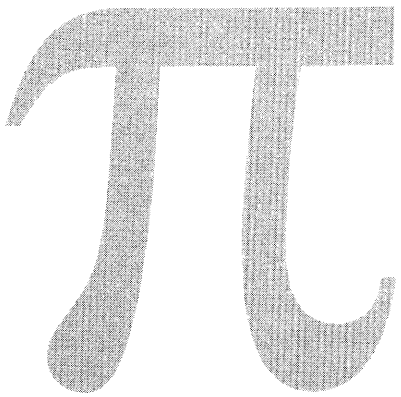
Communicating and Mobile Systems: the π-Calculus

Communication is a fundamental and integral part of computing, whether between different computers on a network, or between components within a single computer. In this book Robin Milner introduces a new way of modelling communication that reflects its position. He treats computers and their programs as themselves built from communicating parts, rather than adding communication as an extra level of activity. Everything is introduced by means of examples, such as mobile phones, job schedulers, vending machines, data structures, and the objects of object-oriented programming. But the aim of the book is to develop a theory, the π-calculus, in which these things can be treated rigorously.

The π-calculus differs from other models of communicating behaviour mainly in its treatment of mobility. The movement of a piece of data inside a computer program is treated exactly the same as the transfer of a message – or indeed an entire computer program – across the internet One can also describe networks which reconfigure themselves.

The calculus is very simple but powerful. Its most prominent notion is that of a name, and it has two important ingredients: the concept of behavioural (or observational) equivalence, and the use of a new theory of types to classify patterns of interactive behaviour. The internet, and its communication protocols, fall within the scope of the theory just as much as computer programs, data structures, algorithms and programming languages.

This book is the first textbook of the subject; it has been long-awaited by professionals and will be welcomed by them, and their students.



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 http://www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA http://www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1999

Printed in the United Kingdom at the University Press, Cambridge

Typeset in Times 10/13pt, in LaTeX 2ε [EPC]

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

Glossary

The important notations, with the section number of their first appearance.

ENTITY SET	ENTITY	DESCRIPTION	
$\mathcal{N}$	$a, \ldots, x, \ldots$	names	3.1
$\overline{\mathcal{N}}$	$\overline{a}, \ldots, \overline{x}, \ldots$	co-names	3.1
$\mathcal{L}$	λ	labels	3.1
$\mathcal{Q}$	$p, q, \ldots$	states	3.1
	$\mathcal{R}, \mathcal{S}, \ldots$	simulation, bisimulation	3.2
$\mathcal{P}^{\mathrm{seq}}$	$P, Q, \ldots$	sequential processes	3.4
Act	$\alpha, \beta, \ldots$	actions	4.2
	τ	silent action	4.2
$\mathcal{P}$	$P, Q, \ldots$	concurrent processes	4.3
	$\mathcal{C}$	process contexts	4.4
$\mathcal{P}^{\pi}$	$P, Q, \ldots$	π-calculus processes	9.1
	π	action prefixes	9.1
Σ	σ	sorts	11.2
Γ	C	sort constructors	11.3
	$F, G, \ldots$	abstractions	12.1
	$C, D, \ldots$	concretions	12.1
$\mathcal{A}^{\pi}$	$A, B, \ldots$	π-calculus agents	12.1

ACTION RELATIONS

$\xrightarrow{\alpha}$	labelled transition	3.1
$\xrightarrow{\alpha}$	commitment	12.2
$\longrightarrow$	reaction	4.5
$\Rightarrow$	empty experiment	6.1
$\overset{s}{\Rightarrow}$	experiment	6.1

BASIC CONSTRUCTIONS

$A(\vec{a}) \stackrel{\mathrm{def}}{=} P_A$	process definition	3.4
$\Sigma\alpha_i.P_i$	summation	3.4
$A\langle a_1, \ldots, a_n\rangle$	process instance	3.4
$\{\vec{b}/\vec{a}\}P$	substitution	3.4
$P \mid Q$	composition	4.3
new a P	restriction	4.3
$P^\frown Q$	linking	4.4

EQUIVALENCES

$\equiv$	structural congruence	4.4
$\sim$	strong equivalence	3.3
$\approx$	weak equivalence	6.2

π-CALCULUS CONSTRUCTIONS

$x(y)$, $\overline{x}\langle y\rangle$	action prefix (monadic)	9.1
$\Sigma\pi_i.P_i$	summation	9.1
$!P$	replication	9.1
$x(\vec{y})$, $\overline{x}\langle\vec{y}\rangle$	action prefix (polyadic)	9.4
$F;G$	sequential composition	9.6
$(\vec{x}).P$	abstraction	12.1
new $\vec{x}$ $\langle\vec{y}\rangle.P$	concretion	12.1
$F@C$	application	12.1
$\Sigma\alpha_i A_i$	summation of agents	12.1

Preface

Over the last thirty years or so, computer science has seriously taken up the challenge to understand the behaviour of communicating systems in the same way as it understands the behaviour of computer programs.

There is little pre-existing theory which can help. This is perhaps surprising, because the theory of computing has developed over a very long period as a part of mathematics and logic, and indeed it influenced the design of early stored-program computers. By comparison, a theory of communication as a smooth extension of programming is in its adolescence.

But theories usually arise to explain practice. Recently there has been a sea-change in computing practice; due to techological advances interactive systems are becoming the norm rather than the exception, and our whole view of computing has changed correspondingly. The new technology has created the need to expand our theory of sequential algorithmic processes to systems where interaction plays a significant and even dominant rôle.

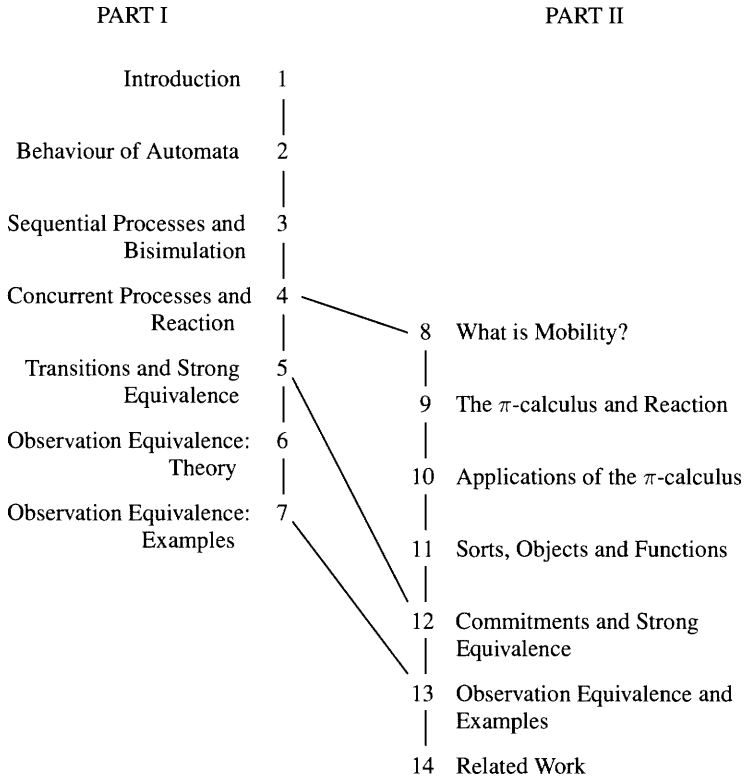
One of the most challenging developments, both technically and conceptually, is the advent of mobile computing. People, computers and software now continually move among each other; moreover, some of the movement is physical and some (e.g. the movement of links) is virtual. As we experience this, we must somehow distil basic ideas which will help us to create reliable mobile systems which do what we want them to do.

Analysing the behaviour of mobile systems at the design stage is much harder than it ever was for sequential computer programs. This is partly because we lack ways even to express such behaviour accurately, in order to specify what must be designed. The π-calculus was developed in the late 1980s with just this goal in mind; this book introduces it with motivation and examples, but also with mathematical precision.

Who should read the book The book has grown out of a lecture course of sixteen lectures to final-year undergraduate students at Cambridge. It is

designed for such a course. In making the book from the lecture notes I have resisted adding more material; I have only added explanations. The material is challenging for undergraduates; the book can also be used as a basis for graduate courses.

How to read the book The book divides clearly into two parts. Part I deals with interactive systems which are not mobile, and represents a self-contained review of previous work on CCS (a Calculus of Communicating Systems) [9, 10]. Part II introduces mobility, in the form of the dynamic creation of new links between active processes. But one need not read all of Part I before Part II. The diagram below shows the dependency of chapters.



There are many paths through some or all of the chapters:

- Part I, by itself is a good introduction to the algebraic treatment of communicating systems; it emphasizes the kind of theoretical problem which arises with concurrency, but preserves a balance between theory and examples.

- Chapters 1–4 and 8–11 make a good introduction to mobile interactive systems, with emphasis on applications and less upon the behavioural theory. The examples of Chapter 7 can be added to this diet; they can be appreciated to a reasonable extent without the preparation of Chapters 5 and 6.
- Chapters 1–5 and 8–12 make a coherent whole, dealing with everything except the concept of observation equivalence. Chapters 6, 7 and 13 can then be tackled together.

Thus the mixture of theory and examples can be varied to taste. The theorems are important, but very often the practical applications can be appreciated without them.

Acknowledgements I would first like to thank the students who have helped me over the last three years to write down these ideas in progressively better form, and particularly those who suffered the earlier attempts. It is a continual source of excitement to me that to teach a new subject is so important for one's understanding of it. The ideas in the π-calculus are due in great part to Mogens Nielsen and Uffe Engberg who took important steps towards it, to Joachim Parrow and David Walker who first worked the calculus out in detail with me, and to Davide Sangiorgi who made important subsequent advances. Alexis Donnelly, Peter Sewell and David Walker have read earlier drafts of the book in considerable detail and made valuable suggestions.

Part I

Communicating Systems

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 the notion of interaction with modern distributed computing systems. 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.

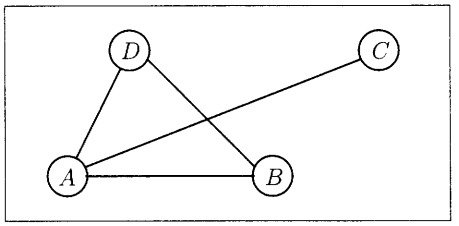
This book is not about design; for example, it will not teach you how best to design a concurrent operating system. Instead, we shall try to isolate concepts which allow designers to think clearly, not only when analysing interactive systems but even when expressing their designs in the first place. 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.

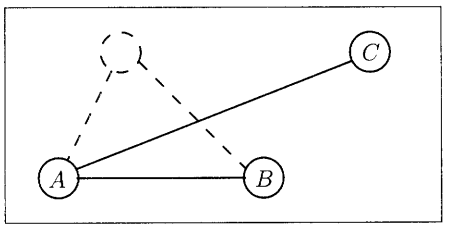
For such systems of interacting automata we shall find it useful to represent their interconnection by diagrams, such as the following:



(1)

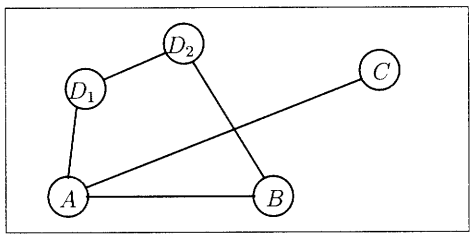
Here, an arc between two component automata A and B of a system means that they *may* interact – that is, A and B may sometimes synchronize their state transitions.

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):



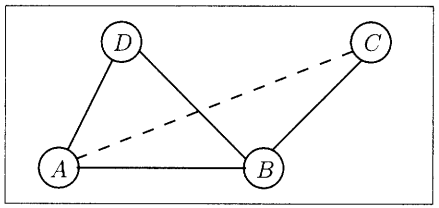
(3)

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 an 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 ($1 \rightarrow 4$):



(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.

2

Behaviour of Automata

We shall begin by considering the components of a system to be automata which interact with one another.

In the standard mathematical definition an automaton has a set of possible *states*, and moves from one state to another by performing certain *actions*. There is a well-developed theory, which we shall call the classical theory, of such automata. It deals with such matters as when two different automata may be regarded as having the same behaviour, and how automata can be classified in terms of this behaviour. Particularly important classes are the *finite state* automata (those with finitely many states) and the *deterministic automata* (those in which, in a given state, any action has only one possible outcome).

In the classical theory, rather little attention is paid to the way in which two automata may interact, in the sense that an action by one entails a complementary action by another. This kind of interaction requires us to look at automata in a new light; in particular, this interdependency of automata via their actions seems to demand a new approach to behavioural equivalence.

Nonetheless, classical automata theory is so important in many aspects of computing (e.g. in parsing theory) that we must take account of it in presenting any new theory, and must clearly indicate the points of agreement and difference. We therefore begin this chapter with a brief review of the classical theory and show where we shall depart from it. For a detailed account, see for example Hopcroft and Ullman [8] or Sudkamp [19].

2.1 Automata

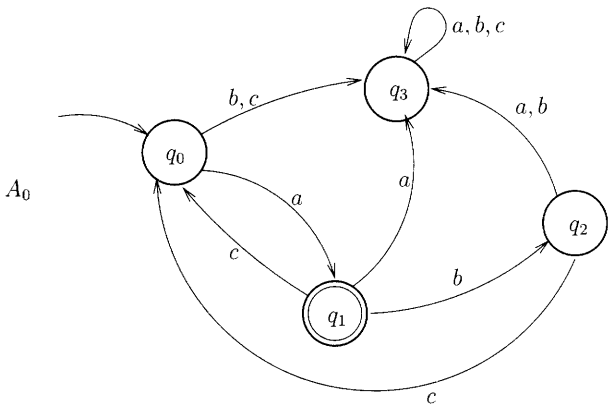
We presume a given set Act of actions, sometimes called an alphabet. The classical definition of an automaton is as follows.

Definition 2.1 Automaton *An* automaton A *over* Act *has four ingredients:*

- *a set* $\mathcal{Q} = \{q_0, q_1, \ldots\}$ *of* states*;*
- *a state* $q_0 \in \mathcal{Q}$ *called the* start state*;*
- *a subset* $\mathcal{F}$ *of* $\mathcal{Q}$ *called the* accepting states*;*
- *a subset* $\mathcal{T}$ *of* $\mathcal{Q} \times Act \times \mathcal{Q}$ *called the* transitions.

A transition $(q, a, q') \in \mathcal{T}$ *is usually written* $q \xrightarrow{a} q'$*. The automaton* A *is said to be* finite-state *if* $\mathcal{Q}$ *is finite, and* deterministic *if for each pair* $(q. a) \in \mathcal{Q} \times Act$ *there is exactly one transition* $q \xrightarrow{a} q'$*.*

An automaton is usually represented by a *transition graph*, whose nodes are the states and whose arcs are the transitions. As an example consider the following finite automaton A_0 over the alphabet $Act = \{a. b, c\}$:



The state-set of A_0 is $\{q_0, q_1, q_2, q_3\}$; it has just one accepting state q_1 (indicated by a double circle) and it is clearly deterministic. In the diagram, two transitions $\xrightarrow{b}$ and $\xrightarrow{c}$ between the same states are combined as $\xrightarrow{b,c}$.

(In some definitions the determinacy condition 'exactly one transition' is relaxed to 'at most one transition'. But there is little difference, because if this weaker condition is satisfied then one can always meet the stronger condition by adding new transitions which all lead to some new non-accepting state. Such a state is called a *sink*; you can think of q_3 in A_0 above as a sink.)

In the classical theory the behaviour of an automaton A is usually taken to be the set of strings over Act which it accepts; this set is often called the language of A, defined as follows:

Definition 2.2 Language of an automaton *Let* A *be an automaton over* Act*, and* $s = a_1 \cdots a_n$ *a string over* Act*. Then* A *is said to* accept s *if there is a path in* A*, from* q_0 *to an accepting state, whose arcs are labelled successively*

$a_1, \ldots, a_n$. *The* language *of A, denoted by* $\widehat{A}$*, is the set of strings accepted by A.*

For example A_0 accepts $abca$ via the path whose successive states are $q_0q_1q_2q_0q_1$.

2.2 Regular sets

We shall be concerned with sets of strings over *Act*, and we shall need to build bigger such sets from smaller ones. Three important operations for building sets of strings are:

$$\begin{array}{rrcl} \text{Union}: & S_1 \cup S_2 & & \\ \text{Concatenation}: & S_1 \cdot S_2 & \stackrel{\text{def}}{=} & \{s_1 s_2 \mid s_1 \in S_1,\ s_2 \in S_2\} \\ \text{Iteration}: & S^* & \stackrel{\text{def}}{=} & \{\epsilon\} \cup S \cup S{\cdot}S \cup S{\cdot}S{\cdot}S \cup \cdots \end{array}$$

Thus the iteration S^* consists of all strings $s_1 s_2 \cdots s_n$ $(n \geq 0)$ such that $s_i \in S$ for each i.

Definition 2.3 Regular set *A set of strings over Act is said to be* regular *if it can be built from the empty set* $\emptyset$ *and the singleton sets* $\{a\}$ *(for each* $a \in Act$*), using just the operations of union, concatenation and iteration.*

In expressions for regular sets we often write a for the singleton set $\{a\}$, and ϵ (the empty string) for the singleton set $\{\epsilon\}$. Conventionally '+' is used for set union, and we let '$\cdot$' bind more strongly than '+'. Thus, for example, $(a+b)\cdot c$ stands for the set $\{ac, bc\}$; on the other hand $a + b \cdot c$ stands for the set $\{a, bc\}$, and so $(a + b \cdot c)^* \cdot c$ stands for the set $\{a, bc\}^* \cdot \{c\}$. The latter set contains exactly all those strings of the form $s_1 s_2 \cdots s_n c$ $(n \geq 0)$ where each s_i is either a or bc.

It is well-known and easy to verify that concatenation obeys the following equations:

$$\begin{array}{lll} (S_1{\cdot}S_2){\cdot}S_3 = S_1{\cdot}(S_2{\cdot}S_3) & & (S_1 + S_2)\cdot T = S_1{\cdot}T + S_2{\cdot}T \\ S{\cdot}\epsilon = S \qquad S{\cdot}\emptyset = \emptyset & & T\cdot(S_1 + S_2) = T{\cdot}S_1 + T{\cdot}S_2\ . \end{array}$$

Iteration also satisfies interesting equations; one which will be useful later is

$$S{\cdot}(T{\cdot}S)^* = (S{\cdot}T)^*{\cdot}S\ .$$

You can see this by noting that each side of the equation contains exactly all those strings of the form $s_1 t_1 s_2 t_2 \cdots s_n t_n s_{n+1}$ with each $s_i \in S$ and each $t_i \in T$.

The algebra of regular sets is interesting in its own right, but we shall not explore it here. The main point which we want to recall is that the language $\widehat{A}$ of any finite-state automaton A is known to be regular. This depend on the following important fact:

Proposition 2.4 Arden's Rule *For any sets of strings S and T, the equation* $X = S\cdot X + T$ *has* $X = S^*\cdot T$ *as a solution. Moreover, this solution is unique if* $\epsilon \notin S$.

For example, the solution of $X = (a+b)\cdot X + c$ is $X = (a+b)^*\cdot c$. From now on we shall often drop the concatenation symbol '$\cdot$', writing SX for $S\cdot X$ etc.

2.3 The language of an automaton

Given an automaton A, we now recall how to find its language $\widehat{A}$.

Let A have states $\{q_0, \ldots, q_n\}$ with start state q_0. For $1 \leq i \leq n$, let X_i denote the set of strings accepted by A starting in state q_i; thus $\widehat{A} = X_0$. We can write an equation for each X_i, defining it in terms of the sets corresponding to its successor states. For example, for A_0 as in Section 2.1 above we have

$$\begin{array}{ll} (0) & X_0 = aX_1 + bX_3 + cX_3 \\ (1) & X_1 = aX_3 + bX_2 + cX_0 + \epsilon \\ (2) & X_2 = aX_3 + bX_3 + cX_0 \\ (3) & X_3 = aX_3 + bX_3 + cX_3\ . \end{array}$$

(Note that X_1 contains ϵ because q_1 is an accepting state.) So we use Arden's Rule to solve the equations. First note that (3) can be written in the form $X_3 = (a+b+c)X_3 + \emptyset$, so Arden's Rule yields

$$X_3 = (a+b+c)^*\emptyset = \emptyset\ .$$

(We are using the fact that $S\emptyset = \emptyset$.) Using this, we can simplify the remaining equations:

$$\begin{array}{ll} (0) & X_0 = aX_1 \\ (1) & X_1 = bX_2 + cX_0 + \epsilon \\ (2) & X_2 = cX_0\ . \end{array}$$

Substituting (0) and (2) in (1) we get $X_1 = (bc+c)aX_1 + \epsilon$, so by Arden's Rule we deduce $X_1 = ((bc+c)a)^*$, and from (0) finally

$$\widehat{A_0} = X_0 = a((bc+c)a)^*\ .$$

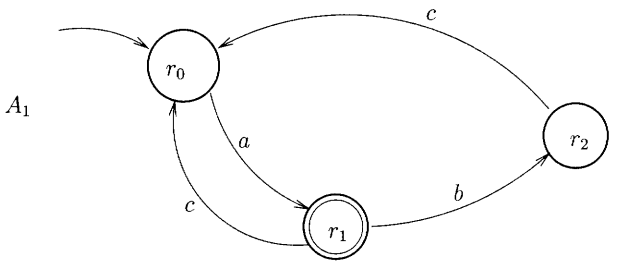
Note that in reaching this solution we have used algebraic properties of union and concatenation; for example, after substituting (0) and (2) in (1) we used $bcaX_1 + caX_1 = (bc+c)aX_1$, justified by the first distributive law

$(S_1 + S_2)T = S_1T + S_2T$. In other examples, one also needs the second distributive law, $T(S_1 + S_2) = TS_1 + TS_2$. But we shall shortly look at another interpretation for our algebra in which the second law is *not* valid.

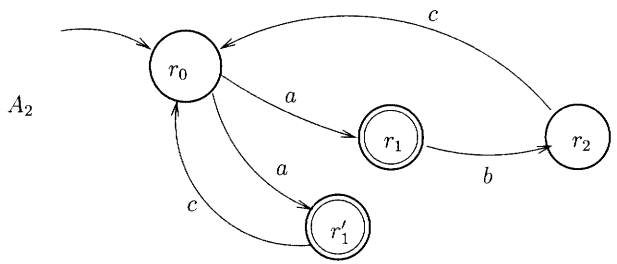
We have seen in this particular case that the language of a finite-state automaton is regular. In fact, this method of solution can be applied to any finite-state automaton, and this constitutes a proof that its language is always regular.

2.4 Determinism versus nondeterminism

Let us now ask whether the languages of deterministic automata differ somehow from those of nondeterministic ones. A simple example will suffice. Consider first A_1, defined by the following transition graph:



A_1 is deterministic, since there is at most one transition for each pair (q, a). In contrast consider A_2; it is very like A_1 but non-deterministic, because there are two a-transitions out of r_0:



By writing down the appropriate equations (as we did for A_0), and solving them, we can show that A_1 and A_2 possess the same language, and indeed it is the same as $\widehat{A_0}$. We leave this as an exercise.

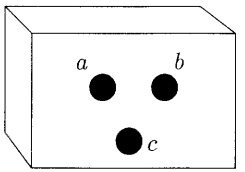
Exercise 2.5 In Section 2.2 the equation $S{\cdot}(T{\cdot}S)^* = (S{\cdot}T)^*{\cdot}S$ was mentioned. What alternative expression can be derived for $\widehat{A_0}$, using this equation? By solving equations using Arden's rule, find the languages $\widehat{A_1}$ and $\widehat{A_2}$; then, using the algebraic equations mentioned in Section 2.2, show that they are identical with $\widehat{A_0}$. ■

We now ask whether it is inevitable – or even desirable – that the automata A_1 and A_2 be regarded as equivalent, just because they accept the same language. Language-equivalence has some advantages; for example, every automaton can be converted to a deterministic one which accepts the same language. (You may recall that the conversion to a deterministic automaton is done by the so-called *subset construction*; see any text-book on automata theory.)

But realistic systems can be nondeterministic in an *intrinsic* way which should not just be explained away like this! We shall now demonstrate this with a simple example from everyday life.

2.5 Black boxes, or reactive systems

Let us now think of an automaton over $Act = \{a, b, c\}$ as a black box



with three buttons marked a, b and c. We interact with it by trying to press the buttons in some sequence. Sometimes the button goes down i.e. we succeed – and sometimes it doesn't; this is the only way we can tell the difference between black boxes with the same alphabet. In particular, we can't tell the difference between an accepting and a non-accepting *state*; we can't tell anything about states except indirectly through the behaviour of the buttons. (We shall often use the words 'button' and 'port', interchangeably, for the means of access to a process.)

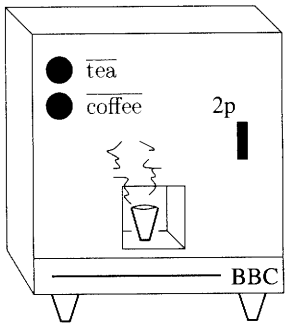
Thus we diverge in a subtle but important way from the classical notion of automaton. What matters about a string s – a sequence of actions – is not whether it drives the automaton into an accepting state (since we cannot detect this by interaction) but whether the automaton is able to perform the sequence s interactively. We reflect this by confining our attention to automata in which *every* state is an accepting one, A consequence is that if an automaton accepts s, then it also accepts any initial part of s. We now make this precise:

Definition 2.6 Prefix closure *If a string s can be expressed in the form s_1s_2, then s_1 is said to be a* prefix *of s. A language S is said to be* prefix-closed *if, whenever $ss' \in S$ then also $s \in S$. The* prefix-closure *of a language S is the larger language $Pref(S)$ which contains all the prefixes of every string in S. It is the smallest prefix-closed language which includes S.*

For example, if $S = \{a, bcd\}$ then its prefix-closure is $Pref(S) = \{\epsilon, a, b, bc, bcd\}$.

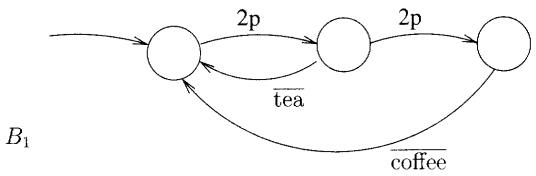
Exercise 2.7 By considering accepting paths (as in Section 2.1), verify informally for any automaton A that if all its states are accepting then $\widehat{A}$ is prefix-closed. ■

Example 2.8 Vending machine Here is a tea/coffee vending machine, a black box with a three-symbol alphabet $\{2p, \overline{tea}, \overline{coffee}\}$. (For now ignore the overbars.)



* Black Box Corporation

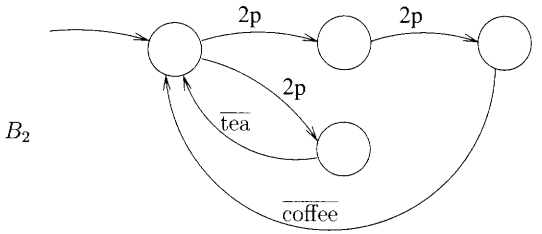
Its internal state-transition diagram could be as follows:



This means that after putting in 2p, you can get tea by pressing the $\overline{tea}$ button, or you can put in another 2p and then get coffee by pressing $\overline{coffee}$, and so on. ■

Exercise 2.9 By solving equations or otherwise, show that the set of strings which take B_1 back to its start state is given by $(2p \cdot (\overline{tea} + 2p \cdot \overline{coffee}))^*$. Then, assuming that every state is accepting, deduce that $\widehat{B_1} = Pref((2p \cdot (\overline{tea} + 2p \cdot \overline{coffee}))^*)$. ■

Exercise 2.10 Consider a variant of the machine B_1, as follows:



Verify that $\widehat{B_2} = \widehat{B_1}$. ■

Despite the fact that B_1 and B_2 are language-equivalent, they are annoyingly different for a thirsty user! This is because B_2 is more nondeterministic; after we have put in the first 2p, it may be in a state in which we can only get tea (it will not accept a further 2p), or it may be in a state in which we can only put in more money to get coffee (it will not allow the $\overline{tea}$ button to be pressed). If you bought a vending machine and found it to behave like this, you would ask for your money back.

So what is wrong with the theory which has allowed you to deduce that $\widehat{B_2} = \widehat{B_1}$, in Exercise 2.10? The answer is that you had to use an instance of the second distributive law, i.e. you used

$$2p \cdot (\overline{tea} + 2p \cdot \overline{coffee}) = 2p \cdot \overline{tea} + 2p \cdot 2p \cdot \overline{coffee},$$

in order to equate a deterministic machine with a less reliable non-deterministic one.

2.6 Summary

We have briefly reviewed classical automata theory, especially its notion of language-equivalence. We found that this notion of equivalence is not suitable for all purposes. In particular, it does not appear to be correct when an automaton's actions consist of *reactions* between it and another automaton; an example was given in the form of a vending machine interacting with a purchaser.

It has appeared that if we are interested in interactive behaviour, then a non-deterministic automaton cannot correctly be equated behaviourally with a deterministic one. So a radical departure from the classical theory is required; we take up this challenge in the chapters which follow.