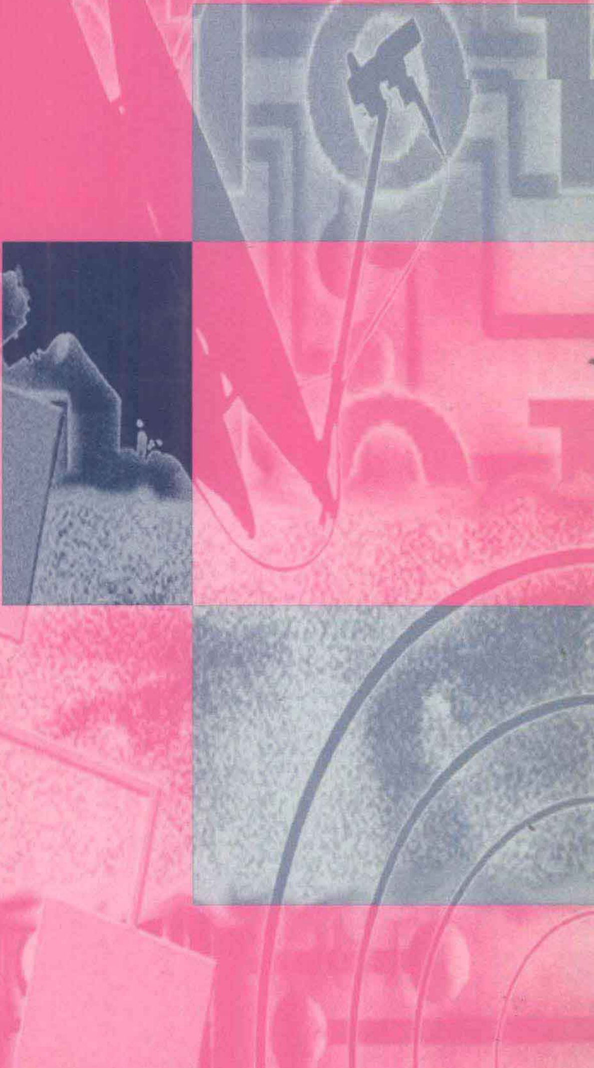
Applied Computing

INTRODUCTION TO THE FORMAL DESIGN OF REAL-TIME SYSTEMS

David Gray



Springer

David Gray

Introduction to the Formal Design of Real-Time Systems

Springer

David Gray, BSc, MSc, PhD, CEng

Series Editors

Professor Peter J. Thomas, BA (Hons), PhD, AIMgt, FRSA, FVRS
Centre for Personal Information Management, University of the West of England, Coldharbour Lane, Bristol, BS16 1QY, UK

Professor Ray J. Paul, BSc, MSc, PhD
Department of Information Systems and Computing, Brunel University, Uxbridge UB8 3PH, UK

ISBN 3-540-76140-3 Springer-Verlag Berlin Heidelberg New York

British Library Cataloguing in Publication Data
Gray, David
Introduction the Formal Design of Real-Time Systems - (Applied computing)
1.Real-time data processing 2.System design
I.Title
004.3'38'1
ISBN 3540761403

Library of Congress Cataloging-in-Publication Data
Gray, David, 1946-
Introduction the formal design of real-time systems / David Gray
p. cm. -- (Applied computing)
Includes bibliographical references and index.
ISBN 3-540-76140-3 (pbk. : alk. paper)
1. Real-time data processing 2. System design I. Title II. Series
QA76.54.G73 1997
005.2'.73--dc21 97-29239

Illustrations by the author
Typesetting by Editburo, Lewes, East Sussex
Printed and bound at the Athenæum Press Ltd., Gateshead, Tyne and Wear
34/3830-543210 Printed on acid-free paper

Applied Computing

Springer
London
Berlin
Heidelberg
New York
Barcelona
Hong Kong
Milan
Paris
Santa Clara
Singapore
Tokyo

No, No, No .....
it only works if we all play the right bits at the right time

Preface

This book is about applying an easily understood formal method, the process algebra SCCS, to the design of concurrent and real time systems. It charts the progress of a non-mathematician through the process of learning a formal method from the creation of designs to their validation using automated tools.

But why formal methods? As a young electrical engineer I was taught by people who believed that engineers, by employing rigorous mathematical theories, correctly predict how designs will behave before they are built; if they couldn't they weren't engineers. It came as something as a shock on first entering the world of software programming to find that such techniques were thought foolish and unnecessary. It was with some relief that I eventually stumbled upon several groups of people applying the rigour of mathematics to the design of correct software systems – perhaps they were taught by the same sort of lecturers as I was?

Based on formal methods, the book's main premise is that, in order to design something correctly, one has to understand it fully. One does not understand something if one's theories about it are incorrect, contradictory or incomplete, and only by expressing our theories in a formal way can they be checked with sufficient rigour. As Martin Thomas of Praxis put it cogently in 1989, 'If you cannot write down a mathematical behaviour of the system you're designing, you don't understand it'. This is not to say that formal methods are a philosopher's stone; rather than replace traditional methods, they complement them. The ideas that drive a design may still come from many sources – your experience, the experience of others, educated guesses, inspiration, hunches, tarot cards, runes, or whatever – but, to check if your ideas hold water, what better than formal tools to animate and apply some rigour to the design process?

The formal method used here is the process algebra SCCS. No previous knowledge of SCCS is presumed: it's both introduced and applied here. SCCS was selected for the task because not only does it capture the behaviour of real-time systems, it does so in a simple and elegant way based on the natural concept of observation – things are considered equivalent if an observer cannot tell them apart. Equally as important, SCCS is a member of an extended family of algebras for which automated tools are available.

Intended Audience

The material that forms the book's content was developed as part of a set of modules given to a variety of students at Brunel and Surrey Universities. The students included undergraduates majoring in electronic and electrical engineering, information technology, mathematics and computer science, as well as graduate students studying telematics, microelectronics and computer engineering. To cater for such disparate backgrounds the material is, as far as possible, self-contained; it is aimed at people who already have some experience with computer systems – about as much as might be achieved by a couple of years of a full-time academic course, or the equivalent in practical experience. Readers should be able to follow the programming examples, and ideally have some understanding of, or may even have met, the problems addressed here. At first I thought a reasonable knowledge of discrete mathematics would be mandatory, but all of the students who followed the original modules took to the formalism like ducks to water, finding the concepts and theories intuitive, process algebras common sense, and only the act of linking the concepts to the notations problematic.

Layout

The book has four main parts:

- The first introduces and defines the problems associated with the correct design of concurrent and real-time systems, and stimulates awareness of the pitfalls of concurrent systems design in addressing some examples by adding special extensions to sequential programming languages.
- The second section introduces the process algebra SCCS in a multi-pass fashion. An informal overview of the complete calculus is followed by a formal introduction to the basic algebra. Examples are given which extend the basic algebra to directly address particular classes of problems, which are then used to devise further extensions.
- The third section dwells on the role of equality. It is one thing to say that our objective is to prove a design equivalent to a specification,

but when we state that A 'equals' B , as well having to know what we mean by A and B we also have know what we mean by 'equals'. This section explores the role of observers; how different types of observer see different things as being equal, and how we can produce algorithms to decide on such equalities. It also explores how we go about writing specifications to which we may compare our SCCS designs.

- The final section is the one which the students like best. Once enough of SCCS is grasped to decide upon the component parts of a design, the 'turning the handle' steps of composition and checking that the design meets its specification are both error-prone and tedious. This section introduces the concurrency work bench, which shoulders most of the burden.

How you use the book is up to you; I'm not even going to suggest pathways. Individual readers know what knowledge they seek, and course leaders know which concepts they are trying to impart and in what order.

Acknowledgements

The list of people I wish to thank includes my parents and teachers; responsible for most of my training, I only hope they're not too dissatisfied with the outcome. I'd also like to thank the following. The students at both Brunel and Surrey Universities, for their forbearance and active feedback while the material was being developed; it was their input that resulted in the multi-pass nature of the book. In particular, one of my PhD students, Jason Cozens, whose work on higher order SCCS gave rise to impassioned discussions which made me rethink certain concepts expressed here. Bernie Cohen, for encouraging my first formal steps. Steve Schumann and Mike Shields, with whom I gave courses and from whom I 'borrowed' the odd example. Stephan Zuldt, an experienced engineer with an interest but no formal training in this area, who gave an outsider's view of the text. Perdita Stephens, who answered my questions about the Concurrency Workbench. My wife, who walked the dogs when it was my turn. But mainly the authors of books and papers and the givers of presentations and seminars, who began and then nurtured my interest in formal methods, foremost amongst whom are Professor Milner, the originator of SCCS, whose book [Mil89] I still return to for insights missed in previous readings, and Professor Stirling, whose clarity of writing does much to demystify modal logic. I also have to thank Professors Milner and Joseph for introducing me to Tomasz Janowski, and Professor Stirling for introducing me to Sibylle Froschle, both of whose reviews changed the eventual content and presentation of the material.

Finally I wish to dedicate this book to Rachel Rochester Warlow.

Contents

1 Scene Set

Introduction

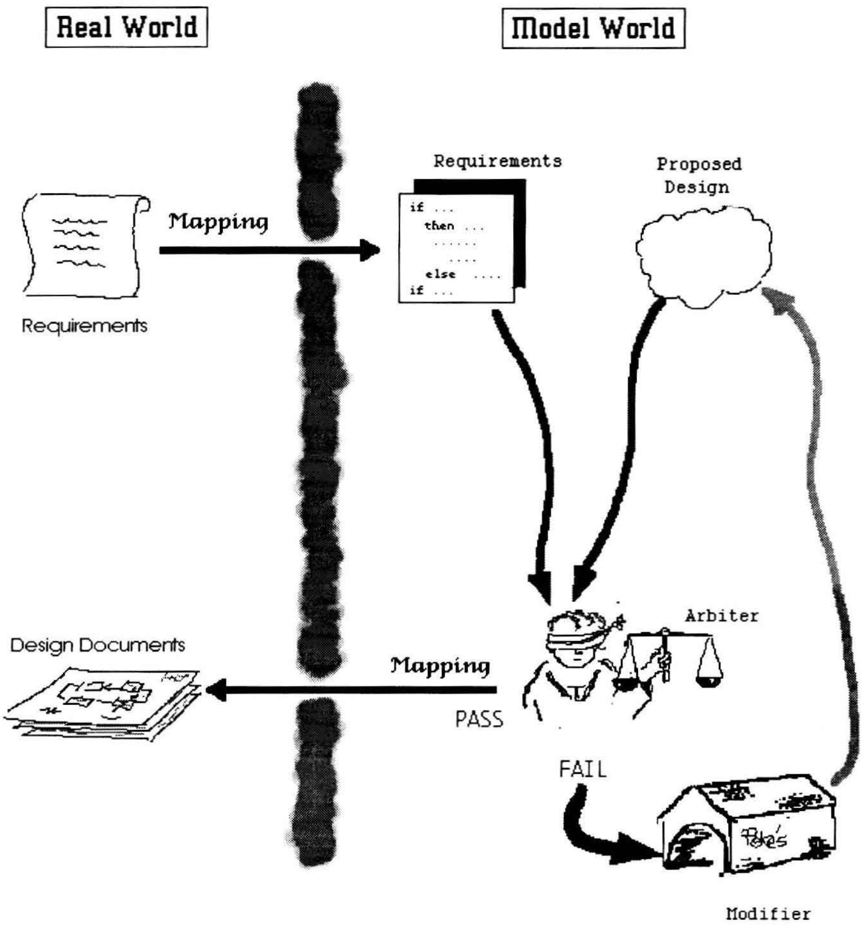
One of the most endearing preoccupations of human beings is the way they attempt to understand the world around them and then try to improve it, with varying degrees of success. Over the years it's been man's insatiable curiosity – '...but what happens next?' – that has driven advances in both science and engineering. As engineers we make our living out of this desire; we don't just design things, we know how objects, yet to be built, will behave, and we can predict whether they will answer to our customers' requirements (and even if we don't, we should). In more formal terms, engineers not only design systems but prove these designs satisfy their specifications.

But how do we build things that behave correctly? The artisan approach is just to go ahead and start building, beginning with something with a behaviour near to what is required and then repeatedly testing and modifying it until we get what we want. One problem with this method is its lack of scalability – we can use it to build simple systems but the method doesn't scale up to complex ones. Proving a system meets its specification also has scalability problems. To prove a system correct we need to test it exhaustively. While smaller systems may need few tests (for example, to test a lamp and switch to exhaustion we only need to show that pressing the switch turns the light on and releasing it turns the light off), to test to exhaustion even medium-sized systems, for example a single computer chip, can take man years. During that time the customer may get impatient and think of the wait to get paid.

Unfortunately, many of the objects we might wish to create are just too complex to build, test and modify as a prototype, or even as an accurate physical model. We need a more abstract modelling system, preferably one with a mathematical framework, so that we can design with pencil on paper, test by computer simulation, and use automated provers to check for correctness. Such an abstract modelling method must faithfully capture the behaviours of objects, enable us to manipulate those behaviours in a predictable way, predict the behaviours of objects composed of other objects, and finally compare behaviours so that we can check whether the object meets its specification. What we need is a modelling method which has operations, and laws over those operations, that replicate objects' behaviours. We need an algebra of objects.

How would such an algebra-based modelling method revise our design approach?

We commence any design with a set of requirements which state how we wish the final object to behave. We then propose, in the abstract world of models, a design that may satisfy those requirements



Using our algebra upon models we can predict how the design will behave; then, using an arbiter, we can test if this behaviour satisfies the requirements. If the arbiter decides the design is satisfactory we can proceed to build the real-world object. If not, we revise the design, before submitting it again to the arbiter. There's a small problem with this scenario. When making comparisons, an arbiter can only compare like with like. To compare an expression in our algebra of models with a specification, both must be defined in the same terms. The specification must either be expressed in terms of the algebra of models or isomorphic to them.

While this process is similar to the artisan method, the design is no longer carried out in the physical world but in the more tractable world of abstract models. When we avail ourselves of strong mathematical underpinnings, the construction, testing, and modification of a design can be carried out more simply, reliably and quickly.

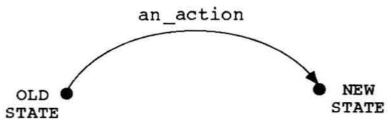
For our design method to be successful, we require that our models behave like the real thing – but only in those aspects in which we are interested. A model aircraft might fly, but it need not contain all the detail of a real one. By reducing complexity we reduce the possibility of error. But how do we know what behaviour is essential? What not? And how do we represent such behaviour in a model?

1.1 Making Models

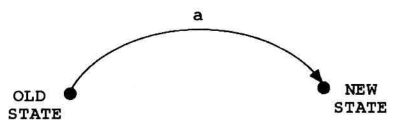
When modelling the behaviour of a thing

we have conflicting goals. We want the model to be simple – the simpler the better – but it must also completely capture the important behaviour of the thing. As we define the behaviour of a real-world object in terms of the actions in which it is seen to engage we will have created an adequate model if an observer cannot distinguish between the actions of our creation and those of the real thing.

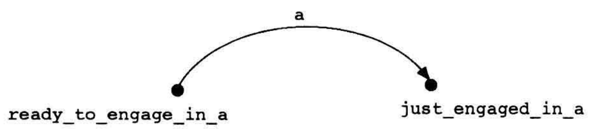
As actions are to be the basis of all our models, we need to know more about them. In the simplest terms, an action is something which moves an object between discernible states.



We can label actions with abstract names. Here, instead of `an_action`, we substitute the label '`a`',

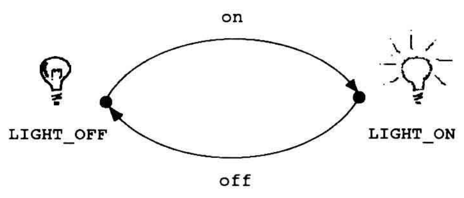


and the states of a system are defined purely in terms of these observed and labelled actions.



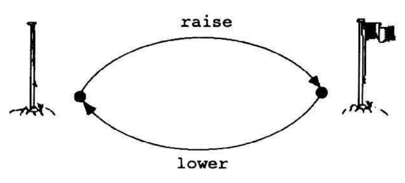
Here, action a moves the system from state ready_to_engage_in_a to state just_engaged_in_a. The states are only significant with respect to actions. If a was the last observed action, we can only be in state just_engaged_in_a. If we've not observed any action, we must be in the initial state ready_to_engage_in_a.

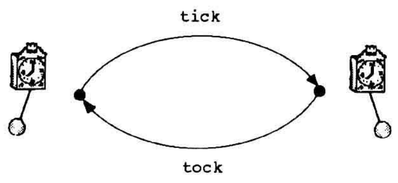
We can, if we wish, label the states and actions with names relevant to the states and actions in the system being modelled. For example, in



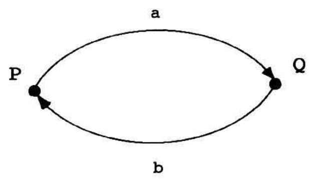
the state of the light is switched between LIGHT_ON and LIGHT_OFF by the on and off actions.

The same model can be interpreted using different objects in different environments. The light model behaves the same as these flag and clock models.





All have two actions moving the systems between two states. We could abstract this to a system in which a and b actions move the system between states P and Q.



Devising laws over behaviours for such generalised models allows us to make subsequent interpretations as to how a particular instantiation will behave. Renaming the general (a,b) in terms of the particular gives us the light system (on,off), the flag system (raise,lower), and the clock system (tick,tock). The benefit of taking abstract views is that one system containing certain action sequences is the same as any other with the same sequences; only the action and state names are changed to better reflect the system modelled.

Instead of the pretty, but difficult to manipulate, pictures, we can represent a model's actions and subsequent state changes in terms of formulae. 'A system in state P can engage in action a and in so doing move to state Q.' The states P and Q represent agents – they portray, in mathematical terms, the behaviour of an associated real-world system. Agent P is defined by expression a→Q.

$$P \stackrel{\text{def}}{=} a \rightarrow Q$$

In other words, 'Agent P can engage in action a and in so doing become agent Q.' Similarly 'Agent Q can engage in action b and in so doing become agent P.'

$$Q \stackrel{\text{def}}{=} b \rightarrow P$$

Finally, the complete system can be expressed as a recursive equation. 'Agent P can engage in actions a then b and so become agent P again.'

$$P \stackrel{\text{def}}{=} a \rightarrow b \rightarrow P$$

These general action and agent names can be instantiated to particular systems.

$$\mathtt{LIGHT} \stackrel{\text{def}}{=} \mathtt{off} \rightarrow \mathtt{on} \rightarrow \mathtt{LIGHT}$$
$$\mathtt{FLAG} \stackrel{\text{def}}{=} \mathtt{lower} \rightarrow \mathtt{raise} \rightarrow \mathtt{FLAG}$$
$$\mathtt{CLOCK} \stackrel{\text{def}}{=} \mathtt{tick} \rightarrow \mathtt{tock} \rightarrow \mathtt{CLOCK}$$

And so on.

Such state- and action-based mathematical models are not new. In the world of electronics, Meally–More diagrams, a type of labelled transition system, are used to design and check the correct operation of digital circuitry. In the field of computer systems vector diagrams, showing how processors change state when they engage in software instructions, have long been used to check if a program performs correctly.

The ability to model single objects is only part of the picture. Objects do not exist in isolation. Something must raise the flag, switch on the light, hear the clock tick, etc.. Objects exist in environments with which they interact. For example, when we observe and record an object's actions, the observer is part of that object's environment. The existence of environments means that an object must engage in