Communicating and Mobile Systems: the π-Calculus

ROBIN MILNER

Computer Laboratory, University of Cambridge

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 1999
Reprinted 2001
Reprinted 2003

Printed in the United Kingdom at the University Press, Cambridge

Typeface Times 10/13 pt *System* LaTeX 2$_\varepsilon$ [TB]

A catalogue record of this book is available from the British Library

Library of Congress Cataloguing in Publication data

Milner, R. (Robin). 1934–
Communicating and mobile systems : the π-calculus / Robin Milner.
p. cm.
ISBN 0 521 64320 1 (hc.). – ISBN 0 521 65869 1 (pbk.).
1. Mobile computing. 2. Telecommunication systems.
3. Pi-calculus. I. Title.
QA76.59.M55 1999
004.6′2–dc21 98-39479 CIP

ISBN 0 521 64320 1 hardback
ISBN 0 521 65869 1 paperback

Contents

1

Introduction

This book introduces a calculus for analysing properties of concurrent communicating processes, which may grow and shrink and move about.

Building communicating systems is not a well-established science, or even a stable craft; we do not have an agreed repertoire of constructions for building and expressing interactive systems, in the way that we (more-or-less) have for building sequential computer programs.

But nowadays most computing involves interaction – and therefore involves systems with components which are concurrently active. Computer science must therefore rise to the challenge of defining an underlying model, with a small number of basic concepts, in terms of which *interactional* behaviour can be rigorously described.

The same thing was done for *computational* behaviour a long time ago; logicians came up with Turing machines, register machines (on which imperative programming languages are built) and the lambda calculus (on which the notion of parametric procedure is founded). None of these models is concerned with interaction, as we would normally understand the term. Their basic activity consists of reading or writing on a storage medium (tape or registers), or invoking a procedure with actual parameters. Instead, we shall work with a model whose basic action is to communicate across an interface with a *handshake*, which means that the two participants synchronize this action.

Let us think about some simple examples of processes which do this handshaking. They can be physical or virtual, hardware or software. As a very physical system, consider a vending machine e.g. for selling drinks. It has links with its environment: the slot for money, the drink-selection buttons, the button for getting your change, the delivery point for a drink. The machine's pattern of interaction at these links is not entirely trivial – as we shall see in Chapter 2.

Physical systems tend to have permanent physical links; they have *fixed*

structure. But most systems in the informatic world are not physical; their links may be virtual or symbolic. An obvious modern example is the linkage among agents on the internet or worldwide web. When you click on a symbolic link on your screen, you induce a handshake between a local process (your screen agent) and a remote process. These symbolic links can also be created or destroyed on the fly, by you and others. Virtual links can also consist of radio connection; consider the linkage between planes and the control tower in an air-traffic control system. Systems like these, with transient links, have *mobile* structure. In Chapter 8 we shall look at a very simple example involving mobile telephones.

We do not normally think of vending machines or mobile phones as doing computation, but they share with modern distributed computing systems the notion of interaction. This common notion underlies a theory of a huge range of modern informatic systems, whether computational or not. This is the theory we shall develop.

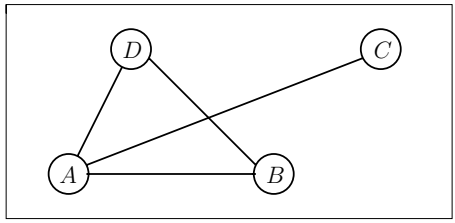
This book is not about design; for example, it will not teach you how best to design a concurrent operating system. Instead, we shall try to isolate concepts which allow designers to think clearly, not only when analysing interactive systems but even when expressing their designs in the first place. So we shall proceed with the help of examples – not large systems, but small ones illustrating key notions and problems.

A central question we shall try to answer is: when do two interactive systems have equivalent behaviour, in the sense that we can unplug one and plug in the other – in any environment – and not tell the difference? This is a theoretical question, but vitally important in practice. Until we know what constitutes similarity or difference of behaviour, we cannot claim to know what 'behaviour' *means* – and if that is the case then we have no precise way of explaining what our systems do!

Therefore our theory will focus on equivalence of behaviour. In fact we use this notion as a means of specifying how a designed system should behave; the designed system is held to be correct if its actual behaviour is equivalent to the specified behaviour. Chapters 7 and 13 contain several examples of how to prove such behavioural equivalence.

We shall begin at a familiar place, the classical theory of automata. We shall then extend these automata to allow them to run concurrently and to interact – which they will do by synchronizing their transitions from one state to another. This allows us to consider each system component, whether elementary or containing subcomponents, as an automaton.

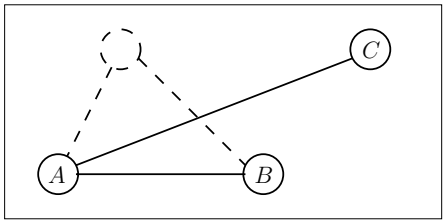
For such systems of interacting automata we shall find it useful to represent their interconnection by diagrams, such as the following:



(1)

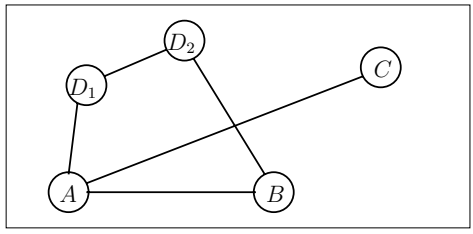
Here, an arc between two component automata A and B of a system means that they *may* interact – that is, A and B may sometimes synchronize their state transitions.

In many systems this linkage, or spatial structure, remains fixed as the system's behaviour unfolds. But in certain applications the spatial structure may *evolve*; for example the component D may *die* (1→2):



(2)

or may *divide* into two components (1→3):



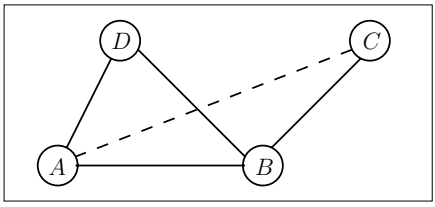
(3)

This mode of evolution covers a large variety of behaviour. For example, in understanding a high-level programming language one can treat each activation of a recursive procedure as a system component, whose lifetime lasts from a call of the procedure to a return from it; this extends smoothly to the case in which concurrent activations of the same procedure are allowed. Again, a communication handler may under certain conditions create a 'subagent' to deal with certain transactions; the subagent will carry out certain delegated interactions, and die when its task is done.

A calculus called CCS (Calculus of Communicating Systems) was devel-

oped along these lines in two books [9, 10] by the author. It was shown to represent not only interactive concurrent systems, such as communications protocols, but also much of what is familiar in traditional computation e.g. data structures and storage regimes. In fact the calculus was used to give a rigorous definition of a fairly powerful concurrent programming language. A similar model known as CSP (Communicating Sequential Processes) is described by Hoare [6]. These two models were independently conceived at roughly the same time, around 1980.

Returning to modes of evolution, there is a further mode in which new links are *created* between existing components, e.g. between B and C (1→4):



(4)

This mode of evolution may be called *mobility*; since links can both die and be created, one can model the movement of links between components. It is also possible to model the movement of the components (automata) themselves. For we may consider the location of a component of an interactive system to be determined by the links which it possesses, i.e. which other components it has as neighbours. If we think this way then movement, or change of location, is represented by change of linkage; so in the example shown – where also A and C have become disconnected – we can think of C having *moved* from A to B.

It can be argued that there are other forms of mobility; for example, a computing agent may move in a *physical* space, which is different from the *virtual* space represented by our links. We take this discussion up again in Chapter 8. Mobility – of whatever kind – is important in modern computing. It was not present in CCS or CSP, and we do not cover it here in Part I; but the theory we develop here extends smoothly to the *π-calculus*, introduced in Part II, which takes mobility of linkage as a primitive notion.

Any conceptual model, particularly in a young subject, has a problem with terminology. Ours is no exception; should we talk of automata, or processes, or systems, or components, or agents? All five have been used in this introduction. We shall mainly talk of *processes*, and of *process expressions* when we discuss mathematical notation for processes. At the beginning of the book we talk of *automata*, but only to relate our process theory to the pre-existing

theory. When we discuss how processes combine to make larger processes we talk of *systems* of *component* processes. For most of the book we shall avoid using the word *agent*, except when we are dealing with examples where the word appears appropriate in a non-technical sense; but in Part II we shall adopt a precise technical meaning for the word.