3 \wedge s_L{\approx}4 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = s_R{\approx}3 \vee b_S{\not\approx}b_R \vee 2{\approx}1).$$

   Allmost all cases are trivial to prove. For instance, in the first case using that the condition provides that $s_S{\approx}2$, the right hand side reduces to $s_R{\approx}3 \vee b_S{\not\approx}b_R = s_R{\approx}3 \vee b_S{\not\approx}b_R$, which is the identity.

   The criterion for summand (9) reduces to

   $$s_R{\approx}3 \wedge s_L{\approx}1 \Rightarrow (true = b_S{\not\approx}{\neg}b_R \vee s_S{\approx}1).$$

   From invariant 1 it follows that $s_S{\approx}1$, making the right hand side equal to $true$.

   The case for summand (13) leads to the proof obligation

   $$s_S{\approx}3 \wedge s_L{\approx}3 \wedge b_S{\approx}b_L \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R = true).$$

   From invariant 3 it follows that $b_L{\not\approx}b_R$. Together with the condition it follows that $b_S{\not\approx}b_R$, from which it is obvious that the proof obligation holds.

3. r_1. Using the focus condition this criterion becomes:

   $$(s_S{\approx}1 \vee s_R{\approx}2) \wedge (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1) \Rightarrow s_S{\approx}1.$$

   Suppose that the conditions would warrant the conclusion that $s_S{\not\approx}1$. Then from the first part of the condition $s_R{\approx}2$. From the second part it follows that $b_S{\not\approx}b_R$. But this is in contradiction with invariant 5.

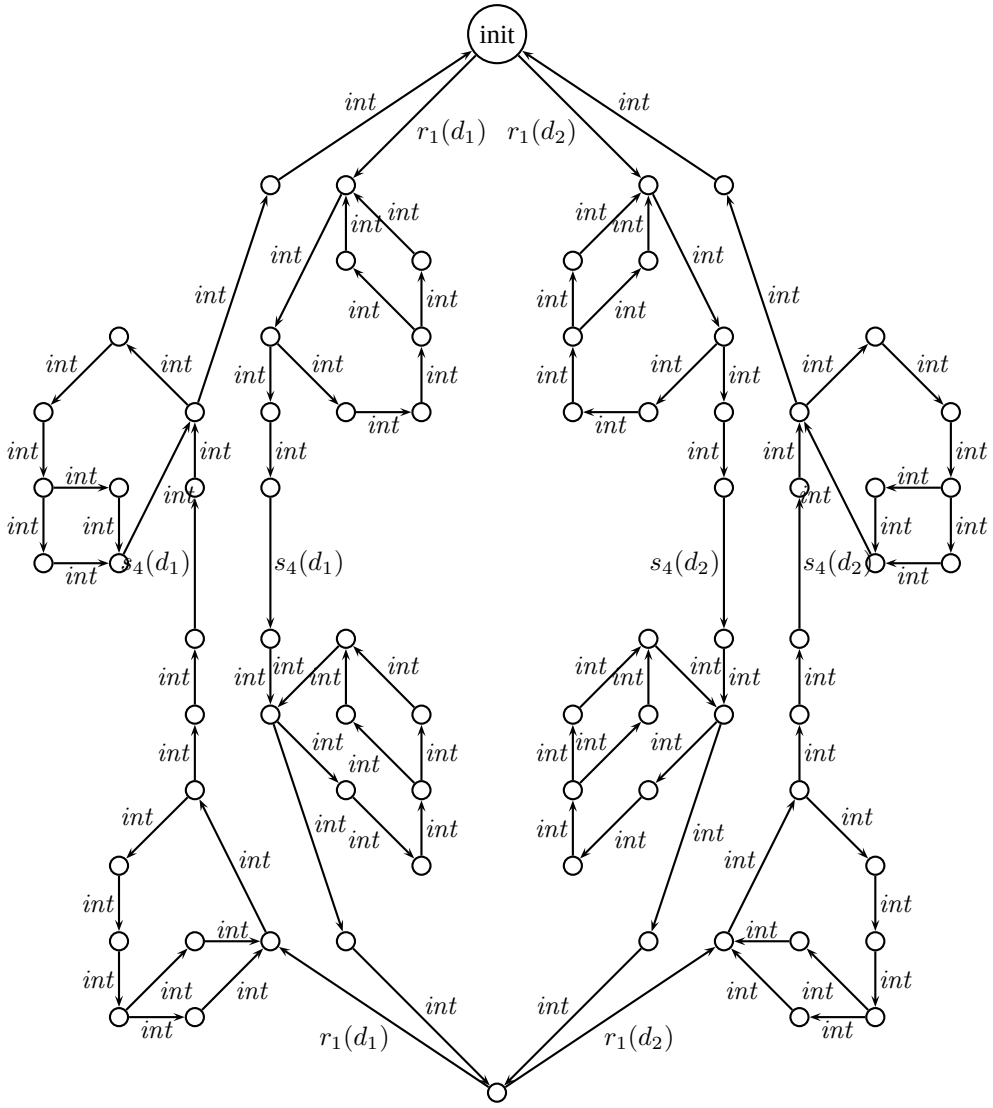


Figure 11.4: The state space of the alternating bit protocol

s_4. We must show that

$$(s_S{\approx}1 \vee s_R{\approx}2) \wedge \neg(s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1) \Rightarrow s_R{\approx}2.$$

From the second part of the condition it follows that $s_S{\not\approx}1$. From the first part we then conclude $s_R{\approx}2$;

4. r_1. This matching criterion is trivial to prove: $s_S{\approx}1 \Rightarrow (s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1)$;

   s_4. In this case the matching criterion is $s_R{\approx}2 \Rightarrow \neg(s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1)$. So, $s_R{\approx}2$ must imply $s_R{\not\approx}3$, $b_S{\approx}b_R$ and $s_S{\not\approx}1$. The first case is trivial and the second case follows immediately from invariant 5. The last case is a direct consequence of invariant 1;

5. r_1. In this case we have a trivial matching criterion $s_S{\approx}1 \Rightarrow d = d$;

   s_4. This matching criterion is a direct consequence of invariant 5 $s_R{\approx}2 \Rightarrow d_R = d_S$;

6. r_1. For this case we must prove equality for the first and the second argument of the buffer. So, this becomes:

$$s_S{\approx}1 \Rightarrow d = d$$
$$s_S{\approx}1 \Rightarrow s_R{\approx}3 \vee b_S{\not\approx}b_R \vee 2{\approx}1 = \mathit{false}$$

The first case is trivial. The second follows immediately from invariant 4 and 1;

s_4. In this case we must show

$$s_R{\approx}2 \Rightarrow d_S = d_S$$
$$s_R{\approx}2 \Rightarrow 3{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1 = \mathit{true}$$

which is fully trivial.

This finishes checking the matching criteria. So, using the General Equality Theorem 11.1.1, we can conclude for state vectors where the invariant holds that:

$$\tau{\cdot}\tau_{\{int\}}(X(s_S, d_S, b_S, s_R, d_R, b_R, s_K, d_K, b_K, s_L, b_L)) = \tau{\cdot}B(d_S, s_R{\approx}3 \vee b_S{\not\approx}b_R \vee s_S{\approx}1).$$

Hence, using equation (11.4), we can conclude for any d_S that

$$\tau{\cdot}B(d_S, \mathit{true}) = \tau{\cdot}\tau_{\{int\}}(\Upsilon_{\{c_2,c_3,c_5,c_6,i\}}(ABP)) = \tau{\cdot}\tau_{\{c_2,c_3,c_5,c_6,i\}}(ABP).$$

As a final touch, because ABP does not start with an internal action, we can even conclude for any $d_S{:}D$:

$$B(d_S, \mathit{true}) = \tau_{\{c_2,c_3,c_5,c_6,i\}}(ABP).$$

Chapter 12

Verification of distributed systems

This chapter contains the verification of three slightly more complex distributed algorithms. The first one is the *tree identify protocol*, which is a leader election protocol for a tree shaped network. Here we only consider the simple case without contention. The proof is taken from [46], where also the more complex verification can be found.

The second verification regards the sliding window protocol, as already introduced in section 6.2. This proof is a simplification of the proof in [1], which was a real tour de force. Fortunately, the proof in section 12.2 does in no way reflect the hardship to find the proof. The sliding window protocol is interesting for two reasons. In the first place it heavily relies on modulo calculations. In the second place, the sliding window protocol is the workhorse for the internet, and so, everybody is continuously using this protocol.

The third verification is about the distributed summing protocol. This verification was published as [21]. The interesting aspect of this algorithm is that it is highly non-deterministic. This actually prompted some authors to claim that such nondeterministic algorithms cannot be proven correct in any effective and precise manner. Section 12.3 shows that this is not true.

12.1 Tree identify protocol

In this section we concentrate on the tree identify phase (TIP) of the IEEE1394 or Firewire protocol [31]. This is a rather simple protocol to determine a leader in a network that does not contain cycles. We assume that the nodes communicate directly with each other. The protocol and the proof are taken from [46]. This article also contains a variant of the TIP protocol where the communication between the nodes is asynchronous. This introduces new problems, for which a root contention phase is required. In our setting root contention does not occur.

We assume a network, consisting of a collection of nodes and connections between nodes. The aim of the TIP is to establish a single leader of the network. This is done by establishing parent-child relations between connected nodes. Every node tries to establish exactly one node as its parent. As the network does not contain cycles, every node will ultimately have at most one parent, and exactly one node has no parent at all. This last node becomes the leader in the network.

In order to establish parent-child relations, a node sends a *parent request* to a neighbouring node, asking that node to become its parent. A parent request from node i to node j is represented by the action $s(i, j)$, which communicates with the read action $r(i, j)$ to $c(i, j)$.

Each node keeps track of the neighbours from which it has not yet received a parent request. Initially this list contains all neighbours. If a node i is the parent of all its neighbours except of some node j, then i is sends a parent request to j. In the case that a node received parent requests from all its neighbours, it is the only parent-less node in the network, and it declares itself root of the network.

Below the specification of a single node is given. The process $Node(i, p, s)$ represents node i in state s, with p as the list of possible parents.

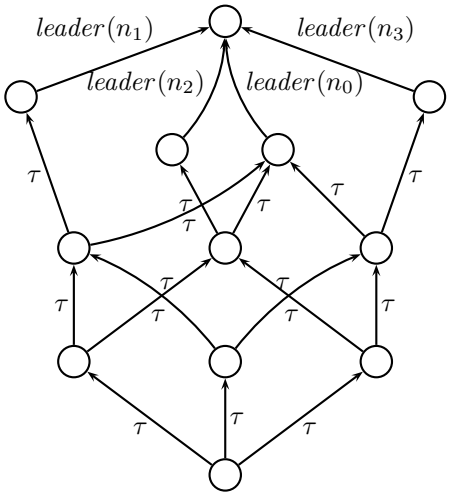


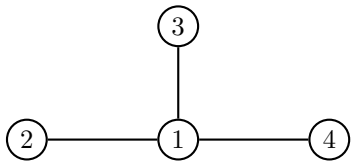
Figure 12.1: External behaviour of the TIP from example 12.1.1

proc $Node(i{:}\mathbb{N}^+, p{:}Set(\mathbb{N}^+), s{:}\mathbb{N}) =$
$\sum_{j:\mathbb{N}^+}(j{\in}p \wedge s{\approx}0) \to r(j,i){\cdot}Node(i, p{-}\{j\}, 0){+}$
$\sum_{j:\mathbb{N}^+}(p{\approx}\{j\} \wedge s{\approx}0) \to s(i,j){\cdot}Node(i, \emptyset, 1){+}$
$(p{\approx}\emptyset \wedge s{\approx}0) \to leader(i){\cdot}Node(i, \emptyset, 1)$

The initial configurations S consists of the parallel composition of the node processes for the nodes $1, \ldots, n$ in state 0. The finite sets $p_1, \ldots, p_n$ contain the neigbours. Initially, p_i contains the neighbours of node i.

proc $S = \tau_{\{c\}}\nabla_{\{c,leader\}}\Gamma_{\{r|s\to c\}}(Node(1,p_1,0) \parallel \cdots \parallel Node(n,p_n,0));$

Example 12.1.1. The network



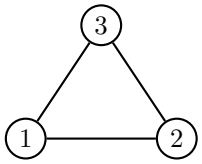
is described by

proc $S = \tau_{\{c\}}\nabla_{\{c,leader\}}\Gamma_{\{r|s\to c\}}$
$(Node(1,\{2,3,4\},0) \parallel Node(2,\{1\},0) \parallel Node(3,\{1\},0) \parallel Node(4,\{1\},0));$

The external behaviour of this system is given in figure 12.1.

Exercise 12.1.2. Explain why the τ-transitions in the external behaviour depicted in Figure 12.1 are not inert.

Exercise 12.1.3. Consider the network



Give a specification of this network. Explain why in this case no root will be elected.

12.1.1 The correctness of the tree identify protocol

In this section we prove using the cones and foci technique that for all connected networks that are free of cycles, the tree identify protocol produces elect a single leader. For this purpose, a network is specified using the prehide operator.

proc $S_{TIP} = \Upsilon_{\{c\}}\nabla_{\{c,leader\}}\Gamma_{\{r|s\to c\}}(Node(1,p_0(1),0)\| \cdots \|Node(n,p_0(n),0));$

where p_0 is a function that for each process i gives its set of neighbours $p_0(i)$ in the network. It is assumed that for all i and j

1. p_0 only contains nodes. If $i{\in}P_0(j)$ the $i \leq n$,
2. p_0 is symmetric, i.e. if $i{\in}p_0(j)$, then $j{\in}p_0$,
3. All nodes are connected. I.e., for all nodes i and j there is a sequence of nodes $i = i_1, \ldots, i_m = j$ such that $i_k{\in}p_0(i_{k+1})$ for all $1{\leq}k < m$, and
4. the network has no cycles. There are no infinite sequences $i_1, i_2, \ldots$ where $i_{k-1}{\neq}i_{k+1}$ for all $k{>}1$.

The aim of this section is to show that the external behaviour of this process, for any connected network without cycles, is $\tau{\cdot}leader{\cdot}\delta$. We use the equation for $Node$ from the previous section, but remove the parameter of the action $leader$.

As we use the cones and foci method, it is necessary to write the specification as a linear process:

proc $Spec(b{:}\mathbb{B}) = b \to leader{\cdot}Spec(false);$

So, we want to prove that $\tau{\cdot}S_{TIP} = \tau{\cdot}Spec(true)$.

As a first step, we use the parallel expansion in section 9.2.3, to transform S_{TIP} to a single linear equation. This equation is given below where S_{TIP} is equal to $Y(n, p_0, \lambda i{:}\mathbb{N}^+.0)$.

$$\begin{aligned}\textbf{proc}\quad & Y(n{:}\mathbb{N}^+, p{:}\mathbb{N}^+{\to}Set(\mathbb{N}^+), s{:}\mathbb{N}^+{\to}\mathbb{N}) = \\ & \quad \textstyle\sum_{i,j:\mathbb{N}^+}(j{\in}p(i){\wedge}p(j){\approx}\{i\}{\wedge}s(i){\approx}0{\wedge}s(j){\approx}0{\wedge}i{\not\approx}j{\wedge}i{\leq}n{\wedge}j{\leq}n) \to \\ & \qquad int{\cdot}Y(n, p[i{:=}p(i){-}\{j\}, j{:=}\emptyset], s[j{:=}1]){+} \\ & \quad \textstyle\sum_{i:\mathbb{N}^+}(p(i){\approx}\emptyset{\wedge}s(i){\approx}0{\wedge}i{\leq}n) \to leader{\cdot}Y(n, p, s[i{:=}1]);\end{aligned}$$

In order to apply the cones and foci theorem, the sum operators in the summands with visible actions need to be the same. Concretely, this means that the sum operator in front of the $leader$ action needs to be replaced. We find that the equation for Y can be rewritten to:

$$\begin{aligned}\textbf{proc}\quad & Y'(n{:}\mathbb{N}^+, p{:}\mathbb{N}^+{\to}Set(\mathbb{N}^+), s{:}\mathbb{N}^+{\to}\mathbb{N}) = \\ & \quad \textstyle\sum_{i,j:\mathbb{N}^+}(j{\in}p(i){\wedge}p(j){\approx}\{i\}{\wedge}s(i){\approx}0{\wedge}s(j){\approx}0{\wedge}i{\not\approx}j{\wedge}i{\leq}n{\wedge}j{\leq}n) \to \\ & \qquad int{\cdot}Y'(n, p[i{:=}p(i){-}\{j\}, j{:=}\emptyset], s[j{:=}1]){+} \\ & \quad (\exists i{:}\mathbb{N}^+.p(i){\approx}\emptyset{\wedge}s(i){\approx}0{\wedge}i{\leq}n) \to leader{\cdot}Y'(n, p, \lambda i{:}\mathbb{N}^+.1);\end{aligned}$$

We prove the correctness of this step using CL-RSP with invariants. For this purpose, we first list four invariants for Y, which are universally quantified over all positive numbers i and j:

1. if $j{\in}p(i)$ then $j{\leq}n$.
2. if $j{\in}p_0(i)$ then $j{\in}p(i) \vee s(j){=}1$.
3. if $s(j){=}1$ then $p(j){=}\emptyset$.
4. if $j{\in}p(i)$ then $j{\in}p_0(i)$.

We skip the proof that these are indeed invariants, as it is straightforward. Note that it is straightforward to show that the invariants are valid in the initial state of the process, when $p = p_0$ and $s = \lambda i{:}\mathbb{N}^+.0$.

The following lemma says that if $p(i){=}\emptyset$, then all other processes j are not a candidate anymore to declare themselves a leader, i.e. $s(j){=}1$. Hence, this means that $s[i{:=}1]{=}\lambda i{:}\mathbb{N}^+.1$, from which it is immediately obvious that $Y(n,p,s){=}Y'(n,p,s)$.

Lemma 12.1.4. Provided the invariants 2 and 3 above hold, we find that for all positive numbers i and j:

$$(p(i) = \emptyset \wedge j \neq i) \Rightarrow s(j) = 1.$$

Proof. Consider two nodes i and j. By connectedness, there are distinct nodes $i = i_1, i_2, \ldots, i_m = j$ with $i_{k+1} \in p_0(i_k)$ for $k = 1, \ldots, m-1$. We derive, by induction on k, that $s(i_k) = 1$ for all $1 < k \leq m$. In particular $s(i_m) = 1$, proving this lemma.

We start with the base case $k=2$. As $i_2 \in p_0(i_1)$ and $p(i_1)=\emptyset$, invariant 2 yields that $s(i_2)=1$.

For the induction step consider a $k>2$. We know that $i_k \in p_0(i_k)$ and by the induction hypothesis $s(i_{k-1}) = 1$. So, $p(i_k) = \emptyset$ by invariant 3. Using invariant 2 we derive that $s(i_k) = 1$. □

We now performed the ground work to make the essential step in proving implementation and specification equal to each other.

Lemma 12.1.5. For all $n{:}\mathbb{N}^+$, $p{:}\mathbb{N}^+ \to Set(\mathbb{N}^+)$ and $s{:}\mathbb{N}^+ \to \mathbb{N}$ for which the invariants hold, we find:

$$\tau \cdot \tau_{\{int\}}(Y'(n, p, s)) = \tau \cdot Spec(h(n, p, s))$$

where the state mapping $h : \mathbb{N}^+ \times (\mathbb{N}^+ \to Set(\mathbb{N}^+)) \times (\mathbb{N}^+ \to \mathbb{N}) \to \mathbb{B}$ is defined by

$$h(n, p, s) = \exists i{:}\mathbb{N}^+.(i \leq n \wedge s(i) \approx 0).$$

Proof. In order to prove this theorem we must check the matching criteria. Note that only the third criterion is not trivial.

1. As by the invariant 1, each set $p(i)$ contains a finite number of elements, this number is reduced by 1 in each int step. Namely, for some positive numbers i and j the condition says $j \in p(i)$ and in the result after the int j is removed from the set $p(i)$. So, only a finite sequence of consecutive int steps is possible.

2. The second matching criterion yields the following awkward looking proof requirement.

$$\begin{aligned}\forall i, j{:}\mathbb{N}^+.(j \in p(i) \wedge p(j) \approx \{i\} \wedge s(i) \approx 0 \wedge s(j) \approx 0 \wedge i \not\approx j \wedge i \leq n \wedge j \leq n \wedge \\ \exists i'{:}\mathbb{N}^+.(i' \leq n \wedge s(i') \approx 0)) \;\Rightarrow\; \exists i''{:}\mathbb{N}^+.(i'' \leq n \wedge s[j{:=}1](i'') \approx 0).\end{aligned}$$

   But actually, the proof of this requirement is trivial by taking i'' equal to i. In this case we must show $i \leq n$ (which is in the premises), and $s[j{:=}1](i) \approx 0$. But as one premise says that $i \not\approx j$, this is equal to $s(i) \approx 0$, which is also in the premises.

3. Note that all other matching criteria regard the $leader$ action, only. For the third matching criterion we need to define the focus condition. As there are no infinite sequences of internal steps, we can take the negation of the condition for the internal step as the focus condition:

$$FC(n, p, s) = \forall i, j{:}\mathbb{N}^+.(j \notin p(i) \vee p(j) \not\approx \{i\} \vee s(i) \not\approx 0 \vee s(j) \not\approx 0 \vee i \approx j \vee i > n \vee j > n).$$

   The third criterion now says:

$$FC(n, p, s) \wedge \exists i{:}\mathbb{N}^+.(i \leq n \wedge s(i) \approx 0) \Rightarrow \exists i'{:}\mathbb{N}^+.(p(i') \approx \emptyset \wedge s(i') \approx 0 \wedge i' \leq n).$$

   We prove this by taking i' equal to i. Hence, the only non trivial conjunct that we must prove in the right hand side of the implication is $p(i) \approx \emptyset$. So, assume $p(i) \not\approx \emptyset$. We show that we can construct an infinite sequence $i_1, i_2, \ldots$ such that $i_{k-1} \neq i_{k+1}$ for all $k>1$ where each i_k has the property that $\{i_{k-1}, i_{k+1}\} \subseteq p(i_k)$. Note that this contradicts that p_0 has no cycles, proving that $p(i) \approx \emptyset$.

   First we show that this property holds for i_2. Take $i_1 = i$. As $p(i) \neq \emptyset$, there is a $j \leq n$ such that $j \in p(i)$. From the focus condition and invariant 2 it follows that $p(j) \neq \{i\}$. Take $i_2 = j$. Suppose $p(j) = \emptyset$. Then $s(i)=1$ via invariant 2, which is a contradiction. Hence, there is an $\ell{:}\mathbb{N}^+$ with $\ell \neq i$ and $\{\ell, i\} \subseteq p(j)$. Take $i_3 = \ell$.

   Now assume that the sequence above has been constructed up till some i_{k-1} for $k>2$. We show that the sequence can be extended up to i_k. Note that $i_k \in p(i_{k-1})$ and hence $i_k \in p_0(i_{k-1})$. By symmetry of p_0 it also holds that $i_{k-1} \in p_0(i_k)$. So, by invariant 2 $i_{k-1} \in p(i_k)$. So, $s(i_k)=0$. By the focus condition it follows that $p(i_k) \neq \{i_{k-1}\}$. So, there is an additional node, call it i_{k+1} such that $\{i_{k-1}, i_{k+1}\} \subseteq p(i_k)$, which is what we had to prove.

4. The fourth matching criterion is trivial by taking i' to be equal to i.

$$(\exists i{:}\mathbb{N}^+.p(i){\approx}\emptyset \wedge s(i){\approx}0 \wedge i{\leq}n) \Rightarrow \exists i'{:}\mathbb{N}^+.(i'{\leq}n \wedge s(i'){\approx}0).$$

5. The fifth matching criterion need not be checked, as the action $leader$ had no parameter.

6. The sixth matching criterion is

$$(\exists i{:}\mathbb{N}^+.p(i){\approx}\emptyset \wedge s(i){\approx}0 \wedge i{\leq}n) \Rightarrow \exists i'{:}\mathbb{N}^+.(i'{\leq}n \wedge s[i{:=}1](i'){\approx}0) = false.$$

This is straightforward by taking i' equal to i. The expression $s[i{:=}1](i'){\approx}0$ in the right hand side of the implication then becomes $s[i{:=}1](i){\approx}0$, which is equivalent to $1{\approx}0$, i.e. $false$. This proves the last criterion.

□

We can now wrap up the work and perform the final steps in the correctness proof of the tree identify protocol

Theorem 12.1.6 (Correctness of the tree identify protocol).

$$\tau{\cdot}\tau_{\{int\}}(S_{TIP}) = \tau{\cdot}leader{\cdot}\delta$$

Proof. Collecting the results provided in this section we get

$$\begin{array}{l}
\tau{\cdot}\tau_{\{int\}}(S_{TIP}) = \\
\tau{\cdot}\tau_{\{int\}}(Y(n, p_0, \lambda i{:}\mathbb{N}^+.0)) = \\
\tau{\cdot}\tau_{\{int\}}(Y'(n, p_0, \lambda i{:}\mathbb{N}^+.0)) \stackrel{12.1.5}{=} \\
\tau{\cdot}Spec(h(n, p_0, \lambda i{:}\mathbb{N}^+.0)) = \\
\tau{\cdot}Spec(true) = \\
\tau{\cdot}leader{\cdot}\delta.
\end{array}$$

Actually, if the network consists of a single node, then no parent requests are exchanged, so in that special case the external behaviour is $leader{\cdot}\delta$. □

12.2 Sliding window protocol

In section 6.2 a unidirectional sliding window protocol is modelled and it is stated that is behaves as a bounded first-in-first-out buffer. More precisely, it is claimed that for any $n{:}\mathbb{N}^+$ and $q, q'{:}DBuf$:

$$\tau.FIFO([], 2n) = \tau.\tau_I(SWP(q, q', n))$$

where $I = \{c_2, c_3, c_4, c_5, i\}$.

In this section we prove this equation. This proof proceeds in three steps. First the description of the sliding window protocol is linearised. Then an alternative linear process is given which is almost the same as the linearised SWP, except that it does not use modulo calculation for the sequence numbers. We prove that this 'non modulo' variant has the same behaviour as the SWP. Finally, we show that the non modulo SWP is equal to the bounded buffer.

12.2.1 Linearization

In this section we give the results of linearising the sliding window protocol. First we provide the straightforward linearisations of the channels K and L form section 6.2.

$$\begin{aligned}\textbf{proc}\quad K(d_K{:}D, k_K{:}\mathbb{N}, s_K{:}\mathbb{N}^+) = {}& s_K{\approx}1 \to \textstyle\sum_{d:D,k:\mathbb{N}} r_2(d,k){\cdot}K(d,k,2)\\ &+ s_K{\approx}2 \to j{\cdot}K(d_K,k_K,1)\\ &+ s_K{\approx}2 \to j{\cdot}K(d_K,k_K,3)\\ &+ s_K{\approx}3 \to s_3(d_K,k_K){\cdot}K(d_K,k_K,1);\end{aligned}$$

$$\begin{aligned}L(k_L{:}\mathbb{N}, s_L{:}\mathbb{N}^+) = {}& s_L{\approx}1 \to \textstyle\sum_{k:\mathbb{N}} r_5(k){\cdot}L(k,2)\\ &+ s_L{\approx}2 \to j{\cdot}L(k_L,1)\\ &+ s_L{\approx}2 \to j{\cdot}L(k_L,3)\\ &+ s_L{\approx}3 \to s_6(k_L){\cdot}L(k_L,1);\end{aligned}$$

As the protocols S and R are already linear, the following linearisation of the sliding window protocol can easily be calculated.

$$\begin{aligned}\textbf{proc}\quad & SWP(\ell, m{:}\mathbb{N}, q{:}DBuf, d_K{:}D, k_K{:}\mathbb{N}, s_K{:}\mathbb{N}^+, k_L{:}\mathbb{N}, s_L{:}\mathbb{N}^+, \ell'{:}\mathbb{N}, q'{:}DBuf, b{:}BBuf, n{:}\mathbb{N}^+)\\ &= \textstyle\sum_{d:D} m{<}n \to r_1(d){\cdot}SWP(m{:=}m{+}1, q{:=}q[((\ell+m)|_n){:=}d]) && \text{(A)}\\ &+ \textstyle\sum_{k:\mathbb{N}} k{<}m \wedge s_K{\approx}1 \to int{\cdot}SWP(d_K{:=}q((\ell+k)|_n), k_K{:=}(\ell+k)|_{2n}, s_K{:=}2) && \text{(B)}\\ &+ s_K{\approx}2 \to i{\cdot}SWP(s_K{:=}1) && \text{(C)}\\ &+ s_K{\approx}2 \to i{\cdot}SWP(s_K{:=}3) && \text{(D)}\\ &+ s_K{\approx}3 \to int{\cdot}SWP(q'{:=}if((k_K{-}\ell')|_{2n}{<}n, q'[(k_K|_n){:=}d_K], q'),\\ &\qquad\qquad if((k_K{-}\ell')|_{2n}{<}n, b[k_K|_n{:=}true], b), s_K{:=}1) && \text{(E)}\\ &+ b(\ell'|_n) \to s_4(q'(\ell'|_n)){\cdot}SWP(\ell'{:=}(\ell'{+}1)|_{2n}, b{:=}b[\ell'|_n{:=}false]) && \text{(F)}\\ &+ s_L{\approx}1 \to int{\cdot}SWP(k_L{:=}nextempty_{mod}(\ell', b, n, n), s_L{:=}2) && \text{(G)}\\ &+ s_L{\approx}2 \to i{\cdot}SWP(s_L{:=}1) && \text{(H)}\\ &+ s_L{\approx}2 \to i{\cdot}SWP(s_L{:=}3) && \text{(I)}\\ &+ s_L{\approx}3 \to int{\cdot}SWP(\ell{:=}k_L, m{:=}m{-}(k_L{-}\ell)|_{2n}, s_L{:=}1); && \text{(J)}\end{aligned}$$

The int in summand (B) comes from a c_2, the one in summand (E) stems from c_3, the one in summand (G) comes from c_5, and the one in summand (F) from c_6.

The following lemma relates the sliding window protocol as defined in section 6.2 to the just given linear variant.

Lemma 12.2.1. For all $q, q'{:}DBuf$, $k_K, k_L{:}\mathbb{N}$, $d_K{:}D$, $n{:}\mathbb{N}^+$ it holds that

$$\Upsilon_U(SWP(q,q',n)) = SWP(0,0,q,d_K,k_K,1,k_L,1,0,q',empty,n)$$

where $U = \{c_2, c_3, c_5, c_6\}$.

12.2.2 Getting rid of modulo arithmetic

For the verification of the sliding window protocol the modulo calculation turns out to be a nuisance. Therefore, it is convenient to introduce an intermediate linear process M, which is obtained from that of the sliding window protocol by removing all wrapping of indices modulo n or modulo $2n$. This means that the sequence numbers and the indices ℓ and ℓ' can increase indefinitely. Furthermore, the data and boolean buffers are assumed to be of unbounded size, as their positions are all used.

It is actually an understatement that the introduction of M is for convenience only. Up till now, it is not even known whether a direct equivalence proof of SWP and $FIFO$ exists, without the use of an intermediate specification.

Obtaining the linear process M from that of SWP is at most places quite straightforward, except for summand (E). Here condition $(k_k - \ell')|_{2n}{<}n$ is replaced by $\ell' \leq k_k \wedge k_K{<}\ell' + n$. This last condition can be much easier understood than the corresponding condition in the sliding window protocol. It says that the value k_K must occur in the range $[\ell', \ell' + n\rangle$ of expected values in the receiver. This is actually also what $(k_k - \ell')|_{2n}{<}n$ says, but the encoding using modulo calculations is much trickier.

Note that in summand (F) it is not recorded that places in buffer b are freed again. As all indices in the buffer are monotonously increasing, this information is not of much use anymore.

In summand (G) the function $nextempty$ is used. This is the modulo free variant of the function $nextempty_{mod}$. Its definition is much easier. It gives the index of the first free position in b starting at position i within the first m next steps. Its definition for all $i, m{:}\mathbb{N}$ and $b{:}BBuf$ is

eqn $nextempty(i, b, m) = if(b(i) \wedge m{>}0, nextempty(i{+}1, b, m{-}1), i);$

So, the defining equation of M is given by:

proc
$$
\begin{aligned}
& M(\ell, m{:}\mathbb{N}, q{:}DBuf, d_K{:}D, k_K{:}\mathbb{N}, s_K{:}\mathbb{N}^+, k_L{:}\mathbb{N}, s_L{:}\mathbb{N}^+, \ell'{:}\mathbb{N}, q'{:}DBuf, b{:}BBuf, n{:}\mathbb{N}^+) \\
&= \textstyle\sum_{d:D} m{<}n \to r_1(d){\cdot}M(m{:=}m{+}1, q{:=}q[(\ell+m){:=}d]) && \text{(A)} \\
&+ \textstyle\sum_{k:\mathbb{N}} k{<}m \wedge s_K{\approx}1 \to int{\cdot}M(d_K{:=}q(\ell+k), k_K{:=}(\ell+k), s_K{:=}2) && \text{(B)} \\
&+ s_K{\approx}2 \to i{\cdot}M(s_K{:=}1) && \text{(C)} \\
&+ s_K{\approx}2 \to i{\cdot}M(s_K{:=}3) && \text{(D)} \\
&+ s_K{\approx}3 \to int{\cdot}M(q'{:=}if(\ell' \le k_K \wedge k_K < \ell'{+}n, q'[k_K{:=}d_K], q'), \\
&\qquad\qquad b{:=}if(\ell' \le k_K \wedge k_K < \ell'{+}n, b[k_K{:=}true], b), s_K{:=}1) && \text{(E)} \\
&+ b(\ell') \to s_4(q'(\ell')){\cdot}M(\ell'{:=}\ell'{+}1) && \text{(F)} \\
&+ s_L{\approx}1 \to int{\cdot}M(k_L{:=}nextempty(\ell', b, n), s_L{:=}2) && \text{(G)} \\
&+ s_L{\approx}2 \to i{\cdot}M(s_L{:=}1) && \text{(H)} \\
&+ s_L{\approx}2 \to i{\cdot}M(s_L{:=}3) && \text{(I)} \\
&+ s_L{\approx}3 \to int{\cdot}M(\ell{:=}k_L, m{:=}m{+}\ell - k_L, s_L{:=}1); && \text{(J)}
\end{aligned}
$$

For M a number of invariants represent the relations between the parameters of the process M. The proof of these invariant properties is straightforward. For this proof note that $\ell' \le \ell + m$ and $k_L \le \ell' + n$ follows from invariant 2 by definition from $nextempty$. The invariant 4 is only used to prove the validity of other invariants. Note that it is not at all trivial, if at all possible, to translate the invariant properties to the linear process SWP with the modulo calculation.

Lemma 12.2.2. The following invariants hold for $M(\ell, m, q, d_K, k_K, s_K, k_L, s_L, \ell', q', b, n)$.

1. $m \le n$
2. $\ell \le k_L \le nextempty(\ell', b, n) \le \ell + m$
3. $\ell'{-}n \le k_K < \ell + m$
4. $\forall i{:}\mathbb{N}.b(i) \Rightarrow i < \ell + m$
5. $\forall i{:}\mathbb{N}.b(i) \Rightarrow i < k_K + n$
6. $q(k_K) = d_K$
7. $\forall i{:}\mathbb{N}.b(i) \Rightarrow q(i) = q'(i)$

All invariants have a very clear intuition. Invariant 1 expresses that as m is the number of elements in q, it can at most be n. Invariant 2 expresses two properties. The first is that $nextempty(\ell', b, n)$ is confined between ℓ and $\ell{+}m$. In the protocol, ℓ can only be increased via an acknowledgement to $nextempty(\ell', b, n)$ and $nextempty(\ell', b, n)$ can only be increased via a message with index k_K, which never exceeds $\ell{+}m$. The second property says is that the value of k_L in the acknowledgement is always between ℓ and $nextempty(\ell', b, n)$. This is self evident as ℓ is only increased by getting the value of k_L, and k_L is only changed by assigning $nextempty(\ell', b, n)$ to it. All these variables are increasing.

Invariant 3 says that k_K lies between between $\ell'{-}n$ and $\ell + m$. Whenever a message is sent, k_K gets a value from ℓ up to $\ell + m$. While k_K is in transit, the value of ℓ' will always stay below $\ell + m$. In other words, $\ell'{-}n$ will stay below k_K.

Invariant 4 says that messages that have been transferred to R have sequence numbers lower than $\ell{+}m$. Hence, positions in buffer q at higher positions are not filled and $b(j) = false$ for $j > \ell + m$.

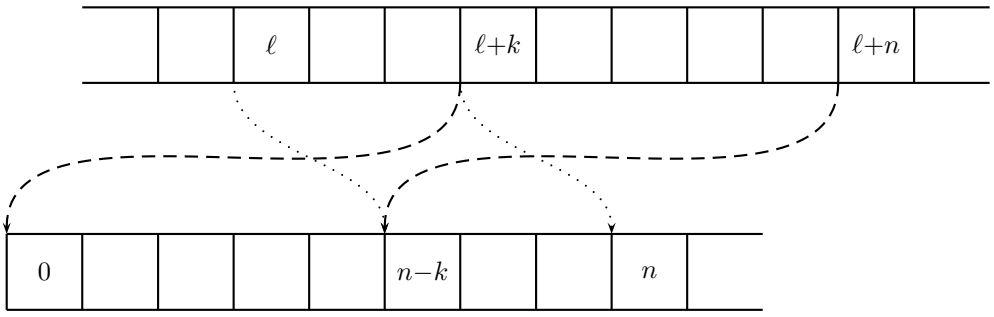


Figure 12.2: A visual representation of the function mod (with $\ell{+}k)|_n = 0$)

Invariant 5 says that if data with sequence number k_K is sent over channel K, a message with a sequence number of $k_K{-}n$ or lower will not be be sent anymore. Or reversely, if there is a message in transit with sequence number k_K, the highest sequence number that a message in K ever had was below $k_K{+}n$.

Invariant 6 says that the contents of channel K matches the contents of q at position k_K. Similarly, 7 says that the contents of buffers q and q' match for those positions that are filled.

We first show that the linearisation SWP of the sliding window protocol in which modulo calculation is used, is equal to the linear process M. This is proven using the cones and foci theorem, by taking M (with the largest state space) as the implementation and SWP as the specification. For the state mapping from the state variables of M to the state variables of SWP we introduce the auxiliary function mod. It takes the elements from a buffer q from a position ℓ to $\ell{+}n$ and maps these circularly onto the first n elements of a new buffer. The function mod is depicted in figure 12.2 and defined as follows:

Definition 12.2.3. For an arbitrary value $d_0{:}D$, the function mod is defined by (for both $q{:}DBuf$ and $q{:}BBuf$):

$$mod(q,\ell,n) = \lambda k{:}\mathbb{N}.\mathit{if}(k{<}n, q(\ell{+}(k{-}\ell)|_n), d_0).$$

The following properties characterises precisely what the mod function is supposed to do:

Lemma 12.2.4. Let $q{:}DBuf$ and $q{:}BBuf$, $i{<}n$ and $(j{-}\ell)|_n = j{-}\ell$.

1. $mod(q,\ell,n)((\ell{+}i)|_n) = q(\ell{+}i)$.
2. $mod(q[j{:=}d],\ell,n) = mod(q,\ell,n)[j|_n{:=}d]$.

Proof. The proofs are direct applications of the definition of mod.

1. $$\begin{array}{lcl} mod(q,\ell,n)((\ell{+}i)|_n) & = & \mathit{if}((\ell{+}i)|_n{<}n, q(\ell{+}((\ell{+}i)|_n{-}\ell)|_n), d_0) \\ & = & q(\ell{+}(\ell{+}i{-}\ell)|_n) \\ & = & q(\ell{+}i|_n) \\ & = & q(\ell{+}i). \end{array}$$

2. Expanding the definition of mod yields:

$$\begin{array}{l} \lambda k{:}\mathbb{N}.\mathit{if}(k{<}n, \mathit{if}(\ell + (k{-}\ell)|_n{\approx}j, d, q(\ell{+}(k{-}\ell)|_n)), d_0) = \\ \qquad \lambda k{:}\mathbb{N}.\mathit{if}(k{\approx}j|_n, d, \mathit{if}(k{<}n, q(\ell{+}(k{-}\ell)|_n), d_0)). \end{array}$$

   If $k \geq n$ both sides yields d_0. So, we need to show that for all $k < n$:

$$\mathit{if}(\ell + (k{-}\ell)|_n{\approx}j, d, q(\ell{+}(k{-}\ell)|_n)) = \mathit{if}(k{\approx}j|_n, d, q(\ell{+}(k{-}\ell)|_n)).$$

   This boils own to proving that $\ell + (k - \ell)|_n = j$ is equivalent to $k = j|_n$.

   $\Rightarrow$) $j|_n = (\ell{+}(k{-}\ell)|_n)|_n = (\ell{+}(k{-}\ell))|_n = k|_n = k$.

$\Leftarrow)$ $\ell + (k{-}\ell)|_n = \ell + (j|_n{-}\ell)|_n = \ell{+}(j{-}\ell)|_n = \ell + (j - \ell) = j$.

□

Using this definition we can state and prove the equality between SWP and M.

Lemma 12.2.5. For all variables $\ell, m, k_K, k_L, \ell'{:}\mathbb{N}$, $q, q'{:}DBuf$, $d_K{:}D$, $s_K, s_L, n{:}\mathbb{N}^+$ and $b{:}BBuf$ it holds that:

$$SWP(\ell|_{2n}, m, mod(q, \ell, n), d_K, k_K|_{2n}, \\ s_K, k_L|_{2n}, s_L, \ell'|_{2n}, mod(q', \ell', n), mod(b, \ell', n), n) = \\ M(\ell, m, q, d_K, k_K, s_K, k_L, s_L, \ell', q', b, n)$$

Proof. We use the cones and foci theorem to show that the SWP implements M. We use the following state mapping where at the left of the assignment we find the name for the variable in SWP in terms of variables of M at the right hand side:

$\ell := \ell\|_{2n}$	$m := m$	$q := mod(q, \ell, n)$
$d_K := d_K$	$k_K := k_K\|_{2n}$	$s_K := s_K$
$k_L := k_L\|_{2n}$	$s_L := s_L$	$\ell' := \ell'\|_{2n}$
$q' := mod(q', \ell', n)$	$b := mod(b, \ell', n)$	$n := n$

We find the following non trivial matching criteria. Note that as there are no τ's proving weak τ-convergence is not necessary. Moreover, there are no matching conditions of category 2.

3 & 4 (s_4). We must show that $b(\ell') = mod(b, \ell', n)(\ell'|_n)$. This follows directly from lemma 12.2.4.1.

6 (s_4). We must show that $b(\ell')$ implies $q'(\ell') = mod(q', \ell', n)(\ell'|_n)$. Use lemma 12.2.4.1 again.

6 (r_1). We must show for all $d{:}D$ that $mod(q[\ell{+}m{:=}d], \ell|_{2n}, n) = mod(q, \ell, n)[(\ell|_{2n}{+}m)|_n{:=}d]$ provided $m{<}n$. The right hand side can be written as:

$$mod(q, \ell, n)[(\ell|_{2n}{+}m)|_n{:=}d] = \\ \lambda k{:}\mathbb{N}.if(k{\approx}(\ell{+}m)|_n, d, if(k{<}n, q(\ell{+}(k{-}\ell)|_n), d_0) = \\ \lambda k{:}\mathbb{N}.if(k{<}n, if(k{\approx}(\ell{+}m)|_n, d, q(\ell{+}(k{-}\ell)|_n), d_0).$$

The left hand side can be expanded to

$$mod(q[(\ell{+}m){:=}d], \ell, n) = \\ \lambda k{:}\mathbb{N}.if(k{<}n, if(\ell{+}(k{-}\ell)|_n{\approx}\ell{+}m, d, q(\ell{+}(k{-}\ell)|_n), d_0)$$

Obviously, both resulting expressions are equal if $k = (\ell{+}m)|_n$ iff $\ell{+}(k{-}\ell)|_n = \ell{+}m$ provided that $k < n$. We prove this by implication in both directions.

$\Rightarrow)$ Suppose $k = (\ell + m)|_n$. Then

$$\ell + (k - \ell)|_n = \ell + (\ell + m - \ell)|_n = \ell + m|_n = \ell + m$$

where in the last step we used that $m < n$.

$\Leftarrow)$ If $\ell + (k - \ell)|_n = \ell + m$, then $m = (k - \ell)|_n$. So,

$$(\ell + m)|_n = (\ell + (k - \ell)|_n)|_n = (\ell + k - \ell)|_n = k|_n = k$$

using in the last step that $k < n$.

6 (B). In this case we must show that for all $k{:}\mathbb{N}$ if $k < m$ that the following two properties hold

1. $(\ell + k)|_{2n} = (\ell|_{2n} + k)|_{2n}$, which follows directly by modulo calculation.
2. We must show that $q(\ell{+}k) = mod(q, \ell, n)(\ell|_{2n}{+}k)|_n$. This is lemma 12.2.4.1, which can be applied as $k{<}n$. This follows from invariant 1 and $k{<}m$.

6 (E). In this case we must show that

$$\begin{aligned}&\mathit{if}(\ell'{\leq}k_K \wedge k_K{<}\ell'{+}n, \mathit{mod}(q'[k_K{:=}d_K], \ell', n), \mathit{mod}(q', \ell', n)) = \\ &\qquad \mathit{if}((k_K|_{2n}{-}\ell'|_{2n})|_{2n}{<}n, \mathit{mod}(q', \ell', n)[k_K|_n{:=}d_K], \mathit{mod}(q', \ell', n)).\end{aligned}$$

By invariants 1, 2 and 3 we find that $\ell' - n \leq k_K < \ell + m \leq k_L + m \leq \ell' + n + m \leq \ell' + 2n$. This means that the conditions $\ell' \leq k_K \wedge k_K < \ell' + n$ and $(k_K - \ell')|_{2n} < n$ are equivalent. Moreover, the else parts are the same. By lemma 12.2.4.2 it follows that the then parts are equivalent too.

6 (s_4). Given that $b(\ell')$ holds, we must show the following two properties:

1. $(\ell'{+}1)|_{2n} = (\ell'|_{2n}{+}1)|_{2n}$, which follows directly from modulo calculation.
2. $\mathit{mod}(b, \ell'{+}1, n) = \mathit{mod}(b, \ell', n)[\ell'|_n := \mathit{false}]$. Expansions of *mod* in this equation yields:

$$\begin{aligned}&\lambda k{:}\mathbb{N}.\mathit{if}(k{<}n, b(\ell'{+}1 + (k{-}\ell'{-}1)|_n), \mathit{true}) = \\ &\qquad \lambda k{:}\mathbb{N}.\mathit{if}(k{\approx}\ell'|_n, \mathit{false}, \mathit{if}(k{<}n, b(\ell'{+}(k{-}\ell')|_n), \mathit{true})).\end{aligned} \tag{12.1}$$

   We consider the following three cases:

   (a) Suppose $k \geq n$. Both sides yield true and hence equation (12.1) holds.

   (b) Suppose $k = \ell'|_n$. The right hand side of equation (12.1) is false. The left hand side reduces to $b(\ell' + 1 + (\ell'|_n - \ell' - 1)|_n) = b(\ell' + 1 + n - 1)) = b(\ell' + n)$. Using invariant 1 and 4 it follows that $b(\ell' + n) = \mathit{false}$.

   (c) If $k < n$ and $k \neq \ell'|_n$. Equation (12.1) reduces to $b(\ell' + 1 + (k - \ell' - 1)|_n) = b(\ell' + (k - \ell')|_n)$, which are equal.

6 (G). We must show that $\mathit{nextempty}(\ell', b, m)|_{2n} = \mathit{nextempty}_{\mathit{mod}}(\ell'|_{2n}, \mathit{mod}(b, \ell', n), m, n)$ with $m = n$. We show this equation by induction on m, for all b, ℓ', n.

If $m{=}0$, both sides of the equation reduce to $\ell'|_{2n}$. If $m{>}0$, we see that the equation becomes:

$$\begin{aligned}&\mathit{if}(b(\ell'), \mathit{nextempty}(\ell' + 1, b, m - 1), \ell')|_{2n} = \\ &\quad \mathit{if}(\mathit{mod}(b, \ell', n)(\ell'|_n), \mathit{nextempty}_{\mathit{mod}}((\ell'{+}1)|_{2n}, \mathit{mod}(b, \ell', n), m{-}1, n), \ell'|_{2n}).\end{aligned}$$

As $\mathit{mod}(b, \ell', n)(\ell'|_n) = b(\ell')$, it suffices to show that

$$\mathit{nextempty}(\ell' + 1, b, m - 1)|_{2n} = \mathit{nextempty}_{\mathit{mod}}((\ell' + 1)|_{2n}, \mathit{mod}(b, \ell', n), m - 1, n).$$

This follows from the induction hypothesis.

6 (J). And finally, we must show that $m + \ell - k_L = m - (k_L - \ell)|_{2n}$. Observe that by invariants 12.2.2.1, and 12.2.2.2, it holds that

$$0 \leq k_L - \ell \leq \ell + m - \ell \leq n.$$

So, $k_L{-}\ell \leq n < 2n$ and hence $(k_L - \ell)|_{2n} = k_L - \ell$, from which the proof obligation follows directly.

□

12.2.3 Proving M equal to a bounded queue

We prove that M is branching bisimilar to the FIFO queue of size $2n$, using the cones and foci theorem. For this, it is useful to define the following auxiliary notation:

$$q[i..j\rangle = [q(i), q(i + 1), \ldots, q(j{-}1)].$$

If $j{\leq}i$, this is the empty queue. Note that if $j \geq i$ then $\#q[i..j\rangle = j - i$.

The state mapping h, which maps the states of M to the states of FIFO (see section 6.2), is defined by:

$$h(\ell, m, q, d_K, k_K, s_K, k_L, s_L, \ell', q', b, n) = q'[\ell'..\mathit{nextempty}(\ell', b, n)\rangle{+}{+}q[\mathit{nextempty}(\ell', b, n)..\ell{+}m\rangle$$

Intuitively, h collects the data elements in the sending and receiving windows, starting at the first cell in the receiving window (i.e., ℓ') until the first empty cell in this window, and then continuing with the sending window until the first empty cell in that window at position $\ell{+}m$. Note that h is independent of d_K, k_K, s_K, k_L, s_L; we therefore write $h(\ell, m, q, \ell', q', b, n)$.

The focus points are those states where either the sending window is empty (in which case $\ell{=}m$), or the receiving window is full and all data elements in the receiving window have been acknowledged (meaning that $\ell{=}\ell' + n$). This exactly characterises the situation where no internal progress is possible in the sliding window protocol. So, the focus condition for M is defined by the following expression:

$$FC(\ell, m, q, d_K, k_K, s_K, k_L, s_L, \ell', q', b, n) \;:=\; \ell{\approx}m \vee \ell{\approx}\ell'{+}n.$$

Lemma 12.2.6. For all $\ell, m{:}\mathbb{N}$, $n{:}\mathbb{N}^+$, $q, q'{:}DBuf$ and $d{:}D$ it holds that

$$\tau.\tau_{\{int,i\}}(M(\ell, 0, q, d, 1, m, 1, \ell, q', empty, n)) = \tau.FIFO([], 2n).$$

Proof. First note that the theorem holds for the initial state. According to the cones and foci method, we obtain the following matching criteria. Trivial matching criteria are left out.

1. From any reachable state of M, a focus point can be reached.

2. if $\ell' \leq k_K \wedge k_K < \ell' + n$ then $h(\ell, m, q, \ell', q'[k_K := d_k], b[k_K := true], n) = h(\ell, m, q, \ell', q', b, n)$;

3 (r_1). if $m < n$ then $\#h(\ell, m, q, \ell', q', b, n) < 2n$;

3 (s_4). if $b(\ell')$ then $\#h(\ell, m, q, \ell', q', b, n) > 0$;

4 (r_1). if $(m{\approx}0 \vee \ell{\approx}\ell' + n) \wedge \#h(\ell, m, q, \ell', q', b, n) < 2n$ then $m < n$;

4 (s_4). if $(m{\approx}0 \vee \ell{\approx}\ell' + n) \wedge \#h(\ell, m, q, \ell', q', b, n) > 0$ then $b(\ell')$;

5 (s_4). if $b(\ell')$ then $q'(\ell') = head(h(\ell, m, q, \ell', q', b, n))$;

6 (r_1). if $m < n$ then $h(\ell, m + 1, q[(\ell + m) := d], \ell', q', b, n) = h(\ell, m, q, \ell', q', b, n) \triangleright d$;

6 (s_4). if $b(\ell')$ then $h(\ell, m, q, \ell' + 1, q', b, n) = tail(h(\ell, m, q, \ell', q', b, n))$.

Below we give the proofs of all these criteria:

1. We show that from any reachable state we can decrease

$$\min(m, \ell' + n - \ell) \tag{12.2}$$

to 0. If this expression has reached 0, a focus point has been reached. Due to invariant 2 and the definition of $nextempty$ the expression $\min(m, \ell' + n - \ell)$ cannot become smaller than 0.

So, suppose that we are in a state where $\min(m, \ell' + n - \ell) > 0$. This means that there is unacknowledged data ready to be sent to the receiver ($m{>}0$) and not all data sent to the receiver is received and/or acknowledged ($\ell'{+}n{>}\ell$).

Assume first that $nextempty(\ell', b, n) = \ell$. This means that all data has been acknowledged but data in the sending buffer has not yet been received. In particular the data at position $\ell + 1$ did not arrive. We show that this data can be sent to the other side, causing the situation where $nextempty(\ell', b, n) > \ell$ (and we deal with that later).

Note that in order to send data from the sender to receiver, we need the channel K. The channel can be in one of the following three states: $s_K = 1$, $s_K = 2$ and $s_K = 3$. If $s_K = 1$ (or $s_K = 2$) we can carry out summands (C), B (with $k = 0$), D and E of M. Summand E can be carried out as its condition in this case is $\ell' \leq \ell \wedge \ell < \ell' + n$. The first part of this condition holds by definition of $nextempty$ as $\ell = nextempty(\ell', b, n) \geq \ell'$. The second part is a direct consequence of the assumption that (12.2) is larger than 0. After executing summand E, $b(\ell) = true$. So, $nextempty(\ell', b, n) > \ell$.

If $s_K = 3$, then first summand E or F must be executed, and subsequently, the same sequence as above, also leading to the situation where $nextempty(\ell', b, n) > \ell$.

Now, consider the situation where $nextempty(\ell', b, n) > \ell$. This means that not all data has been acknowledged. For sending the acknowledgement back, channel L must be used. The state variable s_L of channel L can have the values 1, 2 and 3. If $s_L = 1$ (or $s_L = 2$), we can carry out summands (H), G, I and

J of M and set m to $m+\ell-nextempty(\ell', b, n)$ and ℓ to $nextempty(\ell', b, n)$. As $\ell < nextempty(\ell', b, n)$, the value of m and $\ell' + n - \ell$ are decreased. Hence, (12.2) is decreased.

If $s_L = 3$, we can use a similar argument to show that (12.2) can be decreased.

2. The expression $h(\ell, m, q, \ell', q'[k_K := d_K], b[k_K := true], n)$ is equal to

$$q'[k_K := d_K][\ell'..nextempty(\ell', b[k_K := true], n)\rangle +\!\!+ q[nextempty(\ell', b[k_K := true], n)..m + \ell\rangle$$

Or in other words

$$\begin{array}{l}[q'[k_K := d_K](\ell'), \ldots, q'[k_K := d_K](nextempty(\ell', b[k_K := true], n) - 1),\\ \qquad\qquad q(nextempty(\ell', b[k_K := true], n), \ldots, q(\ell + m - 1)].\end{array} \tag{12.3}$$

This must be equal to $h(\ell, m, q, \ell', q', b, n)$, which can be written as:

$$[q'(\ell'), \ldots, q'(nextempty(\ell', b, n) - 1), q(nextempty(\ell', b, n)), \ldots, q(\ell + m - 1)]. \tag{12.4}$$

As $nextempty(\ell', b[k_K := true], n) \geq nextempty(\ell', b, n)$ the expressions (12.3) and (12.4) are equal if we can show that

$$q'[k_K := d_K](i) = q(i) \text{ for all } nextempty(\ell', b, n) \leq i < nextempty(\ell', b[k_K := true], n). \tag{12.5}$$

For $i \neq k_K$ we know that $b(i)$ must hold. So, by invariant 7 we see that (12.5) holds.

If $i = k_K$, equation (12.5) boils down to $d_K = q(k_K)$. This is exactly invariant 6.

3 (r_1). We see that $\#h(\ell, m, q, \ell', q', b, n) = m + \ell - \ell' < n + \ell - \ell' < 2n$ where the first inequality follows from the condition and the second from invariants 2 and the definition of $nextempty$.

3 (s_4). If $b(\ell')$ holds, $nextempty(\ell', b, n) > \ell'$. Hence, $\#h(\ell, m, q, \ell', q', b, n) > 0$.

4 (r_1). If $m{\approx}0$ this trivially holds as $n > 0$. If $\ell{\approx}\ell' + n$, we use that $\#h(\ell, m, q, \ell', q', b, n) = m+\ell-\ell' < 2n$. So, $m + n < 2n$ or in other words $m < n$.

4 (s_4). As $\#h(\ell, m, q, \ell', q', b, n) > 0$, $m + \ell - \ell' > 0$.

- If $m{\approx}0$, it must be the case that $\ell > \ell'$. By invariant 2 $nextempty(\ell', b, n) \geq \ell > \ell'$. So, $b(\ell')$ must hold.
- If $\ell{\approx}\ell' + n$, then $m + \ell - \ell' = m - n > 0$. This contradicts invariant 1.

5 (s_4). If $b(\ell')$, it follows that $nextempty(\ell', b, n) > \ell'$. So, $head(h(\ell, m, q, \ell', q', b, n)) = q'(\ell')$.

6 (r_1). The expression $h(\ell, m + 1, q[(\ell + m) := d], \ell', q', b, n)$ equals

$$[q'(\ell'), \ldots, q(nextempty(\ell', b, n)), \ldots, q(\ell + m - 1)] \triangleright d.$$

This is exactly the same as $h(\ell, m, q, \ell', q', b, n) \triangleright d$.

6 (s_4). If $b(\ell')$ holds, it is the case that $nextempty(\ell', b, n) > \ell'$. So, $h(\ell, m, q, \ell', q', b, n)$ equals

$$[q'(\ell' + 1), \ldots, q(nextempty(\ell', b, n)), \ldots, q(\ell + m - 1)].$$

This is the same as $tail(h(\ell, m, q, \ell', q', b, n))$.

□

12.2.4 Correctness of the sliding window protocol

We wrap up by proving equality of the sliding window protocol with the first-in-first-out buffer. So, we start this identity for the third time. For any $n{:}\mathbb{N}^+$ and $q, q'{:}DBuf$:

$$\boxed{\tau.FIFO([], 2n) = \tau.\tau_I(SWP(q, q', n))}$$

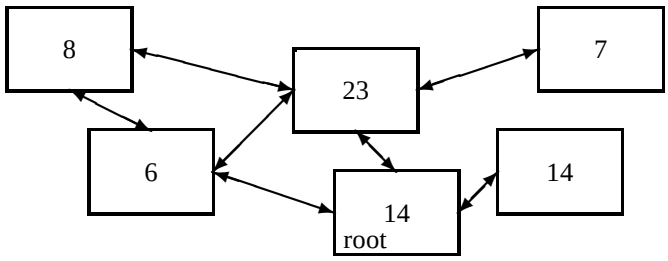


Figure 12.3: A set of distributed processes

where $I = \{c_2, c_3, c_4, c_5, i\}$.

Proof.

$$\begin{array}{ll}
\tau.\tau_{\{c_2,c_3,c_5,c_6,i\}}(SWP(q,q',n)) = & \text{(Lemma 12.2.1)}\\
\tau.\tau_{\{int,i\}}(SWP(0,0,q,d_K,k_K,1,k_L,1,0,q',empty,n)) = & \text{(Lemma 12.2.5)}\\
\tau.\tau_{\{int,i\}}(M(0,0,q,d_K,k_K,1,k_L,1,0,q',empty,n)) = & \text{(Lemma 12.2.6)}\\
\tau.FIFO([],2n) &
\end{array}$$

□

12.3 Distributed summing protocol

We show the correctness of the distributed summing protocol. In this protocol the numbers distributed over a set of processes must be summed. The protocol is interesting because the way the numbers are summed is completely nondeterministic.

We start with a set processes that are all connected via some network of bidirectional links (see e.g. figure 12.3). Each process contains some number, not known to other processes. The algorithm describes how to sum all numbers such that one designated (root) process can output the sum of these numbers.

The algorithm is described as the parallel composition of a (finite) number of processes, indexed by natural numbers. Each process works in exactly the same way, except for the root process, which has number 1. This process differs from the other processes in the sense that initially it is already started, and when it has collected all sums of its neighbours, it issues a $\overline{rep}$ message to indicate the total sum to the outside world, instead of a partial sum to a neighbour.

The overall idea behind the algorithm is that a spanning tree over the links between the processes is constructed with as root the process 1. All partial sums are then sent via this spanning tree to the root.

Initially, a process is waiting for a $start$ message from a neighbour. After it has received the first start message, the process is considered part of the spanning tree and the process by which it is started is called its $parent$. Thereafter it starts all its neighbours except its parent by a $start$ message.

- Those neighbours that were not yet part of the spanning tree will now become part of it with the current process as parent. Eventually, these neighbours will send a partial sum to the current process using an $answer$ message.
- Those neighbours that were already part of the spanning tree ignore the start message. Note however that due to symmetry these processes will also send a start message to the current process.

So, after having sent the start messages, each process gets from each neighbour except its parent either a partial sum or a start message. After having received these messages, it adds all received partial sums to its own value, and sends the result as a partial sum to its parent. Eventually, the root process 1 has received all partial sums, and it can report the total sum.

Theorem 12.3.10 says that this simple scheme is correct, i.e., if each process is connected to the root, processes do not have themselves as neighbours and the neighbour relation is symmetric, then the distributed summation algorithm computes the sum of the values of the individual processes. Note that if any of the stated conditions on the topology does not hold, the algorithm either deadlocks, not yielding a result, or it does not sum up all values.

12.3.1 A description in mCRL2

In this section we give a description of the process in mCRL2 and state the correctness criterion. The algorithm is described as the parallel composition of the algorithms for the individual nodes in the network, which are described generically by means of a linear process equation.

The processes of the network interact via matching actions st, $\overline{st}$ (for *start*), ans, $\overline{ans}$ (for *answer*) and the total sum is communicated using a $\overline{rep}$ (for *report*) action. We think of the overbarred action as a sending activity, and a non-overbarred action as a receiving activity.

Definition 12.3.1. Processes X are described by means of six parameters:

- i: the ID-number of the process.
- t: the *total sum* computed so far by the process. Initially, it contains the value that is the contribution of process i to the total sum.
- N: a list of *neighbours* to which the process still needs to send a $\overline{st}$ message. Initially, this list contains exactly all neigbours. We write $rem(j, N)$ to remove neighbour j from list N.
- p: the index of the initiator, or *parent*, of the process. Variable p is also called the parent link of i.
- w: The number of st and ans messages that the process is still *waiting* for.
- s: the *state* the process is in. The process can be in three states, denoted by 0, 1, 2. If s equals 0, the process is in its initial state. If s equals 1, the process is active. If s equals 2, the process has finished and behaves as deadlock.

proc $X(i{:}\mathbb{N}^+, t{:}\mathbb{N}, N{:}List(\mathbb{N}), p, w, s{:}\mathbb{N}) =$

$$
\begin{array}{l}
(s{\approx}0) \rightarrow \sum_{j:\mathbb{N}} st(i,j)\, X(i,t,rem(j,N),j,\#(N){-}1,1) + \\
\sum_{j:\mathbb{N}} (j{\in}N \wedge s{\approx}1) \rightarrow \overline{st}(j,i)\, X(i,t,rem(j,N),p,w,s) + \\
\sum_{j,m:\mathbb{N}} (s{\approx}1) \rightarrow ans(i,j,m)\, X(i,t+m,N,p,w{-}1,s) + \\
\sum_{j:\mathbb{N}} (s{\approx}1) \rightarrow st(i,j)\, X(i,t,N,p,w{-}1,s) + \\
(i{\approx}1 \wedge N{\approx}[] \wedge w{\approx}0 \wedge s{\approx}1) \rightarrow \overline{rep}(t)\, X(i,t,N,p,w,2) + \\
(i{\not\approx}1 \wedge N{\approx}[] \wedge w{\approx}0 \wedge s{\approx}1) \rightarrow \overline{ans}(p,i,t)\, X(i,t,N,p,w,2)
\end{array}
$$

In line 1 of the definition of X above, process i is in its initial state and an st message is received from some process j, upon which j is stored as the parent and s switches from 0 to 1, indicating that process i has become active. Since it makes no sense to send start messages to one's parent, j is removed from N. The counter w is initialised to the number of neighbours of i, not counting process j. In line 2, a $\overline{st}$ message is sent to a neighbour j, which is thereupon removed from N. In line 3, a sum is received from some process j via an ans message containing the value m, which is added to t, the total sum computed by process i so far. The counter w is decreased. In line 4 a st message is received from neighbour j. The message is ignored, except that the counter w is decreased. In line 5 a $\overline{rep}(t)$ is sent (in case $i = 1$), when process 1 is active, there are no more ans or st messages to be received (formalised by the condition $w = 0$), and a $\overline{st}$ message has been sent to all neighbours (formalised by the condition $N{\approx}[]$). The status variable s becomes 2, indicating that process 1 is no longer active. Line 6 is as line 5 but for processes $i \neq 1$; now an $\overline{ans}$ message is sent to parent p, containing the total sum t computed by process i.

Next, we define the parallel composition of n copies of the process X in the same way as done in section 9.2.3. The actions st and $\overline{st}$ must communicate, as must ans and $\overline{answ}$. The communications are hidden. The only action that remains visible is $\overline{rep}$.

The t-value of the processes are put in a function $\boldsymbol{t}_0$. i.e. $\boldsymbol{t}_0(i)$ indicate the value of t for process i. Similarly, the functions $\boldsymbol{p}$, $\boldsymbol{w}$, $\boldsymbol{s}$ contain the values of the variables p, w and s of all processes, respectively.

Definition 12.3.2. We define the process $Impl$ as the parallel composition of n processes X, and the process $DSum$ as $Impl$ with hiding, allow and communication operator applied to it.

proc $DSum(n{:}\mathbb{N}^+, \boldsymbol{t}_0{:}\mathbb{N}^+{\to}\mathbb{N}, \boldsymbol{n}_0{:}\mathbb{N}^+{\to}List(\mathbb{N}^+)) =$
$\quad \tau_{\{st^\star, ans^\star\}} \nabla_{\{st^\star, ans^\star, \overline{rep}\}}, \Gamma_{\{st|\overline{st}\to st^\star, ans|\overline{ans}\to ans^\star\}}(Impl(k, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0));$
$Impl(n{:}\mathbb{N}^+, \boldsymbol{t}{:}\mathbb{N}^+{\to}\mathbb{N}, \boldsymbol{n}{:}\mathbb{N}^+{\to}Set(\mathbb{N}), \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}{:}\mathbb{N}^+{\to}\mathbb{N}) =$
$\quad (n{\approx}1) \to X(1, \boldsymbol{t}(1), \boldsymbol{n}(1), \boldsymbol{p}(1), \boldsymbol{w}(1), \boldsymbol{s}(1))$
$\quad\quad \diamond\ (X(n, \boldsymbol{t}(n), \boldsymbol{n}(n), \boldsymbol{p}(n), \boldsymbol{w}(n), \boldsymbol{s}(n)) \| Impl(n{-}1, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}));$

Here, $\boldsymbol{p}_0{=}\lambda i{:}\mathbb{N}^+.1$ (each process considers process 1 as its initiator), $\boldsymbol{w}_0{=}\lambda i{:}\mathbb{N}^+.\#\boldsymbol{n}_0(i)$ ($\boldsymbol{w}_0$ contains the number of neighbours of process i), $\boldsymbol{s}_0{=}\lambda i{:}\mathbb{N}^+.if(i{\approx}1, 1, 0)$ (only the root is initially started).

The distributed summing protocol only works, when some requirements on the interconnection of processes are satisfied. With the variable $\boldsymbol{n}_0$ at our disposal, we can formulate these requirements very precisely as follows:

Definition 12.3.3 (Requirements for topology). Let n denote the number of processes in the network. We define $goodtopology(n, \boldsymbol{n}_0)$ as the conjunction of the following properties:

- No process has a link to itself: $\forall i{:}\mathbb{N}^+.i{\notin}\boldsymbol{n}_0(i)$;
- The neighbour relation is symmetric: $\forall i, j{:}\mathbb{N}^+.i{\in}\boldsymbol{n}_0(j) \Leftrightarrow j{\in}\boldsymbol{n}_0(i)$;
- Every process $i \leq n$ is connected to process 1:
  for all $i{\leq}n$ there are $m{\leq}n$ and $i = i_0, \ldots, i_m = 1$ such that $i_{l+1} \in \boldsymbol{n}_0(i_l)$ for all $0{\leq}l{<}m$.
- $\boldsymbol{n}_0$ only contains valid neighbours: $\forall i, j{:}\mathbb{N}^+.j{\leq}n \wedge i \in \boldsymbol{n}_0(j) \Rightarrow i \leq n$.

The theorem below states correctness of the summation algorithm. It says that in a topology as described above, the distributed summation algorithm correctly reports the sum of all values in the processes and halts. The remainder of this paper is devoted to proving this theorem. It is repeated and proven as theorem 12.3.10.

Theorem 12.3.4. For all $n{:}\mathbb{N}^+$, $\boldsymbol{t}_0{:}\mathbb{N}^+{\to}\mathbb{N}$ and $\boldsymbol{n}_0{:}\mathbb{N}^+{\to}List(\mathbb{N}^+)$, it holds that:

$$goodtopology(n, \boldsymbol{n}_0) \Rightarrow \tau{\cdot}DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0) = \tau{\cdot}\overline{rep}(\sum_{i=1}^{n} \boldsymbol{t}_0(i)){\cdot}\delta$$

In the trivial case that the root process 1 has no neighbours, which is equivalent to $n{=}1$, the τ's at the left and right hand side of the equation may be omitted.

12.3.2 Linearisation and invariants

Using the method of section 9.2.3 it is straightforward to write down a linear process for the network of n processes. The result is given in table 12.1 where the process L-$Impl$ is defined. From the technique of linearising n parallel processes, we can conclude that the following relation between L-$Impl$ and $DSum$ holds.

Lemma 12.3.5. For all $n{:}\mathbb{N}^+$, $\boldsymbol{t}_0{:}\mathbb{N}^+{\to}\mathbb{N}$ and $\boldsymbol{n}_0{:}\mathbb{N}^+{\to}List(\mathbb{N}^+)$ (where $\boldsymbol{p}_0$, $\boldsymbol{w}_0$ and $\boldsymbol{s}_0$ are as in definition 12.3.2), it holds that:

$$DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0) = L\text{-}Impl(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0).$$

We provide a number of invariants for L-$Impl$, most of which express that bookkeeping is done properly. The most interesting are invariants 14, 15 and 16. The first of these three implies that process 1 is reachable from each process in state 1 in a finite number of steps by iteratively following parent links (i.e. following variable p). As each process has a unique parent, this is an alternative way of saying that the parent links constitute a tree structure with process 1 as root (and a self-loop at the root). Invariant 15 expresses that

$$
\begin{array}{l}
L\text{-}Impl(n{:}\mathbb{N}^+, \boldsymbol{t}{:}\mathbb{N}^+{\to}\mathbb{N}, \boldsymbol{n}{:}\mathbb{N}^+{\to}List(\mathbb{N}), \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}{:}\mathbb{N}^+{\to}\mathbb{N}) = \\
\quad (\boldsymbol{n}(1){\approx}[] \wedge \boldsymbol{w}(1){\approx}1 \wedge \boldsymbol{s}(1){\approx}1) \to \overline{rep}(\boldsymbol{t}(1)){\cdot}L\text{-}Impl(\boldsymbol{s}{:=}\boldsymbol{s}[1{:=}2]){+} \\
\quad \textstyle\sum_{i,j:\mathbb{N}^+}(\boldsymbol{s}(i){\approx}1 \wedge i \in \boldsymbol{n}(j) \wedge \boldsymbol{s}(j){\approx}1 \wedge i{\not\approx}j \wedge i{\leq}n \wedge j{\leq}n \to \\
\qquad\qquad \tau{\cdot}L\text{-}Impl(\boldsymbol{n}{:=}\boldsymbol{n}[j := rem(i, \boldsymbol{n}(j)), i{:=}rem(j, \boldsymbol{n}(i))], \\
\qquad\qquad\qquad\qquad \boldsymbol{p}{:=}\boldsymbol{p}[i{:=}j], \boldsymbol{w}{:=}\boldsymbol{w}[i := \#\boldsymbol{n}(i){-}1], \boldsymbol{s}{:=}\boldsymbol{s}[i{:=}1]){+} \\
\quad \textstyle\sum_{i,j:\mathbb{N}^+}(\boldsymbol{s}(i){\approx}1 \wedge i \in \boldsymbol{n}(j) \wedge \boldsymbol{s}(j){\approx}1 \wedge i{\not\approx}j \wedge i{\leq}n \wedge j{\leq}n) \to \\
\qquad\qquad \tau{\cdot}L\text{-}Impl(\boldsymbol{n}{:=}\boldsymbol{n}[j{:=}rem(i, \boldsymbol{n}(j))], \boldsymbol{w}{:=}\boldsymbol{w}[i := \boldsymbol{w}(i){-}1]){+} \\
\quad \textstyle\sum_{j:\mathbb{N}^+}(\boldsymbol{n}(j){\approx}[] \wedge \boldsymbol{w}(j){\approx}0 \wedge \boldsymbol{s}(j){\approx}1 \wedge \boldsymbol{s}(\boldsymbol{p}(j)){\approx}1 \wedge j{\not\approx}1 \wedge j{\not\approx}\boldsymbol{p}(j) \wedge j{\leq}n \wedge \boldsymbol{p}(j){\leq}n) \to \\
\qquad\qquad \tau{\cdot}L\text{-}Impl(\boldsymbol{t}{:=}\boldsymbol{t}[\boldsymbol{p}(j){:=}\boldsymbol{t}(\boldsymbol{p}(j)) + \boldsymbol{t}(j)], \boldsymbol{w}{:=}\boldsymbol{w}[\boldsymbol{p}(j) := \boldsymbol{w}(\boldsymbol{p}(j)){-}1], \boldsymbol{s}{:=}\boldsymbol{s}[j{:=}2]).
\end{array}
$$

Table 12.1: Linearisation of the implementation

along each such path all processes are in state 1 too, meaning that they are willing to pass partial results along. Invariant 16 expresses that the total sum in the processes is maintained in the processes that are not in state 2. We will see that at a certain moment all processes, except process 1, are in state 2, which implies that at that moment the total sum is present in process 1.

The invariants mention the functions $Preach$, $Npoint$, $Ppoint$, and $actsum$, which are defined first.

Definition 12.3.6. Let $\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{p}$, $\boldsymbol{s}$ be as in Definition 12.3.2.

- The function $Preach(i, j, \boldsymbol{p}, m)$ expresses that from process i process j can be reached by following the parent links in $\boldsymbol{p}$. So $Preach(i, j, \boldsymbol{p}, m)$ holds if there are $i = i_0, \ldots, i_m = j$ such that, for all $0 \leq l < m$, $\boldsymbol{p}(i_l) = i_{l+1}$.
- $Npoint(i, \boldsymbol{n})$ is the number of sets L in $\boldsymbol{n}$ such that $i \in L$. Intuitively, $Npoint(i, \boldsymbol{n})$ is the number of processes that still need to send a $\overline{st}$ message to process i.
- $Ppoint(i, \boldsymbol{p}, \boldsymbol{s})$ is the number of processes $j \neq 1$ in the list $\boldsymbol{p}$ such that $\boldsymbol{p}(j) = i$ and $\boldsymbol{s}(j) = 1$. That is, $Ppoint(i, \boldsymbol{p}, \boldsymbol{s})$ is the number of active non-root processes that regard process i as their parent.
- $actsum(\boldsymbol{t}, \boldsymbol{s})$ is the sum of the $\boldsymbol{t}(i)$-values of the processes i that have not terminated yet, i.e. such that $\boldsymbol{s}(i) = 0$ or $\boldsymbol{s}(i) = 1$.

Theorem 12.3.7. The following are invariants of $L\text{-}Impl(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$. Here the universal quantification over i and j is left implicit. The conjunction of the invariants is written as $Inv(\boldsymbol{n}_0, \boldsymbol{t}_0, n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$. Note that the initial topology $\boldsymbol{n}_0$ and the initial distribution of values $\boldsymbol{t}_0$ are part of the invariant, although these are not a parameter of $L\text{-}Impl$.

1. $\boldsymbol{s}(i) \leq 2$.
2. $\boldsymbol{p}(i) \leq n$.
3. $i \in \boldsymbol{n}(j) \Rightarrow i \leq n$.
4. $i \notin \boldsymbol{n}(i)$.
5. $\boldsymbol{s}(1) \neq 1$.
6. $\boldsymbol{p}(1) = 0$.
7. $\boldsymbol{s}(i) = 0 \wedge j \in \boldsymbol{n}(i) \Rightarrow i \in \boldsymbol{n}(j)$.
8. $\boldsymbol{s}(i) = 0 \wedge i \in \boldsymbol{n}(j) \Rightarrow j \in \boldsymbol{n}(i)$.
9. $\boldsymbol{s}(i) = 0 \Rightarrow \boldsymbol{n}(i) = \boldsymbol{n}_0(i)$.

10. $\boldsymbol{s}(i) = 2 \Rightarrow \boldsymbol{w}(i) = 0 \wedge \boldsymbol{n}(i) = []$.

11. If a process i is in state 0, then it can't be a parent:
    $\boldsymbol{s}(i) = 0 \Rightarrow \boldsymbol{p}(j) \neq i$.

12. $\boldsymbol{s}(i) = 0 \Rightarrow \boldsymbol{w}(i) = Npoint(i, \boldsymbol{n}) \wedge Npoint(i, \boldsymbol{n}) = \#(\boldsymbol{n}(i)) \wedge Ppoint(i, \boldsymbol{p}, \boldsymbol{s}) = 0$.

13. For every process i, $\boldsymbol{w}(i)$ records exactly the number of messages that are to be received. These can either be st messages, or ans messages:
    $\boldsymbol{w}(i) = Npoint(i, \boldsymbol{n}) + Ppoint(i, \boldsymbol{p}, \boldsymbol{s})$.

14. From every process i process 1 is reachable via parent links in a finite number of steps:
    $\exists m{:}\mathbb{N}.Preach(i, 1, \boldsymbol{p}, m)$.

15. If a process i is in state 1, then its parent is also in state 1: $\boldsymbol{s}(i) = 1 \Rightarrow \boldsymbol{s}(\boldsymbol{p}(i)) = 1$.

16. As long as no $\overline{rep}$ message has been issued by process 1 (i.e. $\boldsymbol{s}(0) \neq 2$), the total sum (i.e. $\sum_{i=1}^{n} \boldsymbol{t}_0(i)$) is present in the processes that are in state 0 or 1: $\boldsymbol{s}(1) \neq 2 \Rightarrow actsum(\boldsymbol{t}, \boldsymbol{s}) = \sum_{i=1}^{n} \boldsymbol{t}_0(i)$.

Proof. The invariants 1 to 12 are easily checked (invariant 6 uses invariant 5). The invariant 13 uses invariants 4, 5, 6, 8 and 12. The invariant 14 uses invariant 11. The invariant 15 uses invariant 13. The last invariant can be proven on its own. □

12.3.3 State mapping, focus points and final lemma

In order to prove that the distributed summing delivers the correct sum using the cones and foci method, we specify a linear process $L\text{-}Spec$ describing the specification.

proc $\quad L\text{-}Spec(b{:}\mathbb{B}) = b \rightarrow \overline{rep}(\sum_{i=1}^{n}(\boldsymbol{t}_0))\ L\text{-}Spec(\neg b)$;

Clearly, $L\text{-}Spec(true) = \overline{rep}(\sum_{i=1}^{n} \boldsymbol{t}_0(i))\cdot\delta$.

Furthermore, we provide a *state mapping* h, that specifies how the control variable b of the specification $L\text{-}Spec$ is constructed out of the parameters $n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}$ of the implementation $L\text{-}Impl$. We define

$$h(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) = \boldsymbol{s}(1) \approx 1.$$

The intuition behind this definition is as follows. In a configuration s of $L\text{-}Impl$ that satisfies $\boldsymbol{s}(0) = 1$, $h(s)$ is $true$, so $L\text{-}Spec$ can perform the $\overline{rep}$-action, after which it halts. $L\text{-}Impl$ may not be able to perform a matching $\overline{rep}$ action directly, since the computation of the value to be reported has not yet finished (i.e., $\boldsymbol{n}(1) \neq []$ or $\boldsymbol{w}(1) \neq 0$). However, using the fact that $L\text{-}Impl$ is convergent, we see that after a finite number of internal τ-steps a configuration s' is reached where no τ-step is enabled, $\boldsymbol{s}(1)$ is still 1 (h will be invariant under the τ-steps), but also $\boldsymbol{n}(1) = []$ and $\boldsymbol{w}(1) = 0$. So the $\overline{rep}$-action can be performed (with the correct value), after which $L\text{-}Impl$ halts. Conversely, it is easy to verify that if in configuration s $L\text{-}Impl$ can perform the $\overline{rep}$ action, then $\boldsymbol{s}(1) = 1$, so in configuration $h(s)$ the control variable $b = h(s)$ of $L\text{-}Spec$ has the value $true$ and the specification $L\text{-}Spec$ can perform the $\overline{rep}$-action (with corresponding value). From these observations it will follow that h is indeed a functional branching bisimulation.

We formalise this intuitive argument, using a *focus condition*, which is a formula that characterises the configurations of $L\text{-}Impl$ in which no τ-step is enabled. Such a formula is extracted from the equation characterising $L\text{-}Impl$ (see table 12.1) by negating the guards that enable τ-steps in $L\text{-}Impl$. As an optimisation, we have put the first two negated guards together, and have restricted the focus condition to configurations satisfying the invariant.

$$\begin{aligned} FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) = \forall i, j \leq n \\ (\boldsymbol{s}[i] = 2 \vee i \notin \boldsymbol{n}[j] \vee \boldsymbol{s}[j] \neq 1 \vee i = j) \wedge \\ (\boldsymbol{n}[j] \neq \emptyset \vee \boldsymbol{w}[j] > 0 \vee \boldsymbol{s}[j] \neq 1 \vee \boldsymbol{s}[\boldsymbol{p}[j]] \neq 1 \vee j = 0) \end{aligned}$$

We distinguish two kinds of focus points of the distributed summation algorithm. One is the set of configurations where the algorithm has reported the sum and is terminated, so $\boldsymbol{s}(1) = 2$. The other one contains the configuration s' mentioned above and is characterised by $\boldsymbol{s}(1) = 1$. At that moment the correct sum should be reported. Items 1 and 2 of the lemma below say that all conditions in the process *L-Impl* for issuing a $\overline{rep}$ action are satisfied; so reporting is possible. Item 3 says that in such a case, all other processes are in state 2. Hence, using invariant 16 (i.e., $\boldsymbol{s}(1) \neq 2 \Rightarrow actsum(\boldsymbol{t}, \boldsymbol{s}) = \sum_{i=1}^{n} \boldsymbol{t}_0(i)$) we may conclude that the total sum is indeed collected in process 1, i.e. process 1 reports the correct sum.

Lemma 12.3.8. $Inv(\boldsymbol{n}_0, \boldsymbol{t}_0, n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$ and $\boldsymbol{s}(1) = 1$ together imply

1. $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \wedge \boldsymbol{s}(i) = 1 \Rightarrow \boldsymbol{n}(i) = []$.
2. $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \Rightarrow \boldsymbol{w}(1) = 0$.
3. $goodtopology(n, \boldsymbol{n}_0) \wedge \boldsymbol{w}(1) = 0 \wedge i \neq 1 \Rightarrow \boldsymbol{s}(1) = 2$.

Proof.

1. Towards a contradiction, assume there exists a process i such that $\boldsymbol{s}(i) = 1$ and $\boldsymbol{n}(i) \neq []$, say $j \in \boldsymbol{n}(i)$. By invariant 4 we have $j \neq i$. The first conjunct of $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$ yields that $\boldsymbol{s}(j) = 2$. By invariant 10, $\boldsymbol{w}(j) = 0$, contradicting invariant 13 (remember that $j \in \boldsymbol{n}(i)$).

2. In order to derive a contradiction, assume that $\boldsymbol{w}(1) > 0$. For arbitrary m, we construct a sequence of m processes $1 = i_1, \ldots, i_m$ such that for all $1 \leq l \leq m$, we have $\boldsymbol{s}(i_l) = 1$, $\boldsymbol{w}(i_l) > 0$, $\boldsymbol{p}(i_{l+1}) = i_l$, and if $l \neq 1$, $i_l \neq 1$. Clearly, if $m > n$, there is one element $i_r \neq 1$ which appears twice in the path. Hence we obtain a cycle in the path starting from i_r that does not contain 1. So, i_1 can't be reachable via parent links from i_r and in particular from i_m, this contradicts the existence of the current sequence.

   Let a process i_l be given such that $\boldsymbol{w}(i_l) > 0$ and $\boldsymbol{s}(i_l) = 1$. According to invariant 13 at least one of the following should hold.

   - There is some i such that $i_l \in \boldsymbol{n}(i)$. By invariant 4, $i_l \neq i$. By the first part of $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$ it follows that $\boldsymbol{s}(i) \neq 1$. So, either $\boldsymbol{s}(i) = 2$, but this leads to a contradiction using invariant 10 (remember that $\boldsymbol{n}(i) \neq \emptyset$). Or, $\boldsymbol{s}(i) = 0$. By invariant 7, $i \in \boldsymbol{n}(i_l)$. So, by $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$, $\boldsymbol{s}(i_l) \neq 1$. Contradiction.
   - Or there is some i such that $\boldsymbol{p}(i) = i_l$, $i \neq 1$ and $\boldsymbol{s}(i) = 1$. By the second part of $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$, we have $\boldsymbol{w}(i) > 0 \vee \boldsymbol{n}(i) \neq \emptyset$. By item 1 of this lemma, $\boldsymbol{n}(i) = \emptyset$. So $\boldsymbol{w}(i) > 0$. We can take $i_{l+1} = i$.

3. First, assume there is some process $i \neq 1$ such that $\boldsymbol{s}(i) = 1$. Using invariants 13, 15 and 14, it follows that there is a sequence of processes $i = i_1, \ldots, i_m = 1$ such that, for all $0 \leq l < m$, $i_l \neq 1$, $\boldsymbol{p}(i_l) = i_{l+1}$, $\boldsymbol{s}(i_l) = 1$ and $\boldsymbol{w}(i_{l+1}) > 0$. In particular $\boldsymbol{w}(1) > 0$ contradicting an assumption.

   So, assume that there is no process $i \neq 1$ such that $\boldsymbol{s}(i) = 1$, but there is some process $i \neq 1$ such that $\boldsymbol{s}(i) = 0$. From the topology requirement it follows that there is a sequence $i = i_1, \ldots, i_m = 1$ such that for all $1 \leq l < m$, $i_{l+1} \in \boldsymbol{n}_0(i_l)$. We show that $\boldsymbol{s}(i_l) = 0$ for all l, $1 \leq l \leq m$. This contradicts the assumption that $\boldsymbol{s}(1) = 1$.

   Note that by assumption $\boldsymbol{s}(i_1) = 0$. So let i_l be such that $\boldsymbol{s}(i_l) = 0$. By invariant 9, it follows that $i_{l+1} \in \boldsymbol{n}(i_l)$. By invariant 13, $\boldsymbol{w}(i_{l+1}) > 0$, so $i_{l+1} \neq 1$ and, by invariant 10, $\boldsymbol{s}(i_{l+1}) \neq 2$. As we have excluded that process i_{l+1} is in state 1, it must hold that $\boldsymbol{s}(i_{l+1}) = 0$, as required.

□

Below we copy the General Equality Theorem (see theorem 11.1.1) instantiated for the distributed summation algorithm. It says that, given the invariant, implementation *L-Impl* and specification *L-Spec* are equivalent. Its proof requires that the 6 matching criteria are checked. Given lemma 12.3.8 this is straightforward.

Lemma 12.3.9. Assume $goodtopology(n, \boldsymbol{n}_0)$.

$$Inv(\boldsymbol{n}_0, \boldsymbol{t}_0, n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \Rightarrow \tau{\cdot}L\text{-}Impl(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) = \tau{\cdot}L\text{-}Spec(\boldsymbol{s}(1){\approx}1)$$

Proof. It suffices to check that the following instances of the matching criteria are implied by the invariant.

1. $L\text{-}Impl$ in table 12.1 is convergent, i.e. does not admit infinite τ-paths. At each τ-step, either a link in $\boldsymbol{n}$ is removed, or a process moves from state 1 to state 2. Hence, the sum of the number of links in $\boldsymbol{n}$ and the number of processes in state 0 or 1 strictly decreases with each τ-step.

2. The following three requirements ensure that the state mapping h is invariant under τ-steps of $L\text{-}Impl$.

   (a) $\boldsymbol{s}(i) = 0 \wedge i \in \boldsymbol{n}(j) \wedge \boldsymbol{s}(j) = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n$ implies $\boldsymbol{s}(1) = \boldsymbol{s}[i{:=}1](1)$. We distinguish two cases. If $i \neq 1$, the condition trivially holds because $(\boldsymbol{s}[i{:=}1](1) = \boldsymbol{s}(1)$. If $i = 1$, one conjunct of the precondition says $\boldsymbol{s}(1) = 0$. This contradicts invariant 5.

   (b) $\boldsymbol{s}(i) = 1 \wedge i \in \boldsymbol{n}(j) \wedge \boldsymbol{s}(j) = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n$ implies $\boldsymbol{s}(1) = \boldsymbol{s}(1)$.
   This requirement clearly holds.

   (c) $\boldsymbol{n}(j) = [] \wedge \boldsymbol{w}(j) = 0 \wedge \boldsymbol{s}(j) = 1 \wedge \boldsymbol{s}(\boldsymbol{p}(j)) = 1 \wedge j \neq 1 \wedge j \neq \boldsymbol{p}(j) \wedge j \leq n \wedge \boldsymbol{p}(j) \leq n$ implies $\boldsymbol{s}(1) = \boldsymbol{s}[j{:=}2](1)$. This requirement is also trivially valid, because the assumption explicitly says $j \neq 1$. Hence, $\boldsymbol{s}[j{:=}2](1) = \boldsymbol{s}(1)$.

3. Next, we verify that when the $\overline{rep}$ action is enabled in $L\text{-}Impl$, it is enabled in $L\text{-}Spec$: $\boldsymbol{n}(1) = [] \wedge \boldsymbol{w}(1) = 0 \wedge \boldsymbol{s}(1) = 1$ implies $\boldsymbol{s}(1) = 1$. This is obviously true.

4. We must show that if $L\text{-}Impl$ is in a focus point (no internal actions enabled) and $L\text{-}Spec$ can perform a $\overline{rep}$-action, $L\text{-}Impl$ can also perform the $\overline{rep}$ action:
   $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \wedge \boldsymbol{s}(1) = 1$ implies $\boldsymbol{n}(1) = [] \wedge \boldsymbol{w}(1) = 0 \wedge \boldsymbol{s}(1) = 1$. This is a direct consequence of lemma 12.3.8.2 and lemma 12.3.8.1.

5. We must show that if the $\overline{rep}$ action is enabled in $L\text{-}Impl$ then the reported sum is equal to the sum reported in $L\text{-}Spec$: $\boldsymbol{n}(1) = [] \wedge \boldsymbol{w}(n) = 0 \wedge \boldsymbol{s}(1) = 1$ implies $\boldsymbol{t}(1) = \sum_{i=1}^{n} \boldsymbol{t}_0(i)$. By invariant 16 we have $\sum_{i=1}^{n} \boldsymbol{t}_0(i) = actsum(\boldsymbol{t}, \boldsymbol{s})$. By definition, $actsum(\boldsymbol{t}, \boldsymbol{s})$ contains the sum of the $\boldsymbol{t}(i)$ values of all processes i that are not in state 2. By lemma 12.3.8.3, only process 1 is not in state 2. Hence $\sum_{i=1}^{n} \boldsymbol{t}_0(i) = actsum(\boldsymbol{t}, \boldsymbol{s}) = \boldsymbol{t}(1)$.

6. Finally, we have to show that the h-mapping commutes with the $\overline{rep}$ action, i.e. $\boldsymbol{s}[1{:=}2](1) \neq 1$. This is easily seen to hold.

□

Theorem 12.3.10. For all $n{:}\mathbb{N}^+$, $\boldsymbol{t}_0{:}\mathbb{N}^+{\to}\mathbb{N}$ and $\boldsymbol{n}_0{:}\mathbb{N}^+{\to}List(\mathbb{N}^+)$, it holds that:

$$goodtopology(n, \boldsymbol{n}_0) \Rightarrow \tau{\cdot}DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0) = \tau{\cdot}\overline{rep}(\sum_{i=1}^{n} \boldsymbol{t}_0(i)){\cdot}\delta$$

If $n = 1$, the initial τ's at the left and right side of the equation can be omitted.

Proof. Apply lemma 12.3.9 with $\boldsymbol{t}_0$ substituted for $\boldsymbol{t}$, $\boldsymbol{n}_0$ for $\boldsymbol{n}$, $\boldsymbol{p}_0$ for $\boldsymbol{p}$, $\boldsymbol{w}_0$ for $\boldsymbol{w}$ and $\boldsymbol{s}_0$ for $\boldsymbol{s}$. As $goodtopology(n, \boldsymbol{n}_0)$ holds, this substitution reduces the invariant to $true$. Thus we have

$$\tau{\cdot}L\text{-}Impl(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0) = \tau{\cdot}L\text{-}Spec(true).$$

By lemma 12.3.5 and the definition of $L\text{-}Spec$ the theorem follows immediately. □

Part IV

Checking properties of systems

Part V

Appendices

Appendix A

Equational definition of built in datatypes

The definitions of predefined sorts are given below. As a general rule, all internal functions and sorts that cannot be accessed directly in a mCRL2 specification start with an '@'-symbol.

For all sorts, a function if (if), equality ($\approx$) and inequality ($\not\approx$) are defined. This includes the basic sorts and all function sorts constructed out of them, used within a specification. As their definitions are all the same, we provide their declarations and definition as a template for an arbitrary sort S. Note that for concrete sorts, there are additional equations for $\approx$. E.g., $true \approx false = false$. The equation $if(f \approx g, f, g) = g$ is called *Bergstra's axiom* and is used to show that $=$ and $\approx$ coincide. Although extremely useful for manual verification, this equation is of no use to tools. So, generally, it is not generated for the use in tools.

map $\approx, \not\approx : S \times S \to \mathbb{B}$;
$if : \mathbb{B} \times S \times S \to S$;
var $x, y : S$;
$b : \mathbb{B}$;
eqn $x \approx x = true$;
$x \not\approx y = \neg(x \approx y)$;
$if(true, x, y) = x$;
$if(false, x, y) = y$;
$if(b, x, x) = x$;
$if(x \approx y, x, y) = y$;

A.1 Bool

The booleans have a straightforward definition. They contain the constructors $true$ and $false$. The semantics of the language mCRL2 prescribes that $true$ and $false$ are different. Adding an equation $true = false$ makes a specification semantically inconsistent, and henceforth it does not define anything anymore.

sort $\mathbb{B}$;
cons $true, false : \mathbb{B}$;
map $\neg : \mathbb{B} \to \mathbb{B}$;
$\wedge, \vee, \Rightarrow : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$;

var $b : \mathbb{B}$;
eqn $\neg true = false$;
$\neg false = true$;
$\neg\neg b = b$;
$b \wedge true = b$;
$b \wedge false = false$;
$true \wedge b = b$;
$false \wedge b = false$;
$b \vee true = true$;
$b \vee false = b$;
$true \vee b = true$;
$false \vee b = b$;
$b \Rightarrow true = true$;
$b \Rightarrow false = \neg b$;
$true \Rightarrow b = b$;
$false \Rightarrow b = true$;
$true \approx b = b$;
$false \approx b = \neg b$;
$b \approx true = b$;
$b \approx false = \neg b$;

A.2 Positive numbers

The internal representation of numbers is such that each unique number has its own unique representation with constructors, and the representation is logarithmic, in the sense that for a number n the representation requires $O(^2\log(n))$ constructors (just as in the ordinary decimal or binary notation of numbers). In order to achieve this, we first define positive numbers and build all other numbers on the basis of these.

For positive numbers ($\mathbb{N}^+$) the number one is represented by the constructor $@c1$. Furthermore there is a constructor $@cDub(b, n)$ with b a boolean and n a natural number. Its meaning is

$$@cDub(b, n) = \begin{cases} 2n+1 & \text{if } b \text{ is } true, \\ 2n & \text{if } b \text{ is } false. \end{cases}$$

When a rewriter is used, the data terms of mCRL2 are translated to internal form. For example a number 2 is translated to $@cDub(false, @c1)$ and 5 becomes $@cDub(true, @cDub(false, @c1))$. The results are generally transformed back using a pretty printer.

sort $\mathbb{N}^+$;
cons $@c1 : \mathbb{N}^+$;
$@cDub : \mathbb{B} \times \mathbb{N}^+ \to \mathbb{N}^+$;

map $\leq, <, \geq, >: \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{B}$;
$max, min : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$;
$abs, succ : \mathbb{N}^+ \to \mathbb{N}^+$;
$+ : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$;
$@addc : \mathbb{B} \times \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$;
$* : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$;
$@multir : \mathbb{B} \times \mathbb{N}^+ \times \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}^+$;

var $b, c : \mathbb{B}$;
$p, q, r : \mathbb{N}^+$;

eqn $@c1 \approx @cDub(b,p) = false$;
$@cDub(b,p) \approx @c1 = false$;
$@cDub(b,p) \approx @cDub(b,q) = p \approx q$;
$@cDub(false,p) \approx @cDub(true,q) = false$;
$@cDub(true,p) \approx @cDub(false,q) = false$;
$@c1 \leq p = true$;
$@cDub(b,p) \leq @c1 = false$;
$@cDub(b,p) \leq @cDub(b,q) = p \leq q$;
$@cDub(false,p) \leq @cDub(b,q) = p \leq q$;
$@cDub(true,p) \leq @cDub(false,q) = p < q$;
$@cDub(b,p) <= @cDub(c,q) = if(b \Rightarrow c, p \leq q, p < q)$;
$p < @c1 = false$;
$@c1 < @cDub(b,p) = true$;
$@cDub(b,p) < @cDub(b,q) = p < q$;
$@cDub(false,p) < @cDub(true,q) = p \leq q$;
$@cDub(b,p) < @cDub(false,q) = p < q$;
$@cDub(b,p) < cDub(c,q) = if(c \Rightarrow b, p < q, p \leq q)$;
$p \geq q = q \leq p$;
$p > q = q < p$;
$max(p,q) = if(p \leq q, q, p)$;
$min(p,q) = if(p \leq q, p, q)$;
$abs(p) = p$;
$succ(@c1) = @cDub(false, @c1)$;
$succ(@cDub(false,p)) = @cDub(true,p)$;
$succ(@cDub(true,p)) = @cDub(false, succ(p))$;
$p + q = @addc(false,p,q)$;
$@addc(false, @c1, p) = succ(p)$;
$@addc(true, @c1, p) = succ(succ(p))$;
$@addc(false, p, @c1) = succ(p)$;
$@addc(true, p, @c1) = succ(succ(p))$;
$@addc(b, @cDub(c,p), @cDub(c,q)) = @cDub(b, @addc(c,p,q))$;
$@addc(b, @cDub(false,p), @cDub(true,q)) = @cDub(\neg b, @addc(b,p,q))$;
$@addc(b, @cDub(true,p), @cDub(false,q)) = @cDub(\neg b, @addc(b,p,q))$;
$p \leq q \rightarrow p * q = @multir(false, @c1, p, q)$;
$p > q \rightarrow p * q = @multir(false, @c1, q, p)$;
$@multir(false, p, @c1, q) = q$;
$@multir(true, p, @c1, q) = @addc(false, p, q)$;
$@multir(b, p, @cDub(false,q), r) = @multir(b, p, q, @cDub(false,r))$;
$@multir(false, p, @cDub(true,q), r) = @multir(true, r, q, @cDub(false,r))$;
$@multir(true, p, @cDub(true,q), r) = @multir(true, @addc(false,p,r), q, @cDub(false,r))$;

A.3 Natural numbers

The sort $\mathbb{N}$ represents the natural numbers. The sort $@NatPair$ is an auxiliary sort used for an efficient implementation of the div and mod functions.

sort $\mathbb{N}$;
$@NatPair$;

Natural number are constructed out of the positive numbers using the constructors $@c0$, representing 0, and $@cNat(p)$ which interprets positive number p as the natural number with the same value.

cons $@c0 : \mathbb{N}$;
$@cNat : \mathbb{N}^+ \rightarrow \mathbb{N}$;
$@cPair : \mathbb{N} \times \mathbb{N} \rightarrow @NatPair$;

map $Pos2Nat : \mathbb{N}^+ \to \mathbb{N}$;
$Nat2Pos : \mathbb{N} \to \mathbb{N}^+$;
$\leq, <, \geq, >: \mathbb{N} \times \mathbb{N} \to \mathbb{B}$;
$max : \mathbb{N}^+ \times \mathbb{N} \to \mathbb{N}^+$;
$max : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}^+$;
$max, min : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;
$abs : \mathbb{N} \to \mathbb{N}$;
$succ : \mathbb{N} \to \mathbb{N}^+$;
$pred : \mathbb{N}^+ \to \mathbb{N}$;
$@dub : \mathbb{B} \times \mathbb{N} \to \mathbb{N}$;
$+ : \mathbb{N}^+ \times \mathbb{N} \to \mathbb{N}^+$;
$+ : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}^+$;
$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;
$@gtesubt : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$;
$@gtesubt : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;
$@gtesubtb : \mathbb{B} \times \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$;
$* : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;
$div : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$;
$div : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}$;
$\_|\_ : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$;
$\_|\_ : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}$;
$exp : \mathbb{N}^+ \times \mathbb{N} \to \mathbb{N}^+$;
$exp : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;
$@even : \mathbb{N} \to \mathbb{B}$;
$@first, @last : @NatPair \to \mathbb{N}$;
$@divmod : \mathbb{N}^+ \times \mathbb{N}^+ \to @NatPair$;
$@gdivmod : @NatPair \times \mathbb{B} \times \mathbb{N}^+ \to @NatPair$;
$@ggdivmod : \mathbb{N} \times \mathbb{N} \times \mathbb{N}^+ \to @NatPair$;

var $f, g : \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$;
$b, c : \mathbb{B}$;
$p, q, r : \mathbb{N}^+$;
$n, m, m', n' : \mathbb{N}$;
$x, y : @NatPair$;

eqn $@c0 \approx @cNat(p) = false$;
$@cNat(p) \approx @c0 = false$;
$@cNat(p) \approx @cNat(q) = p \approx q$;
$Pos2Nat = @cNat$;
$Nat2Pos(p) = p$;
$@c0 \leq n = true$;
$@cNat(p) \leq @c0 = false$;
$@cNat(p) \leq @cNat(q) = p \leq q$;
$n < @c0 = false$;
$@c0 < @cNat(p) = true$;
$@cNat(p) < @cNat(q) = p < q$;
$m \geq n = n \leq m$;
$m > n = n < m$;

$max(p, @c0) = p;$
$max(p, @cNat(q)) = max(p, q);$
$max(@c0, p) = p;$
$max(@cNat(p), q) = max(p, q);$
$max(m, n) = if(m \leq n, n, m);$
$min(m, n) = if(m \leq n, m, n);$
$abs(n) = n;$
$succ(@c0) = @cNat(@c1);$
$succ(@cNat(p)) = succ(p);$
$pred(@c1) = @c0;$
$pred(@cDub(true, p) = @cDub(false, p);$
$pred(@cDub(false, p)) = @dub(true, pred(p));$
$@dub(false, @c0) = @c0;$
$@dub(true, @c0) = @cNat(@c1);$
$@dub(b, @cNat(p)) = @cDub(b, p);$

$p + @c0 = p;$
$p + @cNat(q) = p + q;$
$@c0 + p = p;$
$@cNat(p) + q = p + q;$
$@c0 + n = n;$
$n + @c0 = n;$
$@cNat(p) + @cNat(q) = @addc(false, p, q);$
$@gtesubt(p, q) = @gtesubtb(false, p, q);$
$@gtesubt(n, @c0) = n;$
$@gtesubt(@cNat(p), @cNat(q)) = @gtesubt(p, q);$
$@gtesubtb(false, p, @c1) = pred(p);$
$@gtesubtb(true, p, @c1) = pred(Nat2Pos(pred(p)));$
$@gtesubtb(b, @cDub(c, p), @cDub(c, q)) = @dub(b, @gtesubtb(b, p, q));$
$@gtesubtb(b, @cDub(false, p), @cDub(true, q)) = @dub(\neg b, @gtesubtb(true, p, q));$
$@gtesubtb(b, @cDub(true, p), @cDub(false, q)) = @dub(\neg b, @gtesubtb(false, p, q));$
$@c0 * n = @c0;$
$n * @c0 = @c0;$
$@cNat(p) * @cNat(q) = @cNat(p * q);$

$exp(p, @c0) = @c1;$
$exp(p, @cNat(@c1)) = p;$
$exp(p, @cNat(@cDub(false, q))) = exp(@multir(false, @c1, p, p), q);$
$exp(p, @cNat(@cDub(true, q))) = @multir(false, @c1, p, exp(@multir(false, @c1, p, p), q));$
$exp(n, @c0) = @cNat(@c1);$
$exp(@c0, p) = @c0;$
$exp(@cNat(p), n) = @cNat(exp(p, n));$
$@even(@c0) = true;$
$@even(@cNat(@c1)) = false;$
$@even(@cNat(@cDub(b, p))) = \neg b;$

$p \operatorname{div} @c1 = @cNat(p)$;
$@c1 \operatorname{div} @cDub(b,p) = @c0$;
$@cDub(b,p) \operatorname{div} @cDub(false,q) = p \operatorname{div} q$;
$p \leq q \rightarrow @cDub(false,p) \operatorname{div} @cDub(true,q) = @c0$;
$p > q \rightarrow @cDub(false,p) \operatorname{div} @cDub(true,q) =$
$@first(@divmod(@cDub(false,p), @cDub(true,q)))$;
$p \leq q \rightarrow @cDub(true,p) \operatorname{div} @cDub(true,q) = if(p \approx q, @cNat(@c1), @c0)$;
$p > q \rightarrow @cDub(true,p) \operatorname{div} @cDub(true,q) =$
$@first(@divmod(@cDub(true,p), cDub(true,q)))$;
$@c0 \operatorname{div} p = @c0$;
$@cNat(p) \operatorname{div} q = p \operatorname{div} q$;
$p|_{@c1} = @c0$;
$@c1|_{@cDub(b,p)} = @cNat(@c1)$;
$@cDub(b,p)|_{@cDub(false,q)} = @dub(b, p|_q)$;
$p \leq q \rightarrow @cDub(false,p)|_{@cDub(true,q)} = @cNat(@cDub(false,p))$;
$p > q \rightarrow @cDub(false,p)|_{@cDub(true,q)} = @last(@divmod(@cDub(false,p), @cDub(true,q)))$;
$p \leq q \rightarrow @cDub(true,p)|_{@cDub(true,q)} = if(p \approx q, @c0, @cNat(@cDub(true,p)))$;
$p > q \rightarrow @cDub(true,p)|_{@cDub(true,q)} = @last(@divmod(@cDub(true,p), @cDub(true,q)))$;
$@c0|_p = @c0$;
$@cNat(p)|_q = p|_q$;

$@cPair(m,n) \approx @cPair(m',n') = m \approx m' \wedge n \approx n'$;
$@first(@cPair(m,n)) = m$;
$@last(@cPair(m,n)) = n$;
$@divmod(@c1, @c1) = @cPair(@cNat(@c1), @c0)$;
$@divmod(@c1, @cDub(b,p)) = @cPair(@c0, @cNat(@c1))$;
$@divmod(@cDub(b,p), q) = @gdivmod(@divmod(p,q), b, q)$;
$@gdivmod(@cPair(m,n), b, p) = @ggdivmod(@dub(b,n), m, p)$;
$@ggdivmod(@c0, n, p) = @cPair(@dub(false,n), @c0)$;
$p < q \rightarrow @ggdivmod(p,n,q) = @cPair(@dub(false,n), @cNat(p))$;
$p \geq q \rightarrow @ggdivmod(@cNat(p), n, q) = @cPair(@dub(true,n), @gtesubtb(false,p,q))$;

A.4 Integers

sort $\mathbb{Z}$;

Integers are constructed out of both natural and positive numbers using two additional constructors. The integer $@cInt(n)$ for a natural number n represents the integer with value n. The integer $@cNeg(p)$ for a positive number p represents the integer $-p$.

cons $@cInt : \mathbb{N} \rightarrow \mathbb{Z}$;
$@cNeg : \mathbb{N}^+ \rightarrow \mathbb{Z}$;

map

$Nat2Int : \mathbb{N} \to \mathbb{Z}$;
$Int2Nat : \mathbb{Z} \to \mathbb{N}$;
$Pos2Int : \mathbb{N}^+ \to \mathbb{Z}$;
$Int2Pos : \mathbb{Z} \to \mathbb{N}^+$;
$\leq, <, \geq, >: \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$;
$max : \mathbb{N}^+ \times \mathbb{Z} \to \mathbb{N}^+$;
$max : \mathbb{Z} \times \mathbb{N}^+ \to \mathbb{N}^+$;
$max : \mathbb{N} \times \mathbb{Z} \to \mathbb{N}$;
$max : \mathbb{Z} \times \mathbb{N} \to \mathbb{N}$;
$max, min : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$;
$abs : \mathbb{Z} \to \mathbb{N}$;
$- : \mathbb{N}^+ \to \mathbb{Z}$;
$- : \mathbb{N} \to \mathbb{Z}$;
$-, succ : \mathbb{Z} \to \mathbb{Z}$;
$pred : \mathbb{N} \to \mathbb{Z}$;
$pred : \mathbb{Z} \to \mathbb{Z}$;
$@dub : \mathbb{B} \times \mathbb{Z} \to \mathbb{Z}$;
$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$;
$- : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{Z}$;
$- : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$;
$-, * : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$;
$div : \mathbb{Z} \times \mathbb{N}^+ \to \mathbb{Z}$;
$\_|\_ : \mathbb{Z} \times \mathbb{N}^+ \to \mathbb{N}$;
$exp : \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$;

var

$x, y : \mathbb{Z}$;
$b : \mathbb{B}$;
$n, m : \mathbb{N}$;
$p, q : \mathbb{N}^+$;

eqn

$m \approx n = m \approx n$;
$n \approx -p = false$;
$-p \approx n = false$;
$-p \approx -q = p \approx q$;
$Nat2Int = @cInt$;
$Int2Nat(n) = n$;
$Pos2\mathbb{Z}(p) = p$;
$Int2Pos(n) = Nat2Pos(n)$;
$m \leq n = m \leq n$;
$n \leq -p = false$;
$-p \leq n = true$;
$-p \leq -q = q \leq p$;
$m < n = m < n$;
$n < -p = false$;
$-p < n = true$;
$-p < -q = q < p$;
$x \geq y = y \leq x$;
$x > y = y < x$;

$max(p, n) = max(p, n)$;
$max(p, -q) = p$;
$max(n, p) = max(n, p)$;
$max(-q, p) = p$;
$max(m, n) = max(m, n)$;
$max(n, -p) = n$;
$max(m, n) = max(m, n)$;
$max(-p, n) = n$;
$max(x, y) = if(x \leq y, y, x)$;
$min(x, y) = if(x \leq y, x, y)$;
$abs(n) = n$;
$abs(-p) = p$;

$-p = -p$;
$-@c0 = @c0$;
$-p = -p$;
$-n = -n$;
$- - p = p$;
$succ(n) = succ(n)$;
$succ(-p) = -pred(p)$;
$pred(@c0) = -@c1$;
$pred(p) = pred(p)$;
$pred(n) = pred(n)$;
$pred(-p) = -succ(p)$;
$@dub(b, n) = @dub(b, n)$;
$@dub(false, -p) = -@cDub(false, p)$;
$@dub(true, -p) = -pred(@cDub(false, p))$;

$m + n = m + n$;
$n + -p = n - p$;
$-p + n = n - p$;
$-p + -q = -@addc(false, p, q)$;
$p \geq q \rightarrow p - q = @gtesubt(p, q)$;
$p < q \rightarrow p - q = -@gtesubt(q, p)$;
$m \geq n \rightarrow m - n = @gtesubt(m, n)$;
$m < n \rightarrow m - n = -@gtesubt(n, m)$;

$x - y = x + -y$;
$m * n = m * n$;
$n * -p = -p * n$;
$-p * n = -p * n$;
$-p * -q = p * q$;
$n \text{ div } p = n \text{ div } p$;
$-p \text{ div } q = -succ(pred(p) \text{ div } q)$;
$n|_p = n|_p$;
$-p|_q = Int2Nat(q - succ(pred(p)|_q))$;
$exp(m, n) = exp(m, n)$;
$@even(n) \rightarrow exp(-p, n) = exp(p, n)$;
$\neg @even(n) \rightarrow exp(-p, n) = -exp(p, n)$;

A.5 Reals

The implementation of reals is currently very minimalistic. Research is currently going on to strengthen this, and after the results become available, the reals in mCRL2 will be replaced. Note that unlike other

sorts, the reals do not have constructors. Every constructor sort has at most an enumerable number of elements. The reals are not enumerable and therefore a definition using constructors is not possible.

sort $\mathbb{R}$;
map $@cReal : \mathbb{Z} \to \mathbb{R}$;
$Pos2Real : \mathbb{N}^+ \to \mathbb{R}$;
$Nat2Real : \mathbb{N} \to \mathbb{R}$;
$Int2Real : \mathbb{Z} \to \mathbb{R}$;
$Real2Pos : \mathbb{R} \to \mathbb{N}^+$;
$Real2Nat : \mathbb{R} \to \mathbb{N}$;
$Real2Int : \mathbb{R} \to \mathbb{Z}$;
$\leq, <, \geq, >: \mathbb{R} \times \mathbb{R} \to \mathbb{B}$;
$\max, \min : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$;
$abs, -, succ, pred : \mathbb{R} \to \mathbb{R}$;
$+, -, * : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$;
$exp : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$;

var $p : \mathbb{N}^+$;
$n : \mathbb{N}$;
$x, y : \mathbb{Z}$;
$r, s : \mathbb{R}$;
eqn $@cReal(x) \approx @cReal(y) = x \approx y$;
$Int2Real = @cReal$;
$Nat2Real(n) = @cReal(@cInt(n))$;
$Pos2Real(p) = @cReal(@cInt(@cNat(p)))$;
$Real2Int(@cReal(x)) = x$;
$Real2Nat(@cReal(x)) = Int2Nat(x)$;
$Real2Pos(@cReal(x)) = Int2Pos(x)$;
$@cReal(x) \leq @cReal(y) = x \leq y$;
$@cReal(x) < @cReal(y) = x < y$;
$r \geq s = s \leq r$;
$r > s = s < r$;
$\max(r, s) = if(r \leq s, s, r)$;
$\min(r, s) = if(r \leq s, r, s)$;
$abs(@cReal(x)) = @cReal(@cInt(abs(x)))$;
$-(@cReal(x)) = @cReal(-(x))$;
$succ(@cReal(x)) = @cReal(succ(x))$;
$pred(@cReal(x)) = @cReal(pred(x))$;
$@cReal(x) + @cReal(y) = @cReal(x + y)$;
$@cReal(x) - @cReal(y) = @cReal(x - y)$;
$@cReal(x) * @cReal(y) = @cReal(x * y)$;
$exp(@cReal(x), n) = @cReal(exp(x, n))$;

A.6 Lists

Given a sort S the sort $List(S)$ is translated to some sort $List@0$. For each list over a sort a different sortname of the form $list@n$ is generated including all equations below.

sort $List@0$;
S;

cons $[] : List@0$;
$\triangleright : S \times List@0 \to List@0$;

map $in : S \times List@0 \to \mathbb{B}$;
$\# : List@0 \to \mathbb{N}$;
$\lhd : List@0 \times S \to List@0$;
$+\!\!+ : List@0 \times List@0 \to List@0$;
$. : List@0 \times \mathbb{N} \to S$;
$head : List@0 \to S$;
$tail : List@0 \to List@0$;
$rhead : List@0 \to S$;
$rtail : List@0 \to List@0$;

var $d, e : S$;
$s, t : List@0$;
$p : \mathbb{N}^+$;

eqn $[] \approx d \rhd s = false$;
$d \rhd s \approx [] = false$;
$d \rhd s \approx e \rhd t = d \approx e \wedge s \approx t$;
$in(d, []) = false$;
$in(d, e \rhd s) = d \approx e \vee in(d, s)$;
$\#[] = @c0$;
$\#d \rhd s = @cNat(succ(\#s))$;
$[] \lhd d = d \rhd []$;
$(d \rhd s) \lhd e = d \rhd (s \lhd e)$;
$[] +\!\!+ s = s$;
$(d \rhd s) +\!\!+ t = d \rhd (s +\!\!+ t)$;
$s +\!\!+ [] = s$;
$(d \rhd s).@c0 = d$;
$(d \rhd s).@cNat(p) = s.pred(p)$;
$head(d \rhd s) = d$;
$tail(d \rhd s) = s$;
$rhead(d \rhd []) = d$;
$rhead(d \rhd (e \rhd s)) = rhead(e \rhd s)$;
$rtail(d \rhd []) = []$;
$rtail(d \rhd (e \rhd s)) = d \rhd rtail(e \rhd s)$;

A.7 Sets

The sort $Set(S)$ for some given sort S translates to some sort $Set@n$ where n is some natural number. The notation below is still slightly misleading. Instead of representing a set by $\{d{:}D | f(d)\}$ the internal representation is actually the characteristic function of the set, namely $\lambda d{:}D.f(d)$.

sort $Set@0$;
map $@set : S \to Set@0$;
$\{\} : Set@0$;
$\in: S \times Set@0 \to \mathbb{B}$;
$\subseteq, \subset: Set@0 \times Set@0 \to \mathbb{B}$;
$\cup, -, \cap : Set@0 \times Set@0 \to Set@0$;
$\overline{\ } : Set@0 \to Set@0$;
var $d : S$;
$s, t : Set@0$;
$f, g : S \to \mathbb{B}$;
eqn $\{\} = \{x{:}S | false\}$;
$d \in \{x : S | f(x)\} = f(d)$;
$\{x{:}S | f(x)\} \subseteq \{x{:}S | g(x)\} = \forall x{:}S.f(x) \Rightarrow g(x)$;
$s \subset t = s \subseteq t \wedge s \not\approx t$;
$\{x{:}S | f(x)\} \cup \{x{:}S | g(x)\} = \{x{:}S | f(x) \vee g(x)\}$;
$s - t = s \cap \overline{t}$;
$\overline{\{x{:}S | f(x)\}} \cap \{x{:}S | g(x)\} = \{x{:}S | f(x) \wedge g(x)\}$;
$\overline{\{x{:}S | f(x)\}} = \{x{:}S | \neg f(x)\}$;

A.8 Bags

For a sort $Bag(S)$ of over some arbitrary sort S, the following internal sorts and rewrite rules are generated. Just as with sets, the internal representation of the bags consists of its characteristic function.

sort $Bag@0$;
map $@bag : S \to Bag@0$;
$\{\} : Bag@0$;
$count : S \times Bag@0 \to \mathbb{N}$;
$\in: S \times Bag@0 \to \mathbb{B}$;
$<=, <: Bag@0 \times Bag@0 \to \mathbb{B}$;
$+, -, * : Bag@0 \times Bag@0 \to Bag@0$;
$Bag2Set : Bag@0 \to Set@0$;
$Set2Bag : Set@0 \to Bag@0$;
var $d : S$;
$f, g : S \to \mathbb{N}$;
$s, t : Bag@0$;
$u : Set@0$;
eqn $\{\} = \{x{:}S | @c0\}$;
$count(d, \{x{:}S | f(x)\}) = f(d)$;
$d \in s = count(d, s) > @c0$;
$\{x{:}S | f(x)\} \subseteq \{x{:}S | g(x)\} = \forall x{:}S.f(x) \leq g(x)$;
$s \subset t = s \subseteq t \wedge s \not\approx t$;
$\{x{:}S | f(x)\} \cup \{x{:}S | g(x)\} = \{x{:}S | f(x) + g(x)\}$;
$\{x{:}S | f(x)\} - \{x{:}S | g(x)\} = \{x{:}S | if(f(x) > g(x), @gtesubt(f(x), g(x)), @c0)\}$;
$\{x{:}S | f(x)\} \cap \{x{:}S | g(x)\} = \{x{:}S | \min(f(x), g(x))\}$;
$Bag2Set(s) = \{x{:}S | x \in s\}$;
$Set2Bag(u) = \{x{:}S | if(x \in u, @cNat(@c1), @c0)\}$;

A.9 Structured types

The general form of a structured type is the following, where $n \in \mathbb{N}^+$ and $k_i \in \mathbb{N}$ with $1 \leq i \leq n$:

$$\begin{array}{l}\textbf{struct}\ c_1(pr_{1,1} : A_{1,1},\ \ldots, pr_{1,k_1} : A_{1,k_1})?isC_1\\ \quad\ \mid c_2(pr_{2,1} : A_{2,1},\ \ldots, pr_{2,k_2} : A_{2,k_2})?isC_2\\ \qquad\ \vdots\\ \quad\ \mid c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?isC_n;\end{array}$$

A declaration of this form gives rise to the following equations, which are given schematically. If projection functions and recognizers are not mentioned explicitly, they are generated using internal names, together with the associated equations. The sortname *Struct@0* is an arbitrary internal name used to denote this sort.

sort $Struct@0$;

cons $c_i : A_{i,1} \times \cdots \times A_{i,k_i} \to Struct@0$ for all $1 \leq i \leq n$;

map $isC_i : Struct@0 \to \mathbb{B}$ for all $1 \leq i \leq n$;
$pr_{i,j} : Struct@0 \to A_{i,j}$ for all $1 \leq i \leq n,\ 1 \leq j \leq k_i$;

var $x_{i,j}, y_{i,j} : A_{i,j}$ for all $1 \leq i \leq n,\ 1 \leq j \leq k_i$;

eqn $isC_i(c_i(x_{i,1}, \ldots, x_{i,k_i})) = true$ for all $1 \leq i \leq n$;
$isC_i(c_j(x_{j,1}, \ldots, x_{j,k_j})) = false$ for all $1 \leq i, j \leq n$ and $i \neq j$;
$pr_{i,j}(c_i(x_{i,1}, \ldots, x_{i,k_i})) = x_{i,j}$ for all $1 \leq i \leq n$ and $1 \leq j \leq k_i$;
$c_i(x_{i,1}, \ldots, x_{i,k_i}) \approx c_i(y_{i,1}, \ldots, y_{i,k_i}) = \bigwedge_{1 \leq j \leq k_i} x_{i,j} \approx y_{i,j}$ for all $1 \leq i \leq n$;
$c_i(x_{i,1}, \ldots, x_{i,k_i}) \approx c_j(y_{j,1}, \ldots, y_{j,k_j}) = false$ for all $1 \leq i, j \leq n$ and $i \neq j$;

Note that the projection function $pr_{i,j}$ applied to another constructor than c_i has no equation, and therefore it cannot be determined to which element of the domain $A_{i,j}$ it is equal to.

Appendix B

Syntax of the formalisms

This appendix contains the EBNF description for the syntax of the data, processes, modal formulas and parameterised boolean equation systems. In the text spaces and newlines are ignored, except for comments which start with the symbol $\%$ and and with a newline.

B.1 Lexical part

The following list contains all predefined keywords. These keywords cannot be used as identifiers. The keywords are case sensitive.

```
sort | cons | map | var | eqn | act | proc | init
delta | tau | sum | block | allow | hide | rename | comm
struct | Bool | Pos | Nat | Int | Real | List | Set | Bag
true | false | whr | end | lambda | forall | exists | div | mod | in
```

B.2 Conventions to denote the context free syntax

In the syntax below we use the normal BNF conventions. The syntactic categories are indicated using italics, auxiliary symbols are in roman and keywords and textual symbols occur in teletype font. A line of the form A ::= B means that the syntactic category A consist of sequences as indicated in B. The following operators are allowed in B. A question mark indicates that the preceding symbol or category is optional, a $*$ indicates a sequence of 0 or more of the preceding objects and a $+$ indicates 1 or more occurrences. The bar (|) between two syntactical objects means that either of them is meant. Using brackets syntactical objects can be grouped. The notation [a−z] indicates the range of letters from a to z. Similarly for [A − Z] and [0 − 9].

B.3 Identifiers

Identifiers start with a letter and can consist of letters, numbers, underscores and primes. Numbers have their normal syntax, with an optional − in front.

Number ::= 0 | -?[1−9][0−9]∗
Id ::= ([a−z] | [A−Z] | _) ([a−z] | [A−Z] | [0 − 9] | _ | ')∗
Ids ::= *Id*(, *Id*)∗

B.4 Sort expressions and sort declarations

SortExpr	::= `Bool` | `Pos` | `Nat` | `Int` | `Real`	predefined sort
	| `List(`*SortExpr*`)`	list sort
	| `Set(`*SortExpr*`)`	set sort
	| `Bag(`*SortExpr*`)`	bag sort
	| *Id*	sort reference
	| `(`*SortExpr*`)`	parenthesized sort expression
	| *Domain* `->` *SortExpr*	higher-order sort
Domain	::= *SortExpr*(`#`*SortExpr*)$*$	
SortSpec	::= `sort` *SortDecl*$+$	
SortDecl	::= *Ids* `;`	standard sort
	| *Ids*`=`*SortExpr*`;`	sort expression
	| *Ids*`=struct` *ConstrDecl*(`|` *ConstrDecl*)$*$`;`	structured sort
ConstrDecl	::= *Id*(`(` *ProjDecls* `)`)?(`?`*Id*)?	
ProjDecl	::= (*Id* `:`)?*Domain*	
ProjDecls	::= *ProjDecl*(`,` *ProjDecl*)$*$	

B.5 Declaration of constructors and mappings

IdDecl	::= *Id* `:` *SortExpr*
IdsDecl	::= *Ids* `:` *SortExpr* `;`
OpSpec	::= (`cons` | `map`) *OpDecl*$+$
OpDecl	::= *IdsDecl* `;`

B.6 Declaration of equations

EqnSpec	::= `eqn` *EqnDecl*$+$	
	| `var` *IdsDecl*$+$`eqn` *EqnDecl*$+$	
EqnDecl	::= *DataExpr*`=`*DataExpr*`;`	
	| *DataExpr* `->` *DataExpr*`=`*DataExpr*`;`	

B.7 Data expressions

DataExpr	::=	*Id*	identifier
			number
		`true`	true
		`false`	false
		`[]`	empty list
		{}	empty set/bag
		`[`*DataExprs*`]`	list enumeration
		{*DataExprs*}	set enumeration
		{*BagEnumElts*}	bag enumeration
		{*IdDecl* `|` *DataExpr*}	set/bag comprehension
		`(`*DataExpr*`)`	parenthesized data expression
		DataExpr`(`*DataExprs*`)`	function application
		DataExpr`[`*DataExprs*`->`*DataExpr*`]`	function update
		`!`*DataExpr*	logical negation, set complement
		`-`*DataExpr*	arithmetic negation
		`#`*DataExpr*	list size
		`forall` *IdDecl*`.` *DataExpr*	universal quantification
		`exists` *IdDecl*`.` *DataExpr*	existential quantification
		DataExpr`.`*DataExpr*	list element at position
		DataExpr`*`*DataExpr*	multiplication, set intersection (associative)
		DataExpr `div` *DataExpr*	integer div
		DataExpr `mod` *DataExpr*	integer mod
		DataExpr `+` *DataExpr*	addition, set union (associative)
		DataExpr `-` *DataExpr*	subtraction, set difference
		DataExpr `<` *DataExpr*	less than, proper subset/subbag
		DataExpr `>` *DataExpr*	greater than
		DataExpr `<=` *DataExpr*	less than or equal, subset/subbag
		DataExpr `>=` *DataExpr*	greater than or equal
		DataExpr `in` *DataExpr*	element test
		DataExpr `|>` *DataExpr*	list cons (right associative)
		DataExpr `<|` *DataExpr*	list snoc (left associative)
		DataExpr `++` *DataExpr*	list concatenation (associative)
		DataExpr `==` *DataExpr*	equality (associative)
		DataExpr `!=`*DataExpr*	disequality (associative)
		DataExpr `&&` *DataExpr*	conjunction (associative)
		DataExpr `||` *DataExpr*	disjunction (associative)
		DataExpr `=>` *DataExpr*	implication
		`lambda` *IdDecl*`.` *DataExpr*	lambda abstraction
		DataExpr `whr` *DataExprs* `end`	where clause

DataExprs	::=	*DataExpr*(`,` *DataExpr*)$*$
BagEnumElt	::=	*DataExpr*`:`*DataExpr*
BagEnumElts	::=	*BagEnumElt*(`,` *BagEnumElt*)$*$

B.8 Communication and renaming sets

MAId ::= $Id(\texttt{|}\,Id)*$
MAIdSet ::= $\{(MAId(\texttt{,}\,MAId)*)?\}$
CommExpr ::= $MAId(\texttt{->}Id)?$
CommExprSet ::= $\{(CommExpr(\texttt{,}\,CommExpr)*)?\}$
RenExpr ::= $Id\texttt{->}Id$
RenExprSet ::= $\{(RenExpr(\texttt{,}\,RenExpr)*)?\}$

B.9 Process expressions

ProcExpr	::= Id	action of process reference with 0...
	| $Id\texttt{(}DataExprs\texttt{)}$	and 1 or more arguments
	| `delta`	deadlock
	| `tau`	internal action
	| $\texttt{sum(}IdDecl\texttt{,}\,ProcExpr\texttt{)}$	summation
	| $\texttt{block(}MAIdSet\texttt{,}\,ProcExpr\texttt{)}$	blocking AKA encapsulation
	| $\texttt{allow(}MAIdSet\texttt{,}\,ProcExpr\texttt{)}$	allow AKA nabla
	| $\texttt{hide(}MAIdSet\texttt{,}\,ProcExpr\texttt{)}$	hiding
	| $\texttt{rename(}RenExprSet\texttt{,}\,ProcExpr\texttt{)}$	renaming
	| $\texttt{comm(}CommExprSet\texttt{,}\,ProcExpr\texttt{)}$	communication
	| $\texttt{(}ProcExpr\texttt{)}$	parenthesized process expression
	| $ProcExpr\ \texttt{|}\ ProcExpr$	synchronisation (associative)
	| $ProcExpr\ \texttt{@}\ DataExpr$	timed expression
	| $ProcExpr\ \texttt{.}\ ProcExpr$	sequential (associative)
	| $DataExpr\ \texttt{->}\ ProcExpr$	conditional, or if-then
	| $DataExpr\ \texttt{->}\ ProcExpr\ \texttt{<>}\ ProcExpr$	if-then-else
	| $ProcExpr\ \texttt{<<}\ ProcExpr$	bounded initialisation (left associative)
	| $ProcExpr\ \texttt{||}\ ProcExpr$	parallel (associative)
	| $ProcExpr\ \texttt{||\_}\ ProcExpr$	left merge (left associative)
	| $ProcExpr\ \texttt{+}\ ProcExpr$	choice (associative)

B.10 Action declaration

ActDecl ::= $Ids(\texttt{:}\,Domain)?\texttt{;}$
ActSpec ::= $\texttt{act}\ ActDecl+$

B.11 Process and initial state declaration

ProcDecl ::= $Id(\texttt{(}\,IdsDecl(\texttt{,}\,IdsDecl)*\texttt{)})?\texttt{=}ProcExpr\texttt{;}$
ProcSpec ::= $\texttt{proc}\ ProcDecl+$
Init ::= $\texttt{init}\ ProcExpr\texttt{;}$

B.12 Syntax of an mCRL2 specification

mCRL2Spec ::= $(SortSpec \mid OpSpec \mid EqnSpec \mid ActSpec \mid ProcSpec \mid Init)+$

Appendix C

Axioms for processes

In this appendix all process axioms are grouped together.

MA1	$\alpha \mid \beta = \beta \mid \alpha$
MA2	$(\alpha \mid \beta) \mid \gamma = \alpha \mid (\beta \mid \gamma)$
MA3	$\alpha \mid \tau = \alpha$
MD1	$\tau \setminus \alpha = \tau$
MD2	$\alpha \setminus \tau = \alpha$
MD3	$\alpha \setminus (\beta \mid \gamma) = (\alpha \setminus \beta) \setminus \gamma$
MD4	$(a(d) \mid \alpha) \setminus a(d) = \alpha$
MD5	$(a(d) \mid \alpha) \setminus b(e) = a(d) \mid (\alpha \setminus b(e))$ if $a \not\equiv b$ or $d \not\approx e$
MS1	$\tau \sqsubseteq \alpha = \mathit{true}$
MS2	$a \sqsubseteq \tau = \mathit{false}$
MS3	$a(d) \mid \alpha \sqsubseteq a(d) \mid \beta = \alpha \sqsubseteq \beta$
MS4	$a(d) \mid \alpha \sqsubseteq b(e) \mid \beta = a(d) \mid (\alpha \setminus b(e)) \sqsubseteq \beta$ if $a \not\equiv b$ or $d \not\approx e$
MAN1	$\underline{\tau} = \tau$
MAN2	$\underline{a(d)} = a$
MAN3	$\underline{\alpha \mid \beta} = \underline{\alpha} \mid \underline{\beta}$
A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
A6	$\alpha + \delta = \alpha$
A7	$\delta \cdot x = \delta$
Cond1	$\mathit{true} \rightarrow x \diamond y = x$
Cond2	$\mathit{false} \rightarrow x \diamond y = y$
SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = X(e) + \sum_{d:D} X(d)$
SUM4	$\sum_{d:D} (X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
SUM5	$(\sum_{d:D} X(d)) \cdot y = \sum_{d:D} X(d) \cdot y$

M	$x \parallel y = x \between y + y \between x + x \mid y$
LM1	$\alpha \between x = (\alpha \ll x) \cdot x$
LM2	$\delta \between x = \delta \ll x$
LM3	$\alpha \cdot x \between y = (\alpha \ll y) \cdot (x \parallel y)$
LM4	$(x + y) \between z = x \between z + y \between z$
LM5	$(\sum_{d:D} X(d)) \between y = \sum_{d:D} X(d) \between y$
LM6	$x \triangleleft t \between y = (x \between y) \triangleleft t$
S1	$x \mid y = y \mid x$
S2	$(x \mid y) \mid z = x \mid (y \mid z)$
S3	$x \mid \tau = x$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot x) \mid \beta = \alpha \mid \beta \cdot x$
S6	$(\alpha \cdot x) \mid (\beta \cdot y) = \alpha \mid \beta \cdot (x \parallel y)$
S7	$(x + y) \mid z = x \mid z + y \mid z$
S8	$(\sum_{d:D} X(d)) \mid y = \sum_{d:D} X(d) \mid y$
S9	$x \triangleleft t \mid y = (x \mid y) \triangleleft t$
TB1	$x \ll \alpha = x$
TB2	$x \ll \delta = x$
TB3	$x \ll y \triangleleft t = \sum_{u:\mathbb{R}} u < t \to (x \triangleleft u) \ll y$
TB4	$x \ll (y + z) = x \ll y + x \ll z$
TB5	$x \ll y \cdot z = x \ll y$
TB6	$x \ll \sum_{d:D} Y(d) = \sum_{d:D} x \ll Y(d)$
TC1	$(x \between y) \between z = x \between (y \parallel z)$
TC2	$x \between \delta = x \cdot \delta$
TC3	$(x \mid y) \between z = x \mid (y \between z)$
R1	$\rho_R(\tau) = \tau$
R2	$\rho_R(a(d)) = b(d)$ if $a \to b \in R$ for some b
R3	$\rho_R(a(d)) = a(d)$ if $a \to b \notin R$ for all b
R4	$\rho_R(\alpha \mid \beta) = \rho_R(\alpha) \mid \rho_R(\beta)$
R5	$\rho_R(\delta) = \delta$
R6	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
R7	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$
R8	$\rho_R(\sum_{d:D} X(d)) = \sum_{d:D} \rho_R(X(d))$
R9	$\rho_R(x \triangleleft t) = \rho_R(x) \triangleleft t$
T1	$x + \delta \triangleleft 0 = x$
T2	$c \to x = c \to x \diamond \delta \triangleleft 0$
T3	$x = \sum_{t:\mathbb{R}} x \triangleleft t$
T4	$x \triangleleft t \cdot y = x \triangleleft t \cdot (t \gg y)$
T5	$x \triangleleft t = t > 0 \to x \triangleleft t$
TA1	$\alpha \triangleleft t \triangleleft u = (t \approx u) \to \alpha \triangleleft t \diamond \delta \triangleleft \min(t, u)$
TA2	$\delta \triangleleft t \triangleleft u = \delta \triangleleft \min(t, u)$
TA3	$(x + y) \triangleleft t = x \triangleleft t + y \triangleleft t$
TA4	$(x \cdot y) \triangleleft t = x \triangleleft t \cdot y$
TA5	$(\sum_{d:D} X(d)) \triangleleft t = \sum_{d:D} X(d) \triangleleft t$
TI1	$t \gg \alpha \triangleleft u = t < u \to \alpha \triangleleft u \diamond \delta \triangleleft t$
TI2	$t \gg \delta \triangleleft u = \delta \triangleleft \max(t, u)$
TI3	$t \gg (x + y) = t \gg x + t \gg y$
TI4	$t \gg (x \cdot y) = (t \gg x) \cdot y$
TI5	$t \gg \sum_{d:D} X(d) = \sum_{d:D} t \gg X(d)$

C1	$\Gamma_C(\alpha) = \gamma_C(\alpha)$	C4	$\Gamma_C(x{\cdot}y) = \Gamma_C(x){\cdot}\Gamma_C(y)$
C2	$\Gamma_C(\delta) = \delta$	C5	$\Gamma_C(\sum_{d:D} X(d)) = \sum_{d:D} \Gamma_C(X(d))$
C3	$\Gamma_C(x + y) = \Gamma_C(x) + \Gamma_C(y)$	C6	$\Gamma_C(x \lhd t) = \Gamma_C(x) \lhd t$
V1	$\nabla_V(\alpha) = \alpha$ if $\underline{\alpha} \in V \cup \{\tau\}$	V4	$\nabla_V(x + y) = \nabla_V(x) + \nabla_V(y)$
V2	$\nabla_V(\alpha) = \delta$ if $\underline{\alpha} \notin V \cup \{\tau\}$	V5	$\nabla_V(x{\cdot}y) = \nabla_V(x){\cdot}\nabla_V(y)$
V3	$\nabla_V(\delta) = \delta$	V6	$\nabla_V(\sum_{d:D} X(d)) = \sum_{d:D} \nabla_V(X(d))$
		V7	$\nabla_V(x \lhd t) = \nabla_V(x) \lhd t$
TV1	$\nabla_V(\nabla_W(x)) = \nabla_{V \cap W}(x)$		
E1	$\partial_B(\tau) = \tau$	E5	$\partial_B(\delta) = \delta$
E2	$\partial_B(a(d)) = a(d)$ if $a \notin B$	E6	$\partial_B(x + y) = \partial_B(x) + \partial_B(y)$
E3	$\partial_B(a(d)) = \delta$ if $a \in B$	E7	$\partial_B(x{\cdot}y) = \partial_B(x){\cdot}\partial_B(y)$
E4	$\partial_B(\alpha \mid \beta) = \partial_B(\alpha) \mid \partial_B(\beta)$	E8	$\partial_B(\sum_{d:D} X(d)) = \sum_{d:D} \partial_B(X(d))$
		E9	$\partial_B(x \lhd t) = \partial_B(x) \lhd t$
H1	$\tau_I(\tau) = \tau$	H5	$\tau_I(\delta) = \delta$
H2	$\tau_I(a(d)) = \tau$ if $a \in I$	H6	$\tau_I(x{+}y) = \tau_I(x) + \tau_I(y)$
H3	$\tau_I(a(d)) = a(d)$ if $a \notin I$	H7	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$
H4	$\tau_I(\alpha \mid \beta) = \tau_I(\alpha) \mid \tau_I(\beta)$	H8	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$
		H9	$\tau_I(x \lhd t) = \tau_I(x) \lhd t$
U1	$\Upsilon_U(\tau) = \tau$	U5	$\Upsilon_U(\delta) = \delta$
U2	$\Upsilon_U(a(d)) = int$ if $a \in U$	U6	$\Upsilon_U(x{+}y) = \Upsilon_U(x) + \Upsilon_U(y)$
U3	$\Upsilon_U(a(d)) = a(d)$ if $a \notin U$	U7	$\Upsilon_U(x{\cdot}y) = \Upsilon_U(x){\cdot}\Upsilon_U(y)$
U4	$\Upsilon_U(\alpha \mid \beta) = \Upsilon_U(\alpha) \mid \Upsilon_U(\beta)$	U8	$\Upsilon_U(\sum_{d:D} X(d)) = \sum_{d:D} \Upsilon_U(X(d))$
		U9	$\Upsilon_U(x \lhd t) = \Upsilon_U(x) \lhd t$

B1	$x{\cdot}\tau = x$
B2	$x{\cdot}(\tau{\cdot}(y + z) + y) = x{\cdot}(y + z)$
W1	$x{\cdot}\tau = x$
W2	$\tau{\cdot}x = \tau{\cdot}x + x$
W2	$a{\cdot}(\tau{\cdot}x + y) = a{\cdot}(\tau{\cdot}x + y) + a{\cdot}x$

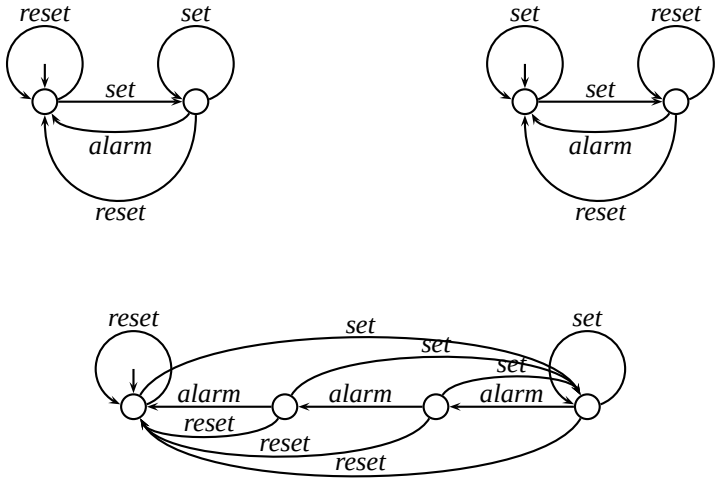
Failures equivalence	$a{\cdot}(b{\cdot}x + u) + a{\cdot}(b{\cdot}y + v) = a{\cdot}(b{\cdot}x + b{\cdot}y + u) + a{\cdot}(b{\cdot}x + b{\cdot}y + v)$ $a{\cdot}x + a{\cdot}(y + z) = a{\cdot}x + a{\cdot}(x + y) + a{\cdot}(y + z)$
Trace equivalence	$x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$
Weak trace equivalence	$x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$ $\tau{\cdot}x = x$ $x{\cdot}\tau = x$

Appendix D

Answers to exercises

2.1.1 It is not possible to give a precise answer to this question, because the relevant interactions depend very much on a concrete system. A possible answer is the following. CD player: *play*, *stop*, *pause*, *Backward*, *Forward*. Text editor: *insertLetter*, *deleteLetter*, *moreCursorRight*, *moveCursorLeft*, *moveCursorDown*, *saveFile*, *openFile*. Data transfer channel: *sendMessage*, *receiveMessage*, *sendUrgentMessage*, *resetChannel*.

2.2.2



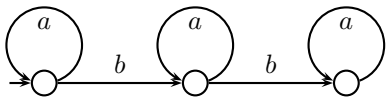
2.2.3 $(\{s_0, s_1\}, \{set, reset, alarm\}, \rightarrow, s_0, \emptyset)$ where

$$\longrightarrow = \{(s_0, set, s_1), (s_1, alarm, s_1), (s_1, alarm, s_0), (s_1, reset, s_0)\}.$$

2.3.2 (1) no. (2) yes. (3) no.

2.3.3 A sequence of a steps can unboundedly be extended in the transition systems to the right if the loop is chosen. None of the finite sequences at the left can mimic that.

2.3.4



2.3.6 They are all trace equivalent.

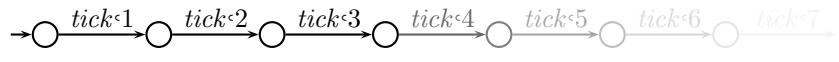
2.3.9 (1) Neither failures nor language equivalent. (2) Language but not failures equivalent. (3) Both failures and language equivalent.

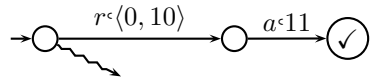
2.5.5 (1) branching bisimilar. Not rooted branching bisimilar. (2) rooted branching bisimilar and hence branching bisimilar. (3) Neither rooted nor branching bisimilar.

2.5.6 All τ transitions in the four leftmost transition systems are inert. None of the τ's in the two transition systems to the right are inert.

2.5.9 All pairs are weakly bisimilar. The first pair is not rooted weakly bisimilar. All τ-transitions are inert with respect to weak bisimulation.

3.1.1 (a) proper timed lts, no time deadlock. (b) proper, two time deadlocks. (c) improper. (d) improper.

3.1.2 

3.1.3 

3.2.2 $A_1 = (\{s_1\}, \emptyset, \emptyset, \rightsquigarrow^1, s_1, \emptyset)$ and $A_2 = (\{s_2\}, \emptyset, \emptyset, \rightsquigarrow^2, s_2, \emptyset)$ where $s_1 \rightsquigarrow^1_t$ holds iff $t < 2$ and $s_2 \rightsquigarrow^2_t$ holds iff $t \leq 2$. The second transition system can be drawn using a single $\rightsquigarrow_2$ transition from the initial state. The first one is hard to draw, because there is no largest real number, smaller than 2.

4.1.1 (1) τ. (2) $a(1)|b(2)$. (3) false.

4.1.3 $((a+a)\cdot(b+b))\cdot(c+c) \stackrel{\text{A3}}{=} (a\cdot b)\cdot c \stackrel{A5}{=} a\cdot(b\cdot c)$;

(2) $(a+a)\cdot(b\cdot c)+(a\cdot b)\cdot(c+c) \stackrel{\text{A3}}{=} a\cdot(b\cdot c)+(a\cdot b)\cdot(c+c) \stackrel{\text{A5}}{=} (a\cdot b)\cdot c+(a\cdot b)\cdot(c+c) \stackrel{\text{A3}}{=} (a\cdot b)\cdot(c+c)+(a\cdot b)\cdot(c+c) \stackrel{\text{A3}}{=} (a\cdot b)\cdot(c+c) \stackrel{\text{A3}}{=} (a\cdot(b+b))\cdot(c+c)$.

4.1.4 Suppose $x \subseteq y$ and $y \subseteq x$. By definition we have (1) $x+y=y$ and (2) $y+x=x$. Thus we obtain: $x \stackrel{(2)}{=} y+x \stackrel{\text{A1}}{=} x+y \stackrel{(1)}{=} y$.

4.1.5 (1) $\mathit{true} \to x \diamond y = x = \neg \mathit{true} \to y \diamond x$.
$\mathit{false} \to x \diamond y = y = \neg \mathit{false} \to y \diamond x$.

(2) $\mathit{true} \vee c' \to x \diamond y = \mathit{true} \to x \diamond y = x = \mathit{true} \to x \diamond (c' \to x \diamond y)$.
$\mathit{false} \vee c' \to x \diamond y = c' \to x \diamond y = \mathit{false} \to x \diamond (c' \to x \diamond y)$.

(3) $x+y = x+y+c \to x+\neg c \to y = x+y+c \to x \diamond y$.

(4) if $b = \mathit{true}$, then by the assumption $(b = \mathit{true} \Rightarrow x = y)$ we have $x = y$ and so $b \to x \diamond z = b \to y \diamond z$.
if $b = \mathit{false}$, then $b \to x \diamond z = z = b \to y \diamond z$.

4.1.6 $\sum_{n:\mathbb{N}} \mathit{read}(n)\cdot((n < 100) \to \mathit{forward}(n) \diamond \mathit{overflow})$.

4.1.7 $X = \sum_{m:\mathit{Message}} \mathit{read}(m)\cdot\mathit{forward}(m)\cdot X$.
$X = \sum_{m:\mathit{Message}} \mathit{read}(m)\cdot(\mathit{empty} + \mathit{forward}(m))\cdot X$.

4.2.1 **map** $\geq: \mathit{Nat} \times \mathit{Nat} \to \mathbb{B}$;
$<: \mathit{Nat} \times \mathit{Nat} \to \mathbb{B}$;
$>: \mathit{Nat} \times \mathit{Nat} \to \mathbb{B}$;

var $n, m{:}Nat$;
eqn $n \geq zero = true$;
$zero \geq successor(n) = false$;
$successor(n) \geq successor(m) = n \geq m$;
$zero < successor(n) = true$;
$n < zero = false$;
$successor(n) < successor(m) = n < m$;
$successor(n) > zero = true$;
$zero > n = false$;
$successor(n) > successor(m) = n > m$;

4.2.2 **map** $minus, \max, power : Nat \times Nat \to Nat$;
var $m, n{:}Nat$;

eqn $power(successor(m), zero) = successor(zero)$;
$power(m, successor(n)) = times(m, power(m, n))$;
$\max(m, zero) = m$;
$\max(zero, m) = m$;
$\max(successor(n), successor(m)) = successor(\max(n, m))$;
$minus(m, zero) = m$;
$minus(zero, m) = zero$;
$minus(successor(m), successor(n)) = minus(m, n)$;

4.2.3 **sort** $\mathbb{B}, Nat, List, D$;
cons $[] :\to List$;
$in : D \times List \to List$;
map $append : D \times List \to List$;
$top, toe : List \to D$;
$tail, untoe : List \to List$;
$nonempty : List \to \mathbb{B}$;
$length : List \to Nat$;
$++ : List \times List \to List$;

var $d, e{:}D, q, q'{:}List$;
eqn $append(d, []) = in(d, [])$;
$append(d, in(e, q)) = in(e, append(d, q))$;
$top(in(d, q)) = d$;
$toe(in(d, [])) = d$;
$toe(in(d, in(e, q))) = toe(in(e, q))$;
$tail(in(d, q)) = q$;
$untoe(in(d, [])) = []$;
$untoe(in(d, in(e, q))) = in(d, untoe(in(e, q)))$;
$nonempty([]) = false$;
$nonempty(in(d, q)) = true$;
$length([]) = 0$;
$length(in(d, q)) = successor(length(q))$;
$++([], q) = q$;
$++(in(d, q), q') = in(d, ++(q, q'))$;

4.2.5 $succ : \mathbb{N}^+ \to \mathbb{N}^+$, $succ : \mathbb{N} \to \mathbb{N}^+$, $succ : \mathbb{Z} \to \mathbb{Z}$, $succ : \mathbb{R} \to \mathbb{R}$. $pred : \mathbb{N}^+ \to \mathbb{N}$, $pred : \mathbb{N} \to \mathbb{Z}$, $pred : \mathbb{Z} \to \mathbb{Z}$, $pred : \mathbb{R} \to \mathbb{R}$.

4.2.6 0 is represented by $@c0$ and 1 by $@cNat(1)$. Suppose $@c0 = @cNat(1)$. Then $true = @c0 \leq @c0 = @cNat(1) \leq @c0 = false$. Contradiction.

4.2.8

map $stretch : List(List(S)) \to List(S)$;
var $l : List(S)$;
$L : List(List(S))$;
eqn $stretch([]) = []$;
$stretch(l \triangleright L) = l ++ stretch(L)$;

4.2.9 **map** $insert : S \times List(S) \to List(S)$;
var $s, s' : S$;
$L : List(S)$;
eqn $insert(s, []) = [s]$;
$insert(s, s' \triangleright L) = if(s \approx s', s \triangleright L, s' \triangleright insert(s, L))$;

The rule above will not terminate with innermost rewriters. To have guaranteed termination, conditional rules are necessary:

var $s, s' : S$;
$L : List(S)$;
eqn $insert(s, []) = [s]$;
$insert(s, s \triangleright L) = s \triangleright L$;
$s \not\approx s' \to insert(s, s' \triangleright L) = s' \triangleright insert(s, L))$;

Prove that $insert(s, L)$ will never contains two equal elements assuming that L contains only unique elements. It then follows with induction on the number of $insert$ operations that created lists always contain only unique elements.

4.2.10 **map** $insert : \mathbb{N} \times List(\mathbb{N}) \to List(\mathbb{N})$;
$is\_in : \mathbb{N} \times List(\mathbb{N}) \to \mathbb{B}$;
var $n, n' : \mathbb{N}$;
$L : List(\mathbb{N})$;
eqn $insert(n, []) = [n]$;
$insert(n, n' \triangleright L) = if(n \leq n', n \triangleright n' \triangleright L, n' \triangleright insert(n, L))$;
$is\_in(n, []) = false$;
$is\_in(n, n' \triangleright L) = if(n < n', false, if(n \approx n', true, is\_in(n, L)))$;

In order to prove that the membership tests yield the same results, one must first relate the original and the ordered lists. A good candidate is the function $sort$ defined as follows:

map $sort : List(\mathbb{N}) \to List(\mathbb{N})$;
var $n : \mathbb{N}$;
$L : List(\mathbb{N})$;
eqn $sort([]) = []$;
$sort(n \triangleright L) = insert(n, sort(L))$;

Due to this definition, it holds that for an arbitrary list L obtained after a sequence of inserts using ($\triangleright$), we obtain the list $sort(L)$ after the same sequence of inserts using $insert$. The following property says that the element tests yield the same results:

$$n \in L \quad \text{iff} \quad is\_in(n, sort(L)).$$

This can be proven with induction on L, using the (to be proven) property that $sort(L)$ must be sorted.

4.2.11 $\{m{:}\mathbb{N}^+ \mid 1 < m \wedge \forall n{:}\mathbb{N}^+ . 2 \leq n < m \Rightarrow m|_n \not\approx 0\}$.

4.2.12 $\{l{:}List(\mathbb{N}) \mid \forall n{:}\mathbb{N}.(l.n) \approx 0\}$ and $\{l{:}List(\mathbb{N}) \mid \#l \approx 2\}$.

4.2.13 **eqn** $map(f, []) = []$;
$map(f, n \triangleright L) = f(n) \triangleright map(f, L)$;

4.2.14

eqn $multiplicity(i,a,n) = if(n{\approx}0, 0, if(a(n{-}1){\approx}i, 1, 0) + multiplicity(i,a,n{-}1));$
$equaltcontents(a,a',n) = \forall i{:}\mathbb{N}.multiplicity(i,a,n) = multiplicity(i,a',n);$

4.2.15 **sort** $CheckSumType, Data;$
$MessageType = \mathbf{struct}\ ack \mid ctrl \mid mes;$
$Message = \mathbf{struct}\ frame(MessageType, CheckSumType, Data) \mid$
$\qquad frame(MessageType, Checksum);$

4.3.1 **proc** $DriftingClock(t{:}\mathbb{R}) = \sum_{u:\mathbb{R}}(t{+}1{-}\epsilon \le u \le t{+}1{+}\epsilon) \to tick{\triangleleft}u{\cdot}DriftingClock(t{+}1);$

4.3.2 **proc** $CM = \sum_{t:\mathbb{R}} coin{\triangleleft}t{\cdot}$
$\sum_{u:\mathbb{R}}((u \le t{+}10)$
$\to (selectCoffee{\triangleleft}u \sum_{v:\mathbb{R}}(v \le u{+}5) \to coffee{\triangleleft}v{+}$
$selectTea{\triangleleft}u \sum_{v:\mathbb{R}}(v \le u{+}5) \to tea{\triangleleft}v)$
$\diamond\ returnCoin{\triangleleft}u){\cdot}CM;$

4.3.3
1. $x = x + \delta{\triangleleft}0 = x + x + y = x + y = \delta{\triangleleft}0.$
2. Induction on b.
3. $a \overset{\text{A6}}{=} a + \delta = a + \sum_{u:\mathbb{R}} \delta{\triangleleft}t \overset{\text{SUM3}}{=} a + \sum_{u:\mathbb{R}} \delta{\triangleleft}u + \delta{\triangleleft}t = a + \delta + \delta{\triangleleft}t = a + \delta{\triangleleft}t.$

4.3.4 $x{\triangleleft}t{\cdot}a{\triangleleft}t \overset{\text{T4}}{=} x{\triangleleft}t{\cdot}(t \gg a{\triangleleft}t) \overset{\text{TI1}}{=} x{\triangleleft}t{\cdot}(t{<}t \to a{\triangleleft}t \diamond \delta{\triangleleft}t) = x{\triangleleft}t{\cdot}(false \to a{\triangleleft}t \diamond \delta{\triangleleft}t) \overset{\text{Cond2}}{=} x{\triangleleft}t{\cdot}\delta{\triangleleft}t.$

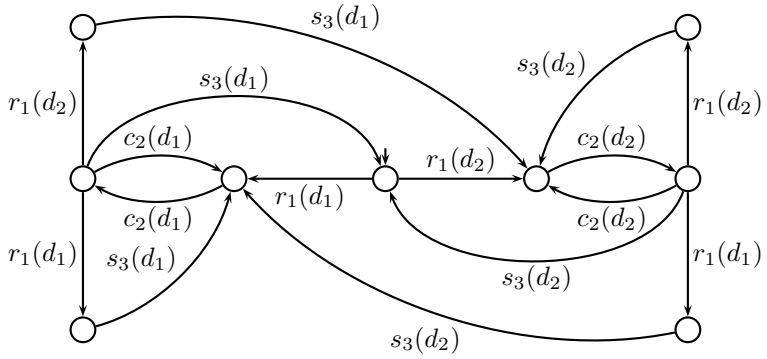
4.4.1 $a{\cdot}(b{\cdot}c{+}c{\cdot}b + b|c) + c{\cdot}a{\cdot}b + (a|c){\cdot}b.$

4.4.2 $a{\triangleleft}1{\cdot}c{\triangleleft}2{\cdot}b{\triangleleft}3.$

4.4.3 $x \parallel y = x \between y + y \between x + x|y = y \parallel x.$
$x \parallel (y \parallel z) =$
$x \between (y \parallel z) + (y \parallel z) \between x + x|(y \parallel z) =$
$x \between (y \parallel z) + (y \between z) \between x + (z \between y) \between x + (y|z) \between x + x|(y \between z) + x|(z \between y) + x|(y|z) =$
$x \between (y \parallel z) + y \between (x \parallel z) + z \between (x \parallel y) + y|(z \between x) + x|(y \between z) + x|(z \between y) + x|(y|z).$
The expression $(x \parallel y) \parallel z$ can similarly be expanded.

4.4.4 Induction on $\mathbb{B}$. If $c = false$, the equation reduces to $x \ll \delta{\triangleleft}0 = \delta{\triangleleft}0$ which is provable using TB2, TB3 and T5.

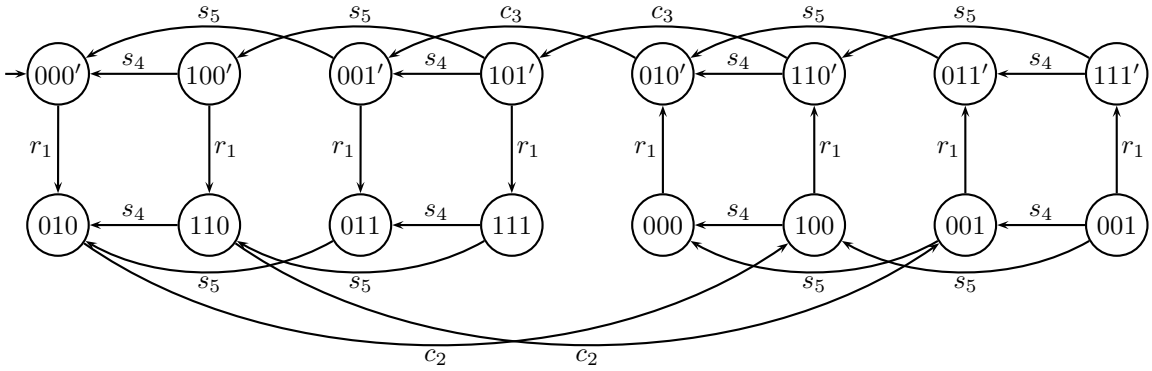
4.4.5 The node in the middle is the root node.



Data elements are read via channel 1 and sent via channel 3 in the same order.

The two shortest execution traces of $\nabla_{\{r_1,c_2\}}(S)$ to a deadlock state are $r_1(d_1)\, c_2(d_1)\, r_1(d_1)$ and $r_1(d_1)\, c_2(d_1)\, r_1(d_2)$.

4.4.6 **act** $r_1, s_2, r_2, c_2, s_3, r_3, c_3, s_4, s_5{:}D;$
proc $X(b{:}\mathbb{B}) = \sum_{d:D} r_1(d){\cdot}(b \to s_2(d) \diamond s_3(d)){\cdot}X(\neg b);$
$Y = \sum_{d:D} r_2(d){\cdot}s_4(d){\cdot}Y;$
$Z = \sum_{d:D} r_3(d){\cdot}s_5(d){\cdot}Z;$
init $\nabla_{\{r_1,s_4,s_5,c_2,c_3\}}(\Gamma_{\{s_2|r_2\to c_2, s_3|r_3\to c_3\}}(X \parallel Y \parallel Z));$



The three bits in the states above denote whether there is a datum in the buffer of Y, X or Z, respectively. In a state with or without prime the next incoming datum is sent on via channel 2 or 3, respectively. The initial state is $000'$.

4.5.1

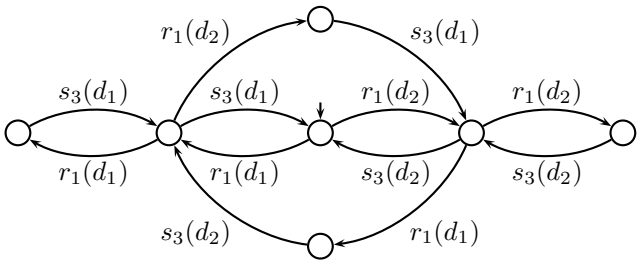
(1) $a(\tau b + b) \stackrel{\text{A3}}{=} a(\tau(b+b)+b) \stackrel{\text{B2}}{=} a(b+b) \stackrel{\text{A3}}{=} ab$.

(2) $a(\tau(b+c)+b) \stackrel{\text{B2}}{=} a(b+c) \stackrel{\text{A1}}{=} a(c+b) \stackrel{\text{B2}}{=} a(\tau(c+b)+c) \stackrel{\text{A1}}{=} a(\tau(b+c)+c)$.

(3) $\tau_{\{a\}}(a(a(b+c)+b)) \stackrel{\text{TI1-4}}{=} \tau(\tau(b+c)+b) = \tau(\tau(b+c)+c) \stackrel{\text{TI1-4}}{=} \tau_{\{d\}}(d(d(b+c)+c))$.

(4) If $x+y=x$, then $\tau(\tau x + y) = \tau(\tau(x+y)+y) \stackrel{\text{B2}}{=} \tau(x+y) = \tau x$.

4.5.2



The reduced state space is deterministic and contains no τ's. Therefore reducing it further using weak bisimulation or weak trace equivalence does not make sense.

4.6.4 $e{\cdot}f + f{\cdot}e$.

5.1.1 1. $\langle a\rangle(\langle b\rangle \mathit{true} \wedge \neg\langle c\rangle \mathit{true})$. 2. $[a]\neg(\langle b\rangle \mathit{true} \vee \langle c\rangle \mathit{true})$. 3. $[b]\mathit{false} \vee [a][b]\mathit{false}$.

5.1.2 Use the identities between modal formulas. The modalities containing c_1 and c_2 can be removed.

5.2.1 1. $[\overline{\mathit{error}}^\star]\langle \mathit{true}\rangle \mathit{true}$. 2. $[\mathit{true}^\star{\cdot}a]\langle \mathit{true}^\star{\cdot}(b \cup c)\rangle \mathit{true}$. 3. $[\mathit{true}^\star{\cdot}a{\cdot}\overline{b}^\star]\langle \mathit{true}^\star{\cdot}b\rangle \mathit{true}$.

5.2.2 Use the rules in table 5.1. $[R_1{\cdot}(R_2+R_3)]\phi = [R_1]([R_2]\phi \wedge [R_3]\phi) = [R_1][R_2]\phi \wedge [R_1][R_3]\phi = [R_1{\cdot}R_2 + R_1{\cdot}R_2]\phi$. The case with the diamond modalitities is similar.

5.3.1 The formula $\mu X.[a]X$ is invalid for an a-loop, whereas the formula $\nu X.[a]X$ is valid. As $\nu X.[a]X$ is equivalent to true, there is no transition system for which the minimal fixed point formula holds, and the maximal one does not.

5.3.2 $b{\cdot}(a + b{\cdot}X)$ where $X = b{\cdot}X$.

5.3.3 The process P with $P = \mathit{send}{\cdot}P + \mathit{receive}{\cdot}\delta$ makes the first formula valid, and not the second.

5.3.4 Each sequence consisting of only a and b actions ends in an infinite sequence of a's. Each sequence of a and b actions contains only finite subsequences of a's. The first formula is valid and the second is not valid for the process P defined by $P = a{\cdot}P$.

5.4.1 $\forall n{:}\mathbb{N}.[true^\star{\cdot}generate(n)true^\star{\cdot}generate(n)]false$.

5.4.2 $\forall f{:}\mathbb{N}\to\mathbb{N}.[true^\star{\cdot}deliver(f)]\forall n{:}\mathbb{N}f(n{+}1) > f(n)$.

5.4.3 $\forall p{:}Product.\nu X(n{:}\mathbb{N}{:=}0).[\overline{enter(p)\cup leave(p)}]X(n)\wedge[enter(p)]X(n{+}1)\wedge[leave(p)]X(n{-}1)\wedge Y(p,n)$.
$\mu Y(p{:}Product, n{:}\mathbb{N}).(n{\approx}0)\vee([\overline{enter(p)\cup leave(p)}]Y(p,n)\wedge[enter(p)]Y(p,n{+}1)\wedge[leave(p)]Y(p,n{-}1))$.

5.5.1 $\neg\nabla = \neg\exists t:\mathbb{R}.\nabla{<}t = \forall t:\mathbb{R}.\Delta{<}t = \Delta$.

5.5.2 $[true^\star]\Delta.\ \langle true^\star\rangle\nabla$.

5.5.3 $[true^\star]\forall p{:}Product, t{:}\mathbb{R}.[enter(p){<}t]\mu X.([\forall u{:}\mathbb{R}.\overline{u{\le}t+5\wedge leave(p){<}u}]X\wedge\langle true\rangle true)$
$\exists m{:}\mathbb{N}.\nu X(n{:}\mathbb{N}{:=}0, sum{:}\mathbb{R}{:=}0).\forall p{:}Product, t, u{:}\mathbb{R}.[\overline{enter(p)\cup leave(p)}^\star{\cdot}enter(p){<}t{\cdot}$
$\overline{enter(p)\cup leave(p)}^\star{\cdot}leave(p){<}u](X(n{+}1, sum{+}u{-}t)\wedge(n{>}m\Rightarrow\frac{sum}{n}{\le}3))$.

6.3.1 Properties 3, 5 and 6 are valid. The others are not valid.

6.3.2 According to requirement 2, the scanner must be manually movable out of the scanning room, meaning that no brake is applied, the motor is off and the platform is undocked. But this contradicts with the first requirement saying that when undocked, the brakes must be applied and the bed is in outermost position. Most likely it is needed to weaken requirement 2, and rethink the procedure in case of an emergency.

7.1.9 1. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$. 2. $Set(\mathbb{R})$. 3. $List(D)$ for any sort D. 4. Not well-typed.

7.1.11 1. yes. 2. no. 3. no. 4. yes.

7.1.13 1. $\lambda x{:}\mathbb{N}.\lambda y{:}\mathbb{N}.x{+}y{+}u$. 2. $\lambda z{:}\mathbb{N}.x$. 3. $\lambda u{:}\mathbb{N}.\lambda y{:}\mathbb{N}.u$. 4. $\lambda u{:}\mathbb{N}.\lambda y{:}\mathbb{N}.u$.

8.2.5 Alpha reduction yields $\lambda d{:}D.X(d) = \lambda e{:}D.X(e)$. Put Σ in front using congruence.

8.2.6 Assume $p_1 = q_1, \ldots, p_n = q_n$ are provable. We list these assumptions below as items 1 to n.

1.	$\Gamma \vdash p_1 = q_1$
	$\vdots$
n.	$\Gamma \vdash p_n = q_n$
$n{+}1$.	$\Gamma \vdash f = f$ reflexivity
$n{+}2$.	$\Gamma \vdash f\,p_1 = f\,q_1$ congruence using 1 and $n{+}1$
$n{+}3$.	$\Gamma \vdash f\,p_1\,p_2 = f\,q_1\,p_2$ congruence using 2 and $n{+}2$
	$\vdots$
$2n$.	$\Gamma \vdash f\,p_1\,p_2\ldots p_{n-1} = f\,q_1\,q_2\ldots q_{n-1}$ congruence using $n{-}1$ and $2n{-}1$
$2n{+}1$.	$\Gamma \vdash f\,p_1\,p_2\ldots p_{n-1}\,p_n = f\,q_1\,q_2\ldots q_{n-1}\,q_n$ congruence using n and $2n$

Note that $f\,p_1\,p_2\ldots p_{n-1}\,p_n$ and $f(p_1, p_2, \ldots, p_{n-1}, p_n)$ are alternative ways of writing the same object.

8.2.7 Write p for $\sum_{d:D}\sum_{e:E}X(d,e)$. Note that d and e do not occur freely in p. Assume that the context Γ contains at least SUM1. We derive

1. $\Gamma \vdash \sum_{d:D} x = x$ by start using that SUM1 is in Γ;
2. $\Gamma \vdash \sum_{d:D} p = p$ by substitution using 1;
3. $\Gamma \vdash \sum_{d:D}\sum_{d:D} p = \sum_{d:D} p$ by substitution using 1;
4. $\Gamma \vdash \sum_{d:D}\sum_{d:D} p = \sum_{d:D} p$ by alpha conversion, see text on page 117;

5. $\Gamma \vdash \sum_{e:D} \sum_{d:D} p = 0$ by transitivity using 2 and 4.

8.4.1
1. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash true = true \Rightarrow x = y$ start (generalised);
2. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash true = true$ reflexivity;
3. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash x = y$ modus ponens using 1 and 2;
4. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash \lambda z{:}\mathbb{P}.true{\to}z = \lambda z{:}\mathbb{P}.true{\to}z$ reflexivity;
5. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash (\lambda z{:}\mathbb{P}.true{\to}z)x = (\lambda z{:}\mathbb{P}.true{\to}z)y$ congruence using 3 and 4;
6. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash (\lambda z{:}\mathbb{P}.true{\to}z)x = true{\to}x$ beta conversion;
7. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash (\lambda z{:}\mathbb{P}.true{\to}z)y = true{\to}y$ beta conversion;
8. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash true{\to}x = (\lambda z{:}\mathbb{P}.true{\to}z)x$ symmetry from 6;
9. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash true{\to}x = (\lambda z{:}\mathbb{P}.true{\to}z)y$ transitivity 5 and 8;
10. $\Gamma \cup \{true = true \Rightarrow x = y\} \vdash true{\to}x = true{\to}y$ transitivity from 7 and 9;
11. $\Gamma \vdash (true = true \Rightarrow x = y) \Rightarrow true{\to}x = true{\to}y \Rightarrow$ introduction from 10.

8.4.2 Use induction on c_1, and within that induction on c_2.

8.4.3

$$\frac{\Gamma \vdash \phi[x{:=}@c1], \quad \Gamma \vdash \phi[x{:=}p] \Rightarrow \phi[x{:=}@cDub(b,p)]}{\Gamma \vdash \phi} \text{ induction on } \mathbb{N}^+$$

$$\frac{\Gamma \vdash \phi[x{:=}@c0], \quad \Gamma \vdash \phi[x{:=}@cNat(p)]}{\Gamma \vdash \phi} \text{ induction on } \mathbb{N}$$

$$\frac{\Gamma \vdash \phi[x{:=}@cInt(n)], \quad \Gamma \vdash \phi[x{:=}@cNeg(p)]}{\Gamma \vdash \phi} \text{ induction on } \mathbb{Z}$$

8.4.4 To prove the second rule from the first assume some given formula ϕ, for which $\phi(1)$, $\phi(n){\Rightarrow}\phi(2n)$ and $\phi(n){\Rightarrow}\phi(2n{+}1)$ can be proven. Define $\psi = \lambda n{:}\mathbb{N}.\forall i{:}\mathbb{N}.(2^{n-1} \leq i < 2^n) \to \phi(i)$. For ψ it can be shown that $\psi(1)$ and $\psi(n) \Rightarrow \psi(n{+}1)$. Hence, from rule 1 we conclude that $\psi(n)$ for any n. From this $\phi(n)$ follows.

To prove the first rule from the second, define $\psi = \lambda n{:}\mathbb{N}^+.\phi(f(n))$ where $f = \lambda n{:}\mathbb{N}^+.\min(\lceil {}^2\log(n)\rceil, 1)$. Use that $f(2^n) = n$, $f(2n) = 1{+}f(n)$ for $n > 1$ and $f(2n{+}1) = 1{+}f(n)$ for $n > 1$.

8.5.2 First note that $x \lhd t \overset{\text{T5}}{=} t > 0 \to x \lhd t$ for any $t{:}\mathbb{R}$. Using lambda abstraction, congruence and SUM1, it follows that $x = \sum_{t:\mathbb{R}} x \lhd t = \sum_{t:\mathbb{R}} (t > 0) \to x \lhd t$.

8.5.3 ($\supseteq$) By SUM3

$$\sum_{b:\mathbb{B}} b \to x \diamond y = \sum_{b:\mathbb{B}} b \to x \diamond y + true \to x \diamond y + false \to x \diamond y = \sum_{b:\mathbb{B}} b \to x \diamond y + x + y.$$

So $x + y \subseteq \sum_{b:\mathbb{B}} b \to x \diamond y$.

($\subseteq$) $true \to x \diamond y = x \subseteq x + y$ and $false \to x \diamond y = y \subseteq x + y$, so by induction on booleans $b \to x \diamond y \subseteq x + y$. Then by abstraction, congruence, SUM4 and SUM1 $\sum_{b:\mathbb{B}} b \to x \diamond y \subseteq x + y$.

8.5.4 ($\subseteq$) By SUM3

$$\sum_{d:D} c(d){\to}x = \sum_{d:D} c(d){\to}x + c(e){\to}x = \sum_{d:D} c(d){\to}x + x.$$

So $x \subseteq \sum_{d:D} c(d){\to}x$.

($\supseteq$) If $b = true$ then $b{\to}x = x$, and if $b = false$ then $b{\to}x = \delta \lhd 0 \subseteq x$, so $b{\to}x \subseteq x$. By substitution of $c(d)$ for b it folllows that $c(d){\to}x \subseteq x$ where d does not occur in the context. Then by abstraction, congruence, SUM4 and SUM1 $\sum_{d:D} c(d) \to x \subseteq x$.

8.5.5 Using SUM3, it can be shown that $X(0)+X(1)+X(2) \subseteq \sum_{n:\mathbb{N}} n{\leq}2 \rightarrow X(n)$. Using induction on n it can be shown that $n{\leq}2 \rightarrow{\subseteq}\ X(0)+X(1)+X(2)$. This is intricate, when done precisely using the induction principles. A possibility is to use the induction principle for $\mathbb{N}$, and within that the induction principle for $\mathbb{N}^+$. A more 'handwaving' proof can be given by assuming that either $n \leq 2$ or $n > 2$. In the first case, it is 'generally known' that n can only be 0, 1 or 2, and hence the result follows. In the second case the left hand side is equal to $\delta@0$ and the proof follows using axiom T1.

8.6.4 Show that X is a solution for Y. So, it must be shown that $X = a{\cdot}a{\cdot}a{\cdot}X + a{\cdot}a{\cdot}X$. Prove this using the defining equation for X and axiom A3.

8.6.5 First show that $Y = Y{\cdot}\delta$ by showing that $Y{\cdot}\delta$ is a solution for Y. This leads to the proof obligation that $Y{\cdot}\delta = a{\cdot}(Y{\cdot}\delta){\cdot}\delta$. It is now straightforward to show that Y is a solution for X by proving $Y = a{\cdot}Y$. This follows from the defining equation for Y by removing the δ at the end.

8.6.6 Show that $\lambda l{:}List(\mathbb{N}).X(\#(l))$ is a solution for Y. Remove the sum operator with SUM1.

8.6.7 Show that $\lambda n{:}\mathbb{N}.X(n|_N)$ is a solution for Y. Use the calculation rules in section 4.2.4.

8.6.8 Show that $\lambda n{:}\mathbb{N}^+.(n{\approx}1{\rightarrow}X + n{\approx}2{\rightarrow}b{\cdot}c{\cdot}X + n{\approx}3{\rightarrow}c{\cdot}X)$ is a solution for Y.

8.6.9 Define an auxiliary process Z by $Z(n{:}\mathbb{N}) = n{\approx}0{\rightarrow}X \diamond \nabla_{\{a,b\}}(b \| Z(n-1))$. Show that Z is a solution for Y by induction on n. If $n = 0$, it must be shown that $Z(0) = a{\cdot}Z(1)$, $Z(1) = a{\cdot}Z(2){+}b{\cdot}Z(0)$. If $n = m{+}1$ with $m > 1$, the proof obligation is $Z(m{+}1) = a{\cdot}Y(m{+}2){+}b{\cdot}Z(m)$. Note that a useful identity is $Z(m) = \nabla_{\{a,b\}}(Z(m))$.

8.7.1 The conclusion from KFAR is $\tau{\cdot}\tau_{\{head\}}(X) = \tau{\cdot}\tau_{\{head\}}(tail)$.

8.8.1 $S = a{\cdot}S + d{\cdot}S_1,\ \ S_1 = a{\cdot}S_1 + c{\cdot}S$.

9.1.4 Substitute $\lambda n{:}\mathbb{N}, b{:}\mathbb{B}.b{\rightarrow}X \diamond s_2(n){\cdot}X$ in Y.

9.1.5 $X(n{:}\mathbb{N}^+) = (n{\approx}1){\rightarrow}a{\cdot}X(2) + (n{\approx}2){\rightarrow}b{\cdot}X(1) + (n{\approx}1){\rightarrow}b{\cdot}X(3) + (n{\approx}3){\rightarrow}a{\cdot}X(4){+}$ $(n{\approx}4){\rightarrow}a{\cdot}X(2) + (n{\approx}4){\rightarrow}a{\cdot}X(1)$. $X(n{:}\mathbb{N}^+){=}(n{\not\approx}2){\rightarrow}a{\cdot}X((n{+}1)|_4) + (n{<}2){\rightarrow}b{\cdot}X((n{+}2)|_4)$. The equation for Y is unguarded and has no unique solution. It does not unequivocally define a process.

9.1.8 $X(n{:}\mathbb{N}^+) = (n{\leq}2){\rightarrow}a{\cdot}X(n{+}1) + (n{\approx}3){\rightarrow}b{\cdot}X(1)$.

9.2.9 The process declaration is captured by $Z(true)$, where

$$Z(b'{:}\mathbb{B}) = b'{\rightarrow}a{\cdot}Z(false) + b'{\rightarrow}b{\cdot}Z(false) + \neg b'{\rightarrow}d{\cdot}Z(true).$$

9.2.10 **proc** $X(pc{:}\mathbb{N}^+) =$
$(pc{\approx}1){\rightarrow}a{\cdot}X(2) + (pc{\approx}2){\rightarrow}a{\cdot}X(1) + (pc{\approx}2){\rightarrow}b{\cdot}X(4){+}$
$(pc{\approx}3){\rightarrow}b{\cdot}X(4) + (pc{\approx}4){\rightarrow}a{\cdot}X(2) + (pc{\approx}4){\rightarrow}b{\cdot}X(4)$;

9.2.11 **proc** $X_1(pc{:}\mathbb{N}^+, n{:}\mathbb{N}) =$
$\sum_{m:\mathbb{N}}(pc{\approx}1){\rightarrow}a(m){\cdot}X_1(2,m) + \sum_{m:\mathbb{N}}(pc{\approx}2){\rightarrow}a(m){\cdot}X_1(2,m) + (pc{\approx}2){\rightarrow}c(n){\cdot}X_1(1,0)$;
$X_2 = \sum_{n:\mathbb{N}} a(n){\cdot}X_2$;
sort $Stack = \mathbf{struct}\ empty?isEmpty \mid push(getpc{:}\mathbb{N}^+, getn{:}\mathbb{N}, pop{:}Stack)$;
proc $X_3(s{:}Stack) =$
$\sum_{n:\mathbb{N}}(\neg isEmpty(s) \wedge getpc(s){\approx}1){\rightarrow}a(n){\cdot}X_3(push(2,n,pop(s))){+}$
$\sum_{n:\mathbb{N}}(\neg isEmpty(s) \wedge getpc(s){\approx}2){\rightarrow}a(n){\cdot}X_3(push(2,n,push(3,getn(s),pop(s)))){+}$
$(\neg isEmpty(s) \wedge getpc(s){\approx}2){\rightarrow}c(getn(s)){\cdot}X_3(pop(s)){+}$
$(\neg isEmpty(s) \wedge getpc(s){\approx}3){\rightarrow}c(getn(s)){\cdot}X_3(pop(s))$;

9.2.12 **proc** $X(pc{:}\mathbb{N}^+) = (pc{\approx}1){\rightarrow}a{\cdot}3{\cdot}X(2) + (pc{\approx}2){\rightarrow}c{\cdot}4{\cdot}X(1)$;
$Y(pc{:}\mathbb{N}^+, n{:}\mathbb{N}) = \sum_{m:\mathbb{N}}(pc{\approx}1){\rightarrow}r(m){\cdot}Y(2,m) + (pc{\approx}2 \wedge n{>}10){\rightarrow}a{\cdot}n{\cdot}Y(1,n)$;

9.3.13 We prove that $\mathcal{I}_i$ is an invariant for the LPE X in Definition 9.1.1; i.e., $\mathcal{I}_i(d) \wedge h(d,e) \Rightarrow \mathcal{I}_i(g(d,e))$, for $i = 1, 2$.

If $i = 1$, then we obtain $true \wedge h(d,e) \Rightarrow true$, which is true.

If $i = 2$, then we obtain $false \wedge h(d,e) \Rightarrow false$, which is true.

10.1.3 $\{s_0 \xrightarrow{\tau} s_1, s_2 \xrightarrow{\tau} s_3\}$; $\emptyset$, $\{s_0 \xrightarrow{\tau} s_1, s_2 \xrightarrow{\tau} s_3, s_4 \xrightarrow{\tau} s_5\}$, $\{s_3 \xrightarrow{\tau} s_5, s_4 \xrightarrow{\tau} s_5\}$.

10.1.4 $\{s_0 \xrightarrow{\tau} s_1\}$; $\emptyset$, $\{s_0 \xrightarrow{\tau} s_1\}$.

10.2.6 Prioritising transition systems 1, 3 and 7 maintains branching bisimulation, in accordance with the theory.

11.1.3 $FC(n) = n|_3 \not\approx 2$, $M(n) = n|_3$ and $h(n) = n|_3 \not\approx 1$.

11.1.4 Note that from $n \approx 3m$ it follows that $m \approx n \text{ div } 3$ (but not vice versa). So, the condition $n \approx 3m$ is equivalent to $n \approx 3m \wedge m \approx n \text{ div } 3$. Applying the sum elimination theorem simplifies the condition to $n \approx 3(m \text{ div } 3)$. This is equivalent to $n|_3 \approx 0$. In this way the first summand of exercise 11.1.3 is obtained. The second and third summand can be obtained in a similar way.

11.1.5 The focus condition for $n{:}\mathbb{N}$ is: $\exists m{:}\mathbb{N}.(n = 3m \vee n = 3m + 1)$.

$$\mathcal{I}(n) = \begin{cases} true & \text{if } \exists m{:}\mathbb{N}(n = 3m \vee n = 3m + 2) \\ false & \text{if } \exists m{:}\mathbb{N}(n = 3m + 1) \end{cases}$$

The matching criteria are fulfilled for all n with $\mathcal{I}(n) = true$:

- $n = 3m + 2 \Rightarrow \phi(n) = \phi(n + 1) = nil$;
- $n = 3m \Rightarrow h'_a(\phi(n)) = true$;
- $n = 3m \Rightarrow n = 3m$;
- actions do not carry data parameters;
- $n = 3m \Rightarrow \phi(n + 2) = nil$.

12.1.2 Each τ-transition loses the possibility to execute one of the $leader(n)$-actions at the end.

12.1.3 The network is defined by

proc $S = \tau_{\{c\}} \nabla_{\{c, leader\}} \Gamma_{\{r|s \to c\}}$
$(Node(1, \{2, 3\}, 0) \parallel Node(2, \{1, 3\}, 0) \parallel Node(3, \{1, 2\}, 0))$;

No node can start sending a parent request.

Bibliography

[1] B. Badban, W. Fokkink, J.F. Groote, J. Pang and J.C. van de Pol. Verification of a sliding window protocol in μCRL and PVS. Formal Aspects of Computing 17(3):342-388, 2005.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[4] H.P. Barendregt. Lambda Calculi with Types. In S. Abramsky, D.M. Gabbay, T.S.E. Maibaum. Handbook of Logic in Computer Science, pp. 117-309, Oxford Science Publications, 1992.

[5] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[6] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[7] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, eds, *Proceedings 5th Conference on Concurrency Theory (CONCUR'94)*, Uppsala, Sweden, LNCS 836, pp. 401–416. Springer-Verlag, 1994.

[8] S.C.C. Blom. Partial τ-confluence for efficient state space generation. Technical Report SEN-R0123, CWI, 2001.

[9] G. von Bochmann. Logical verification and implementation of protocols. In *Proceedings 4th ACM-IEEE Data Communications Symposium*, Quebec, Canada, pp. 7-15–7-20. IEEE, 1975.

[10] J.J. Brunekreef. Sliding window protocols. In S. Mauw and G.J. Veltink, eds, *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[11] D.E. Carlson. Bit-oriented data link control. In P.E. Green Jr, ed., *Computer Network Architectures and Protocols*, pp. 111–143. Plenum Press, 1982.

[12] V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.

[13] E.W. Dijkstra. A disciplne of programming. Prentice-Hall, Englewood Cliffs, 1976.

[14] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. Journal of Algorithms, 3: 245-260, 1982.

[15] W.J. Fokkink, J.F. Groote and M.A. Reniers. Modelling distributed systems. Unpublished. 2006.

[16] W.J. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In A.D. Gordon, editor, Proceedings of the 6th Conference on Foundations of Software Science and Computation Structures - FOSSACS'03, Warsaw, Lecture Notes in Computer Science 2620, pp. 267–281, Springer Verlag, April 2003.

[17] R.J. van Glabbeek. A complete axiomatization for branching bisimulation congruence of finite-state behaviours. In Proceedings Mathematical Foundations of Computer Science 1993 (MFCS), Gdansk, Poland, August/September 1993 (A.M. Borzyszkowski and S. Sokolowski, editors.), LNCS 711, Springer-Verlag, pp. 473–484, 1993.

[18] R.J. van Glabbeek. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse and S.A. Smolka, Handbook of process algebra, Elsevier, pp. 3–99, 2001.

[19] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[20] J.F. Groote. The syntax and semantics of timed μCRL. Technical report SEN-R9709, CWI, Amsterdam, 1997.

[21] J.F. Groote, F. Monin and J. Springintveld. A computer checked algebraic verification of a distributed summation algorithm. Formal Aspects of Computing 17:19-37. Springer-Verlag. 2005.

[22] J.F. Groote and J.C. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, eds, *Proceedings 25th Symposium on Mathematical Foundations of Computer Science (MFCS'00)*, Bratislava, Slovakia, LNCS 1893, pp. 383–393. Springer-Verlag, 2000.

[23] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes (ACP'94)*, Utrecht, The Netherlands, Workshops in Computing, pp. 26–62. Springer-Verlag, 1995.

[24] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse and S.A. Smolka. Handbook of Process Algebra, pages 1151-1208, Elsevier, Amsterdam, 2001.

[25] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1/2):47–81, 1996.

[26] J.F. Groote and J. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60, 2001.

[27] M.C.B. Hennessy and H. Lin. Unique fixpoint induction for message-passing process calculi. In *Proceedings 3rd Australasian Theory Symposium on Computing (CATS'97)*, Sydney, Australia, pp. 122–131. Australia Computer Science Communications, 1997.

[28] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[29] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[30] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[31] IEEE Computer Society. *IEEE Standard for a High Performance Serial Bus*. Std. 1394-1995, August 1996.

[32] ISO. Informationon processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour ISO/TC97/SC21/N DIS8807, 1987.

[33] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[34] N.A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, Inc. 1996.

[35] W.C. Lynch. Reliable full duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407–410, 1968.

[36] Efficient on-the-fly model checking for regular alternation free mu-calculus. Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany), April 2000

[37] S. Mauw and J.C. Mulder. Regularity of BPA-Systems is Decidable. In B. Jonsson and J. Parrow, editors, Proc. CONCUR '94, LNCS 836, pp. 34–47. Uppsala, Springer Verlag, 1994.

[38] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamentae Informaticae* XIII:85–139, 1990.

[39] R. Milner. An approach to the semantics of parallel programs. In *Proceedings Cenvegno di Informatica Teorica, Pisa*, pages 283-302, 1973.

[40] R. Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium 1973*, pages 158-173. North-Holland, 1973.

[41] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[42] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[43] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[44] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[45] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, ed., *Proceedings 5th GI (Gesellschaft für Informatik) Conference*, Karlsruhe, Germany, LNCS 104, pp. 167–183. Springer-Verlag, 1981.

[46] C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10(5/6):509–531, 1998.

[47] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[48] Terese. Term rewriting systems. M. Bezem, J.W. Klop and R. de Vrijer, editors. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge 2003.

[49] Y.S. Usenko. Linearization in μCRL. PhD. thesis. Eindhoven University of Technology. 2002.

[50] R.F.C. Walters. Number theory. An introduction. Carslaw Publications. 1987.

[51] M.J. van Weerdenburg. Process Algebra with Local Communication. Computer Science Report 05/05, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2005.