Part I

Modelling

Chapter 1

Introduction

1.1 Motivation

In today's world, virtually all human-made systems contain computers which 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, it is a major aspect of their task to design and understand the behaviour of such systems. 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 systems. 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 power to the computer was disconnected, and neither the computer nor the switch would attempt to perform whatever task anymore, until switched on again.

In a modern computer, the on/off switch is connected to the central processor. If the on/off switch is pressed, the processor is signalled that the computer 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, even the simplest system can perform complex, and often counter-intuitive behaviour. Switching a system off will lead to more activity, although generally only temporarily.

Given the complexity of systems, it is not at all self-evident anymore whether a system will behave as expected. Existing techniques such as testing to guarantee the quality of such interacting systems are inadequate. There are too many different runs of the system to inspect and test them all.

A standard engineering approach to contain phenomena with a complexity beyond human grasp is to use mathematical models. State machines appear to be the right mathematical notion to model behaviour of computer systems. A system starts in its initial state and whenever something happens it moves to some other state. This 'something' is called an action, which can be any activity such as an internal calculation or a communication with some other system. The act of going from one state to another is called a transition. State machines are also called automata or labelled transition systems.

The core problem of labelled transition systems is that they tend to become large. This is called the *state space explosion problem*. Especially, the use of data and parallel behaviour can easily lead to automata of which the sizes are colossal.

Modelling a four byte integer as a state machine, where each value is a state and changing a value is a transition, already leads to a state machine with $4{\cdot}10^9$ states. For parallel systems 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 numbers indicating the number of states in a labelled transition system are typically 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*.

The sheer size of labelled transition systems has two consequences. The first one is that we cannot write transition systems down as a state machine, but require more concise notation. We use the language mCRL2 for this, which was inspired by the process algebra ACP, extended with data and time. And we use the modal mu-calculus, also extended with data and time to denote properties about automata.

The second consequence is that – as in ordinary mathematical modelling – we must restrain ourselves when modelling behaviour. If our models have too many states, analysis becomes cumbersome. But of course, the contrary also holds. If our models do not contain sufficient detail, we may not be able to study those phenomena we are interested in.

But mastering the art of behavioural modelling will turn out to be rewarding. If the state space explosion problem can be avoided, it is often very efficient to check a few modal formulas representing correctness requirements of a system. Our own experience is that we found problems in all existing systems that we modelled thoroughly. Several examples of our experiments and the discovered problems are reported in [9, 10, 103, 104, 157]. If these systems, protocols and algorithms were modelled and analysed using methods and tools available today, many of those problems would have been uncovered easily. Interestingly, some of the faulty protocols and algorithms have been published in scientific venues; it is therefore not too exaggerated to say that formal analyses (e.g., model checking) before publication of distributed algorithms and protocols ought to be obligatory.

Without proper mathematical theories, appropriate proof and analysis methods and adequate computer tools, it is impossible to design correct communicating systems with the complexity that is common today. Using formal modelling and analysis, a far better job is possible. In this book we provide the required ingredients.

1.2 The mCRL2 approach

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* [133, 25, 101], which are algebraic formalisms for compositional specification of concurrent systems. More specifically, we propose a formalism, called *mCRL2*, which extends the Algebra of Communicating Processes (ACP) [25] with various features including notions of data, time and multi-actions.

Specifications in mCRL2 are built upon atomic (inter)actions, which can be composed using various algebraic operators. The specified behaviour can then be simulated, visualised 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, namely an action, is introduced. Using transition systems, it is explained how actions can be combined into behaviour. The circumstances under which behaviours can be considered the same are also investigated.

In chapter 3, data types are explained. In particular it is explained how data types are built upon constructors, mappings and functions using equations. In appendix A the exact definition of all pre-defined data types is given.

In chapter 4, sequential and non-deterministic behaviour is described in an algebraic way. This means that there are a number of composition operators, of which the properties are characterised by axioms. In chapter 5, it is shown how parallel and communication components are specified, again including the required axioms.

The next chapter explains how to describe behavioural properties in a very expressive logic, called the modal mu-calculus. By integrating data in this calculus, we cannot only state simple properties, such as that a system is deadlock-free, but also very complex properties that depend on fair behaviour or require data storage and computation in the modal formula.

Chapter 7 contains a number of example descriptions of the behaviour of some simple systems. In chapter 8, 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 9 provides basic technologies to transform one process into another. Typical techniques are induction, the recursive specification principle and the expansion theorem.

Chapter 10 describes how to transform processes to the so-called linear form, which will play an important role in all subsequent manipulations. Moreover, recursive specifications and different methods of reasoning about them are presented in this chapter.

Chapter 11 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 12, the cones and foci technique is explained, which allows to prove that a specification and an implementation of some system are equivalent. In chapter 13, the verification techniques are used to validate some complex distributed algorithms.

In chapter 14 it is shown how parameterised boolean equations can be used to verify modal formulas on processes in the linear form.

In chapter 15, the formal semantics of mCRL2 and the related formalisms 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 relationship between the process equivalences given in chapter 2 and the axioms in the subsequent chapters.

In appendix A, the equations characterising data types are given. In appendix B the notation used in the book is related to the notation used in the tools. Appendix C provides an overview of the syntaxes of all different formalisms. Appendix D summarises all process-algebraic axioms. Appendix E 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, although students can deal with more complex systems). For larger modules, chapters 8 to 12 and chapter 14 can also be included. Chapter 13 gives examples of verifications of more complex protocols, while chapter 15 provides formal background material, such as type-checking rules and the semantics of various formalisms.

More advanced material and more difficult exercises in each chapter are designated by a black star (★) and can be skipped for shorter and / or undergraduate courses.