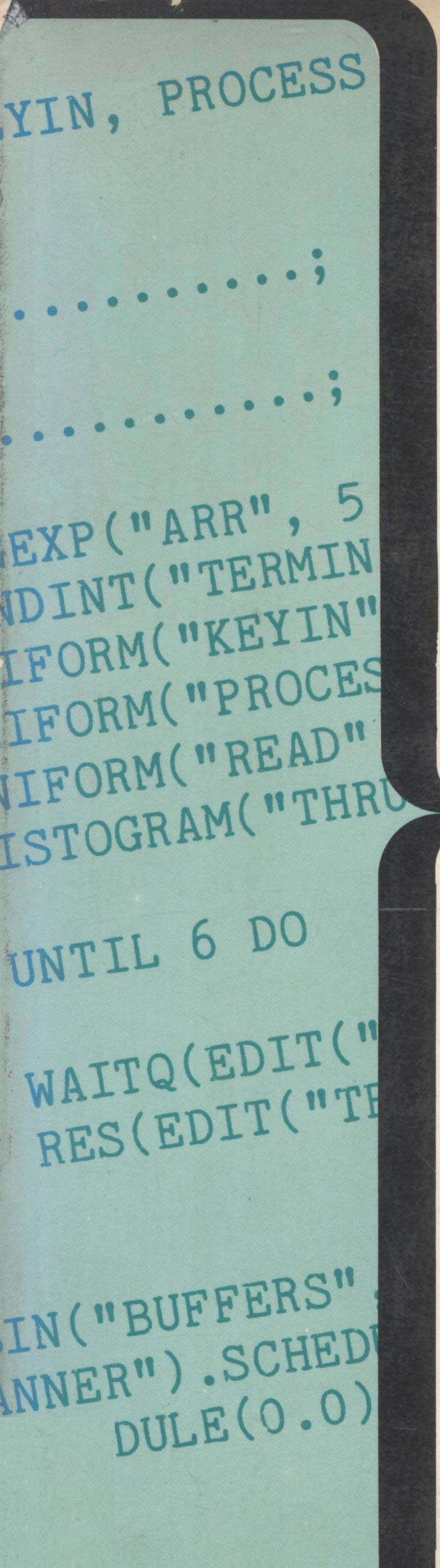

Discrete Event Modelling on Simula

G.M. Birtwistle

DEMOS

A System for

Discrete Event Modelling on Simula

G. M. Birtwistle

University of Bradford

M

First published 1979 by
THE MACMILLAN PRESS LTD
London and Basingstoke
Associated companies in Delhi Dublin
Hong Kong Johannesburg Lagos Melbourne
New York Singapore and Tokyo

Printed in Hong Kong

British Library Cataloguing in Publication Data

Birtwistle, G M
A system for discrete event modelling on SIMULA.–(Macmillan computer science series).
1. Computer simulation 2. DEMOS (Computer program)
I. Title
001.4′24 QA76.9.C65

ISBN 0-333-23881-8

PREFACE

This book is a primer on discrete event simulation modelling using the DEMOS package. It is written in informal style as a teaching text and is not meant as a reference manual. It should thus be read from start to finish and not dipped into at random. The book covers DEMOS fairly completely and uses it as a vehicle in which to describe several simulation models. As we have not aimed to produce a general text, no attempt has been made to cover the statistical side of discrete event simulation.

DEMOS is implemented in the general purpose language SIMULA (an extension to ALGOL 60). Thus DEMOS programs may be run on any computer that supports SIMULA (see references [3 - 10]). DEMOS is available free of charge as a SIMULA source program (see page 145 for more details).

SIMULA (Dahl et al. [1]) itself contains simulation primitives sufficient to build any simulation model, but leaves it to the user himself to flesh out the primitives in the style of his choice. While this puts the SIMULA expert in an enviable position, it is at first sight unfortunate for the beginner or occasional user of SIMULA. For it would seem that he has to acquire considerable expertise in SIMULA before he can start building out these primitives and actually get down to describing the simulation model itself. But this is not so. The situation was foreseen by the designers of SIMULA and they provided a way round the problem, namely the CONTEXT (= block prefix) mechanism. A context is a package written in SIMULA which extends that language towards a specific problem area. It will define the basic concepts and methods associated with the area, but leaves it to the modeller to apply them in his own way.

DEMOS is a context intended to help beginners in discrete event simulation get off the ground. It augments SIMULA with a few building blocks which provide a standardised approach to a wide range of problems. DEMOS invites model description in terms of ENTITIES and how they compete for RESOURCES. Written in terms of these concepts, DEMOS programs are bona fide SIMULA programs, but SIMULA programs which conform to a very simple format. They can thus be written and understood without a specialist knowledge of SIMULA. (This is very typical of contexts: their use requires much less SIMULA expertise than their writing.)

This book is based on material developed for undergraduate and postgraduate courses in Discrete Systems given at Bradford University, England, and for other courses given to industry. It has been written in tutorial style, each new feature being first motivated and then used in an illustrative example. Inevitably with a book written in this style, one or two DEMOS facilities could not be fitted in (a list of these is given in chapter 8, page 142). The DEMOS Reference Manual (Birtwistle [14]) covers the implementation of DEMOS in full; it gives proper documentation and SIMULA source listings for all the facilities of DEMOS.

For nearly all the DEMOS models in this book, we have first outlined our solution pictorially by means of activity diagrams and then given the corresponding DEMOS code. Input and output details are also recorded where appropriate. The main text is spread over eight chapters. Chapter 1 provides a brief introduction to discrete event modelling, and explains why DEMOS came to be written.

Chapter 2 provides a tutorial on that small sub-set of SIMULA we require (a check list of SIMULA declaration and statement types is given in appendix A). N.B. The reader is assumed to have a working knowledge of ALGOL 60. ALGOL 60, SIMULA and DEMOS programs all take the form

```
BEGIN
  declarations;
  statements;
END;
```

ASIDE: All programmers have idiosyncrasies and one of the author's is to include semicolons before each END after the final statement in a block or compound statement, and also after the final END. These are optional in SIMULA, and hence in DEMOS.

Chapter 3 illustrates the basic DEMOS approach to discrete event simulation model building (which has been inherited from SIMULA). With this approach, a system is described in terms of its constituent components (we call them ENTITIES) and a full action history describing its behaviour pattern is given for each entity. The separate entity descriptions are pieced together to describe the behaviour of the system as a whole. This approach enables the system modeller to focus his attention on the description of one entity at a time and is very natural.

In discrete event simulations, entities may compete with each other for system resources, cooperate with each other to perform a sequence of tasks, or even interrupt one another. Chapters 4, 5, 6 and 7 consider these basic synchronisation problems in turn, and show how they can be described in terms of DEMOS mechanisms.

In chapter 8, we first tidy up a few loose ends and then remark on the implementation of DEMOS as a package in SIMULA.

Each chapter contains several exercises which are best attempted when met in the text. They form an important part of the book: several reinforce or extend points just made in the main text, and some form a lead into the next section. Answers to all but two exercises are given at the end of the book. Regretfully, space considerations prevented us from including activity diagrams and output for all our solutions. That would have been nice.

Many individuals have helped make this book possible by their advice and encouragement through the years. Very special thanks are due to my gurus over several years: Kristen Nygaard (for imparting the SIMULA ethos) and Robin Hills (the same for discrete event simulation). Alan Benson, Ole-Johan Dahl, Roy Francis, Lars Enderin, Paul Luker, Mats Ohlin, Rod Wild and Norman Willis read the manuscript and helped remove several errors and infelicities of style. Any remaining errors are solely mine. Sorry.

DEMOS itself, and all the programs contained in this book, were developed on the Leeds University DEC System 10 computer using the excellent SIMULA compiler written by the Swedish Defense Research Establishment, Stockholm. Thanks are due to the Leeds University Centre for Computer Studies for permission to use their machine, and to several individuals there for their able, cheerful, and willing assistance.

This manuscript was prepared by the author using the DEC utilities SOS and RUNOFF. The page size of 50 lines by 62 characters has meant some compromises. Firstly not all our diagrams could be fitted on to a single page: these we have managed to split in a reasonable manner. Secondly program listings, which can usually be spread over 72 columns, have had to be narrowed down. However, because SIMULA is a free-format language, only a few programs seem to have suffered. Finally, the output from DEMOS programs also reckons on a 72 character line. Accordingly all output listings have been 'doctored' by squeezing out some unnecessary blanks.

In the formal description of SIMULA there are several symbols which are not reproducible on standard line printers. The representation of SIMULA programs in this book follows the recommendations of the SIMULA Standards Group. Key words are reserved and written in upper case (e.g. BEGIN, PROCEDURE, IF). Other changes are: array brackets('[' replaced by '(' and ']' by ')'), exponentiation ('**'), integer division ('//'), greater than or equal ('>='), less than or equal ('<='), not equal (NE), logical and (AND), logical or (OR), logical not (NOT), and power of 10 (E).

CONTENTS

1 INTRODUCTION

All around us in everyday life are complex systems of men and machines. Automobile plants, steel foundries, telephone exchanges, ticket reservation systems, banking systems, air flight control systems, local transport systems, etc. spring to mind. For these to function properly, we need to be able to understand them and how they react to emergencies (perhaps a bus breaks down in the rush hour), continual high pressures (rush hour traffic) as well as under normal circumstances (traffic in off-peak periods). Since the world is continually changing, systems have to adapt to new circumstances, e.g. how does the building of a new town nearby affect the local bus company? Which extra services should be provided and thus how many extra buses and crew will be needed? We may also need to implement totally fresh systems - how then do we justify and test our designs?

For all but the very simplest systems, we cannot just go ahead, implement a change and see what happens. It may prove too costly (who would build a new metro system in a town 'just to see if it is needed'?); it may even prove catastrophic (a new air traffic control system, or a new control program for a chemical plant). We have thus a distinct need to be able to experiment with adaptations of existing systems and test proposed designs without actually disturbing them or building them respectively. Here simulation can help.

Simulation is a technique for representing a dynamic system by a model in order to gain information about the underlying system. If the behaviour of the model correctly matches the relevant behaviour characteristics of the underlying system, we may draw inferences about the system from experiments with the model and thus spare ourselves any disasters.

Practical simulation work involves:

1. specification of the problem and satisfactory answers to such questions as: "Is it worth doing?", "Can it be done within our time scale and budget?", etc.

2. building a model which describes the system. We have used an adaptation of the well known activity diagram technique (explained in chapter 3) to represent pictorially the logic of the models developed in this book. In real life situations, it is important to have such a high level representation of the model so that the modeller can discuss his understanding of reality with the specialists who run the actual system. Whoever they are, be they managers, foremen, or workers, they are unlikely to understand computer programs and so cannot be expected to read a program text and point out logical flaws in a model. Yet feedback from them is essential. They must understand (at least) how their part of the system is represented in the model and so be able to confirm what has been done correctly, point out what has been omitted, and draw attention to those parts which do not function exactly as the official rule book states. Not many systems work exactly as planned and the modeller has to describe a given system as it actually is.

3. converting the model into an operating DEMOS program. This step is quite straightforward, almost mechanical, from the appropriate activity diagram - a second important reason for using them. Indeed, activity cycle diagrams can be used as high level flow charts for simulations written in activity, event, process or transaction mode.

4. validating the model by checking its consistency with the underlying system before any changes are made. The success of this validation establishes a basis of confidence in the results that the model generates under new conditions. Inadequate consistency will cause the modeller to try again from step 2 or step 3 above.

5. using the computer simulation program as an experimental tool to study proposed changes in the underlying system that the program represents.

This book makes no attempt to cover steps 1, 4, or 5 above. For thorough accounts of the important topics of model validation, output analysis, and the design of experiments, etc., the reader is instead referred to the excellent texts of Fishman [32] and Shannon [37].

In this book we cover steps 2 and 3, first representing our models by activity diagrams and then presenting the corresponding DEMOS programs.

Unlike most languages used for discrete event simulation, SIMULA does not force the user into one style of modelling. (See Hills [13] for a non-trivial model coded first in activity, then event, and finally process mode.) The designers of SIMULA included a standard context called SIMULATION which contains a sort of common denominator to all these three styles, but left it to the user to build this out. Thus if SIMULATION is to be used as it stands, a style of model building has to be developed and the user has to write his own synchronisation routines, data collection routines, etc. Some of these prove to be fairly subtle.

DEMOS extends SIMULATION by a few basic concepts which provide the operational research worker with a standardised approach to a wide range of discrete event problems. These are primarily the ENTITY for mirroring major dynamic model components whose complete life cycles warrant description in the model, and the RESOURCE for representing minor components. In addition, DEMOS automates as much as possible (scheduling, data collection, report generation), and provides event tracing to help in model validation and debugging. Happily these turn out to be the very areas in which the deepest knowledge of SIMULA itself is required. Along with the simplifications inherent in a prescribed model structure, this means that DEMOS programs can be written in a surprisingly small sub-set of SIMULA. Teaching experience has shown that this can be learnt quickly, and the beginner is very soon able to concentrate his attention squarely on the construction of the model.

The approach to model building that we have used remains viable as the range of problems widens and their degree of difficulty sharpens. Despite its modest design aims, DEMOS has been successfully used to tackle some realistic industrial simulations. Nothing learnt by the beginner need be unlearnt as his experience grows. But DEMOS is not the panacea for all discrete event problems: eventually the user will surely run into a problem which is not capable of being modelled cleanly in complete detail in DEMOS. Then the user can fall back on the host language SIMULA. Because DEMOS programs are SIMULA programs, all the power of SIMULA is directly available behind the building blocks provided by DEMOS. Any feature not provided by DEMOS can be written directly into a DEMOS program as SIMULA code. Again, any user can add or even replace DEMOS features by standard SIMULA mechanisms. Notice that at this stage of his career, the user will have already written several DEMOS (= SIMULA) programs and picking up the required expertise in SIMULA proper is no longer such a problem. Much has been absorbed by osmosis.

DEMOS has taken some time to evolve. Vaucher [19] long ago suggested writing a GPSS-like package in SIMULA and implemented such a package himself. The author did the same and learnt some valuable lessons. In particular, GPSS allows only one transaction type (which closely parallels a process in SIMULA or an entity in DEMOS). For many examples this is sufficient, but the rest have to be bent into this format. It certainly concentrates the mind wonderfully well. Experience with GPSS teaches one how to do a lot within a simple framework - how to separate out and de-emphasize minor components and resist the urge to overmodel. GPSS also teaches the value of resource types, and standard methods of synchronisation, automatic report generation and data collection.

About this time, the author collaborated with Robin Hills. Robin already had a considerable background in both practical simulation work and simulation language design (see Hills [22, 23]). This background in activity based languages proved especially valuable when we sought for ways of tackling models involving complicated decisions - an area in which GPSS is weak. The product of our joint efforts, called SIMON 75 (see Hills and Birtwistle [16]), used WAITUNTIL statements to make the scheduling of events as easy as possible and in a uniform style. Waits until are expensive on machine time, but the package had some merit in that it was easy to learn and resulted in concise yet readable programs.

It came as a pleasant surprise when some 50 or so non-trivial SIMON 75 programs were analysed by the author for their usage of wait until. They proved necessary in only a few cases, and it was at once apparent that a much faster new version could be implemented which would retain the ease of learning and textual clarity of the old. Along with a few other improvements, this was developed into DEMOS.

2 THE SIMULA FOUNDATION

This chapter is a short introduction to the highlights of SIMULA. It is not meant to be exhaustive: it merely aims to give the reader with little or no prior knowledge of SIMULA enough understanding to follow through the later chapters on DEMOS. Full accounts of SIMULA are found in Birtwistle et al. [11] and Rohlfing [12]. The central new ideas in SIMULA are those of the OBJECT and of the CONTEXT. An OBJECT is used in SIMULA to mirror the characteristics and behaviour of a major component in the system under description. For example, a boat in a harbour simulation or a furnace in a steel mill simulation. Objects with similar characteristics and the same behaviour pattern have the same single definition called a CLASS DECLARATION. A CONTEXT is roughly a library of object definitions common to one particular topic, e.g. a HARBOUR context may contain class declarations for boats, cranes, tugs, the tide, etc., and a TRAFFIC context may contain class declarations for cars, trucks, etc. Once defined, a context serves as a library of predefined building blocks. It is available to any number of programs by its very occurrence (almost, see later) as prefix to a program, e.g.

```
TRAFFIC
  BEGIN
    program using cars, trucks, etc.;
  END;
```

The remainder of this chapter is a tutorial on the purpose and usage of objects and contexts.

2.1 OBJECTS

Objects are used in SIMULA programs to mirror major components in the actual system under investigation. Each major component in the actual system is mapped into a corresponding object in the SIMULA program. As an example, consider a harbour simulation involving boats, lorries, etc. Each actual boat will be represented in the SIMULA program by a corresponding boat object. It follows that the boat object has to reflect all those features of the actual boat deemed relevant in the model: not only its physical characteristics such as its tonnage, current load, etc., but also the actions it carries out as it wends its own way through the harbour system.

```
          @                --------------
           @               !    BOAT    !
            @              --------------
            -              ! TONNAGE 15 !
          ! !              ! LOAD     4 !
          ! !              --------------
                         ->! sail in;   !
     ______________        !            !
     !            /        ! unload;    !
     !           /         ! sail out;  !
     ___________/          --------------
```

Figure 2.1 A boat and the corresponding boat object.

Figure 2.1 introduces our standard way of depicting objects - as rectangular boxes divided into three levels. The top level gives the class of the object (here BOAT), the middle level gives the ATTRIBUTES (data characteristics) of the object (here TONNAGE and LOAD shown with current values of 15 and 4 respectively, perhaps in units of 1000 tons), and the bottom level gives the life history of the boat object as a sequence of actions. Here, these are informally shown as

sail in; unload; sail out;

N.B. The middle and bottom layers may be empty, in which case they will be omitted.

Where it sheds light on the situation, the current action of an object will be marked with an arrow, thus '->'. This marker is called its LOCAL SEQUENCE CONTROL (or LSC for short). The boat object in figure 2.1 represents an actual boat sailing in. Figure 2.2 shows how a real world situation involving three boats (one sailing out, one sailing in, and one unloading) and one lorry (loading) would be mapped into a SIMULA program. Notice how the LSCs of the boat objects move on in tandem as the actual boats they represent progress through the harbour system.

```
         @
        @                   @
       @        @           @
                 @          @
        _         @
       ! !                  _
       ! !                 ! !
                  _
                 ! !       ! !
    _______                       _
    \      !     ! !             ! !
                         _______ ___! !__
     \     !             \     / !  ! ! !
               _______
      \_____   !      /   \   /   ______
        .      !     /     \_/    O.  O
        .      ______/      .     .
        .         .         .     . REAL
        .         .         .     .WORLD
===================================
        .         .         .     .MODEL
        .         .         .     .
    -----------  -----------  -----------  .
    !   BOAT  !  !   BOAT  !  !   BOAT  !  .
    -----------  -----------  -----------  .
    !TONNAGE 9!  !TONNAGE 8!  !TONNAGE 5!  .
    !LOAD    0!  !LOAD    4!  !LOAD    2!  .
    -----------  -----------  -----------  .
    !sail in; ! ->!sail in; !  !sail in; !  .
    !unload;  !  !unload;  ! ->!unload;  !  .
  ->!sail out;!  !sail out;!  !sail out;!  .
    -----------  -----------  -----------  .
                                           .
                 -----------               .
                 !  LORRY  !.................
                 -----------
                 !REG  1975!
                 !LOAD    0!
                 -----------
               ->!load;    !
                 !deliver; !
                 -----------
```

Figure 2.2 Objects representing 3 boats and 1 lorry.

Now although their individual data values are different, and they are currently performing different actions, the three boat objects have exactly the same layout of attributes and the same action sequence. The objects are said to be 'of the same class' and are defined by a single CLASS DECLARATION. Here it is in SIMULA (partly informally)

```
CLASS BOAT;
BEGIN INTEGER TONNAGE, LOAD;
  sail in;
  unload;
  sail out;
END***BOAT***;
```

N.B. In this program segment (and in others scattered throughout this book), we use a blending of formal SIMULA and natural English wherever it suits us. Upper case letters and punctuation are formal language elements which are part of SIMULA itself. They have precisely defined meanings and must be used strictly according to the rules of SIMULA. (In the above we have the key words CLASS, BEGIN, INTEGER, and END, and the comma ',' and semicolon ';' as formal elements. The phrase 'END***BOAT***;' is exactly equivalent to 'END;' - we use this form of comment, which is inherited from ALGOL 60, often as it helps delineate the textual end of class and procedure declarations quite clearly.) When it suits us to be informal, we use lower case letters. Above, we have sketched the action sequence of CLASS BOAT informally as its precise formulation in SIMULA is of no immediate relevance. In this way we can postpone detail until it is really necessary.

We need a class declaration for each type of object appearing in a SIMULA program. Each declaration can be thought of as a mould from which objects of the same layout can be created as and when required. Several objects of the same class may be in existence and operating at the same time. To create a boat object in a SIMULA program, we execute the command NEW BOAT. A fresh boat object is created each time this command is executed. If we have one or several boat objects in a SIMULA program, we may wish to name them individually. To create and name two boat objects QE2 and MARIECELESTE respectively we would write

```
QE2          :- NEW BOAT;
MARIECELESTE :- NEW BOAT;
```

(:- is read 'denotes'). QE2 and MARIECELESTE are SIMULA variables of a type not found in ALGOL 60. They are REFERENCE VARIABLES of type REF(BOAT) (which is read as 'ref to BOAT') and are declared so

```
REF(BOAT)QE2, MARIECELESTE;
```

QE2 and MARIECELESTE are variables capable of referencing boat