Part I

A Classic Theory of Reactive Systems

1

Introduction

Aims of this book

The aim of the first part of this book is to introduce three basic notions that we shall use to describe, specify and analyse reactive systems, namely

- Milner's calculus of communicating systems (CCS) (Milner, 1989),
- the model known as labelled transition systems (LTSs) (Keller, 1976), and
- Hennessy–Milner logic (HML) (Hennessy and Milner, 1985) and its extension with recursive definitions of formulae (Larsen, 1990).

We shall present a general theory of reactive systems and its applications. In particular, we intend to show the following:

1. how to describe actual systems using terms in our chosen models (i.e. either as terms in the process description language CCS or as labelled transition systems);
2. how to offer specifications of the desired behaviour of systems either as terms of our models or as formulae in HML; and
3. how to manipulate these descriptions, possibly (semi-)automatically, in order to analyse the behaviour of the model of the system under consideration.

In the second part of the book, we shall introduce a similar trinity of basic notions that will allow us to describe, specify and analyse real-time systems – that is, systems whose behaviour depends crucially on timing constraints. There we shall present the formalisms of timed automata (Alur and Dill, 1994) and timed CCS (Yi, 1990, 1991a, b) to describe real-time systems, the model of timed

labelled transition systems (TLTSs) and a real-time version of Hennessy–Milner logic (Laroussinie, Larsen and Weise, 1995).

After having worked through the material in this book, you should be able to describe non-trivial reactive systems and their specifications using the aforementioned models and verify the correctness of a model of a system with respect to given specifications either manually or by using automatic verification tools such as the Edinburgh Concurrency Workbench (Cleaveland, Parrow and Steffen, 1993) and the model checker for real-time systems UPPAAL (Behrmann, David and Larsen, 2004).

Our, somewhat ambitious, aim is therefore to present a model of reactive systems that supports their design, specification and verification. Moreover, since many real-life systems are hard to analyse manually, we should like to have computer support for our verification tasks. This means that all the models and languages that we shall use in this book need to have a *formal* syntax and semantics. (The *syntax* of a language consists of the rules governing the formation of statements, whereas its *semantics* assigns meaning to each of the syntactically correct statements in the language.) These requirements of formality are not only necessary in order to be able to build computer tools for the analysis of system descriptions but are also fundamental in agreeing upon what the terms in our models are actually intended to describe in the first place. Moreover, as Donald Knuth once wrote:

> A person does not really understand something until after teaching it to a computer, i.e. expressing it as an algorithm. . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.

The pay-off from using formal models with an explicit formal semantics to describe our systems will therefore be the possibility of devising algorithms for the animation, simulation and verification of system models. These would be impossible to obtain if our models were specified only in an informal notation.

Now that, it is hoped, you know what to expect from this book, it is time to get to work. We shall begin our journey through the beautiful land of concurrency theory by introducing a prototype description language for reactive systems and its semantics. However, before setting off on such an enterprise, we should describe in more detail what we actually mean by the term ‘reactive system’.

1.1 What are reactive systems?

The ‘standard’ view of computing systems is that, at a high level of abstraction, these may be considered as black boxes that take inputs and provide

appropriate outputs. This view agrees with the description of algorithmic problems. An *algorithmic problem* is specified by a collection of legal inputs, and, for each legal input, its expected output. In an imperative setting, an abstract view of a computing system may therefore be given by describing how it transforms an initial *state* – i.e. a function from variables to their values – to a final state. This function will, in general, be *partial*, i.e. it may be undefined for some initial states, in order to capture that the behaviour of a computing system may be non-terminating for some input states. For example, the effect of the program

$$S = z \leftarrow x; x \leftarrow y; y \leftarrow z$$

is described by the function $[\![S]\!]$ from states to states, defined thus:

$$[\![S]\!] = \lambda s.\ s[x \mapsto s(y), y \mapsto s(x), z \mapsto s(x)],$$

where the new state $s[x \mapsto s(y), y \mapsto s(x), z \mapsto s(x)]$ is that in which the value of variable x is the value of y in state s and that of variables y and z is the value of x in state s. The values of all the other variables are those they had in state s. This state transformation is a way of describing formally that the intended effect of S is essentially to swap the values of the variables x and y.

However, the effect of the program

$$U = \textbf{while}\ \text{true}\ \textbf{do skip},$$

where we use **skip** to stand for 'no operation', is described by the *partial* function from states to states given by

$$[\![U]\!] = \lambda s.\ \text{undefined},$$

i.e. the always undefined function. This captures the fact that the computation of U never produces a result (final state), irrespective of the initial state.

In this view of computing systems, non-termination is a highly undesirable phenomenon. An algorithm that fails to terminate on some inputs is not one the users of a computing system would expect to have to use. A moment of reflection, however, should make us realize that we already use many computing systems whose behaviour cannot be readily described as a function from inputs to outputs – not least because, at some level of abstraction, these systems are inherently meant to be non-terminating. Examples of such computing systems are

- operating systems,
- communication protocols,
- control programs, and
- software running in embedded system devices such as mobile telephones.

At a high level of abstraction, the behaviour of a control program can be seen to be governed by the following pseudocode algorithm skeleton:

loop
read the sensors' values at regular intervals
depending on the sensors' values trigger the relevant actuators
forever

These and many others are examples of computing systems that interact with their environment by exchanging information with it. Like the neurons in a human brain, these systems react to stimuli from their computing environment (in the example control program above, these are variations in the values of the sensors) by possibly changing their state or mode of computation and, in turn, influence their environment by sending back some signals to it or initiating some operations that affect the computing environment (this is the role played by the actuators in the example control program). David Harel and Amir Pnueli coined the term *reactive system* in Harel and Pnueli (1985) to describe a system that, like those mentioned above, computes by reacting to stimuli from its environment.

As the above examples and discussion indicate, reactive systems are inherently parallel systems and a key role in their behaviour is played by communication and interaction with their computing environment. A 'standard' computing system can also be viewed as a reactive system in which interaction with the environment takes place only at the beginning of the computation (when inputs are fed to the computing device) and at the end (when the output is received). However, all the example systems given before maintain a continuous interaction with their environment, and we may think of the computing system and its environment as parallel processes that communicate with each other. In addition, as again nicely exemplified by the skeleton of a control program given above, non-termination is a *desirable* feature of some reactive systems. In contrast to the setting of 'standard' computing systems, we certainly do *not* expect the operating systems running on our computers or the control program monitoring a nuclear reactor to terminate!

Now that we have an idea of what reactive systems are, and of the key aspects of their behaviour, we can begin to consider what an appropriate abstract model for this class of systems should offer. In particular, such a model should allow us to describe the behaviour of collections of (possibly non-terminating) parallel processes that may compute independently or interact with one another. It should provide us with facilities for the description of well-known phenomena that appear in the presence of concurrency and are familiar to us from the world of operating systems and parallel computation in general (e.g., deadlock, livelock, starvation and so on). Finally, in order to abstract from implementation-dependent issues having to do with, say, scheduling policies, the chosen model should permit a clean description of *non-determinism* – a most useful modelling tool in computer science.

Our aim in the remainder of this book will be to present a general-purpose theory that can be used to describe, and reason about, *any* collection of interacting processes. The approach that we shall present will make use of a collection of models and formal techniques that is often referred to as *process theory*. The key ingredients in this approach are

- (process) algebra,
- automata or labelled transition systems,
- structural operational semantics, and
- logic.

These ingredients give the foundations for the development of (semi-)automatic verification tools for reactive systems that support various formal methods for validation and verification, which can be applied to the analysis of highly non-trivial computing systems. The development of these tools requires in turn advances in algorithmics and via the increasing complexity of the analysed designs feeds back to the theory-development phase by suggesting the invention of new languages and models for the description of reactive systems.

Unlike in the setting of sequential programs, where we would often prefer to believe that the development of correct programs can be done without any recourse to 'formalism', it is a well-recognized fact of life that the behaviour of even very short parallel programs may be very hard to analyse and understand. Indeed, analyzing these programs requires a careful consideration of issues related to the interactions amongst their components, and even imagining these is often a mind-boggling task. As a result, the techniques and tools that we shall present in this book are becoming widely accepted in the academic and industrial communities that develop reactive systems.

1.2 Process algebras

The first ingredient in the approach to the theory of reactive systems presented in this book is a prototypical example of a *process algebra*. Process algebras are prototype specification languages for reactive systems. They have evolved from the insights of many outstanding researchers over the last 30 years, and a brief history of the ideas that led to their development may be found in Baeten (2005). (For an accessible, but more advanced, discussion of the role that algebra plays in process theory, the reader could consult the survey paper Luttik (2006).) A crucial initial observation at the heart of the notion of process algebra is due to Milner, who noticed that concurrent processes have an algebraic structure. For example, once we have built two separate processes P and Q, we can form a new process by

combining P and Q sequentially or in parallel. The results of these combinations will be new processes whose behaviour depends on that of P and Q and on the *operation* that we have used to compose them. This is the first sense in which these description languages are algebraic: they consist of a collection of operations for building new process descriptions from existing ones.

Since these languages aim at specifying parallel processes that may interact with one another, a key issue that needs to be addressed is how to describe communication or interaction between processes running at the same time. Communication amounts to information exchange between a process that produces the information (the *sender*) and a process that consumes it (the *receiver*). We often think of this communication of information as taking place via a *medium* that connects the sender and the receiver. If we are to develop a theory of communicating systems based on this view, we have to decide upon the communication medium used in inter-process communication. Several possible choices immediately come to mind. Processes may communicate via, for example, (un)bounded buffers, shared variables, some unspecified ether or the tuple spaces used by Linda-like languages (Gelernter, 1985). Which one do we choose? The answer is not at all clear, and each specific choice may in fact reduce the applicability of our language and the models that support it. A language that can properly describe processes that communicate via, say, FIFO buffers may not readily allow us to specify situations in which processes interact via, say, shared variables.

The solution to this riddle is both conceptually simple and general. A crucial original insight of figures such as Hoare and Milner was that we need not distinguish between active components, such as senders and receivers, and passive ones such as the communication media mentioned above. They may all be viewed as processes – i.e. as systems that exhibit behaviour. All these processes can interact via message-passing modelled as *synchronized communication*, which is the only basic mode of interaction. This is the key idea underlying Hoare's Communicating Sequential Processes (CSP) (Hoare, 1978, 1985), a highly influential proposal for a programming language for parallel programs, and Milner's Calculus of Communicating Systems (CCS) (Milner, 1989), the paradigmatic process algebra.

2

The language CCS

We shall now introduce Milner's Calculus of Communicating Systems (CCS). We begin by informally presenting the process constructions allowed in this language, and their semantics, in Section 2.1. Then, in Section 2.2, we proceed to put our developments on a more formal footing.

2.1 Some CCS process constructions

It is useful to begin by thinking of a CCS process as a black box. This black box may have a name that identifies it, and it has a *process interface*. This interface describes the collection of *communication ports*, also referred to as *channels*, that the process may use to interact with other processes that reside in its environment, together with an indication of whether it uses these ports for inputting or outputting information. For example, the drawing in Figure 2.1 pictures the interface for a process whose name is CS (for computer scientist). This process may interact with its environment via three ports, or communication channels, namely 'coffee', '$\overline{\text{coin}}$' and '$\overline{\text{pub}}$'. The port 'coffee' is used by process CS for input, whereas the ports '$\overline{\text{coin}}$' and '$\overline{\text{pub}}$' are used for output. In general, given a port name a we use $\bar{a}$ for the output on port a. We shall often refer to labels such as coffee or $\overline{\text{coin}}$ as *actions*.

A description like the one given in Figure 2.1 only gives static information about a process. What we are most interested in is the *behaviour* of the process being specified. The behaviour of a process is described by giving a 'CCS program'. The idea is that, as we shall soon see, the process constructions that are used in building the program allow us to describe both the structure of the process and its behaviour.

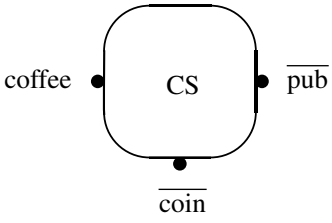

Figure 2.1 The interface for the process CS.

Inaction, prefixing and recursive definitions Let us begin by introducing the constructs of the language CCS, by means of examples. The most basic process of all is the process $\mathbf{0}$ (read 'nil'). This performs no action whatsoever. The process $\mathbf{0}$ offers the prototypical example of a deadlocked behaviour – one that cannot proceed any further in its computation.

The most basic process constructor in CCS is *action prefixing*. Two example processes built using $\mathbf{0}$ and action prefixing are a match and a complex match, described by the expressions

$$\text{strike}.\mathbf{0} \quad \text{and} \quad \text{take}.\text{strike}.\mathbf{0},$$

respectively. Intuitively, a match is a process that dies after it has been performed (i.e. that becomes the process $\mathbf{0}$ after executing the *action* strike), and a complex match is one that needs to be taken hold of before it can behave like a match. More generally, the formation rule for action prefixing says that

if P is a process and a is a label then $a.P$ is also a process.

The idea is that a label such as strike or $\overline{\text{pub}}$ will denote an input or output action on a communication port and that the process $a.P$ is one that begins by performing action a and behaves like P thereafter.

We have already mentioned that processes can be given names, very much as procedures can. This means that we can introduce names for (complex) processes and that we can use these names in defining other process descriptions. For instance, we can give the name Match to the complex match defined thus:

$$\text{Match} \stackrel{\text{def}}{=} \text{take}.\text{strike}.\mathbf{0}.$$

The introduction of names for processes allows us to give recursive definitions of process behaviours – compare with the recursive definition of procedures or methods in your favourite programming language. For instance, we may define the behaviour of an everlasting clock thus:

$$\text{Clock} \stackrel{\text{def}}{=} \text{tick}.\text{Clock}.$$

Note that, since the process name Clock is a short-hand for the term on the right-hand side of the above equation, we may repeatedly replace the name Clock with its definition to obtain that

$$\begin{aligned} \mathrm{Clock} &\stackrel{\mathrm{def}}{=} \mathrm{tick.Clock} \\ &= \mathrm{tick.tick.Clock} \\ &= \mathrm{tick.tick.tick.Clock} \\ &\vdots \\ &= \underbrace{\mathrm{tick.}\ldots.\mathrm{tick}}_{n \text{ times}}\mathrm{.Clock}, \end{aligned}$$

for each positive integer n.

As another recursive process specification, consider that of a simple coffee vending machine:

$$\mathrm{CM} \stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{coffee}}.\mathrm{CM}. \tag{2.1}$$

This is a machine that is willing to accept a coin as input, deliver coffee to its customer and thereafter return to its initial state.

Choice The CCS constructs that we have presented so far would not allow us to describe the behaviour of a vending machine that allows its paying customer to choose between tea and coffee, say. In order to allow for the description of processes whose behaviour may follow different patterns of interaction with their environment, CCS offers the *choice operator*, which is written '+'. For example, a vending machine offering either tea or coffee may be described thus:

$$\mathrm{CTM} \stackrel{\mathrm{def}}{=} \mathrm{coin}.(\overline{\mathrm{coffee}}.\mathrm{CTM} + \overline{\mathrm{tea}}.\mathrm{CTM}). \tag{2.2}$$

The idea here is that, after having received a coin as input, the process CTM is willing to deliver either coffee or tea, depending on its customer's choice. In general, the formation rule for choice states that

if P and Q are processes then so is $P + Q$.

The process $P + Q$ is one that has the initial capabilities of both P and Q. However, choosing to perform initially an action from P will pre-empt the further execution of actions from Q, and vice versa.

Exercise 2.1 *Give a CCS process which describes a clock that ticks at least once and may stop ticking after each clock tick.* ♦

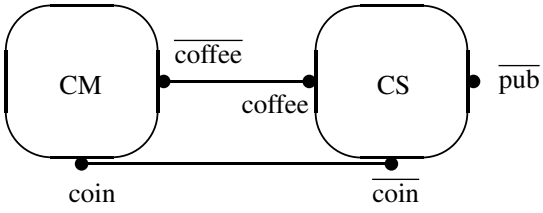

Figure 2.2 The interface for the process CM | CS.

Exercise 2.2 *Give a CCS process which describes a coffee machine that may behave like that given by (2.1) but may also steal the money it receives and fail at any time.* ♦

Exercise 2.3 *A finite process graph T is a quadruple $(\mathcal{Q}, A, \delta, q_0)$, where*

- *$\mathcal{Q}$ is a finite set of states,*
- *A is a finite set of labels,*
- *$q_0 \in \mathcal{Q}$ is the start state, and*
- *$\delta : \mathcal{Q} \times A \to 2^{\mathcal{Q}}$ is the transition function.*

Using the operators introduced so far, give a CCS process that describes T. ♦

Parallel composition It is well known that a computer scientist working in a research university is a machine for turning coffee into publications. The behaviour of such an academic may be described by the CCS process

$$\mathrm{CS} \stackrel{\mathrm{def}}{=} \overline{\mathrm{pub}}.\overline{\mathrm{coin}}.\mathrm{coffee}.\mathrm{CS}. \tag{2.3}$$

As made explicit by the above description, a computer scientist is initially keen to produce a publication – possibly straight from her doctoral dissertation – but she needs coffee to produce her next publication. Coffee is only available through interaction with the departmental coffee machine CM. In order to describe systems consisting of two or more processes running in parallel, and possibly interacting with each other, CCS offers the *parallel composition operation*, which is written ' | '. For example, the CCS expression CM | CS describes a system consisting of two processes – the coffee machine CM and the computer scientist CS – that run in parallel one with the other. These two processes may communicate via the communication ports they share and use in complementary fashion, namely 'coffee' and 'coin'. By complementary, we mean that one process uses the port for input and the other for output. Potential communications are represented in Figure 2.2 by the solid lines linking complementary ports. The port 'pub', however, is used by the computer scientist to communicate with her research environment or, more