Modelling and Analysis of Communicating Systems
(abridged version)

Jan Friso Groote and Mohammad Reza Mousavi
Department of Computer Science
Eindhoven University of Technology, Eindhoven
Email: J.F.Groote@tue.nl, M.R.Mousavi@tue.nl

Revision: 4268
Date: 2011-09-06 20:07:37 +0200 (Tue, 06 Sep 2011)
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 [40, 41]. As a consequence, he set out working on his seminal book [42, 44] in which he developed 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, [30, 31]).

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 [33] and PSF [39]) 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 [24, 21]) 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 [25]. This chapter formed the basis of a reader [16] used for courses at several universities (published as [?]). These earlier publications were based on the modelling language µCRL (micro Common Representation Language, [24, 21]) 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, and contains far more material than [?].

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 [42] 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 courses Requirements, Analysis, Design and Verification (RADV) and System Validation (SV) at Eindhoven University of Technology and Delft University of Technology. Many thanks go to Jeroen Keiren and Michel Reniers for his careful proofreading. Valuable feedback also came from Muhammad Atif, Ruud Bauhaus, Harsh Beohar, Debjyoni 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, Chidi Okwudire, Paul Mulders, Mathijs Opdam, Chidi Okwudire, Eva Ploum, André van Renssen, Marcel Roeloffzen, Koos Rooda, Frank Stappers, Carst Tankink, Maks Ververs, Sander Verdonschot, Amrita Vikas Sinha, Migiel de Vos, Umar Waqas and many others.

Jan Friso Groote and Mohammad Reza Mousavi
September 2010, Eindhoven, The Netherlands
## Contents

1 Modeling system behaviour .......................................................... 7
   1 Introduction ............................................................................. 9
      1.1 Motivation ........................................................................ 9
      1.2 The mCRL2 approach ......................................................... 10
      1.3 An overview of the book .................................................. 10
      1.4 Audience and suggested method of reading ....................... 11

2 Actions, behaviour, equivalence and abstraction .......................... 13
   2.1 Actions ............................................................................... 13
   2.2 Labelled transition systems ............................................... 14
   2.3 Equivalence of behaviours .................................................. 16
      2.3.1 Trace equivalence ...................................................... 16
      2.3.2 Language equivalence ............................................... 17
      2.3.3 Failures equivalence .................................................. 17
      2.3.4 Strong bisimulation equivalence ................................. 19
      2.3.5 The Van Glabbeek linear time – branching time spectrum 21
   2.4 Behavioural abstraction ....................................................... 23
      2.4.1 The internal action \( \tau \) .............................................. 24
   2.5 Behavioural equivalence for the internal action .................... 24
      2.5.1 Weak trace equivalence ............................................. 24
      2.5.2 Rooted branching bisimulation .................................. 25
      2.5.3 Rooted weak bisimulation ......................................... 28

3 Data types ............................................................................... 31
   3.1 Basic data type definition mechanism ................................. 31
   3.2 Standard data types ............................................................ 34
      3.2.1 Booleans .................................................................... 35
      3.2.2 Numbers .................................................................... 35
   3.3 Function data types ............................................................. 38
   3.4 Structured data types .......................................................... 38
   3.5 Constructed data types ......................................................... 40
      3.5.1 Lists ........................................................................... 40
      3.5.2 Sets and bags ............................................................ 41
   3.6 Terms and where expressions ............................................. 41

4 Sequential processes ................................................................. 43
   4.1 Actions .............................................................................. 43
   4.2 Multi-actions ...................................................................... 44
   4.3 Alternative and sequential composition ............................... 45
   4.4 Deadlock ........................................................................... 47
   4.5 The conditional and sum operator ..................................... 47
   4.6 Recursive processes ........................................................... 49
5 Parallel processes
5.1 The parallel operator .............................................................. 51
5.2 Communication among parallel processes ................................. 52
5.3 Blocking and renaming ............................................................ 56
5.4 Hiding internal behaviour ......................................................... 56
5.5 Alphabet axioms ..................................................................... 58

6 The modal $\mu$-calculus .................................................................. 63
6.1 Hennessy-Milner logic .............................................................. 63
6.2 Regular formulas ...................................................................... 65
6.3 Fixed point modalities ............................................................... 66
6.4 Modal formulas with data .......................................................... 69
6.5 Equations ................................................................................. 70

7 Modelling of system behaviour ..................................................... 73
7.1 Alternating bit protocol ............................................................. 73
7.2 Sliding window protocol ............................................................ 76
7.3 A patient support platform ......................................................... 79

A Answers to exercises .................................................................. 87

References ....................................................................................... 93
Part I

Modeling system behaviour
Chapter 1

Introduction

1.1 Motivation

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 fulfill 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 theories, appropriate proof- and anal-
ysis methods and adequate computer tools, it is impossible to design correct(ly) communicating systems. In this book we provide the required ingredients.

### 1.2 The mCRL2 approach

In this book, we present a rigorous approach to behavioural specification and verification of concurrent and distributed systems. Our approach builds upon the rich literature of process algebras [42, 6, 31], which are algebraic formalisms for compositional specification of concurrent systems. More specifically, we propose a process algebra, called mCRL2, which extends the Algebra of Communicating Processes (ACP) [6] with various features including notions of data, time and multi-actions.

Specification in mCRL2 are built out of using atomic (inter)actions which can be composed using various algebraic operators it provides. The specified can then be simulated, visualized or verified against its requirements. Requirements are defined by using a rich logic, namely the modal mu-calculus with data and time. This logic is very suitable to express patterns of (dis)allowed behaviour. In order to mechanically verify the requirements, an extensive toolset has been developed for mCRL2. It can be downloaded from www.mcrl2.org.

### 1.3 An overview of the book

In chapter 2 our basic building block, i.e., the notion of action, is introduced. 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 the data types being used are explained. In particular it is explained how all datatypes are built upon constructors, mappings and functions using equations. In appendix ?? the exact definition of all pre-defined data types is given.

In chapter 4 non parallel 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 chapter 5 it is shown how parallel and communication components are specified, again including the required axioms.

The next chapter, i.e. chapter 6, 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.

Chapter 7 contains a number of example descriptions of the behaviour of some simple systems. In chapter ?? all components of the framework are extended with time. This chapter concludes the first part of the book.

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

Chapter ?? 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 ?? 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 ?? 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 ?? the accumulated techniques are used to prove a number of typical protocols and distributed algorithms correct. In chapter ?? 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 chapter ?? 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 the subsequent chapters.
In appendix ?? the equations characterising datatypes are given. In appendix ?? an overview of the syntaxes of all different formalisms are given. Appendix ?? summarises all process algebraic axioms. The last appendix A contains answers to the exercises.

1.4 Audience and suggested method of reading

The authors have used this book for several years to teach graduate courses on formal specification and verification. It is suitable for graduate (or upper level undergraduate) students of computer science and related fields (e.g., embedded systems and software engineering) with some background in logic and the theory of formal languages and automata.

For a short course (of 7 to 10 lectures) chapters 2 to 6 can be used to treat the theory of parallel processes together with a small project on specification and verification of a small system controller or a protocol (examples of such systems and protocols can be found in chapters 7 and ??). For larger modules, chapters 9 to 12 and chapter 14 can also be included.

More advance material in each chapter is designated by a vertical bar and can be skipped for shorter and / or undergraduate courses.
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.

Also actions can be parameterized with data; an action $a$ taking data parameter $d$ is denoted by $a(d)$; examples of such actions are read(1), write(true, 2) and draw(1.5, 2.5, sqrt(2)), where sqrt(2) is a data expression defining the square root of 2. This feature is essential in modeling reactive systems that communicate data and make decisions based on the values of communicated data.

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., $a:3$ 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 ?? and 4.

We use an alarm clock as our running example throughout this chapter. Our simple alarm clock has three basic actions, namely, set, reset and alarm. Also, we specify a variant of alarm clock in which the
number of alarms can be given to the clock; in this variant we use actions of the form set(n), where \( n \) is a natural number denoting the number of times the alarm should go off.

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?

### 2.2Labelled 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, i.e., graph vertices, and a set of transitions, i.e., graph edges, 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 (\( \checkmark \)). 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](image)

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](image)

Our model of alarm clock can be extended with the feature for specifying the number of alarms to go off. This is achieved by parameterizing the action set, with a natural number as depicted in Figure 2.3. In this picture we assume that the pattern of behavior is repeated and hence, we have an infinite labelled transition system.

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 2.4. 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 times before stopping, it can be useful to describe it using the model of 2.4. 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 [40, 42]. 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 that 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 \( A = (S, \text{Act}, \rightarrow, s, T) \) where

- \( S \) is a set of states.
- \( \text{Act} \) is a set of actions, possibly multi-actions.
- \( \rightarrow \subseteq S \times \text{Act} \times S \) 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 \( (t, a, t') \in \rightarrow \).

Often, when not relevant or clear from the context, the set \( T \) of terminating states and \( s \) or the initial state are omitted from 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 2.4 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 [19].

2.3.1 Trace equivalence

One of the coarsest (most unifying) notions of 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 $\sigma$ and $\rho$ to represent traces. The termination symbol $\top$ can also be part of a trace. The symbol $\epsilon$ represents the empty trace.

**Definition 2.3.1 (Trace equivalence).** Let $A = (S, Act, \rightarrow, s, T)$ 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. $\top \in Traces(t)$ 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.

Consider the labelled transition systems for the two alarm clocks depicted in figure 2.5. The alarm clock in the left-hand side has a nondeterministic choice between two set transitions: if it moves with the set transition to right, it behaves the same as the right-hand-side labelled transition system. However, if it moves to left with the other set transition, it deadlocks. Hence, the observational behavior of the two transition systems is different: the left-hand-side one sometimes stands still while the right-hand-side keeps interacting with its environment. This is the reason why trace equivalence generally is not used and a finer notion, called language equivalence, is used which refines trace equivalence by taking deadlocks into account.

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 observe quiescence or influence the behaviour, trace equivalence is exactly the right notion. In other words, if one can neither interact with a system, nor distinguish a slow system from one that has come to a stand still, then trace equivalence is appropriate. 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 2.5, 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 2.3.2.** Which of the labelled transition systems of exercise 2.3.7 are trace equivalent.
2.3.2 Language 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 2.3.3 (Language equivalence).** Let $A = (S, \text{Act}, \rightarrow, s, T)$ be a labelled transition system.

We define the language $\text{Lang}(t)$ of a state $t \in S$ as the minimal set satisfying:

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

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

Note that our notion of language equivalence slightly deviates from the language-theoretic notion. Our notion has the additional constraint that the set of traces should also be equivalent. This is essential for distinguishing systems with infinite behavior. For example, the language of a non-deadlocking alarm clock (e.g., the one on the right-hand side of figure 2.5) is empty, which is equal to the language of a labelled transitions system with a single deadlocking initial state. However, the observable behavior from the two and also the traces of the two labelled transition systems are different. The aforementioned labelled transition systems are thus considered different by definition 2.3.3, while they are identified by the notion of language equivalence in language theory.

Note that the transition systems in figure 2.5 are not language equivalent. The language of the one at the left is \{set\} whereas the language of the one at the right is the empty set. Consider the labelled transition systems depicted in figure 2.6; they are certainly trace equivalent since they both have the empty trace, set, set reset and set alarm as their set of traces. Moreover, they are also language equivalent since their languages are both empty. However, they are clearly distinguishable when interacting with them, upon interacting with the set set action, the left-hand-side system already makes a decision whether it takes the left-hand-side set transition or the right-hand-side one. Assume, without loss of generality, that it takes the left-hand-side one; then, if the environment want to synchronize on the action reset, the system fails to accept this, i.e., the combination of system and its environment deadlocks. However, in the right-hand-side system, after synchronizing on set, the system still offers possibilities of synchronizing on both reset as well as alarm and hence, the aforementioned deadline scenario can never materialise. To summarise, although the notion of language equivalence may be a suitable of observational equivalence for closed systems, it is not robust against interaction and hence is not appropriate for reactive systems. This motivates the finer (more distinguish) notions to come.

2.3.3 Failures equivalence

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 2.3.4 (Failures equivalence). Let $A = (S, \text{Act}, \rightarrow, s, T)$ be a labelled transition system. A set $F \subseteq \text{Act} \cup \{✓\}$ 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 $✓ \in F$, then $t \notin T$, i.e. $t$ cannot terminate.

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

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

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

Arguably, failures equivalence does not provide enough distinguishing power. The labelled transition systems depicted in figure 2.7 are indeed failures equivalent. (Checking this fact is left to the interested reader.) They share the behavior demonstrating the lack of choice, i.e., when by performing a set action the system non-deterministically chooses to take the path in which either a reset, or an alarm action is possible. That is why they are intuitively considered failures equivalent, namely, because they offer the same possibility of deadlock. However, the right-hand-side labelled transition system offers some extra piece of behavior allowing the user to make a choice after performing a set action. Since this difference is neglected by failures equivalence, it seems appropriate to consider finer, i.e., more distinguishing, notions of behavioral equivalence such as strong bisimulation defined next.

Exercise 2.3.5. State whether the following pairs of transition systems are language and/or failures equivalent.

Figure 2.7: Two failure-equivalent alarm clocks
2.3.4 Strong bisimulation equivalence

Bisimulation equivalence (also referred to as strong bisimulation equivalence) or (strong) bisimilarity is the most important process equivalence [3, 43, 46]. 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 2.3.6 (Bisimulation). Let $A=(S, Act, \rightarrow, s, T)$ be a labelled transition system. A binary relation $R \subseteq S \times S$ is called a strong bisimulation relation iff for all $s, t \in S$ such that $sRt$ holds, it also holds for all actions $a \in Act$ that:

1. if $s \xrightarrow{a} s'$, then there is a $t' \in S$ such that $t \xrightarrow{a} t'$ with $s'Rt'$,
2. if $t \xrightarrow{a} t'$, then there is a $s' \in S$ such that $s \xrightarrow{a} s'$ with $s'Rt'$, and
3. $s \in T$ if and only if $t \in T$.

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

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 [34, 45].

For small transition systems a more straightforward technique generally is adequate. Consider the transition systems in figure 2.8. 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 2.8). 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
extended as indicated in the fourth picture of figure 2.8. Finally, it needs to be checked that all related states satisfy the requirements in definition 2.3.6.

Now consider the transition systems in figure 2.9. 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 2.9 the same sequence of actions can be performed, namely $ab$ and $ac$. 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 2.3.7.** State for each of the following transition systems whether they are pairwise bisimilar:
Exercise 2.3.8. 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 2.3.9. Give the unique minimal labelled transition system that is bisimilar to the following one:

2.3.5 The Van Glabbeek linear time – branching time spectrum

As stated before, there is a myriad of process equivalences. A nice classification of some of these has been made by Van Glabbeek [19]. He produced the so-called linear time – branching time spectrum which is depicted in figure 2.10. 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 that precisely 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
Figure 2.10: The Van Glabbeek linear – branching time spectrum
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 \( A = (S, Act, \rightarrow, s, T) \) deterministic \( \text{iff} \) 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 \( A = (S, Act, \rightarrow, s, T) \) be a deterministic transition system. For all states \( t, t' \in S \) it holds that

\[
\text{Traces}(t) = \text{Traces}(t') \iff t \xrightarrow{\sim} t'.
\]

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

In order to show that \( t \xrightarrow{\sim} t' \), we need to show the existence of a bisimulation relation \( R \) such that \( t R t' \).

We coin the following relation for any states \( u \) and \( u' \):

\[
R(u, u') \iff \text{Traces}(u) = \text{Traces}(u').
\]

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 2.3.6. So, assume that for states \( u \) and \( v \) we have that \( u R v \). Then

1. Suppose \( u \xrightarrow{a} u' \). So, according to definition 2.3.1 \( a \sigma \in \text{Traces}(u) \) for all traces \( \sigma \in \text{Traces}(u') \).

Furthermore, as \( \text{Traces}(u) = \text{Traces}(v) \), it holds that \( a \sigma \in \text{Traces}(v) \) or in other words, \( v \xrightarrow{a} v' \) for some state \( v' \in S \). So, we are left to show that \( u' R v' \), or in other words:

\[
\text{Traces}(u') = \text{Traces}(v').
\]

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

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

3. If \( u \in T \), then \( \checkmark \in \text{Traces}(u) \). As \( u \) and \( v \) are related, it follows by definition of \( R \) that \( \checkmark \in \text{Traces}(v) \). 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 to decide (P-space hard).

**Exercise 2.3.12.** Prove that bisimilarity on a given labelled transition system is an equivalence relation, i.e., it is reflexive (\( s \equiv s \) for any \( s \in S \)), symmetric (if \( s \equiv t \) then \( t \equiv s \) for all states \( s \) and \( t \)) and transitive (if \( s \equiv t \) and \( t \equiv u \), then \( s \equiv u \) for all states \( s, t, u \)).

### 2.4 Behavioural abstraction

Although the examples given hitherto 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 [42] together with an associated process equivalence, called weak bisimulation. We are mainly using branching bisimulation which was defined by Van Glabbeek and Weijland [20]. 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 $\tau$

An action is internal, if we have no way of observing it directly. We use the special symbol $\tau$ to denote any internal action. We generally assume that it is available in a labelled transition system, i.e., $\tau \in \mathit{Act}$. 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 2.11 cannot be distinguished, because the $\tau$ after the $a$ cannot be observed. Such internal actions are called inert.

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 2.12. 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 $\tau$ 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 Weak trace equivalence

The notion of weak trace is obtained by absorbing the internal action in a trace. Two processes are weakly trace equivalent, if their sets of weak traces, i.e., traces in which $\tau$-transitions are neglected, are the same.

**Definition 2.5.1 (Weak trace equivalence).** Let $A = (S, \mathit{Act}, \rightarrow, s, T)$ be a labelled transition system. The set of weak traces $W\mathit{Traces}(t)$ for a state $t \in S$ is the minimal set satisfying:

1. $\epsilon \in W\mathit{Traces}(t)$.
2. $✓ \in W\mathit{Traces}(t)$ iff $s \in T$, and
3. if there is a state $t' \in S$ such that $t \xrightarrow{a} t'$ ($a \neq \tau$) and $\sigma \in W\mathit{Traces}(t')$ then $a\sigma \in W\mathit{Traces}(t)$.
4. if there is a state $t' \in S$ such that $t \xrightarrow{\tau} t'$ and $\sigma \in W\mathit{Traces}(t')$ then $\sigma \in W\mathit{Traces}(t)$.

Two states $t, u \in S$ are called weak trace equivalent iff $W\mathit{Traces}(t) = W\mathit{Traces}(u)$. 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.

2.5.2 Rooted branching bisimulation

In [18] it is shown that the observable equality of the two behaviours in figure 2.11 leads to rooted branching bisimulation as the finest equivalence incorporating internal actions [20]. 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 that single action. See the diagram at the left of figure 2.13. Note that all states that are visited via \( \tau \) actions are related.

If the action to be simulated is a \( \tau \) it can be simulated by any number of internal transitions. Even by none, as the diagram in the middle of figure 2.13 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 2.13.

**Definition 2.5.2 (Branching bisimulation).** Consider the labelled transition system \( A = (S, \text{Act}, \rightarrow, s, T) \). We call a relation \( R \subseteq S \times S \) a branching bisimulation relation if for all \( s, t \in S \) such that \( sRt \), the following conditions hold for all actions \( a \in \text{Act} \):

1. If \( s \xrightarrow{a} s' \), then
   - either \( a = \tau \) and \( s'Rt \), or
   - there is a sequence \( t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t' \) of (zero or more) \( \tau \)-transitions such that \( sRt' \) and \( t' \xrightarrow{a} t'' \) with \( s'Rt'' \).

2. Symmetrically, if \( t \xrightarrow{a} t' \), then
   - either \( a = \tau \) and \( sRt' \), or
   - there is a sequence \( s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s' \) of (zero or more) \( \tau \)-transitions such that \( s'Rt \) and \( s' \xrightarrow{a} s'' \) with \( s''Rt' \).

3. If \( s \in T \), then there is a sequence of (zero or more) \( \tau \)-transitions \( t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t' \) such that \( sRt' \) and \( t' \in T \).
4. Again, symmetrically, if \( t \in T \), then there is a sequence of (zero or more) \( \tau \)-transitions \( s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s' \) such that \( s'Rt \) and \( s' \in T \).

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.

**Example 2.5.3.** In figure 2.14 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 assumes a notion of **fairness**. That is, if a \( \tau \)-loop exists, then no infinite execution sequence will remain in this \( \tau \)-loop forever if there is a possibility to leave it. This phenomenon is also called **divergence**. The intuition is that there is zero chance that any exit from the \( \tau \)-loop will never be chosen. It is straightforward to show that the initial states in the two labelled transition systems in figure 2.15 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. Figure 2.16 illustrates the problem:
The two transition systems at the left of figure 2.16 are branching bisimilar, but the diagrams at the right reflect the transition systems of figure 2.9. They are not branching bisimilar, because doing the $\tau$ in the third transition system means that the option to do $a \ b$ disappears, and this is not possible in the fourth graph.

Milner [44] showed that this problem can be overcome by adding a rootedness condition: initial $\tau$-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 2.5.4 (Rooted branching bisimulation).** Let $A = (S, Act, \rightarrow, s, T)$ be a labelled transition system. A relation $R \subseteq S \times S$ is called a rooted branching bisimulation relation iff it is a branching bisimulation relation and it satisfies for all $s \in S$ and $t \in S$ such that $sRt$:

1. if $s \xrightarrow{a} s'$, then there is a $t' \in S$ such that $t \xrightarrow{a} t'$ and $s' \leftrightarrow b t'$.
2. symmetrically, if $t \xrightarrow{a} t'$, then there is an $s' \in S$ such that $s \xrightarrow{a} s'$ and $s' \leftrightarrow b t'$.

Two states $s \in S$ and $t \in S$ are rooted branching bisimilar, denoted by $s \leftrightarrow_{rb} t$, 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:

$$\leftrightarrow \subset \leftrightarrow_{rb} \subset \leftrightarrow_{b}.$$ 

In the absence of $\tau$, bisimulation and branching bisimulation coincide.

**Exercise 2.5.5.** Show using the definition of rooted branching bisimulation that the two labelled transition systems in figure 2.11 are rooted branching bisimilar. Show also that the two transition systems in figure 2.12 are neither rooted branching bisimilar nor branching bisimilar.

**Exercise 2.5.6.** Which of the following pairs of transition systems are branching and/or rooted branching bisimilar.

**Exercise 2.5.7.** With regard to the examples in the previous exercise 2.5.6 which $\tau$-transitions are inert with respect to branching bisimulation, i.e., for which $\tau$-transitions $s \xrightarrow{\tau} s'$ are the states $s$ and $s'$ branching bisimilar.
2.5.3 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 2.5.6 are weakly bisimilar, although the initial \( \tau \) 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 2.5.8 (Weak bisimulation).** Consider the labelled transition system \( A = (S, \text{Act}, \rightarrow, s, T) \).

A relation \( R \subseteq S \times S \) is called a weak bisimulation relation if for all \( s, t \in S \) such that \( s \xlongleftarrow{R} t \), the following conditions hold:

1. If \( s \xrightarrow{a} s' \), then
   - either \( a = \tau \) and \( s' \xlongleftarrow{R} t \), or
   - there is a sequence \( t \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{a} \xrightarrow{\tau} \cdots t' \) such that \( s' \xlongleftarrow{R} t' \).

2. Symmetrically, if \( t \xrightarrow{a} t' \), then
   - either \( a = \tau \) and \( s \xlongleftarrow{R} t' \), or
   - there is a sequence \( s \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{a} \xrightarrow{\tau} \cdots s' \) such that \( s' \xlongleftarrow{R} t' \).

3. If \( s \in T \), then there is a sequence \( t \xrightarrow{\tau} \cdots t' \) such that \( t' \in T \).

4. Again, symmetrically, if \( t \in T \), then there is a sequence \( s \xrightarrow{\tau} \cdots s' \) such that \( s' \in T \).

Two states \( s \) and \( t \) are weakly bisimilar, denoted by \( s \xrightarrow{\text{wb}} t \), iff there is a weak bisimulation relation \( R \) such that \( s \xlongleftarrow{R} t \). Two labelled transition systems are weakly bisimilar iff their initial states are weakly bisimilar.

In figure 2.17 weak bisimulation has been depicted. Compare this figure with figure 2.13 for branching bisimulation. Note that weak bisimulation is more relaxed in the sense that \( \bar{R} \) 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 2.5.9 (Rooted weak bisimulation).** Let \( A = (S, \text{Act}, \rightarrow, s, T) \) be a labelled transition system. A relation \( R \subseteq S \times S \) is called a rooted weak bisimulation relation if \( R \) is a weak bisimulation relation and it satisfies for all \( s, t \in S \) such that \( s \xlongleftarrow{R} t \):
1. if \( s \xrightarrow{\tau} s' \), then there is a sequence \( t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t' \) of at least length 1 and \( s' \xleftrightarrow{w} t' \), and

2. symmetrically, if \( t \xrightarrow{\tau} t' \), then there is a sequence of at least length 1 of \( \tau \)-steps \( s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s' \) and \( s' \xleftrightarrow{w} t' \).

Two states \( s \in S \) and \( t \in S \) are rooted weak bisimilar, denoted by \( p \xleftrightarrow{\text{rw}} q \), 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.

\[
\xleftrightarrow{\text{w}} \subset \xleftrightarrow{\text{b}} \subset \xleftrightarrow{\text{rb}} \subset \xleftrightarrow{\text{rw}} \subset \xleftrightarrow{\text{w}}.
\]

Note that rooted weak bisimulation and branching bisimulation are incomparable.

**Exercise 2.5.10.** Which of the pairs of transition systems of exercise 2.5.6 are (rooted) weakly bisimilar. Which \( \tau \)-transitions are inert with respect to weak bisimulation.

**Exercise 2.5.11.** Prove that any branching bisimulation is a weak bisimulation relation.
Chapter 3

Data types

Components of reactive systems often exchange messages containing data items among themselves and with the environment. For example, recall from chapter 2 the alarm clock example with multiple alarms, of which the labelled transition system reproduced in figure 3.1. In this figure, the data parameter of action \textit{set} is to be used by the environment to communicate the number of times the alarm should sound. In order to specify such actions, we need a specification language for data types and their elements; ideally the specification language should feature both built-in common data types such as natural numbers, booleans and real numbers and allow the user to define new data types such as enumerate (structured data) types, lists and sets. A strong feature of mCRL2 is a very rich specification language for both built-in and user-defined data types. This chapter is dedicated to the data type specification language and in the next chapter we study how data types can be attached to actions and how data expressions can influence the behavior of processes.

In mCRL2, one may define data types (called sorts), by defining its elements (constructors), its operations (maps) and the equations governing the operations. Moreover, mCRL2 provides facilities to define data types in a more structured mannered. These facilities allow for four kinds of data types, namely, standard (built-in) data types, structured data types, function types and constructed types. But before explaining these, the general mechanism to define new data types is explained.

![Figure 3.1: An alarm clock with multiple alarm feature](image)

3.1 Basic data type definition mechanism

In mCRL2, we have a straightforward data definition mechanism using which all data sorts are built. One can declare arbitrary sorts using the keyword \texttt{sort}. Sorts are non-empty, possibly infinite sets with data elements. For a sort one can define constructor functions using the keyword \texttt{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 $\text{Nat}$ representing the natural numbers. (This is not the actual built-in sort $\mathbb{N}$ in mCRL2, which for efficiency purposes has a different internal structure. The definition of the built-in datatype can be found in appendix ??)

sort $\text{Nat}$;
cons $\text{zero} : \text{Nat}$;
successor : $\text{Nat} \rightarrow \text{Nat}$;

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

$$\text{successor}(\text{successor}(\ldots \text{successor}($zero$)\ldots)).$$

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 $\text{Nat}$ must differ.

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

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

The sort $\mathbb{B}$ plays a special role. In the first place the semantics of the language prescribes that the constructors $\text{true}$ and $\text{false}$ must be different. (This is the only exception to the rule that constructors are not presumed to be different; using this exception one can build up sorts with necessarily different elements.) So, there are exactly two booleans. In the second place, booleans are used to connect data specification with behavior, namely, conditions in processes and conditional equations must be of sort $\mathbb{B}$. In appendix ?? 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 $f : S \rightarrow S$;

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 $\text{Message}$ without constructors. In this way, one can model for instance data transfer protocols without assuming anything about messages being transferred.

sort $\text{Message}$;

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 $\text{eq} : \text{Nat} \times \text{Nat} \rightarrow \mathbb{B}$;
plus, times : $\text{Nat} \times \text{Nat} \rightarrow \text{Nat}$;

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:
By applying these equations, one can show which terms are equal to others. For instance showing that $2 * 2 = 4$ respectively):

$$
times(succes\text{}\text{sor}(zero), succee(succ\text{}sor(zero)))) =
plus(succ\text{isor}(zero), times(succ\text{isor}(zero), zero)) =
plus(succ\text{isor}(zero), plus(succ\text{isor}(zero), zero)) =
succ\text{esor}(plus(succ\text{isor}(zero), succ\text{esor}(zero), zero)) =
succ\text{esor}(succ\text{esor}(plus(succ\text{esor}(zero), zero)))) =
succ\text{esor}(succe\text{esor}(zero))
$$

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 → B;
var n : Nat;
eqn even(zero) = true;
eqn even(successor(zero)) = false;
eqn even(successor(successor(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, zero) → even(n) = true;
eqn eq(n, successor(zero)) → even(n) = false;
eqn even(successor(successor(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:
Table 3.1: The predefined sorts ($S$ and $T$ are sorts)

<table>
<thead>
<tr>
<th>Sort</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>$\mathbb{B}$</td>
<td>Bool</td>
</tr>
<tr>
<td>Positive numbers</td>
<td>$\mathbb{N}^+$</td>
<td>Pos</td>
</tr>
<tr>
<td>Natural numbers</td>
<td>$\mathbb{N}$</td>
<td>Nat</td>
</tr>
<tr>
<td>Real numbers</td>
<td>$\mathbb{R}$</td>
<td>Real</td>
</tr>
<tr>
<td>Structured types</td>
<td>struct</td>
<td>....</td>
</tr>
<tr>
<td>Functions</td>
<td>$S \rightarrow T$</td>
<td>$S \rightarrow T$</td>
</tr>
<tr>
<td>Lists</td>
<td>List($S$)</td>
<td>List($S$)</td>
</tr>
<tr>
<td>Sets</td>
<td>Set($S$)</td>
<td>Set($S$)</td>
</tr>
<tr>
<td>Bags</td>
<td>Bag($S$)</td>
<td>Bag($S$)</td>
</tr>
</tbody>
</table>

Table 3.1: The predefined sorts ($S$ and $T$ are sorts)

map \( \text{less} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B} \);
var \( n, m : \mathbb{N} \);
eqn \( \text{less}(n, \text{zero}) = \text{false} \);
\( \text{less} (\text{zero}, \text{successor}(n)) = \text{true} \);
\( \text{less}(\text{successor}(n), \text{successor}(m)) = \text{less}(n, m) \);

Now assume that \text{zero} and \text{successor} (\text{zero}) are equal, more precisely,
\( \text{zero} = \text{successor}(\text{zero}) \).

Then, we can derive:
\[
\text{true} = \text{less} (\text{zero}, \text{successor}(\text{zero})) \quad \text{assumption} \quad \text{less} (\text{zero}, \text{zero}) = \text{false}.
\]

So, under this assumption, \text{true} and \text{false} coincide, leading to the conclusion that \text{zero} and \text{successor} (\text{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 \text{less} is present.

**Exercise 3.1.1.** Give equational specifications of ‘greater than or equal’ \( \geq \), ‘smaller than’ \(<\) and ‘greater than’ \( >\) on the natural numbers.

**Exercise 3.1.2.** Give specifications of \( \text{max} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \), \( \text{min} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) and \( \text{power} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) where \( \text{power} (m, n) \) equals \( m^n \).

**Exercise 3.1.3.** Define a sort \( \text{List} \) on an arbitrary non-empty domain \( D \), with as constructors the empty list \( [] \) : \( \text{List} \) and \( \text{in} : D \times \text{List} \rightarrow \text{List} \) to insert an element of \( D \) at the beginning of a list. Extend this with the following non-constructor functions: \( \text{append} : D \times \text{List} \rightarrow \text{List} \) to insert an element of \( D \) at the end of a list; \( \text{top} : \text{List} \rightarrow D \) and \( \text{toe} : \text{List} \rightarrow D \) to obtain the first and the last element of a list, respectively; \( \text{tail} : \text{List} \rightarrow \text{List} \) and \( \text{untoe} : \text{List} \rightarrow \text{List} \) to remove the first and the last element from a list, respectively; \( \text{nonempty} : \text{List} \rightarrow \mathbb{B} \) to check whether a list is empty, \( \text{length} : \text{List} \rightarrow \mathbb{N} \) to compute the length of a list, and \( ++ : \text{List} \times \text{List} \rightarrow \text{List} \) to concatenate two lists.

### 3.2 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 3.1.

All predefined sorts have functions for if-then-else (\( \text{if}(\_\_, \_\,, \_\_, \_\,) \)), 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 \text{true} or \text{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 $\text{if}(\cdot, \cdot, \cdot)$ are as follows:

\[
\begin{align*}
\text{map} & \approx, \not\approx : S \times S \to \mathbb{B}; \\
\text{if} & : \mathbb{B} \times S \times S \to S; \\
\text{var} & x, y : S; \quad b : \mathbb{B}; \\
\text{eqn} & x \approx x = \text{true}; \\
& x \not\approx y = \neg(x \approx y); \\
& \text{if}(\text{true}, x, y) = x; \\
& \text{if}(\text{false}, x, y) = y; \\
& \text{if}(b, x, x) = x; \\
& \text{if}(x \approx y, x, y) = y;
\end{align*}
\]

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 3.2.1.** For any data sort $S$ for which the equations above are defined, it holds that:

\[x \approx y = \text{true} \text{ if } x = y.\]

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

\[x = \text{if}(\text{true}, x, y) = \text{if}(x \approx y, x, y) = y.\]

For the direction from right to left, we derive:

\[\text{true} = x \approx x = x \approx y.\]

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

### 3.2.1 Booleans

The sort boolean is already introduced as $\mathbb{B}$. It consists of exactly two different constructors $\text{true}$ and $\text{false}$. For this sort, the operations are listed in table 3.2, 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 ??.

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.

As demonstrated by the above-given example, 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.

### 3.2.2 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.

35
The operators on numbers are given in table 3.3. 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 \( x \mid n \) 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 ‘\( \ast \)’.

In some cases the sort of the result must be upgraded. For numbers, we have the explicit type conversion operations \( A \to B \) (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 \( \text{Int2Nat}(n-1) \). 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 ???. The numbers have been designed such that each number has a unique and efficient internal representation.

The operator \( x \mid n \) is the modulo operation. For a natural number \( x \) and a positive number \( n \), the expression \( x \mid n \) yields a natural number, being the remainder of \( x \mod n \). This situation is best characterised by the valid equation \( x = n(x \mod n) + x \mid n \).

Calculations with expressions involving modulo operations is very much simplified using the following properties (contributed to Gauss [51]).

1. \( (x \mid nm) \mid n = x \mid n \);
2. \( (x + y) \mid n = (x + y) \mid n \);
3. \( (x \cdot y) \mid n = (x \cdot y) \mid n \);
4. \( x \mid n = x \) if \( 0 \leq x < n \);
5. \( x \mid n < n \).

**Exercise 3.2.2.** What are the sorts of the successor function \( \text{succ}(\_), \) and what are the sorts of the predecessor function \( \text{pred}(\_). \)

**Exercise 3.2.3.** Prove using the equations in appendix ?? that the numbers 0 and 1 are different.

**Exercise 3.2.4.** Prove that \( (1 + x) \mid n = (1 + x) \mid n \).
<table>
<thead>
<tr>
<th>Operator</th>
<th>Rich</th>
<th>Plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>positive numbers</td>
<td>$\mathbb{N}^+$ (1,2,3,...)</td>
<td>Pos.(1,2,3,...)</td>
</tr>
<tr>
<td>natural numbers</td>
<td>$\mathbb{N}$ (0,1,2,...)</td>
<td>Nat.(0,1,2,...)</td>
</tr>
<tr>
<td>integers</td>
<td>$\mathbb{Z}$ (...,−2,−1,0,1,2,...)</td>
<td>Int.(...,−2,−1,0,1,2,...)</td>
</tr>
<tr>
<td>reals</td>
<td>$\mathbb{R}$</td>
<td>Real</td>
</tr>
<tr>
<td>equality</td>
<td>$\approx$</td>
<td>$\ == \ $</td>
</tr>
<tr>
<td>inequality</td>
<td>$\neq$</td>
<td>$\ != \ $</td>
</tr>
<tr>
<td>conditional</td>
<td>$\text{if}(__,__,__)$</td>
<td>$\text{if}(__,__,__)$</td>
</tr>
<tr>
<td>conversion</td>
<td>$A2B(_)$</td>
<td>$A2B(_)$</td>
</tr>
<tr>
<td>less than or equal</td>
<td>$\leq$</td>
<td>$\ &lt;= \ $</td>
</tr>
<tr>
<td>less than</td>
<td>$&lt;$</td>
<td>$\ &lt; \ $</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>$\geq$</td>
<td>$\ &gt;= \ $</td>
</tr>
<tr>
<td>greater than</td>
<td>$&gt;$</td>
<td>$\ &gt; \ $</td>
</tr>
<tr>
<td>maximum</td>
<td>$\max(__,__)$</td>
<td>$\max(__,__)$</td>
</tr>
<tr>
<td>minimum</td>
<td>$\min(__,__)$</td>
<td>$\min(__,__)$</td>
</tr>
<tr>
<td>absolute value</td>
<td>$\text{abs}(__)$</td>
<td>$\text{abs}(__)$</td>
</tr>
<tr>
<td>negation</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>successor</td>
<td>$\text{succ}(__)$</td>
<td>$\text{succ}(__)$</td>
</tr>
<tr>
<td>predecessor</td>
<td>$\text{pred}(__)$</td>
<td>$\text{pred}(__)$</td>
</tr>
<tr>
<td>addition</td>
<td>$__ + __$</td>
<td>$__ + __$</td>
</tr>
<tr>
<td>subtraction</td>
<td>$__ - __$</td>
<td>$__ - __$</td>
</tr>
<tr>
<td>multiplication</td>
<td>$__ _ \ast __$</td>
<td>$__ _ \ast __$</td>
</tr>
<tr>
<td>integer div</td>
<td>$__ _ \text{div} __$</td>
<td>$__ _ \text{div} __$</td>
</tr>
<tr>
<td>integer mod</td>
<td>$__ _ _ \mod __$</td>
<td>$__ _ _ \mod __$</td>
</tr>
<tr>
<td>exponentiation</td>
<td>$__ _ _ \exp(__,__)$</td>
<td>$__ _ _ \exp(__,__)$</td>
</tr>
</tbody>
</table>

Table 3.3: Operations on numbers
Operator | Rich | Plain
--- | --- | ---
function application | $(\ldots,\cdot)$ | $(\cdot,\ldots,\cdot)$
lambda abstraction | $\lambda D_0,\ldots,D_n.\cdot$ | lambda $\cdot$:D0,\ldots,:Dn.\cdot$
equality | $\approx$ | $==$
inequality | $\not\approx$ | $!=\cdot$
conditional | if($\cdot,\cdot,\cdot$) | if($\cdot,\cdot,\cdot$)

Table 3.4: Lambda abstraction and function application

3.3 Function data 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,

\[
\begin{align*}
\text{sort } F &= \mathbb{N} \rightarrow \mathbb{N}; \\
G &= \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}; \\
H &= \mathbb{R} \rightarrow F \rightarrow \text{List}(G);
\end{align*}
\]

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 $\text{List}(G)$. The brackets of function types associate to the right. Hence, the sort $H$ equals $\mathbb{R} \rightarrow (F \rightarrow \text{List}(G))$. If the sort $(\mathbb{R} \rightarrow F) \rightarrow \text{List}(G)$ were required, explicit bracketing is needed.

Functions can be made using the lambda abstraction and application (see table 3.4). 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 $((f)(g))(h)$.

**Exercise 3.3.1.** Specify a function `map` that gets a function and applies it to all elements of a given list.

**Exercise 3.3.2.** 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.

3.4 Structured data 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 = struct up?isUp | down?isDown | left?isLeft | right?isRight;
```
This says the sort \textit{Direction} has four constructors characterising different elements. The optional recognisers such as \textit{isUp} are functions from \textit{Direction} to \mathbb{B} and yield true iff they are applied to the constructor to which they belong. E.g., \textit{isUp}(\textit{up}) = \textit{true} and \textit{isUp}(\textit{down}) = \textit{false}.

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

\begin{verbatim}
sort Pair = struct pair (fst: A, snd: B);
\end{verbatim}

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

\begin{align*}
\textit{fst}(\textit{pair}(a,b)) &= a; \\
\textit{snd}(\textit{pair}(a,b)) &= b;
\end{align*}

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 \textit{Tree} for binary trees has the following minimal definition:

\begin{verbatim}
sort Tree = struct leaf(A) | node(Tree, Tree);
\end{verbatim}

By adding projection and recogniser functions it looks like:

\begin{verbatim}
sort Tree = struct leaf(val:A)?isLeaf | node(left:Tree, right:Tree)?isNode;
\end{verbatim}

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

\begin{verbatim}
map HE : (A → \mathbb{B}) × Tree → \mathbb{B};
var f : A → \mathbb{B};
    t, u : Tree;
    a : A;
eqn HE(f, leaf(a)) = f(a);
    HE(f, node(t, u)) = HE(f, t) ∧ HE(f, u);
\end{verbatim}

The following definition of sort \textit{Tree} allows the definition of operation \textit{HE} without pattern matching.

\begin{verbatim}
var f : A → \mathbb{B};
    t : Tree;
eqn HE(f, t) = if(isLeaf(t), f(val(t)), f(HE(f, left(t))) ∧ f(HE(f, right(t))));
\end{verbatim}

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{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Operator & Rich & Plain \\
\hline
constructor of summation \(i\) & \(\text{ci}(\_, \_, \ldots, \_)\) & \(\text{ci}(\_, \ldots, \_)\) \\
recogniser for constructor \(i\) & \(\text{is}_i(\_)\) & \(\text{is}_i(\_)\) \\
projection \((i, j)\), if declared & \(\text{pr}_{i,j}(\_)\) & \(\text{pr}_{i,j}(\_)\) \\
equality & \(\_ == \_\) & \(\_ == \_\) \\
inequality & \(\_ != \_\) & \(\_ != \_\) \\
conditional & \(\text{if}(\_ \ldots, \_\ldots, \_)\) & \(\text{if}(\_ \ldots, \_, \_)\) \\
\hline
\end{tabular}
\caption{Operators for structured types}
\end{table}
Operator | Rich | Plain
--- | --- | ---
construction | \[\_\cdots\_] | \[\_\cdots\_\]
element test | \(\_\in\_\) | \(\_\text{ in }\_\)
length | \#\_ | \#\_
cons | \(\_\rhd\_\) | \(\_\mid\_\)
snoc | \(\_\rhd\_\) | \(\_\mid\_\)
concatenation | \(\_\rhd\_\) | \(\_\rhd\_\)
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 | \(\_\approx\_\) | \(\_\approx\_\)
inequality | \(\_\not\approx\_\) | \(\_\not\approx\_\)
conditional | if(\_,\_,\_) | if(\_,\_,\_,\_) | if(\_,\_,\_,\_,\_)

Table 3.6: Operations on lists

\[\text{struct } c_1(pr_{1,1} : A_{1,1}, \ldots, pr_{1,k_1} : A_{1,k_1})?\text{isC}_1\\| c_2(pr_{2,1} : A_{2,1}, \ldots, pr_{2,k_2} : A_{2,k_2})?\text{isC}_2\\| \cdots\\| c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?\text{isC}_n;\]

This declares \(n\) constructors \(c_i\), projection functions \(pr_{i,j}\) and recognisers \(\text{isC}_i\). All names have to be chosen such that no ambiguity can arise. The operations in table 3.5 are available after declaring the sort above.

Exercise 3.4.1. Define the sort \(\text{Message}\) that contains message frames with a header containing the type of the message (\(\text{ack}, \text{ctrl}, \text{mes}\)), checksum field, and optionally a data field. Leave the data and checksums unspecified.

### 3.5 Constructed data types

#### 3.5.1 Lists

Lists, where all elements are of sort \(A\), are declared by the sort expression \(\text{List}(A)\). The operations in table 3.6 are predefined for this sort. Lists consist of constructors \([\_]\), the empty list, and \(\rhd\), 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. \([2, 4, 1] \_\) equals 4). The concatenation operator ++ can be used to append one list to the end of another. The \(\_\) operator can be used to put an element to the end of a list. So, the lists \([a, b], a \rhd [b], [a] \_\) and \([a]+[b] \_\) are all equivalent. The precise equations for lists are given in appendix ??.

Exercise 3.5.1. Specify a function \(\text{stretch}\) that given a list of lists of some sort \(S\), concatenates all these lists to one single list.

Exercise 3.5.2. Define an \(\text{insert}\) 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 3.5.3. Define an \(\text{insert}\) function on lists of naturals such that the list is sorted. The smallest elements must occur first in a list. Define a function \(\text{isIn}\) that checks whether a number \(n\) occurs in an
ordered list $l$ using the smallest number of steps necessary. Prove that the functions $\text{insert}$ and $\text{is\_in}$ behave exactly the same as the functions $\triangleright$ and $\in$ from appendix ??.

### 3.5.2 Sets and bags

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

$$
\text{sort } S = \text{Set}(A);
B = \text{Bag}(A);
$$

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 \(\{n: \mathbb{N} \mid \text{true}\}\). Similarly, the infinite set of all even numbers is easily denoted as \(\{n: \mathbb{N} \mid n \mid 2 \approx 0\}\). 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 3.5.4.** Specify the set of all prime numbers.

**Exercise 3.5.5.** 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 $2$.

### 3.6 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 3.8 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 ⊲, ⊳ 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 + +:

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>plain</th>
<th>associativity</th>
</tr>
</thead>
</table>
| * | .div, | .
| +, − | +, − | left
| ⊳ | |
| ⊲ | |
| ⊤ | |
| ⊥ | |
| <, >, ≤, ≥, ∈ | <, >, ≤, ≥, in |
| ≈, ≠ | ≈, ≠ |
| ∧, ∨ | ∧, ∨ |
| ⇒ | ⇒ |

Table 3.8: 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 $e \text{ where } a_1 = e_1, \ldots, a_n = e_n \text{ end}$, with $n ∈ \mathbb{N}$. 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.
Chapter 4

Sequential processes

In chapter 2, we described behaviour by labelled transition systems. If behaviour becomes more complex this technique falls short and we need a proper syntax to describe complex labelled transition systems. In this chapter we describe processes using an extended process algebra. The building blocks of our process algebra are (multi-)actions with data. There are operators to combine behaviour sequentially, nondeterministically and in parallel. The language defined henceforth (in this chapter and the next one) provides us with several means to model reactive systems.

We start in this chapter with a language for basic processes, i.e., processes without parallelism. Axioms are used to characterise the meaning of the various constructs.

4.1 Actions

As in chapter 2, 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 \texttt{receive} action can carry a message, and an \texttt{error} action can carry a natural number, for instance indicating its severity. Actions can have any number of parameters. They are declared as follows:

\begin{verbatim}
act timeout;
    error : \mathbb{N};
    receive : \mathbb{B} \times \mathbb{N}^+;
\end{verbatim}

This declares parameterless action name \texttt{timeout}, action name \texttt{error} with a data parameter of sort \mathbb{N} (natural numbers), and action name \texttt{receive} with two parameters of sort \mathbb{B} (booleans) and \mathbb{N}^+ (positive numbers) respectively. For the above action name declaration, \texttt{timeout}, \texttt{error}(0) and \texttt{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:

\begin{verbatim}
act a_{begin}, a_{end};
\end{verbatim}

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.
### 4.2 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:

\[
\alpha ::= \tau \mid a \mid a(\vec{d}) \mid \alpha \mid \beta,
\]

where \(a\) denotes an action name and \(\vec{d}\) a vector of data parameters. The term \(\tau\) 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 \(\alpha \mid \beta\) consists of the actions from both the multi-actions \(\alpha\) and \(\beta\), which all must happen simultaneously.

Typical examples of multi-actions are \(\tau\), \(\text{error} \mid \text{error} \mid \text{send}(\text{true})\), \(\text{send}(\text{true}) \mid \text{receive}(\text{false}, 6)\) and \(\tau \mid \text{error}\). 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 \(\tau\) 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 \(\alpha\) that associates with a multi-action \(\alpha\) the multiset of action names that is obtained by omitting all data parameters that occur in \(\alpha\). 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.

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

**Table 4.1: Axioms for multi-actions**

**Exercise 4.2.1.** Simplify the following expressions using the equations in table 4.1.

1. \((a(1)\mid b(2)) \setminus (b(2)\mid a(1))\).
2. \((a(1)\mid b(2)) \setminus (b(3)\mid c(2))\).

---

\(^1\) BNF stands for Backus-Naur Form which is a popular notation to denote context free grammars.
Table 4.2: Axioms for the basic operators

3. \( a(1) \mid b(2) \subseteq b(2) \).

**Exercise 4.2.2.** Prove using induction on the structure of multi-actions that the following three properties hold for all multi-actions \( \alpha \) and \( \beta \):

1. \( \alpha \setminus \alpha = \tau \).
2. \( \alpha \sqsubseteq \alpha = \text{true} \).
3. \( (\alpha \mid \beta \mid \gamma) \setminus \beta = \alpha \mid \gamma \).

### 4.3 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](image-url)
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 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 it does not matter whether behaviour stands at the left or right of the choice, and that brackets to group more than one choice 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 in general not behaviourally equivalent, as argued in the previous chapter (see figure 2.9). 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 axiom A6 is designated with \( \boxdot \) to indicate that it is only sound in the untimed settings; a weaker version of this axiom, which holds also in the timed settings is presented in chapter ??.

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. 2.8). This goes as follows:

\[
(a\cdot b + a\cdot b) \triangleq A3 \triangleright a\cdot b + a\cdot b.
\]

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.

\[
((a + b)\cdot c + a\cdot c\cdot d) \triangleq A4 \triangleright (a\cdot c + b\cdot c + a\cdot c)
\bigtriangleup A3 \triangleright (a\cdot c + b\cdot c)
\bigtriangleup A4 \triangleright (a + b)\cdot c
\bigtriangleup A5 \triangleright (b + a)\cdot (c\cdot d).
\]

**Exercise 4.3.1.** 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 4.3.2.** Prove that if \( x \subseteq y \) and \( y \subseteq x \), then \( x = y \).
4.4 Deadlock

A remarkable but very essential process is deadlock, also called inaction. This is denoted as $\delta$ 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 $\alpha$ and $\delta$, we must choose $\alpha$ because $\delta$ has no first action that would cause the $\delta$ 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 $\text{error} \cdot \delta$. 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 $\delta$ exists. One feature of $\delta$ 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 $x + \delta = x$.

4.5 The conditional and sum operator

Data influences the run of processes using the conditional operator. For a condition $c$ of sort $B$ (boolean) and processes $p$ and $q$ we write $c \rightarrow p \odot q$ 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:

$$(\text{waterLevel}> \text{limit}) \rightarrow \text{soundAlarm} \odot \text{sendOK}.$$ 

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

$$(\text{desiredColor} \approx \text{Red}) \rightarrow \text{setSignalToRed} + (\text{desiredColor} \approx \text{Yellow}) \rightarrow \text{setSignalToYellow} + (\text{desiredColor} \approx \text{Green}) \rightarrow \text{setSignalToGreen}$$

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 $c \rightarrow x \odot x = x$ we must show true $\rightarrow x \odot x = x$, which follows directly from Cond1, and false $\rightarrow x \odot x = x$ which follows directly from Cond2.
Exercise 4.5.1. Derive the following equations:

1. \( c \to x \odot y = \neg c \to y \odot x \);
2. \( c \lor c' \to x \odot y = c \to x \odot (c' \to y) \);
3. \( x + y \geq c \to x \odot y \);
4. if assuming that \( c \) holds, we can prove that \( x = y \) then \( c \to x \odot z = c \to y \odot z \).

The sum operator \( \sum_{d \in D} p(d) \) is a generalisation of the choice operator. The notation \( p(d) \) is used to stress that \( d \) can occur in the process \( p \). Where \( p \) and \( q \) allows a choice among processes \( p \) and \( q \), \( \sum_{d \in D} p(d) \) 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:

\[
\sum_{c \in \mathbb{B}} p(c) = p(\text{true}) + p(\text{false}).
\]

For sums over infinite domains, e.g. \( \sum_{n \in \mathbb{N}} p(n) \), 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:

\[
\sum_{n, m : \text{Message}} \text{read}(m) \cdot \text{forward}(m).
\]

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 \( \text{read}_1 \) and \( \text{read}_2 \).

\[
\sum_{n_1, m_1 : \text{Message}} \text{read}_1(m_1) \cdot \sum_{n_2, m_2 : \text{Message}} \text{read}_2(m_2) \cdot \text{forward}(m_1, m_2).
\]

In the first (correct) process, the message \( m_2 \) is chosen when the action \( \text{read}_2 \) takes place. In the second process, the message \( m_2 \) to be read is chosen when action \( \text{read}_1 \) takes place. When doing \( \text{read}_2 \), 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 ?? . 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.

Below, we describe a process that maintains an unbounded array in which natural numbers can be stored. There are actions \( \text{set}, \text{get} \) and \( \text{show} \). The action \( \text{set}(n, m) \) sets the \( n \)th entry of the array to value \( m \). After an action \( \text{get}(n) \) an action \( \text{show}(m) \) shows the value \( m \) stored at position \( n \) in the array.

\[
\begin{align*}
\text{act} & \quad \text{set} : \mathbb{N} \times \mathbb{N}; \\
& \quad \text{get}, \text{show} : \mathbb{N}; \\
\text{proc} & \quad P(a) : \mathbb{N} \to \mathbb{N} \\
& \quad = \sum_{n, m : \mathbb{N}} \text{set}(n, m) \cdot P(\lambda z : \mathbb{N}. \text{if}(z \approx n, m, a(z))) \\
& \quad + \sum_{n : \mathbb{N}} \text{get}(n) \cdot \text{show}(a(n)) \cdot P(a);
\end{align*}
\]

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 \( \text{sorted} \) expresses that the numbers in an array are increasing and the predicate \( \text{equalcontents} \) expresses that each of the arrays \( a \) and \( a' \) contain the same elements. But note that \( \text{equalcontents} \) does not preserve the number of occurrences of numbers.
\[ \text{act} \quad \text{read}, \text{deliver} : \mathbb{N} \rightarrow \mathbb{N}; \]
\[ \text{map} \quad \text{sorted}, \text{equalcontents}, \text{includes} : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}; \]
\[ \text{var} \quad a, a' : \mathbb{N} \rightarrow \mathbb{N}; \]
\[ \text{eqn} \quad \text{sorted}(a) = \forall i : \mathbb{N}. a(i) \leq a(i+1); \]
\[ \text{equalcontents}(a, a') = \text{includes}(a, a') \land \text{includes}(a', a); \]
\[ \text{includes}(a, a') = \forall i : \mathbb{N}. \exists j : \mathbb{N}. a(i) \approx a'(j); \]
\[ \text{proc} \quad P = \sum_{a : \mathbb{N} \rightarrow \mathbb{N}} \text{read}(a) \cdot \sum_{a' : \mathbb{N} \rightarrow \mathbb{N}} (\text{sorted}(a') \land \text{equalcontents}(a, a')) \rightarrow \text{deliver}(a') \cdot P; \]

Exercise 4.5.2. 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.

### 4.6 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:

\[ \text{act} \quad \text{set}, \text{alarm}, \text{reset}; \]
\[ \text{proc} \quad P = \text{set} \cdot Q; \]
\[ Q = \text{reset} \cdot P + \text{alarm} \cdot Q; \]

This declares process variables \( P \) and \( Q \) (often just called processes). Note that we use the keyword \texttt{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 \texttt{init} can be used to indicated the initial behaviour. In accordance with figure 2.2 this ought to be variable \( P \).

\[ \text{init} \quad P; \]

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.

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

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,
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 4.6.1.** 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.

**Exercise 4.6.2.** 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.
Chapter 5

Parallel processes

In chapter 4, we showed how it is possible to describe sequential processes that can interact with their environment. In this 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.

5.1 The parallel operator

<table>
<thead>
<tr>
<th>M</th>
<th>x ∥ y = x ∥ y + y ∥ x + x ∥ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM1</td>
<td>α ∥ x = α·x</td>
</tr>
<tr>
<td>LM2</td>
<td>δ ∥ x = δ</td>
</tr>
<tr>
<td>LM3</td>
<td>α·x ∥ y = α·(x ∥ y)</td>
</tr>
<tr>
<td>LM4</td>
<td>(x + y) ∥ z = x ∥ z + y ∥ z</td>
</tr>
<tr>
<td>LM5</td>
<td>(Σ_d:D X(d)) ∥ y = Σ_d:D X(d) ∥ y</td>
</tr>
<tr>
<td>S1</td>
<td>x∥y = y∥x</td>
</tr>
<tr>
<td>S2</td>
<td>(x∥y)∥z = x∥(y∥z)</td>
</tr>
<tr>
<td>S3</td>
<td>x∥τ = x</td>
</tr>
<tr>
<td>S4</td>
<td>α∥δ = δ</td>
</tr>
<tr>
<td>S5</td>
<td>(α·x)∥β = α∥β·x</td>
</tr>
<tr>
<td>S6</td>
<td>(α·x)(β·y) = α·β·(x ∥ y)</td>
</tr>
<tr>
<td>S7</td>
<td>(x + y)∥z = x∥z + y∥z</td>
</tr>
<tr>
<td>S8</td>
<td>(Σ_d:D X(d))∥y = Σ_d:D X(d)∥y</td>
</tr>
<tr>
<td>TC1</td>
<td>(x ∥ y) ∥ z = x ∥ (y ∥ z)</td>
</tr>
<tr>
<td>TC2</td>
<td>x ∥ δ = x·δ</td>
</tr>
<tr>
<td>TC3</td>
<td>(x∥y) ∥ z = x∥(y∥z)</td>
</tr>
</tbody>
</table>

Table 5.1: Axioms for the parallel composition operators
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 5.1 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 $(\llfloor)$, the synchronisation merge $(\mid)$ and the before operator $(\llless)$, which is specially introduced to deal with time.

The process $p \parallel q$ (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 $p\mid q$ (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, these are two different operators, we use them interchangeably as their meaning in both cases is the same. More concretely, $a\mid b$ both represents a multi-action and a synchronisation of two processes both consisting of a single action.

The first axiom marked $M$ in table 5.1 characterises our view on parallelism. The first action in $x \mid y$ 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 $\alpha$ must happen before the process $x$ must do an action. The axiom $LM\exists\varnothing$ is only valid in a timed setting. The variant valid in a timed setting can be found in chapter ??.

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:

$$a\cdot b \parallel c\cdot d = a \llfloor (b \mid c\cdot d) + c \llfloor (a\cdot b \mid d) + a\cdot b|c\cdot d =$$

$$a\cdot (b \mid c\cdot d) + c \llfloor (a\cdot b \mid d) + d \llfloor a\cdot b + a\cdot b|d) + (a|c)-(b \mid d) =$$

$$a\cdot (b\cdot c\cdot d + c\cdot (b\mid d) | (b|c\cdot d) + c\cdot (a\cdot b \mid d) + d \cdot a\cdot b + (a|d\cdot b) + (a|b)-(b\cdot c\cdot d + (b\cdot d + b\cdot b|d)) + (a\cdot b\cdot d + b\cdot b) + (a|c)-(b\cdot d + b\cdot b|d)$$

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 5.1).

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: $|, \cdot, \cdot, (\gg, \llless)$, $\rightarrow, \{\llfloor, \|\}, \sum, +$.

Exercise 5.1.1. Expand the process $a\cdot b \parallel c$. Indicate precisely which axioms have been used.

Exercise 5.1.2. Give a rough estimate the size of the expansion of $a\cdot a \parallel b\cdot b \mid c\cdot c\cdot c$.

Exercise 5.1.3. Prove that the parallel operator is both commutative and associative, i.e. $x \mid y = y \mid x$ and $x \mid (y \mid z) = (x \mid y) \mid z$.

5.2 Communication among parallel processes

Processes that are put in parallel can execute actions simultaneously, resulting in multi-actions. The communication operator $Γ_C(p)$ 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
to this process. We get the state space as depicted in figure 5.1. As a next operation we say

\[ \text{communicates with} \]

\[ \text{c} \]

\[ \text{p} \]

\[ \text{ters of the multi-actions in} \]

\[ \gamma \]

\[ \text{would not necessarily hold. Hence,} \]

\[ \text{allow operator} \]

\[ \text{exactly which multi-actions from} \]

\[ \text{cannot enforce communication. We must explicitly allow those actions that we want to see (e.g. communications), and implicitly block other actions.} \]

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

\[
\begin{align*}
\gamma_\emptyset(\alpha) &= \alpha \\
\gamma_{C_1 \cup C_2}(\alpha) &= \gamma_{C_1}(\gamma_{C_2}(\alpha)) \\
\gamma_{\{a_1|\cdots|a_n\rightarrow b\}}(\alpha) &= \begin{cases} 
\gamma_{\{a_1|\cdots|a_n\rightarrow b\}}(\alpha \setminus \{a_1(1)|\cdots\}|a_n(d))) & \text{if } a_1(1)|\cdots\]|a_n(d) \subseteq \alpha \text{ for some } d. \\
\alpha & \text{otherwise.}
\end{cases}
\end{align*}
\]

For example, \( \gamma_{\{a_b\rightarrow c\}}(aיבc) = aיבc \) and \( \gamma_{\{a_1|\cdots|a_n\rightarrow b\}}(aיבd럽c) = aיבc \). 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) and a right hand side of a communication cannot occur in a left hand side. 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 \( \gamma_C \) would not be a properly defined function.

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. \( \nabla_{\{b\}}(a|b\rightarrow c, a|d\rightarrow e) \) is not allowed. The empty multi-action \( \tau \) is not allowed to occur in set \( V \) because it cannot be blocked. The axioms are given in table 5.3.

So, consider again the process \( a\cdot b \parallel c\cdot d \) as depicted in figure 5.1. 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 \( \Gamma_{\{a|c\rightarrow e, b|d\rightarrow f\}} \) to this process. We get the state space as depicted in figure 5.2 at the left. As a next operation we say

\[
\begin{array}{c|c}
C_1 & \Gamma_C(\alpha) = \gamma_C(\alpha) \\
C_2 & \Gamma_C(\delta) = \delta \\
C_3 & \Gamma_C(x+y) = \Gamma_C(x) + \Gamma_C(y) \\
C_4 & \Gamma_C(x\cdot y) = \Gamma_C(x) \cdot \Gamma_C(y) \\
C_5 & \Gamma_C(\sum_{d,D} X(d)) = \sum_{d,D} \Gamma_C(X(d))
\end{array}
\]

Table 5.2: Axioms for the communication operator

synchronise. Here \( C \) is a set of allowed communications of the form \( a_1|\cdots|a_n\rightarrow c \), with \( n \geq 1 \) 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 5.2.

Figure 5.1: The behaviour of \( a\cdot b \parallel c\cdot d \)
\( \nabla_V(\alpha) = \alpha \text{ if } \alpha \in V \cup \{ \tau \} \)

\( \nabla_V(x + y) = \nabla_V(x) + \nabla_V(y) \)

\( \nabla_V(x \cdot y) = \nabla_V(x) \cdot \nabla_V(y) \)

\( \nabla_V(\sum_d D X(d)) = \sum_d \nabla_V(X(d)) \)

\( \nabla_V(\nabla_W(x)) = \nabla_{V \cap W}(x) \)

Table 5.3: Axioms for the allow operator

---

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 5.2 at the right belongs to the expression \( \nabla_{(e,f)}(\Gamma_{[a|c\rightarrow e,b|d\rightarrow f]}(a \cdot b \parallel c \cdot d)) \).

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:

\[
\text{act } r_1, s_2, s_3, r_3, c_3 : D;
\]

\[
\text{proc } X = \sum_{d:D} (r_1(d) + r_3(d)) \cdot (s_2(d) + s_3(d)) \cdot X;
\]

\[
B = \sum_{d:D} r_3(d) \cdot s_3(d) \cdot B;
\]

Consider the behaviour \( S = \nabla_{\{r_1, r_2, c_2\}}(\Gamma_{\{r_3|s_3|c_3\}}(X \parallel B)) \). In order to depict the labelled transition system we let \( D \) be equal to \( \{d_1, d_2\} \). In figure 5.3 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 5.2.1. 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 3, or it returns this datum to \( X \) via channel 2. In the latter case, \( X \) returns the datum to \( Y \) via channel 2.

\[
\begin{align*}
X &\xrightarrow{r_1, s_2, r_2, c_2, s_3, D} Y \\
Y &\xrightarrow{r_3, d_1} X
\end{align*}
\]

\( X \) and \( Y \) are defined by the following recursive specification:

\begin{align*}
\text{act} &\quad r_1, s_2, r_2, c_2, s_3 : D; \\
\text{proc} \quad X = \sum_{d : D} (r_1(d) + r_2(d)) \cdot s_2(d) \cdot X; \\
&\quad Y = \sum_{d : D} r_2(d) \cdot (s_3(d) + s_2(d)) \cdot Y;
\end{align*}

Let \( S \) denote \( \nabla_{\{r_1, s_2, c_3\}} (\Gamma_{\{r_3, s_3 \to c_3\}} (X \parallel Y)) \), 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 5.2.2. 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.

\[
\begin{align*}
&\quad Y \xrightarrow{1} X \xrightarrow{2} Y \xrightarrow{3} Z \\
&\quad Z \xrightarrow{4} Y \xrightarrow{2} X \xrightarrow{3} Z \xrightarrow{5}
\end{align*}
\]

(1) Specify the independent processes \( X, Y \) and \( Z \) 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.
5.3 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 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 5.4.

| 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  | $\partial_B(x-y) = \partial_B(x) \cdot \partial_B(y)$       |
| 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  | $\partial_B(\alpha|\beta) = \partial_B(\alpha)|\partial_B(\beta)$  |
| E10 | $\partial_H(\partial_H-(x)) = \partial_H-\partial_H-(x)$ |

Table 5.4: Axioms for the blocking operator

The renaming operator $\rho_R$ is used to rename action names. The set $R$ contains renamings of the form $a \rightarrow b$. 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. $\rho(a(0) + a(3)) = b(0)$. To avoid unclarieties, every action name may only occur once as a left-hand side of $a \rightarrow b$ in $R$. All renamings are applied simultaneously, i.e., a renamed action cannot be renamed twice in one application of the renaming operator. So $\rho_{\{a \rightarrow b, b \rightarrow c\}}$ renames action label $a$ to $b$, not to $c$. The axioms are given in table 5.5.

| R1  | $\rho_R(\tau) = \tau$                          |
| R2  | $\rho_R(a(d)) = b(d)$ if $a \rightarrow b \in R$ for some $b$ |
| R3  | $\rho_R(a(d)) = a(d)$ if $a \rightarrow b \notin R$ 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  | $\rho_R(\sum_{d:D} X(d)) = \sum_{d:D} \rho_R(X(d))$ |

Table 5.5: Axioms for the renaming operator

5.4 Hiding internal behaviour

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 $\tau_I$ is defined. The action names in the set $I$ are removed from multi-actions. So, $\tau_I(a) = b$ and $\tau_I(a) = \tau$. The axioms for hiding are listed in table 5.6.

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 $\tau$. For this purpose the straightforward pre-hide operator $\tau_U$ 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 $\tau_{U\{int\}}(x) = \tau_{(int)}(\tau_U(x))$. The axioms for the pre-hide operator are in table 5.7.
Exercise 5.4.1. Derive the following equations using the axioms valid in rooted branching bisimulation.

1. \( a \cdot (\tau \cdot b + b) = a \cdot b; \)
\[ x \cdot \tau = x \]
\[ \tau \cdot x = \tau \cdot x + x \]
\[ a \cdot (\tau \cdot x + y) = a \cdot (\tau \cdot x + y) + a \cdot x \]

Table 5.9: Axioms for \( \tau \), valid in weak bisimulation for untimed processes

| Failures equivalence | \[ a \cdot (b \cdot x + u) + a \cdot (b \cdot y + v) = a \cdot (b \cdot x + b \cdot y + u) + a \cdot (b \cdot x + b \cdot y + v) \]
| Trace equivalence    | \[ a \cdot x + a \cdot (y + z) = a \cdot x + a \cdot (x + y) + a \cdot (y + z) \]
| Weak trace equivalence| \[ x \cdot (y + z) = x \cdot y + x \cdot z \]

Table 5.10: Axioms for some other equivalences for untimed processes

(2) \[ a \cdot (\tau \cdot (b + c) + b) = a \cdot (\tau \cdot (b + c) + c); \]
(3) \[ \tau \cdot d \left( a \cdot (b + c) + b \right) = \tau \cdot d \left( d \cdot (b + c) + c \right); \]
(4) If \( y \subseteq x \), then \( \tau \cdot (\tau \cdot x + y) = \tau \cdot x \).

See exercise 4.3.2 for the definition of \( \subseteq \).

Exercise 5.4.2. Consider the labelled transition system drawn for the system in example 5.2.1 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?

5.5 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.

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

- \( \alpha(\alpha) = \{ \alpha \} \) if \( \alpha \neq \tau \).
- \( \alpha(\tau) = \emptyset \).
- \( \alpha(\delta) = \emptyset \).
- \( \alpha(x + y) = \alpha(x) \cup \alpha(y) \).
- \( \alpha(x \cdot y) = \alpha(x) \cup \alpha(y) \).
- \( \alpha(\sum_{d \in D} X(d)) = \alpha(X(e)) \) where \( e \) is an arbitrary data term of sort \( D \).
- \( \alpha(x \cdot t) = \alpha(x) \).
Table 5.11: Alphabet Axioms
We define the set with multi-sets of action names $\mathcal{N}(V)$ of actions as follows:

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

We define the set with multi-sets of action names $\psi(V)$ by $\psi(V) = \{\beta \subseteq \alpha \mid \alpha \in V\}.

Let $C = \{a_1^1 \ldots a_n^1 \rightarrow a_1^2 \ldots , a_n^2 \ldots a_n^n \rightarrow a^n\}$ 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 $\text{dom}(C)$ and $\text{rng}(C)$ as follows:

$$\text{dom}(C) = \{a_1^1 \ldots a_n^1, \ldots , a_1^n \ldots a_n^n\}$$

$$\text{rng}(C) = \{a_1 \ldots , a_n\}$$

If $R$ is a renaming we write $R(a(d_1, \ldots , d_n)) = b(d_1, \ldots , d_n)$ if $a \rightarrow b \in R$. If $\alpha$ is a multi-action, we apply $R$ to all individual action names. I.e., if $\alpha$ 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 $\alpha \rightarrow b \in C$.

**Example 5.5.3.** We show how the alphabet axioms can be used to reduce the number of multi-actions when simplifying a process. Consider

$$\nabla_{\{a,d\}}(\Gamma_{\{b|c\rightarrow d\}}(a \parallel b \parallel c)). \quad (5.1)$$

Straightforward expansion of the parallel operators yields the following term:

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

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 (5.1) to:

$$\nabla_{\{a,d\}}(a \parallel \nabla_{\{a,d\}}(\Gamma_{\{b|c\rightarrow d\}}(b \parallel c))).$$
Expansion of the innermost $b \parallel c$ yields $b \cdot c + c \cdot d + b \cdot c$ and application of the communication and allow operator shows that (5.1) 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 5.5.4.** Simplify $\nabla_{\{e,f\}}(\Gamma_{\{a \parallel b \parallel c \parallel d \parallel e \parallel f\}}(a \parallel b \parallel c \parallel d))$. 

Chapter 6

The modal $\mu$-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 ??.

6.1 Hennessy-Milner logic

Hennessy-Milner logic is the underlying modal logic for our property language [29]. Its syntax is given by the following BNF grammar:

$$\phi ::= \text{true} | \text{false} | \neg \phi | \phi \land \phi | \phi \lor \phi | (\alpha)\phi | [\alpha]\phi.$$

The modal formula $\text{true}$ is true in each state of a process and $\text{false}$ is never true. The connectives $\land$ (and), $\lor$ (or) and $\neg$ (not) have their usual meaning. E.g. the formula $\phi_1 \land \phi_2$ is valid wherever both $\phi_1$ and $\phi_2$ 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 $(\alpha)\phi$ is valid whenever an $\alpha$-action can be performed such that $\phi$ is valid after this $\alpha$ has been done. So, the formula $(\alpha)\phi$ expresses that a process can do an $\alpha$ followed by $\phi$ followed by $c$. 

63
Using the connectives more complex properties can be formulated. Expressing that after doing an action \( a \) both a \( b \) and \( c \) must be possible is done by the formula \( (a)(b)true \wedge (c)true \). Expressing that after an action \( a \) no \( b \) is possible can be done by \( (a)\neg(b)true \).

The box modality \([a]\phi\) is more involved. It is valid when for every action \( a \) that can be done, \( \phi \) holds after doing that \( a \). So, the formula \([a]\langle b \rangle true\) says that whenever an \( a \) can be done, \( b \) a 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, \([a]\langle b \rangle false\) 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 \( \langle a \rangle \phi \) is valid and \( [a]\phi \) is not. In the second transition system the situation is reversed, namely \([a]\phi \) is valid \( \langle a \rangle \phi \) is not. Both formulas are true in the third transition system and both are invalid in the fourth.

\[
\begin{array}{c}
\phi \\
a \quad a \\
\neg \phi \\
\end{array}
\quad
\begin{array}{c}
\phi \\
a \\
\end{array}
\quad
\begin{array}{c}
\phi \\
a \\
\neg \phi \\
\end{array}
\quad
\begin{array}{c}
\phi \\
a \\
\end{array}
\quad
\begin{array}{c}
\phi \\
\neg \phi \\
\end{array}
\]

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

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 \( \neg \langle a \rangle \false \) can be hard to comprehend. Yet the equivalent \([a]\langle b \rangle true\) clearly says that whenever an action \( a \) can be done, an action \( b \) must be possible after that \( a \). 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.

\[
\begin{align*}
\neg \langle a \rangle \phi &= [a]\neg \phi \\
\neg [a]\phi &= \langle a \rangle \neg \phi \\
\langle a \rangle \false &= \false \\
[a]true &= true \\
\langle a \rangle (\phi \vee \psi) &= \langle a \rangle \phi \vee \langle a \rangle \psi \\
[a](\phi \wedge \psi) &= [a]\phi \wedge [a]\psi \\
\langle a \rangle \phi \wedge [a]\psi &= \langle a \rangle(\phi \wedge \psi)
\end{align*}
\]

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

**Exercise 6.1.1.**

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 6.1.2.** Give an argument why the following two formulas are equivalent.

\( \langle a \rangle (b)true \vee (c)false \), \quad \neg[a] (b)false \wedge (c)true \)
6.2 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 [37], 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:

$$\alpha ::= a_1 | \cdots | a_n \mid true \mid false \mid \bar{\alpha} \mid \alpha \land \alpha \mid \alpha \lor \alpha.$$ 

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 $(true)(a)true$ expresses that an arbitrary action followed by an action $a$ can be performed. The formula $[true]false$ expresses that no action can be done.

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

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

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

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

$$R ::= \varepsilon \mid \alpha \mid R ; R \mid R^+ \mid R^*.$$ 

The formula $\varepsilon$ 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 $R_1 ; R_2$ represents the concatenation of the sequences of actions in $R_1$ and $R_2$. For instance, $(a\cdot b\cdot c)true$ is the same as $\langle a \rangle \langle b \rangle \langle c \rangle true$ 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, $[a \cdot b + c \cdot d]false$ expresses that neither the sequence $a \cdot b$ nor the sequence $c \cdot d$ is possible.

The definitions of both operators is the following:

$$\langle R_1 + R_2 \rangle \phi = \langle R_1 \rangle \phi \lor \langle R_2 \rangle \phi \quad \quad [R_1 + R_2] \phi = [R_1] \phi \land [R_2] \phi$$

$$\langle R_1 ; R_2 \rangle \phi = \langle R_1 \rangle \langle R_2 \rangle \phi \quad \quad [R_1 ; R_2] \phi = \langle R_1 \rangle [R_2] \phi.$$ 

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 $R^*$ and $R^+$ this improves substantially, because they allow iterative behaviour.

The regular formula $R^*$ denotes zero or more repetitions of the sequences in $R$. Similarly, the formula $R^+$ stands for one or more repetitions. So, $\langle a^* \rangle true$ expresses that any sequence of $a$ actions is possible. And $[a^*] \phi$ expresses that the formula $\phi$ 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 $\phi$ 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 $\phi$ holds. Using regular formulas these can be written as follows:

$$\Box \phi = [true^*] \phi \quad \quad \Diamond \phi = \langle true^* \rangle \phi.$$ 

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 enters without a leave action in between:

\[ [\text{true}^* \cdot \text{enter} \cdot \text{leave} \cdot \text{enter}] \text{false}. \]

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

\[ [\text{true}^*] \langle \text{true} \rangle \text{true}. \]

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:

\[ [\text{send}] \langle \text{true}^* \cdot \text{receive} \rangle \text{true}. \]

Compare this to the following formula

\[ [\text{send} \cdot \text{receive}] \langle \text{true}^* \cdot \text{receive} \rangle \text{true} \]

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

**Exercise 6.2.1.** 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 6.2.2.** 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.

### 6.3 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 \(\mu\)-calculus. As a sign of its expressiveness, it is possible to translate regular formulas to the modal \(\mu\)-calculus. However, the expressiveness comes at a price. Formulating properties using the modal \(\mu\)-calculus is far from easy.

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

\[ \phi ::= \text{true} \mid \text{false} \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi \mid \langle a \rangle \phi \mid [a] \phi \mid \mu X. \phi \mid \nu X. \phi \mid X. \]

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 \(\phi\). Here we abuse notation, by thinking of \(X\) as the set of states where \(\phi\) 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 \(\mu X. X = \text{false}\). 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, \(\nu X. X = \text{true}\).

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 $false = \langle a \rangle false$, 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 $\phi$ can eventually become valid. More precisely, there is a run starting in the current state on which $\phi$ becomes valid. Very often a stronger property is required, namely that $\phi$ will eventually become valid along every path. The formula to express this is:

$$\mu X.([true]X \lor \phi).$$

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

$$\mu X.(([true]X \land [true]true) \lor \phi).$$

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:

$$\mu X.([\overline{a}]X \land [true]true).$$

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.

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

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:

\[ \neg \nu X. \phi = \mu X. \neg \phi \quad \text{and} \quad \neg \mu X. \phi = \nu X. \neg \phi. \]

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 \( \phi \). This means that \( X \) in \( \phi \) must be preceded by an even number of negations. For counting negations \( \phi_1 \rightarrow \phi_2 \) ought to be read as \( \neg \phi_1 \lor \phi_2 \). So, for instance \( \mu X. \neg X \) and \( \nu X. \neg ([a] \neg X \lor X) \) 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 \( \nu X. \neg ([a] \neg X \lor \neg X) \) has a well defined solution as both occurrences of the variable \( X \) are preceded by two negations.

Regular formulas containing a \( * \) or \( + \) are straightforwardly translated to fixed point formulas. The translation is:

\[
\begin{align*}
\langle R^* \rangle \phi &= \mu X. ([R] X \lor \phi) \\
\langle R^+ \rangle \phi &= \langle R \rangle \langle R^* \rangle \phi \\
\langle R^+ \rangle \phi &= \langle R \rangle \langle R^* \rangle \phi
\end{align*}
\]

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

\[ \mu X. \nu Y. ((\langle a \rangle \text{true} \land [b] X) \lor (\neg \langle a \rangle \text{true} \land [b] Y)). \]

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

\[ \nu X. \mu Y. ((\langle a \rangle \text{true} \land [b] X) \lor (\neg \langle a \rangle \text{true} \land [b] Y)) \]

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 6.3.1.** Consider the formulas \( \phi_1 = \mu X. [a] X \) and \( \phi_2 = \nu X. [a] X \). If possible, give transition systems where \( \phi_1 \) is valid in the initial state and \( \phi_2 \) is not valid and vice versa.

**Exercise 6.3.2.** Give a labelled transition system that distinguishes between the following formulas

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

**Exercise 6.3.3.** Are the following formulas equivalent, and if not, explain why:

\[ [\overline{\text{send}} \cdot \overline{\text{receive}} ] ([\text{true}^* \cdot \text{receive}] \text{true}) \quad \text{and} \quad [\overline{\text{send}} ] [\mu X. ([\overline{\text{receive}}] X \lor \langle \text{true} \rangle \text{true}). \]

**Exercise 6.3.4.** What do the following formulas express:

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

Is there a process that shows that these formulas are not equivalent?
### 6.4 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 \( \alpha \) stands for a multi-action, \( R \) represents a regular formula, \( \phi \) stands for a modal formula and \( af \) stands for an action formula.

\[
\begin{align*}
\alpha & ::= \tau \mid a(t_1, \ldots, t_n) \mid \alpha_1 \alpha_2 \\
af & ::= t \mid true \mid false \mid \alpha \mid af \mid af \land af \mid af \lor af \mid \forall d. af \mid \exists d. af \\
R & ::= \varepsilon \mid af \mid R \cdot R \mid R + R \mid R^* \mid R^+ \\
\phi & ::= true \mid false \mid t \mid \lnot \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \forall d. \phi \mid \exists d. \phi \mid (R) \phi \mid [R]\phi \\
 & \quad \mu X(d_1;D_1:=t_1, \ldots, d_n;D_n:=t_n) \phi \mid \nu X(d_1;D_1:=t_1, \ldots, d_n;D_n:=t_n) \phi \mid X(t_1, \ldots, t_n).
\end{align*}
\]

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 \( \exists d. af \) represents \( \bigcup_{d:D} af \). Dually, \( \forall d. af \) represents \( \bigcap_{d:D} af \).

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

\[
[true^* \exists n: \mathbb{N}. error(n)] \mu X.([\text{shutdown}]X \land (true true))
\]

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:

\[
[true^* \exists n: \mathbb{N}. (fatal(n) \land error(n))] \mu X.([\text{shutdown}]X \land (true true)).
\]

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

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

\[
[(\forall n: \mathbb{N}. (fatal(n) \land error(n)))^*](true true)
\]

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 \( \phi \) holds for all values from the domain \( D \) substituted for \( d \) in \( \phi \). For the existential quantifier, \( \phi \) 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:

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

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.

\[
\forall m: \text{Message}. (true^* \cdot send(m)) (true^* \exists n: \mathbb{N}. deliver(m, n)) true.
\]
Note that it does not really matter where the quantifiers are put. I.e.
\[ \forall d:D. [\text{true} \cdot a(d)] \text{false} = [\text{true}] \forall d[D] a(d) \text{false} \]

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:

\[ \nu X(n:N:=0). [\text{deliver} \cup \text{receive}] X(n) \land [\text{receive}] X(n+1) \land [\text{deliver}](n > 0 \land X(n-1)). \]

Here \( n \) counts the number of received messages that have not been delivered. The notation \( n:N:=0 \) 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 6.1). 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 \( m_1, m_2 \) and \( out \) contain the last numbers read and delivered.

\[ \nu X\left(\begin{array}{l}
(\text{in}_1:N:=0, \text{in}_2:N:=0, \text{out}:N:=0) \\
\forall l:N. ([r_1(l)](l \geq \text{in}_1 \rightarrow X(l, \text{in}_2, \text{out})) \land \\
[r_2(l)](l \geq \text{in}_2 \rightarrow X(\text{in}_1, l, \text{out})) \land \\
[s(l)](l \geq \text{out} \land X(\text{in}_1, \text{in}_2, l))
\end{array}\right) \]

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

**Exercise 6.4.1.** Specify a unique number generator that works properly if it does not generate the same number twice.

**Exercise 6.4.2.** Express the property that a sorting machine only delivers sorted arrays. Arrays are represented by a function \( f:N \rightarrow \mathbb{N} \).

**Exercise 6.4.3.** Specify that a store with products of sort \( \text{Prod} \) is guaranteed to refresh each product. The only way to see this, is that the difference in the number of \( \text{enter}(p) \) and \( \text{leave}(p) \) is always guaranteed to become zero within a finite number of steps.

### 6.5 Equations

In tables 6.1 and 6.2 identities 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.
<table>
<thead>
<tr>
<th>Action formulas</th>
<th>Regular formulas</th>
<th>Proposition logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{true} = \text{false} )</td>
<td>( \langle \varepsilon \rangle \phi = \phi )</td>
<td>( \phi \land \psi = \psi \land \phi )</td>
</tr>
<tr>
<td>( \alpha_1 \cup \alpha_2 = \alpha_1 \cap \alpha_2 )</td>
<td>( \langle \text{false} \rangle \phi = \text{false} )</td>
<td>( (\phi \land \psi) \land \chi = \phi \land (\psi \land \chi) )</td>
</tr>
<tr>
<td>( \exists d : D.A(d) = \forall d : D.A(d) )</td>
<td>( \langle a f_1 \cup a f_2 \rangle \phi = \langle a f_1 \rangle \phi \lor \langle a f_2 \rangle \phi )</td>
<td>( \phi \lor \psi = \psi \lor \phi )</td>
</tr>
<tr>
<td>( \langle a f_1 \cap a f_2 \rangle \phi \Rightarrow \langle a f_1 \rangle \phi \land \langle a f_2 \rangle \phi )</td>
<td>( \langle \text{false} \rangle \phi \Rightarrow \langle \text{false} \rangle \phi \land \langle \text{true} \rangle \phi )</td>
<td>( (\phi \lor \psi) \lor \chi = \phi \lor (\psi \lor \chi) )</td>
</tr>
<tr>
<td>( \langle \text{false} \rangle \phi \Rightarrow \langle \text{false} \rangle \phi \land \langle \text{false} \rangle \phi )</td>
<td>( \langle \text{false} \rangle \phi \Rightarrow \langle \text{false} \rangle \phi \land \langle \text{true} \rangle \phi )</td>
<td>( \phi \lor \phi = \phi )</td>
</tr>
<tr>
<td>( \langle \forall \rangle \phi = (\forall \phi \land \langle \exists \rangle \phi) )</td>
<td>( \langle \forall \rangle \phi = (\forall \phi \land \langle \exists \rangle \phi) )</td>
<td>( \phi \lor \text{true} = \text{true} )</td>
</tr>
<tr>
<td>( \langle \exists \rangle \phi = (\forall \phi \lor \langle \forall \rangle \phi) )</td>
<td>( \langle \exists \rangle \phi = (\forall \phi \lor \langle \forall \rangle \phi) )</td>
<td>( \phi \lor \text{true} = \text{true} )</td>
</tr>
<tr>
<td>( R_1 + R_2 \phi = (R_1) \phi \lor (R_2) \phi )</td>
<td>( \langle \text{false} \rangle \phi \Rightarrow \langle \text{true} \rangle \phi \lor \langle \forall \rangle \phi )</td>
<td>( \phi \lor \text{false} = \phi )</td>
</tr>
<tr>
<td>( R_1.R_2 \phi = (R_1)(R_2) \phi )</td>
<td>( \langle \exists \rangle \phi \Rightarrow \langle \exists \rangle \phi \lor \langle \forall \rangle \phi )</td>
<td>( \phi \lor \text{false} = \phi )</td>
</tr>
<tr>
<td>( \langle R \rangle \phi = \mu X . (\langle R \rangle X \lor \phi) )</td>
<td>( \langle R \rangle \phi = \mu X . (\langle R \rangle X \lor \phi) )</td>
<td>( \phi \Rightarrow \psi = \neg \phi \lor \psi )</td>
</tr>
<tr>
<td>( \langle R \rangle \phi \Rightarrow \langle R \rangle \phi \lor \langle R \rangle \phi )</td>
<td>( \langle R \rangle \phi \Rightarrow \langle R \rangle \phi \lor \langle R \rangle \phi )</td>
<td>( \phi \Rightarrow \psi = \neg \phi \lor \psi )</td>
</tr>
<tr>
<td>( \langle R \rangle \phi \Rightarrow \langle R \rangle \phi \lor \langle R \rangle \phi )</td>
<td>( \langle R \rangle \phi \Rightarrow \langle R \rangle \phi \lor \langle R \rangle \phi )</td>
<td>( \phi \Rightarrow \psi = \neg \phi \lor \psi )</td>
</tr>
</tbody>
</table>

Table 6.1: Equivalences between modal formulas (part I)

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

A weaker version, in casu \( \langle a \cup b \rangle \phi = \langle a \rangle \phi \lor \langle b \rangle \phi \) is valid. This is also the reason that the following identities have weaker formulations:

\[
\begin{align*}
[a f_1 \cap a f_2] \phi &\Leftrightarrow [a f_1] \phi \lor [a f_2] \phi, \\
(\forall d : D.AF(d)) \phi &\Rightarrow (\forall d : D.(AF(d)) \phi, \\
[\forall d : D.AF(d)] \phi &\Leftrightarrow (\exists d : D.(AF(d)) \phi, \\
[\exists d : D.AF(d)] \phi &\Leftrightarrow (\forall d : D.(AF(d)) \phi.
\end{align*}
\]
Predicate logic
\( \forall d : D. \phi = \phi \)
\( \neg \forall d : D. \Phi(d) = \exists d : D. \neg \Phi(d) \)
\( \forall d : D. (\Phi(d) \land \Psi(d)) = \forall d : D. \Phi(d) \land \forall d : D. \Psi(d) \)
\( \forall d : D. (\Phi(d) \lor \psi) = \forall d : D. \Phi(d) \lor \psi \)
\( \forall d : D. \Phi(d) \Rightarrow \Phi(e) \)

Hennessy-Milner logic
\( \neg \langle a \rangle \phi = [a] \neg \phi \)
\( \langle a \rangle \text{false} = \text{false} \)
\( \langle a \rangle (\phi \lor \psi) = (\langle a \rangle \phi) \lor (\langle a \rangle \psi) \)
\( \langle a \rangle (\phi \land [a] \psi) \Rightarrow (\langle a \rangle (\phi \land \psi) \)

Fixed point equations
\( \mu X. \phi \Rightarrow \nu X. \phi \)
\( \neg \nu X. \phi = \mu X. \neg \phi \)
\( \mu X. \phi(X) = \phi(\mu X. \phi(X)) \)
\( \text{if } \phi(\psi) \Rightarrow \psi \text{ then } \mu X. \phi(X) \Rightarrow \psi \)

\( \exists d : D. \phi = \phi \)
\( \neg \exists d : D. \Phi(d) = \forall d : D. \neg \Phi(d) \)
\( \exists d : D. (\Phi(d) \lor \Psi(d)) = \exists d : D. \Phi(d) \lor \exists d : D. \Psi(d) \)
\( \exists d : D. (\Phi(d) \land \psi) = \exists d : D. \Phi(d) \land \psi \)
\( \Phi(e) \Rightarrow \exists d : D. \Phi(d) \)

\( \neg \langle a \rangle \phi = [a] \neg \phi \)
\( [a] \text{true} = \text{true} \)
\( [a] (\phi \land \psi) = [a] \phi \land [a] \psi \)
\( [a] (\phi \lor \psi) \Rightarrow (\langle a \rangle (\phi \lor [a] \psi) \)

Table 6.2: Equivalences between modal formulas (part II)
Chapter 7

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 craftsmanship 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 reproducible. The explanation for this is that the number of states in a system typically vary from $10^{10}$ to $10^{10^{10^{10}}}$, 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.

7.1 Alternating bit protocol

We start out by providing a model of a simple system, namely the alternating bit protocol (ABP) [5, 36]. The alternating bit protocol ensures successful transmission of data through lossy channels. The protocol is depicted in Figure 7.1. 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 7.1: Alternating bit protocol

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

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

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 $\text{Bits}$, 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 $\perp$ is delivered:

$$\text{proc } K = \sum_{d:D, b:B} r_2(d, b) \cdot (i \cdot s_3(d, b) + i \cdot s_3(\perp)) \cdot K$$

$$L = \sum_{b:B} r_5(b) \cdot (i \cdot s_6(b) + i \cdot s_6(\perp)) \cdot L$$

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, 36]. 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, 36] 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 $\perp$ 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 \( b \in \{0,1\} \):

\[
\text{proc } S(b;\mathbb{B}) = \sum_{d \in D} r_1(d) \cdot T(d, b), \\
T(d;D, b;\mathbb{B}) = s_2(d, b) \cdot (r_6(b) \cdot S(b) + (r_6(\neg b) + r_6(\perp)) \cdot T(d, b)).
\]

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 \( \perp \), 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 \( b \in \{0,1\} \):

\[
\text{proc } R(b;\mathbb{B}) = \sum_{d \in D} r_3(d, b) \cdot s_4(d) \cdot s_5(b) \cdot R(\neg b) + (\sum_{d \in D} r_3(d, \neg b) + r_3(\perp)) \cdot s_5(\neg b) \cdot R(b).
\]

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 \( \perp \) 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

\[
\text{proc } ABP = \nabla I(\Gamma_G(S(true) \mid K \mid L \mid R(true)))
\]

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

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:

\[
B = \tau_{\{c_2, c_3, c_6, i\}}(ABP) \tag{7.1}
\]

In section ?? 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 7.1.1.** In the mCRL2 toolset a description of the alternating bit protocol can be found in the directory `examples/academic/abp.mcrl2`. Use the tool `mcrll22lps 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 7.1.2.** Make a copy of the file `abp.mcrl2` from exercise 7.1.1 and remove the bits. Show with the simulator `xsim` that data elements can be duplicated and/or lost.
7.2 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 [48]), 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 [48] was modelled in 1991 using techniques as described in this book [10]. This model revealed a deadlock. When confronted with this, the author of [48] 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 [48] this problem has been removed.

![Figure 7.2: Sliding window protocol](image)

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 \( \text{FIFO}([], 2n) \):

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

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, \( s_4(\text{head}(l)) \) 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 7.2.

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.
\[
\begin{align*}
\text{proc } K &= \sum_{d,D,k:N} r_2(d,k) \cdot (i \cdot s_3(d,k) + i) \cdot K; \\
L &= \sum_{k:N} r_5(k) \cdot (i \cdot s_6(k) + i) \cdot L;
\end{align*}
\]

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.

\[
\begin{align*}
sort \quad DBuf &= \mathbb{N} \to D; \\
BBuf &= \mathbb{N} \to \mathbb{B};
\end{align*}
\]

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 \( j\mod n \) 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 \( \text{nextempty}_\text{mod}(i, b, m, n) \). It yields the first position in buffer \( b \) starting at \( i\mod n \) that contains \( false \). If the first \( m \) positions from \( i\mod n \) of \( b \) are all \( true \), it yields the value \((i+m)\mod 2n \).

The variable \( m \) is used to guarantee that the function \( \text{nextempty} \) is well defined if \( b \) is false at all its first \( n \) positions. The variables have the following sorts: \( d:D, i,j,m,n: \mathbb{N}; q:DBuf, c:B \) and \( b:BBuf \).

\[
\begin{align*}
eqn \quad empty &= \lambda j:\mathbb{N}.false; \\
q[i:=d] &= \lambda j:\mathbb{N}.if(i\approx j, d, q(j)); \\
b[i:=c] &= \lambda j:\mathbb{N}.if(i\approx j, c, b(j)); \\
\text{nextempty}_\text{mod}(i, b, m, n) &= \lambda \text{if}((b[i\mod n]) \wedge m>0, \text{nextempty}_\text{mod}((i+1)\mod 2n, b, m-1, n, i\mod 2n));
\end{align*}
\]

Below we model the sender process \( S \). The variable \( \ell \) 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 \( k \)th datum via gate 2 with sequence number \((\ell+k)\mod 2n \) and an acknowledgement can arrive via gate 6. This acknowledgement is the index of the first message that has not yet been received by the receiver.

\[
\begin{align*}
\text{proc } S(\ell, m:N, q:DBuf, n: \mathbb{N}^+ ) &= \sum_{d:D} m < n \to r_1(d) \cdot S(\ell, m+1, q[((\ell+m)\mod 2n)\mod n]):=d, n) \\
&\quad + \sum_{k:N} k < m \to s_2(q[(\ell+k)\mod 2n], (\ell+k)\mod 2n) \cdot S(\ell, m, q, n) \\
&\quad + \sum_{k:N} r_6(k) \cdot S(k, (m-k+\ell)\mod 2n, q, n);
\end{align*}
\]

The receiver \( R \) is modelled by the process \( R(\ell', q', b, n) \) where \( \ell' \) 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 \( \ell' \leq k < \ell' + n \). The condition \((k-\ell')\mod 2n < n \) 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., \( b(\ell'\mod n) \)) holds, then this message can be delivered via gate 4. Moreover, the oldest message is now \((\ell'+1)\mod 2n \) and the message at position \( \ell'\mod n \) 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.

\[
\begin{align*}
\text{proc } R(\ell':N, q':DBuf, b:BBuf, n: \mathbb{N}^+) &= \sum_{d,D,k:N} r_3(d,k) \cdot (k-\ell')\mod 2n < n \\
&\quad \to R(\ell', q'[k\mod n]:=d, b[k\mod n] := \text{true}, n) \\
&\quad \circ R(\ell', q', b, n)) \\
&\quad + b(\ell'\mod n) \to s_4(q'(\ell'\mod n)) \cdot R((\ell'+1)\mod 2n, q', b[\ell'\mod n] := \text{false}, n) \\
&\quad + s_5(\text{nextempty}_\text{mod}(\ell', b, n, n)) \cdot R(\ell', q', b, n);
\end{align*}
\]

The behaviour of the SWP is characterized by:
proc \textit{SWP}(q, q':DBuf, n:\mathbb{N}^+) = \nabla_H (\Gamma_G(S(0, 0, q, n) \parallel K \parallel L \parallel R(0, q', \text{empty}, n)));

where the set \( H = \{c_2, c_3, c_5, i, r_1, s_4\} \) 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 \( q, q':DBuf \):

\[
\tau.\text{FIFO}([], 2n) = \tau.\tau_I(\textit{SWP}(q, q', n))
\]

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

**Exercise 7.2.1.** Suppose the buffer size is two and messages are sent modulo \( 4 \) (see the file \textit{swp.mcrl2} in the directory \texttt{examples/academic}). Give an execution trace of \textit{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 7.3). 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 7.3: Two-way SWP](image)

**Exercise 7.2.2.** 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 7.2.3.** Give a specification of the Sender in the two-way SWP with piggybacking.
Piggybacking as described in exercise 7.2.3 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 [48] for more details.

**Exercise 7.2.4.** 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.

### 7.3 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 7.4: A movable patient support unit.](image)

The patient support unit is depicted in figure 7.4. 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 7.5. 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
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 7.1. 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 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 7.1) and the sliding window protocol (section 7.2). 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

| 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 7.1: The interactions of the MPSU controller
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 intentionally 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.

1a. 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.

1b. 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.

1c. 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.

\[\neg (\text{applyBrake} \land \text{unlockDock})\]

\[\neg (\text{atOutermost} \land \text{unlockDock})\]

\[\neg (\text{motorOff} \land \text{unlockDock})\]

\[\neg (\text{true} \land \text{releaseBrake} \cdot \text{applyBrake} \cdot \text{unlockDock})\]

\[\neg (\text{true} \land \text{motorLeft} \cdot \text{atOutermost} \land \text{unlockDock})\]

\[\neg (\text{true} \land (\text{motorLeft} + \text{motorRight}) \cdot \text{motorOff} \land \text{unlockDock})\]

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

2b. After a pressStop event takes place, a motorOff, a releaseBrake and an unlockDock action must inevitably follow.
2c. We can write this requirement as three compact modal formulas.

- \([true \cdot pressStop] \mu X [motorOff] X\).
- \([true \cdot pressStop] \mu X [releaseBrake] X\).
- \([true \cdot pressStop] \mu X [unlockDock] X\).

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

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

3c. \([true \cdot (atOutermost + atInnermost)] \mu X [motorOff] X\).

Interaction properties are the following:

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

4b. 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.

4c. \([true \cdot atInnermost \cdot motorLeft \cup releaseBrake \cdot pressUndock] \mu X [unlockDock \cap motorLeft \cap releaseBrake] X\).

5a. 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 switch on to move the bed inside the bore.

5b. 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.

5c. 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:

\[
\nu X(b_1; B := false, b_2; B := true, b_3; B := true, b_4; B := true).
\]

\[
isDocked] X (true, b_2, b_3, b_4) \land [unlockDock] X (false, b_2, b_3, b_4) \land
\]

\[
pressResume] X (b_1, true, b_3, b_4) \land [pressStop] X (b_1, false, b_3, b_4) \land
\]

\[
atInnermost] X (b_1, b_2, true, b_4) \land [motorRight] X (b_1, b_2, false, b_4) \land
\]

\[
motorOff \cup motorRight] X (b_1, b_2, b_3, true) \land [motorLeft] X (b_1, b_2, b_3, false) \land
\]

\[
(b_1 \land b_2 \land b_3 \land b_4 \Rightarrow [pressLeft] [motorLeft \cap unlockDock \cap pressStop \cap atInnermost] Y).
\]

6a. 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 switch on for an outward movement.

6b. 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.
6c. This formula very much resembles the formula at 5c.

\[ \nu X (b_1 : \mathbb{B} := \text{false}, b_2 : \mathbb{B} := \text{true}, b_3 : \mathbb{B} := \text{true}, b_4 : \mathbb{B} := \text{true}) \]

\[ [\text{isDocked}]X (\text{true}, b_2, b_3, b_4) \land [\text{unlockDock}]X (\text{false}, b_2, b_3, b_4) \land \]

\[ [\text{pressResume}]X (b_1, \text{true}, b_3, b_4) \land [\text{pressStop}]X (b_1, \text{false}, b_3, b_4) \land \]

\[ [\text{atOutermost}]X (b_1, b_2, \text{true}, b_4) \land [\text{motorLeft}]X (b_1, b_2, \text{false}, b_4) \land \]

\[ [\text{motorOff} \cup \text{motorLeft}]X (b_1, b_2, b_3, \text{true}) \land [\text{motorRight}]X (b_1, b_2, b_3, \text{false}) \land \]

\[ (b_1 \land b_2 \land b_3 \land b_4 \Rightarrow [\text{pressRight}] \]

\[ \mu Y ([\text{motorRight} \cap \text{unlockDock} \cap \text{pressStop} \cap \text{atOutermost}]Y). \]

A possible behaviour of the controller is given below. The idea is that there are two primary modes, namely Normal and Emergency. In 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** Mode = struct Normal | Emergency;

MotorStatus = struct moveleft | moveright | stopped;
**proc** Controller\((m:\text{Mode}, \text{docked}, \text{rightmost}, \text{leftmost}; B, ms:\text{MotorStatus})\)

\[= \text{pressStop}\cdot \text{releaseBrake}\cdot \text{motorOff}\cdot \]

\[\quad \text{Controller}(\text{Emergency, docked, rightmost, leftmost, stopped})\]

\[+ \text{pressResume}\cdot \text{Controller}(\text{Normal, docked, rightmost, leftmost, ms})\]

\[+ \text{pressUndock}\cdot\]

\[\quad (\text{docked} \land \text{rightmost})\]

\[\quad \rightarrow \text{applyBrake}\cdot \text{unlockDock}\cdot \text{Controller}(m, \text{false}, \text{rightmost}, \text{leftmost}, \text{ms})\]

\[\quad \circ \text{Controller}(m, \text{docked}, \text{rightmost}, \text{leftmost}, \text{ms})\]

\[+ \text{pressLeft}\cdot\]

\[\quad (\text{docked} \land \text{ms} \neq \text{moveleft} \land \neg \text{leftmost} \land m \approx \text{Normal})\]

\[\quad \rightarrow \text{releaseBrake}\cdot \text{motorLeft}\cdot \text{Controller}(m, \text{docked}, \text{false}, \text{leftmost}, \text{moveleft})\]

\[\quad \circ \text{Controller}(m, \text{docked}, \text{rightmost}, \text{leftmost}, \text{ms})\]

\[+ \text{pressRight}\cdot\]

\[\quad (\text{docked} \land \text{ms} \neq \text{moveRight} \land \neg \text{rightmost} \land m \approx \text{Normal})\]

\[\quad \rightarrow \text{releaseBrake}\cdot \text{motorRight}\cdot \text{Controller}(m, \text{docked}, \text{rightmost}, \text{false}, \text{moveRight})\]

\[\quad \circ \text{Controller}(m, \text{docked}, \text{rightmost}, \text{leftmost}, \text{ms})\]

\[+ \text{isDocked}\cdot \text{Controller}(m, \text{true}, \text{rightmost}, \text{leftmost}, \text{ms})\]

\[+ \text{atInnermost}\cdot \text{motorOff}\cdot \text{applyBrake}\cdot \text{Controller}(m, \text{docked}, \text{true}, \text{false}, \text{stopped})\]

\[+ \text{atOutermost}\cdot \text{motorOff}\cdot \text{applyBrake}\cdot \text{Controller}(m, \text{docked}, \text{false}, \text{true}, \text{stopped})\].

Initially, the support unit starts in \text{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\((\text{Normal, true, false, false, stopped})\);

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 7.3.1.** Which of the modal formulas are valid for the given model of the patient support platform? You may want to use the tools mcrl2lps, lps2pbes and pbes2bool to verify the modal formulas automatically. Also the state space generation tool lps2ltts and the visualisation tools ltsgraph and ltsview can be used to understand whether the patient support platform works properly.

**Exercise 7.3.2.** Are the requirements consistent in the sense that there exists a behavioural description such that all requirements are valid?

**Exercise 7.3.3.** 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 [35].

In order to overcome this problem extra hardware primitives are available, such as \text{LoadLock} and \text{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:

\[\text{CMPSWP}(n, m_1, m_2) = \text{if } M[n] \approx m_1 \text{ then } M[n] := m_2; \text{ return true else return false}\]

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.

84
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 $hd$. Furthermore, 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.

\[
\text{list } hd = NULL;
\]

\[
\text{void release(node } n) =
\]
\[
\text{bool } b;
\]
\[
\text{repeat}
\]
\[
\text{n.next := } hd;
\]
\[
b := \text{CMPSWP}(hd, n.next, n);
\]
\[
\text{until } b;
\]

\[
\text{node get(node } n) =
\]
\[
\text{bool } b; \text{ node } n;
\]
\[
\text{repeat}
\]
\[
n := hd;
\]
\[
\text{if } n \approx NULL \text{ return } NULL;
\]
\[
b := \text{CMPSWP}(hd, n, n.next);
\]
\[
\text{until } b;
\]
\[
\text{return } n;
\]

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 limited 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.
Appendix A

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, moveCursorRight, 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

\[ \rightarrow = \{(s_0, set, s_1), (s_1, alarm, s_1), (s_1, alarm, s_0), (s_1, reset, s_0)\} \]

2.3.7 (1) no. (2) yes. (3) no.

2.3.8 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.

2.3.9
2.3.2 They are all trace equivalent.

2.3.5 (1) Neither failures nor language equivalent. (2) Language but not failures equivalent. (3) Both failures and language equivalent.

2.5.6 (1) branching bisimilar. Not rooted branching bisimilar. (2) rooted branching bisimilar and hence branching bisimilar. (3) Neither rooted nor branching bisimilar.

2.5.7 All \( \tau \) transitions in the four leftmost transition systems are inert. None of the \( \tau \)'s in the two transition systems to the right are inert.

2.5.10 All pairs are weakly bisimilar. The first pair is not rooted weakly bisimilar. All \( \tau \)-transitions are inert with respect to weak bisimulation.

3.1.1 \textbf{map} \( \geq \): \( \text{Nat} \times \text{Nat} \rightarrow \mathbb{B} \);
\( < \): \( \text{Nat} \times \text{Nat} \rightarrow \mathbb{B} \);
\( > \): \( \text{Nat} \times \text{Nat} \rightarrow \mathbb{B} \);

\textbf{var} \( n, m : \text{Nat} \);
\textbf{eqn} \( n \geq \text{zero} = \text{true} \);
\( \text{zero} \geq \text{successor}(n) = \text{false} \);
\( \text{successor}(n) \geq \text{successor}(m) = n \geq m \);
\( n < \text{zero} = \text{false} \);
\( \text{successor}(n) < \text{successor}(m) = n < m \);
\( \text{successor}(n) > \text{zero} = \text{true} \);
\( \text{zero} > n = \text{false} \);
\( \text{successor}(n) > \text{successor}(m) = n > m \);

3.1.2 \textbf{map} \( \text{minus}, \text{max}, \text{power} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \);
\textbf{var} \( m, n : \text{Nat} \);
\textbf{eqn} \( \text{power} \left( \text{successor}(m), \text{zero} \right) = \text{successor}(\text{zero}) \);
\( \text{power} \left( m, \text{successor}(n) \right) = \text{times} \left( m, \text{power} \left( m, n \right) \right) \);
\( \text{max}(m, \text{zero}) = m \);
\( \text{max}(\text{zero}, m) = m \);
\( \text{max} \left( \text{successor}(n), \text{successor}(m) \right) = \text{successor} \left( \text{max}(n, m) \right) \);
\( \text{minus}(m, \text{zero}) = m \);
\( \text{minus}(\text{zero}, m) = \text{zero} \);
\( \text{minus} \left( \text{successor}(m), \text{successor}(n) \right) = \text{minus}(m, n) \);

3.1.3 \textbf{sort} \( \mathbb{B}, \text{Nat}, \text{List}, D \);
\textbf{cons} \[ ] :\rightarrow \text{List} ;
\textbf{in} : D \times \text{List} \rightarrow \text{List} ;
\textbf{map} \text{append} : D \times \text{List} \rightarrow \text{List} ;
\text{top}, \text{toe} : \text{List} \rightarrow D ;
\text{tail}, \text{untoe} : \text{List} \rightarrow \text{List} ;
\text{nonempty} : \text{List} \rightarrow \mathbb{B} ;
\text{length} : \text{List} \rightarrow \text{Nat} ;
\text{++}: \text{List} \times \text{List} \rightarrow \text{List} ;
\[\begin{align*}
\textbf{var} & \quad d, c, D, q, q': \text{List}; \\
\textbf{eqn} & \quad \text{append}(d, []) = \text{in}(d, []); \\
& \quad \text{append}(d, \text{in}(e, q)) = \text{in}(e, \text{append}(d, q)); \\
& \quad \text{top}(\text{in}(d, q)) = d; \\
& \quad \text{toe}(\text{in}(d, [])) = d; \\
& \quad \text{toe}(\text{in}(d, \text{in}(e, q))) = \text{toe}(\text{in}(e, q)); \\
& \quad \text{tail}(\text{in}(d, q)) = q; \\
& \quad \text{untoe}(\text{in}(d, [])) = []; \\
& \quad \text{untoe}(\text{in}(d, \text{in}(e, q))) = \text{in}(d, \text{untoe}(\text{in}(e, q)));
\end{align*}\]

3.2.2 \textit{succ} : \mathbb{N}^+ \rightarrow \mathbb{N}^+, \textit{succ} : \mathbb{N} \rightarrow \mathbb{N}^+, \textit{succ} : \mathbb{Z} \rightarrow \mathbb{Z}, \textit{succ} : \mathbb{R} \rightarrow \mathbb{R}, \textit{pred} : \mathbb{N}^+ \rightarrow \mathbb{N}, \textit{pred} : \mathbb{N} \rightarrow \mathbb{Z}, \textit{pred} : \mathbb{Z} \rightarrow \mathbb{Z}, \textit{pred} : \mathbb{R} \rightarrow \mathbb{R}.

3.2.3 \textit{0} is represented by @c\textit{0} and 1 by @c\textit{cNat}(1). Suppose @c\textit{0} = @c\textit{cNat}(1). Then true = @c\textit{0} \leq @c\textit{c0} = @c\textit{cNat}(1) \leq @c\textit{c0} = \textit{false}. Contradiction.

3.5.1 \textbf{map} \quad \textit{stretch} : \text{List}(\text{List}(S)) \rightarrow \text{List}(S); \\
\textbf{var} & \quad l: \text{List}(S); \\
& \quad L: \text{List}(\text{List}(S)); \\
\textbf{eqn} & \quad \text{stretch}([]) = []; \\
& \quad \text{stretch}(l \triangleright L) = l + \text{stretch}(L);
\]

3.5.2 \textbf{map} \quad \textit{insert} : \mathbb{S} \times \text{List}(S) \rightarrow \text{List}(S); \\
\textbf{var} & \quad s, s': \mathbb{S}; \\
& \quad L: \text{List}(S); \\
\textbf{eqn} & \quad \text{insert}(s, []) = [s]; \\
& \quad \text{insert}(s, s' \triangleright L) = \text{if}(s \approx s', s \triangleright L, s' \triangleright \text{insert}(s, L));
\]

The rule above will not terminate with innermost rewriters. To have guaranteed termination, conditional rules are necessary:

\[\begin{align*}
\textbf{var} & \quad s, s': \mathbb{S}; \\
& \quad L: \text{List}(S); \\
\textbf{eqn} & \quad \text{insert}(s, []) = [s]; \\
& \quad \text{insert}(s, s' \triangleright L) = s \triangleright L; \\
& \quad s \neq s' \rightarrow \text{insert}(s, s' \triangleright L) = s' \triangleright \text{insert}(s, L);
\end{align*}\]

Prove that \textit{insert}(s, L) will never contain two equal elements assuming that \(L\) contains only unique elements. It then follows with induction on the number of \textit{insert} operations that created lists always contain only unique elements.

3.5.3 \textbf{map} \quad \textit{insert} : \mathbb{N} \times \text{List}(\mathbb{N}) \rightarrow \text{List}(\mathbb{N}); \\
\textbf{var} & \quad n, n': \mathbb{N}; \\
& \quad L: \text{List}(\mathbb{N}); \\
\textbf{eqn} & \quad \textit{insert}(n, []) = [n]; \\
& \quad \textit{insert}(n, n' \triangleright L) = \text{if}(n \leq n', n \triangleright n' \triangleright L, n' \triangleright \text{insert}(n, L)); \\
& \quad \textit{is_in}(n, []) = \text{false}; \\
& \quad \textit{is_in}(n, n' \triangleright L) = \text{if}(n < n', \text{false}, \text{if}(n \approx n', \text{true}, \textit{is_in}(n, L)));
\]

89
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:

\[
\begin{align*}
\text{map} & \quad \text{sort} : \text{List}(\mathbb{N}) \rightarrow \text{List}(\mathbb{N}); \\
\text{var} & \quad n : \mathbb{N}; \\
& \quad L : \text{List}(\mathbb{N}); \\
\text{eqn} & \quad \text{sort}([],[]) = []; \\
& \quad \text{sort}(n \triangleright L) = \text{insert}(n, \text{sort}(L));
\end{align*}
\]

Due to this definition, it holds that for an arbitrary list \( L \) obtained after a sequence of inserts using \( \triangleright \), we obtain the list \( \text{sort}(L) \) after the same sequence of inserts using \( \text{insert} \). The following property says that the element tests yield the same results:

\[
n \in L \iff \text{is\_in}(n, \text{sort}(L)).
\]

This can be proven with induction on \( L \), using the (to be proven) property that \( \text{sort}(L) \) must be sorted.

3.5.4 \( \{m : \mathbb{N}^+ \mid 1 < m \land \forall n : \mathbb{N}^+. 2 \leq n < m \Rightarrow m | a \neq 0\} \).

3.5.5 \( \{l : \text{List}(\mathbb{N}) | \forall n : \mathbb{N}. (l.n) \approx 0\} \) and \( \{l : \text{List}(\mathbb{N}) | \#(l) \approx 2\} \).

3.3.1 \textbf{eqn} \quad \text{map}(f,[]) = []; \quad \text{map}(f,n \triangleright L) = f(n) \triangleright \text{map}(f,L);

3.3.2 \textbf{eqn} \quad \text{multiplicity}(i,a,n) = \text{if}(n \equiv 0,0,\text{if}(a(n-1) \equiv i,1,0) + \text{multiplicity}(i,a,n-1)); \quad \text{equalcontents}(a,a',n) = \forall i : \mathbb{N}. \text{multiplicity}(i,a,n) = \text{multiplicity}(i,a',n);

3.4.1 \textbf{sort} \quad \text{ChecksumType, Data};
\quad \text{MessageType} = \textbf{struct} \quad \text{ack} \mid \text{ctrl} \mid \text{mes};
\quad \text{Message} = \textbf{struct} \quad \text{frame}(\text{MessageType, CheckSumType, Data}) | \text{frame}(\text{MessageType, Checksum});

4.2.1 (1) \( \tau \). (2) \( a(1)|b(2) \). (3) \text{false}.

4.3.1 \((a + a) \cdot (b + b) \cdot (c + c) \overset{A3}{=} (a \cdot b) \cdot c \overset{A5}{=} a \cdot (b \cdot c)\); 
\( (a + a) \cdot (b + c) \cdot (c + c) \overset{A3}{=} (a \cdot b) \cdot c + (a \cdot b) \cdot (c + c) \overset{A3}{=} (a \cdot b) \cdot (c + c) + (a \cdot b) \cdot (c + c) \overset{A3}{=} (a \cdot (b + b)) \cdot (c + c)\).

4.3.2 Suppose \( x \subseteq y \) and \( y \subseteq x \). By definition we have \( (1) \; x + y = y \) and \( (2) \; y + x = x \). Thus we obtain:
\[
x = y + x = x = y = x + y = (1).
\]

4.5.1 (1) \( \text{true} \rightarrow x \circ y = x \) \( \equiv \text{true} \rightarrow y \circ x \).
\[
\text{false} \rightarrow x \circ y = y \equiv \text{false} \rightarrow y \circ x.
\]
(2) \( \text{true} \lor c' \rightarrow x \circ y = \text{true} \rightarrow x \circ y = x = \text{true} \rightarrow x \circ (c' \rightarrow x \circ y) \).
\[
\text{false} \lor c' \rightarrow x \circ y = c' \rightarrow x \circ y = \text{false} \rightarrow x \circ (c' \rightarrow x \circ y).
\]
(3) \( x + y = x + y + c \rightarrow x + c \rightarrow y = x + y + c \rightarrow x \circ y \).
(4) if \( b = \text{true} \), then by the assumption \( (b = \text{true} \Rightarrow x = y) \) we have \( x = y \) and so \( b \rightarrow x \circ z = b \rightarrow y \circ z \).
\[
\text{if} \; b = \text{false}, \; \text{then} \; b \rightarrow x \circ z = z = b \rightarrow y \circ z.
\]

4.5.2 \( \sum_{n: \mathbb{N}} \text{read}(n) \cdot ((n < 100) \rightarrow \text{forward}(n) \circ \text{overflow}) \).

4.6.1 \( X = \sum_{m: \text{Message}} \text{read}(m) \cdot \text{forward}(m) \cdot X \cdot X = \sum_{m: \text{Message}} \text{read}(m) \cdot (\text{empty} + \text{forward}(m)) \cdot X \).
5.1.1 \( a(b+c+c\cdot b + b(c) + c \cdot a \cdot b + (a|c) \cdot b). \)

5.1.2 Approx. 2 pages.

5.1.3 \( x \parallel y = x \parallel y \parallel x \parallel x | y = y \parallel x. \)

5.2.1 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 \( \nabla \) to a deadlock state are \( r_1(d_1) \parallel c_2(d_1) \parallel r_1(d_1) \) and \( r_1(d_1) \parallel c_2(d_1) \parallel r_1(d_2). \)

5.2.2 \( \text{act } r_1, s_2, r_2, c_2, s_3, r_3, c_3, s_4, s_5; \)

\( \text{proc } X(b;y) = \sum_{d:D} r_1(d) \cdot (b \rightarrow s_2(d) \parallel s_3(d)) \cdot X(\neg b); \)

\( Y = \sum_{d:D} r_2(d) \cdot s_4(d) \cdot Y; \)

\( Z = \sum_{d:D} r_3(d) \cdot s_5(d) \cdot Z; \)

\( \text{init } \nabla_{(r_1,s_4,s_5,c_2,c_3)}(\Gamma_{(r_2=s_2,c_2=s_3,c_3)}(X \parallel Y \parallel Z)); \)

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'.

5.4.1

1. \( a(\tau b + b) \overset{A_3}{=} a(\tau (b + b) + b) \overset{B_2}{=} a(b + b) \overset{A_3}{=} ab. \)

2. \( a(\tau (b + c) + b) \overset{B_2}{=} a(b + c) \overset{A_1}{=} a(c + b) \overset{B_2}{=} a(\tau (c + b) + c) \overset{A_1}{=} a(\tau (b + c) + c). \)
6.2.1 Use the rules in table 6.1.

6.4.1 The formula \( \mu X. \forall X \) is equivalent to true, there is no transition system for which the minimal fixed point formula holds, and the maximal one does not.

6.4.2 The reduced state space is deterministic and contains no \( \tau \)'s. Therefore reducing it further using weak bisimulation or weak trace equivalence does not make sense.

5.5.4 \( e \cdot f + f \cdot e \).

5.6.1 1. \( (a \cdot (b) \text{true} \land \neg (c) \text{true}) \). 2. \( [a] \neg (\langle b \rangle \text{true} \lor \langle c \rangle \text{true}) \). 3. \([b] \text{false} \lor [a][b] \text{false} \).

5.6.2 Use the identities between modal formulas. The modalities containing \( c_1 \) and \( c_2 \) can be removed.

5.6.3 The process \( P \) with \( P = \text{send} \cdot P + \text{receive} \cdot \delta \) makes the first formula valid, and the second invalid.

5.6.4 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 \( P = a \cdot P \).

5.6.5 \( \forall n : \text{N}. [\text{true} \cdot \text{generate}(n)] \text{true} \cdot \text{generate}(n)] \text{false} \).

5.6.6 \( \forall f : \text{N} \rightarrow \text{N}. [\text{true} \cdot \text{deliver}(f)] \forall n : \text{N} f(n + 1) > f(n) \).

5.6.7 \( \forall p : \text{Product}, \nu X (n : \text{N} : = 0). [\text{enter}(p) \cup \text{leave}(p)] X(n) \land [\text{enter}(p)] X(n + 1) \land [\text{leave}(p)] X(n - 1) \land Y(p, n), \nu Y(p : \text{Product}, n : \text{N}). (n \approx 0) \lor ([\text{enter}(p) \cup \text{leave}(p)] Y(p, n) \land [\text{enter}(p)] Y(p, n + 1) \land [\text{leave}(p)] Y(p, n - 1)) \).

7.3.1 Properties 3, 5 and 6 are valid. The others are not valid.

7.3.2 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.
Bibliography


