Demos—a package for Discrete Event Modelling on Simula

by

Graham Birtwistle,
School of Computer Science,
University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield, England S1 4DP
Tel: (+44) 114 222 1842
Fax: (+44) 114 222 1810
Net address: graham@dcs.shef.ac.uk

Revised July 2005

Contents

List of Figures

List of Tables

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, itself an extension to ALGOL 60, and so Demos programs may be run on any computer that supports Simula (see references [26, 70, 27, 28, 69, 68, 78, 87, 54]). Simula (Dahl et al. [25, 24]) contains simulation primitives sufficient to build any simulation model, but leaves it to the user to flesh out the primitives in his or her own style. 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 one has to acquire considerable expertise in Simula before one can start building out these primitives and actually get down to describing the simulation model itself.

But this is not the case. The situation was foreseen by the designers of Simula and they provided a way round the problem, namely the *context*, or *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 or her 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.

Structure of the book

This book is based on material developed for undergraduate and postgraduate courses in Discrete Systems given in the late 1970's at Bradford University, England,

and for other courses given to industry. It has been written in tutorial style with each new feature being first motivated and then used in an illustrative example. 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.

Inevitably with a book written in this style, one or two Demos facilities could not be fitted in. A list of these is can be found in section 8.1. The Demos Reference Manual (Birtwistle [5]) gives a full explanation of their intent, code structure and source listing. Both the Simula source code for Demos and the implementation guide are available from the author (see section 8.2 for details).

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, the *entities*; a full action history describing its behaviour pattern is given for each entity. The separate entity descriptions are then pieced together to describe the behaviour of the system as a whole. This approach is very natural and enables the system modeller to focus his or her attention on the description of one entity at a time.

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 and more recent Demos-based work.

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 decades Ole-Johan Dahl, Björn Myhrhaug and Kristen Nygaard (for a lifetime's guidance and imparting the Simula ethos); Robin Hills (the facts of life and insights into discrete event simulation); and alphabetically last, but certainly not least, Jean Vaucher whose exemplary papers on simulation data structures, event list handling and waits until both started me off a guided me on the way. Alan Benson, Roy Francis, Ralph Huntsinger, Lars Enderin, Paul Luker, Mats Ohlin, Brian Unger, 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 originally developed on the Leeds University DEC System 10 computer using the excellent Simula compiler written by the Swedish Defence Research Establishment, Stockholm. Sincere thanks are due to Jim Cunningham, Henry Islo, Henk Sol, and Jean Vaucher for spotting errors in the original release and sending fixes.

For the second edition of the book, the programs were all re-run on Sun workstations at the University of Calgary under the equally excellent Lund Simula system.

Typesetting

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 lower case (e.g. `begin`, `procedure`, `if`). Other changes are: 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 (`&`).

Latexed version, 2005

The Macmillan text is now out of print. As Demos is still around and in use, I am putting this version on the web. I have reformatted the original text using LaTeX. I have reworked all the diagrams and put the full trace of chapter 3 in pictures rather than lines of text (something I always fancied doing, but the original editor only supported ascii pictures). I have added a fresh example as example 7 in chapter 5, page 103..111. Otherwise I have tried to resist the urge to tinker with the text—but have added some extra references and several extra appendices, E..I.

Over the last few years I had the pleasure of working with Chris Tofts, now at HP Labs in Bristol. We applied the techniques of process algebra to Demos models and showed how to test models for deadlock, livelock, safety and liveness properties [8, 9, 10, 12, 13, 101, 102, 11, 103]. Appendices E..I are included to show the style and spirit of our endeavours. Chris has produced a new system extending this work. Full details of *demos2k* can be found at `http://www.demos2k.org` together with documentation and references as the work evolved. It seems the best way to fly now.

Chapter 1

Introduction

All around us in everyday life are complex systems of workers 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 an amulance breaks down), 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 satellite 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: its quicker, cheaper and spares us from real 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 or her 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 their feedback is essential. Therefore 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 1, 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 the 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 Bratley, Fox and Shrage [14], Cave Brown [15], Evans [29], Fishman [30, 31], Franta [32], Law and Kelton [50], Shannon [90], and Tocher [99].

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 Birtwistle et al [7] or Hills [39] for non-trivial models coded in activity, event, and process modes.) The designers of Simula included a standard context called `SIMULATION` which contains an 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 one has to write one's 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 `res`ource for representing minor components whose detailed life cycles can be abstracted away. In addition, Demos automates as much as possible (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 or her 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. Importantly, nothing learned by the beginner need be unlearned as one's 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 or her 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 [104] long ago suggested writing a GPSS-like package in Simula and implemented a prototype package himself. The author did the same and learnt some valuable lessons. In particular, GPSS then allowed 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 for a while with Robin Hills. Robin already had a considerable background in both practical simulation work and simulation language design. This background in activity based languages proved especially valuable when we sought 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 [41]), 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 100 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`.

Despite its modest design aims, `Demos` has been successfully used to tackle some realistic industrial simulations. The author has applied `Demos` to problems in the steel industry, for work on operating systems (segmentation and paging algorithms), and designing real time processes (long haul and local area networks, multiple cpu configurations). `Demos` has been/is used at a number of research institutes and universities as well as in the aerospace, automobile, oil, gas, steel, and telecommunications industries by Asea, British Telecom, Burroughs, GE (UK), Intel, US Naval Research Labs, Norwegian Computing Center, Norwegian Defence Research Establishment, Philips, Plessey, Swedish Defence Research Establishment, and Volvo.

Chapter 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. [6], Poley [80], and Rohlfing [35]. 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. A context serves as a library of predefined building blocks for a particular area. A context will normally be externally pre-compiled and then be available to any number of programs by an *external declaration* and its occurrence as prefix to a program block, e.g.

```
external class Traffic;

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 model 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, trucks, etc. Each actual boat will be represented in the Simula program by a corresponding boat object. It follows that each boat object has to reflect all those features of an 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.

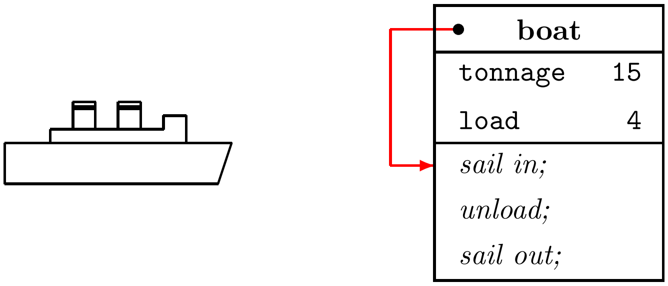


Figure 2.1: A boat and the corresponding boat object.

Figure 2.1 introduces our standard way of depicting objects—as rectangular boxes divided into three levels:

1. the top level gives the class of the object (here `boat`)

2. the middle level gives the *attributes* of the object (here `tonnage` and `load` shown with current values of 15 and 4 respectively, perhaps in units of 1000 tons), and

3. 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 of objects may be empty (no local data or no actions), in which cases 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 whose current action is *sail in;*.

Figure 2.2 shows how a real world situation involving two boats (one sailing out and one unloading) and one truck (loading) would be mapped into a Simula program. Notice how the LSCs of objects move on as they progress through the harbour model and that the LSC's of the unloading boat and the truck which is being loaded are *synchronised*.

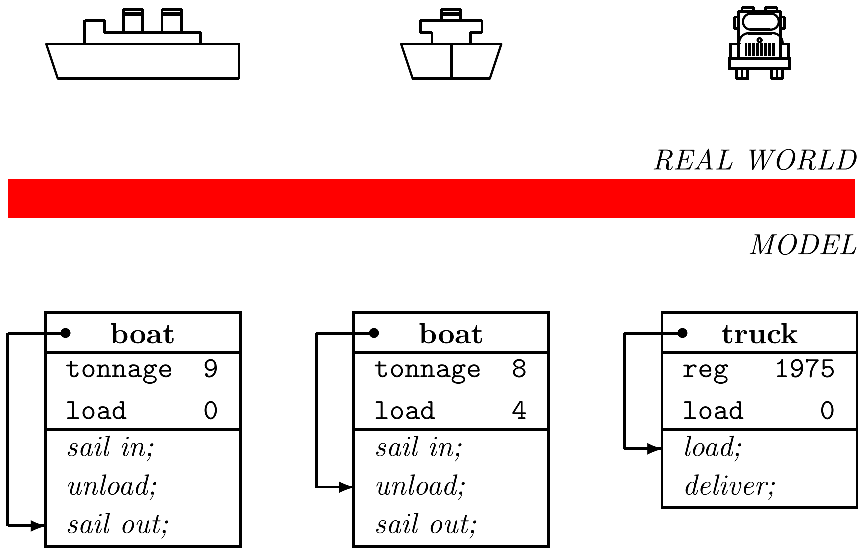


Figure 2.2: Objects representing 2 boats and 1 truck

Now although their individual data values are different, and they are currently performing different actions, the 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***;
```

In this program segment, and in others scattered throughout this book, we use a blending of formal Simula and natural English. `Teletype letters`, both upper and lower case, 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.) For contrast, we use *italics* when it suits us to be informal. Above, we have sketched the action sequence of `class boat`

sail in; unload; sail out;

informally as its precise formulation in Simula is not the point at issue. 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 template from which objects of the same form 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 `MARIE_CELESTE` and `QE2` respectively we would write

```
MARIE_CELESTE :- new boat;
QE2           :- new boat;
```

(The reference assignment operator `:-` is read *denotes*). `MARIE_CELESTE` and `QE2` 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) MARIE_CELESTE, QE2;
```

References variables are defined with a *qualification* which restricts the range of objects they may access. The qualification of `MARIE_CELESTE` and `QE2` is `boat`. Thus `MARIE_CELESTE` and `QE2` are reference variables capable of referencing boat objects[1].

In the same way, should we wish to create and name a truck `T`, we would declare `ref(truck) T` and execute the reference assignment

```
T :- new truck;
```

EXERCISES 2

Exercise 2.1 Give an informal declaration of `class truck` based on the truck object depicted in figure 2.2.

[1] This will be made more flexible in section 2.3.

2.2 Class declarations

We now start to put things on a more formal footing. Consider the class of road vehicles. Although vehicles are of many shapes and sizes and are built for different purposes, they do have certain characteristics in common. We let them be typified (fairly arbitrarily) by their "year of registration", their "unladen weight", and by whether or not they are currently "broken down".

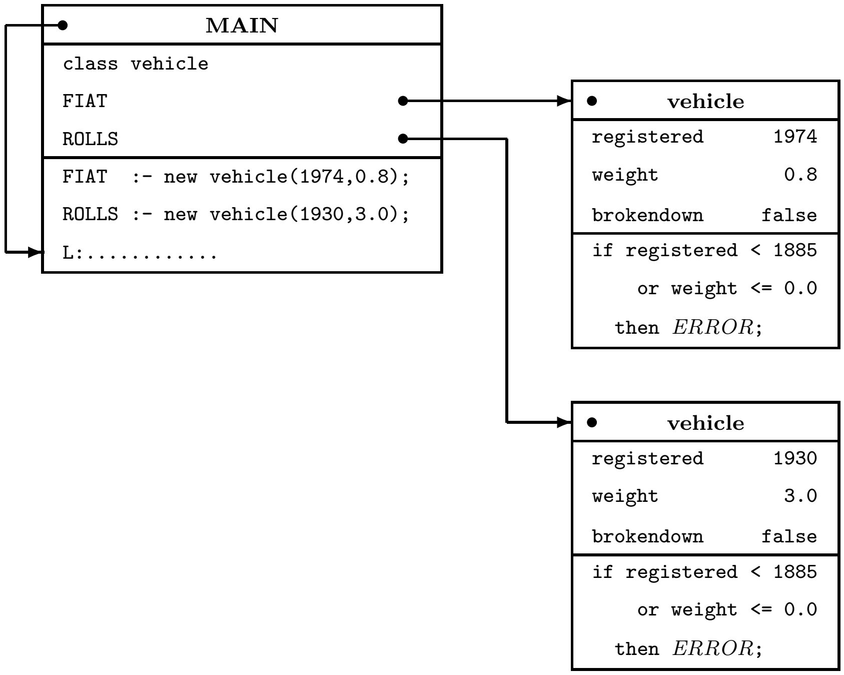


Figure 2.3: Two terminated vehicle objects

The following program declares the prototype for such vehicles (lines 2–7), declares a pair of reference variables (`FIAT` and `ROLLS`) capable of referencing vehicle objects (line 9) and then creates a 1974 vehicle of weight 0.8 tons (`FIAT`) and a 1930 vehicle of weight 3.0 tons (`ROLLS`). Neither vehicle is broken down. The informal call on the undefined *procedure ERROR* indicates that corrective action is to be taken should the actual parameter values to an object under creation prove to be invalid.

```
begin
  class vehicle(registered, weight);
    integer registered; real weight;
  begin
    boolean brokendown;
    if registered < 1885 or weight <= 0.0 then ERROR;
  end***vehicle***;

  ref(vehicle) FIAT, ROLLS;

  FIAT :- new vehicle(1974, 0.8);
  ROLLS :- new vehicle(1930, 3.0);
L:................;
end;
```

Figure 2.3 is a representation of the structures created at the label `L`. Initially there are no objects in the system and the reference variables `FIAT` and `ROLLS` each take the standard value of `none`, which represents no object[2].

Two vehicle objects are created by our program (lines 11 and 12). We detail the creation of the first object by following through the reference assignment on line 11, namely `FIAT :- new vehicle(1974, 0.8);`. First an object is created with layout as defined by `class vehicle`. The parameter values 1974 and 0.8 are transmitted (by `value` in the ALGOL 60 sense), and its local variable `brokendown` is set to the standard initial value `false`. Then the actions of the object are entered. These perform a rough check on the validity of the parameters. Once the actions of the object have been exhausted, their LSCs are no longer required, and the objects are said to be *terminated*. Program control returns to the generator `new` and the whereabouts of the object is assigned to the reference variable `FIAT`.

The second object is created in the same way, and so at the label `L`, `FIAT` and `ROLLS` will reference two distinct vehicle objects. We can now access the current data values of their attributes (parameters and local data values) from the main program by the *dot notation*. Below we tabulate all the possible main program accesses, their current values and their types.

Possible statements in a main program using these accesses are

```
[ integer vintage; real tonnage; ]

if ROLLS.registered < 1930 then vintage := vintage+1;
if FIAT.brokendown then take FIAT off the road;
tonnage := FIAT.weight + ROLLS.weight;
```

The dot notation can be used to re-assign attribute values as well as read them. Should the unthinkable happen and `ROLLS` later break down, we could write `ROLLS.brokendown := true` in our main program.

Notice that when the program action is inside a particular object — such as when carrying out the parameter checking on object creation—the object refers to

[2] In Simula all declared variables have standard initial values according to their type. In particular, booleans are initialised to `false`, arithmetics to zero, and reference variables to `none`.

Access	Value	Type
`FIAT.registered`	1974	integer
`FIAT.weight`	0.8	real
`FIAT.brokendown`	false	boolean
`ROLLS.registered`	1930	integer
`ROLLS.weight`	3.0	real
`ROLLS.brokendown`	false	boolean

its own attributes directly, e.g. `weight` and `registered`. When the action is in the main program, we have to specify the particular object we require as well as the name of the attribute. These remote accesses have the format

`<object_reference>.<attribute_name>`, e.g. `ROLLS.weight`

A run time (= execution time) error results if the value of the object reference in a remote access is `none`. The null object certainly has no attributes. Accordingly, the program will stop executing and print out a suitable error message.

The remote access problem has its analogues in everyday life. For example, when requiring the telephone number Edinburgh 123 4567, we dial 123 4567 from inside Edinburgh itself, but 0131 123 4567 from the rest of Britain, and 0044 131 123 4567 from outside Britain. 0131 is the dialling code for Edinburgh. When outside Edinburgh, omitting the 0131 prefix gives us quite a different telephone number. Indeed 123 4567 may not even be valid (compare asking for `brokendown` within the main block).

EXERCISES 2 (continued)

Exercise 2.2 Give an informal declaration of `class customer` describing the actions of customers who enter a barber's shop for a haircut. Draw a customer object which represents an actual customer whose hair is currently being cut.

Exercise 2.3 Write the declaration of `class car`. Each car object is registered, has a weight, and may or may not be broken down. In addition to these `vehicle` attributes, it has a maximum speed and some seats. Remember to check as many parameter values as you can.

Write Simula code to create a 1970, 1.2 ton car with a maximum speed of 240 kph and 2 seats referenced by `ref(car) JAG`. Cause `JAG` to breakdown. Draw the object referenced by `JAG`, filling in its attribute values to represent its status after the breakdown.

Exercise 2.4 Write a declaration for `class boat`. Each boat object has a tonnage, a current load (not a parameter), and a crew. The owners man the boats according to the formula: 5 permanent officers (including the captain), plus one seaman for every 200 tons of tonnage (calculated by rounding up).

Write Simula code to create a boat object (referenced by) `B` of tonnage 2600 tons and then load it with 1600 tons. Use the actions of the class body to compute the size of the crew. Draw the object.

Exercise 2.5 Give another everyday analogue of the remote access problem in addition to the one given in the text (telephoning a number in Edinburgh).

2.2.1 Sub-classes (of vehicle)

We now turn our attention to defining more specific kinds of road vehicles, for example, cars, trucks and pick-ups. All are vehicles and will be registered, have an unladen weight and (hopefully) will be roadworthy. But they have distinguishing qualities too—trucks carry loads, cars carry passengers and pick-ups have cranes for towing. We could start from scratch and define

```
class truck(registered, weight, maxload);
  integer registered; real weight, maxload;
begin
  boolean brokendown, juggernaut;
  real load;

  if registered < 1885 or weight <= 0.0 or maxload <= 0.0
    then ERROR;
  juggernaut := maxload >= 25.0;
end***truck***;
```

and in the same way

```
class car(registered,weight,maxspeed,seats);...;

class pickup(registered,weight);..............;
```

But we have done so much of this work before. If trucks, cars, and pick-ups really are "vehicles plus" we should be able to build on the definition of vehicle that we have already worked out. This we can do in Simula by employing the *prefix notation.* We simply declare

```
vehicle class truck(maxload); real maxload;
begin
  boolean juggernaut;
  real load;

  if maxload <= 0.0 then ERROR;
  juggernaut := maxload >= 25.0;
end***truck***;

vehicle class car(maxspeed, seats); integer maxspeed, seats;
begin
  if maxspeed < 50 or seats < 1 then ERROR;
end***car***;

vehicle class pickup;
begin
  ref(vehicle) VICTIM;

  procedure towin(V); ref(vehicle) V;
  begin
  if V.brokendown
    then VICTIM :- V
    else FALSE ALARM;
  end***tow in***;

end***pickup***;
```

`truck`, `car`, and `pickup` are said to be *sub-classes* of `vehicle`. Figure 2.4 overleaf shows the attribute structures of the four types of object we have defined so far and typical accesses.

Objects of the three sub-classes `truck`, `car`, and `pickup` are compound objects which inherit all the attributes and all the actions of their `vehicle` prefix. When such objects are created, the actions of the prefix level(s) are executed first, and then the actions at the new level.

Prefixing can be carried out to any depth, and so we may now use `truck`, `car`, or `pickup` as prefix if we wish.

Trucks initially carry no payload. They are deemed juggernauts if their maximum payload is 25 tons or over. To create a 1970, 5 ton truck with a maximum payload of 37.5 tons we may write

```
T :- new truck(1970, 5.0, 37.5);              [ ref(truck) T;]
```

Note that each new truck object requires 3 parameters — two are inherited from the `vehicle` prefix. Later, to give `T` a load of 16 tons, we write

```
T.load := 16.0;
```

Cars have maximum speeds and seats as attributes in addition to `vehicle` attributes. We create a new 1969, 1.5 ton car with a top speed of 145 kph and with 5 seats by

```
 C :- new car(1969, 1.5, 145, 5);             [ ref(car) C; ]
```

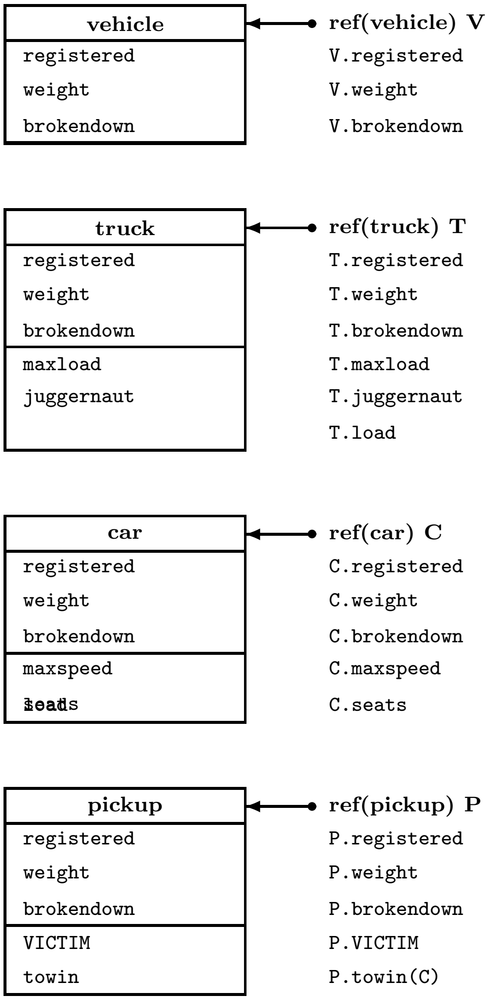


Figure 2.4: Attribute structures and accesses

Finally, pick-up objects are furnished with a reference `VICTIM` to the current vehicle (if any) that they are towing. `VICTIM` is initially `none`. The `procedure towin` is the means of supplying a suitable value to `VICTIM`. It may be any `vehicle`, `truck`, `car` or `pickup` object. Local procedures, such as `towin`, are treated as attributes in just the same way as are parameters and local data values. They too are accessible via the dot notation.

We create a 1976, 4 ton pick-up by

```
P :- new pickup(1976, 4.0);                    [ ref(pickup) P; ]
```

If `ref(car)C` breaks down, `P` can be deputed to tow in `C` by such a coding sequence as

```
    C.brokendown := true;
    P.towin(C);
L2: ...................
```

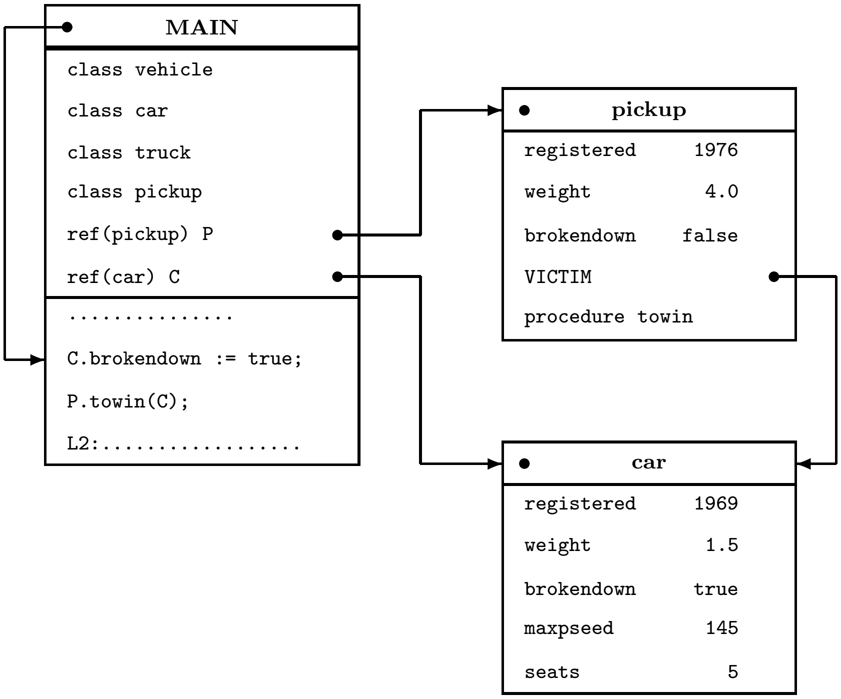


Figure 2.5: Result of the call P.towin(C)

Figure 2.5 pictures the situation at the label `L`. When the `procedure towin` is called, the actual parameter value `C` is passed by *reference*—a mechanism not found in ALGOL 60. Informally this method of passing reference parameters is equivalent to the reference assignment

```
<formal parameter> :- <actual parameter>;   (here V :- C;)
```

and is subject to the compatibility checks outlined in the next sub-section.

Rather interestingly, if `P` now breaks down, another pick-up (`Q` say) can be instructed to tow in `P` (and `C` with it!) by

```
        P.brokendown := true;
        Q.towin(P);
```

In our examples, the actions belonging to an object have been concerned with checking parameter values on object creation and initialising local quantities. They are executed at once each time a fresh object of that class is created.

But their scope is much broader than this. As indicated earlier, class body actions may be used to describe complete life histories. While such action sequences may be executed all at once, they may also be executed as sequences of separate sub-tasks or active phases. That is, between two active phases of a given object, any number of active phases of other objects may occur. This is the very basis of the process approach and is fully developed in the next chapter.

2.2.2 Security of data access

For the examples of this section we assume the class declarations of the previous section and the existence of the reference variables

```
                    ref(vehicle) V;

         ref(truck) T;    ref(car) C;    ref(pickup) P;
```

So far we have concentrated on the objects themselves. We now turn our attention to reference variables and reference assignments. Reference variables are given a *qualification* on declaration: the qualification of `ref(vehicle) V` is `vehicle` and of `ref(truck) T` is `truck`. This qualification restricts the kind of object to which a reference variable is allowed to refer. The typical reference assignment is

```
          <reference variable> :- <object expression>;
```

Reference variables may be assigned to reference

1. objects of their qualifying class, e.g.
   `V :- new vehicle(1976, 1.4)`
   `T :- new truck(1974, 5.0, 37.2)`

2. objects of classes prefixed by their qualifying class, e.g.
   `V :- new pickup(1970, 10.0)`
   `V :- T`   (an existing object or `none`)

3. no object at all, e.g.
   `V :- none`

By means of qualification, a Simula compiler can check the compatibility of the left and right hand sides `none` can be thought of as having a universal qualification here. Such attempted assignments as

```
T :- new car(1939, 1.4, 100, 4);
C :- new vehicle(1976, 1.9);
```

must always fail this compatibility check. This is sensible because later, apparently good attempts to access (such as `T.load` or `C.seats` above) would be in error. There is one genuine case of doubt exemplified by the reference assignment

```
T :- V;
```

which may be valid. It is valid if `V` currently references a truck object or `none`: but is illegal if `V` currently references a vehicle, a car, or a pick-up object. This check can only be made during program execution. On meeting such assignments in the source program, the Simula compiler prints a warning message and plants a run time check which is carried out when the assignment is actually attempted. If this compatibility check fails, execution of the program stops at once. The compiler can thus guarantee that, at run time, a reference variable refers either to `none`, or to an object of its qualifying class, or to an object prefixed by its qualifying class. That is, if a `ref(A)` variable does reference an object at run time, then that object is at least *A sized*.

The qualification of a reference variable acts as a key opening up the inside of the object it currently references. In a remote access such as `C.seats`, we are guaranteed that `C` references `none` (causing a run time error) or that the object referenced by `C`, being at least `car` sized, does indeed possess an `integer` attribute `seats`. The qualification `car` of `C` gives both the offset of `seats` within the object and its type (`integer`). The compiler can also use qualification to reject such attempted accesses as `C.driver` or `T.seats` as undefined, and trap such attempted illegal uses as

```
C.brokendown := C.brokendown + 1;
```

The prefix notation permits a useful flexibility in object referencing illustrated by the `procedure towin` local to `class pickup`. Any vehicle object, or object of a class prefixed by `vehicle`, is a suitable potential victim. This is very desirable. A weak qualification permits a wide range of objects to be referenced at the cost of run time checks on the validity of all remote accesses to attributes declared at stronger levels. We do not go into this aspect of Simula here as we never need to use it in the sequel. Indeed it is not frequently required in Demos programs. The reader is referred to Birtwistle et al. [11, chapter 4 on `qua`, `inspect`, and `virtual`].

2.2.3 ==, =/=, is, in

To complete this section on security of access, we introduce four more operators which enable us to distinguish at run time between references to objects (`==`, `=/=`) and ascertain the class of an object (`is`, `in`).

==, =/=: Because such a remote access as

```
V.brokendown                    [ ref(vehicle) V; ]
```

causes a run time error if the value of `V` is `none`, it is important to have a means of checking against that eventuality. To this end, Simula includes the *reference comparators* `==` and `=/=`. Let `V` and `W` be references to objects or `none`. Then

- `V == W` is `true` only if `V` and `W` reference the same object, or both are `none`
- `V =/= W` is equivalent to `not(V == W)`

Typical uses are furnished by:

1. `ROLLS` gets special treatment

```
if V == ROLLS
  then special treatment
  else send V to end of queue ;
```

2. we expressed the `procedure towin` local to `class pickup` as

```
procedure towin(V); ref(VEHICLE) V;
begin
   if V.brokendown then VICTIM :- V else
     FALSE ALARM;
end***towin***;
```

A better formulation might be to use `if V == none` to guard against the actual parameter value being `none` and use `if VICTIM =/= none` to check that this pick-up is not already busily towing in a victim. Only if these two checks are passed do we enter the third `if` condition.

```
procedure towin(V); ref(VEHICLE) V;
begin
   if V == none         then   ERROR: argument none    else
   if VICTIM =/= none   then   pick-up already in use  else
   if V.brokendown      then   VICTIM :- V             else
     FALSE ALARM;
end***towin***;
```

is, in: is and in may be used to ascertain at run time the type of object a variable is currently referencing. We use them in conditions of the formats (a little restricted)

```
<reference variable> is <qualification>, e.g. V is CAR

<reference variable> in <qualification>, e.g. V in VEHICLE
```

Given class A and ref(A) X. Then at run time the value of X can only be a) none, b) an A object, or c) an object of a class prefixed by A.

- X is A is true only if X references an A object; it is false if X has the value none or X references an object of a class prefixed by A
- X in A is true if X references an A object or an object of a class prefixed by A; it is false if X == none

We could thus extend the procedure towin to

```
procedure towin(V); ref(VEHICLE) V;
begin
   if V == none         then   ERROR: argument none    else
   if VICTIM =/= none   then   pick-up already in use  else
   if V.brokendown      then   VICTIM :- V             else
     FALSE ALARM;
   if V is car          then   tow to nearest garage   else
   if V is truck        then   tow to its owner        else
     tow to my garage;
end***towin***;
```

where a call on this new towin not only locates the victim as before, but takes it to the nearest garage if it is a car, to the victim's owning garage if it is a truck, and to the rescuer's garage if the victim is either another pick-up or another vehicle object.

EXERCISES 2 (continued)

Exercise 2.6 Define a class order which describes a class of objects each of which possesses a serial number, an arrival number, a set-up time, and a processing time.

Define a class batch which describes a class of objects which have all the attributes of order objects plus a batch size.

Define a class single which describes a class of objects which have all the attributes of order objects plus a finishing time and a weight.

Define a class plate which describes a class of objects which have all the attributes of order objects plus weight, length, width, and finishing time.

NB: In our suggested solution, all attributes are given as local variables and none as parameters.

2.3 Contexts

A Simula program must contain the class declarations which give the patterns of the objects it uses. For example, a traffic simulation involving the classes we developed in the last section, would have the format

```
begin
   class vehicle.............;
   vehicle class car..........;
   vehicle class truck........;
   vehicle class pickup.......;

   other relevant declarations;

   actions involving objects of
   the above classes;
end;
```

N.B. In a proper Simula program, the declarations must, of course, appear in full. We have used the dots for brevity's sake, a device we will often resort to in the sequel.

When working in a particular problem area, e.g. traffic simulation, it is clear that the same basic declarations may be useful over a whole range of programs. It is tiresome and error prone to prepare much the same program several times. Instead, the inter-related definitions of vehicle types can be collected together in Simula to define a *context*. In this case, we choose to call it *Traffic* and define it by

```
class Traffic;
begin
   class vehicle.............;
   vehicle class car..........;
   vehicle class truck........;
   vehicle class pickup.......;
end;
```

Normally, `Traffic` would now be separately compiled and the object code be retained in a library. Users with an interest in this particular field and *local* access to the library can pick up the compiled context code by an `external declaration` as below

```
external class Traffic;

Traffic
begin
   other relevant declarations;

   actions involving objects of
   classes defined within Traffic
   and this prefixed block;
end;
```

As a particular example, the user wishing to write a program involving 2 cars and 1 pick-up merely codes

```
external class Traffic;

Traffic
begin
   ref(car) C1, C2;
   ref(pickup) AUTO_REX;

   C1       :- new car......;
   C2       :- new car......;
   AUTO_REX :- new pickup...;
   .........................
end;
```

All the concepts defined inside `Traffic` (namely `vehicle`, `car`, `truck`, `pickup`) are directly available within the user-defined block.

In general, contexts make available to the programmer a set of problem-oriented and familiar concepts for use as building blocks in programs. Given the right abstractions, this may be enough, and the ordinary user does not have to know the full Simula language. But the experienced programmer has the general language available and may extend the application language by new concepts that be desirable. For example,

```
external class Traffic;

Traffic class Police_Surveillance;
begin
   class traffic_lights...........;
   vehicle class black_maria......;
   car class police_car...........;
   ...............................
end***Police_Surveillance***;
```

Any block prefixed by this new context, e.g.

```
external class Police_Surveillance;

Police_Surveillance
begin
   ...............................
end;
```

has available all the definitions at the `Traffic` level *plus* the new ideas of traffic lights and police vehicles. Notice that black marias and police cars may be towed in by pick-ups when broken down since their respective class declarations both contain the prefix `vehicle` (directly in the case of `class black_maria`, and implicitly in the case of `class police_car` since its explicit prefix `car` is itself prefixed by `vehicle`). Trafficants unconcerned by police activities will continue to use `Traffic` as prefix to their programs.

N.B. Typically, when a context has been developed and is stable, it will be placed in a group or system library and be made generally available. In such cases, `external` declarations require additional search path information. On my system at Calgary, I located the compiled version of Demos by:

```
        external class Demos = "/usr/local/simulabin/demos.atr"
```

The details will differ on your system—look it up in the appropriate Simula implementation manual.

EXERCISES 2 (continued)

Exercise 2.7 Which items would you like already defined in a context for a harbour simulation? Write down the skeleton of its definition in Simula and how you would use it.

Chapter 3

Modelling with entities

We begin with an analogy. Consider splitting the text of a play (∼ a system) into separate scripts for each role (∼ each entity). For example, Act 1, Scene 1 of Shakespeare's Macbeth starts

1 Witch. When shall we three meet again.
In thunder, lightning, or in rain?
2 Witch. When the hurlyburly's done,
When the battle's lost and won.
3 Witch. That will be ere the set of sun.
1 Witch. Where the place?
2 Witch. Upon the heath.
3 Witch. There to meet with Macbeth.
....

This can be split into three separately described roles, as below. (Asterisks represent pauses in between speeches.)

1 Witch. When shall we three meet again.
In thunder, lightning or in rain?
*
Where the place?
*
....

2 Witch. When the hurlyburly's done,
When the battle's lost and won.
*
Upon the heath.
*
....
3 Witch. That will be ere the set of sun.
*
There to meet with Macbeth.
*
....

The separate roles are sequences of active speaking phases and passive waits until it is one's turn again.

In much the same way, when we come to describe a simulation model, we first split the totality into suitable components (cf. 1 Witch, 2 Witch, 3 Witch, ... in Macbeth), and provide an object with a full life history to act out the role of each. Once we have sorted out how each object synchronises its actions with other objects (a major modelling problem), the description of the rest of an object's life history can be completed separately. We simply psyche ourselves into each role in turn and then write out its actions from its own viewpoint.

Example 1: Port system

A port has 2 jetties each of which can be used for unloading by one boat at a time. Boats arrive at the port periodically and must wait if no jetty is currently free. When a jetty is available, a boat may dock and start to unload. When this activity has been completed, the boat leaves the jetty and sails away. The port authority has a pool of 3 tugs. Two are required for docking: only one when a boat leaves its jetty.

It is instructive to hand simulate the system with some "easy" numbers. Assume that

- boats arrive at times 0.0, 1.0, 15.0
- tug manoeuvres take 2.0 time units
- unloading takes 14.0 time units.

Table 3.1 gives a full *trace* of the port system using this data. We have named the boats `B1`, `B2`, and `B3`. As the state of the system changes only at certain critical times, only these times have been recorded (hence *discrete event simulation*).

The trace records the essential behaviour of the system as a time ordered sequence of events. The first three columns of table 3.1 give the 'when', the 'who' and the 'what' of each event. The who of each event is *always* a boat, and so the behaviour of the complete system can be rephrased in terms of the actions and interactions of `B1`, `B2`, and `B3`.

time	boat	current action	next event
0.0	B1	arrive	
	B1	request 1 jetty	
	B1	seize 1 jetty	
	B1	request 2 tugs	
	B1	seize 2 tugs	
	B1	start docking	2.0
1.0	B2	arrive	
	B2	request 1 jetty	
	B2	seize 1 jetty	
	B2	request 2 tugs	
2.0	B1	release 2 tugs	
	B1	start unloading	16.0
	B2	seize 2 tugs	
	B2	start docking	4.0
4.0	B2	release 2 tugs	
	B2	start unloading	18.0
15.0	B3	arrive	
	B3	request 1 jetty	
16.0	B1	request 1 tug	
	B1	seize 1 tug	
	B1	start leaving	18.0
18.0	B2	request 1 tug	
	B2	seize 1 tug	
	B2	start leaving	20.0
	B1	release 1 tug	
	B1	release 1 jetty	
	B1	quit	****
	B3	seize 1 jetty	
	B3	request 2 tugs	
	B3	seize 2 tugs	
	B3	start docking	20.0
20.0	B2	release 1 tug	
	B2	release 1 jetty	
	B2	quit	****
	B3	release 2 tugs	
	B3	start unloading	34.0
34.0	B3	request 1 tug	
	B3	seize 1 tug	
	B3	start leaving	36.0
36.0	B3	release 1 tug	
	B3	release 1 jetty	
	B3	quit	****

Table 3.1: Trace of the Port System

We follow this lead in table 3.2 and split the trace narrative into three separate columns, one for each boat. Notice that each and every event appears once under the appropriate boat name, and that no events have been omitted. The whole is precisely the sum of its parts.

	time		
event sequence	B1	B2	B3
arrive	0.0	1.0	15.0
request 1 jetty	0.0	1.0	15.0
seize 1 jetty	0.0	1.0	18.0
request 2 tugs	0.0	1.0	18.0
seize 2 tugs	0.0	2.0	18.0
start docking	0.0	2.0	18.0
release 2 tugs	2.0	4.0	20.0
start unloading	2.0	4.0	20.0
request 1 tug	16.0	18.0	34.0
seize 1 tug	16.0	18.0	34.0
start leaving	16.0	18.0	34.0
release 1 tug	18.0	20.0	36.0
release 1 jetty	18.0	20.0	36.0
quit	18.0	20.0	36.0

Table 3.2: Table of event times for each boat

Table 3.2 is just a rehash of the trace in which we have followed through the actions of each boat as an individual. Importantly, the action sequence of each boat may be framed in exactly the same way—as the sequence of activities

dock; unload; leave;

and invites the declaration of a single `class boat`: informally

```
class boat;
begin
   request 1 jetty; seize 1 jetty;
      request 2 tugs; seize 2 tugs;
         dock;
      release 2 tugs;

         unload;

      request 1 tug; seize 1 tug;
         leave;
      release 1 tug;
   release 1 jetty;
end***boat***;
```

a template from which three boat objects will be created. Notice that a fresh arrival will be taken care of by executing `new boat` at an appropriate time in the main program. Quitting is taken care of automatically by the Simula garbage collector when a boat object exhausts its action sequence.

EXERCISES 3

Exercise 3.1 Give another class declaration-role analogue besides the one given in the text (Macbeth).

3.1 Activities

Before developing Demos code for `class boat`, we take a closer look at the notion of a *task* or *activity*. The life history of each boat is a sequence of three activities and their most general pattern would seem to be:

1. `acquire` (= *request* and *seize* when available) the extra resources needed for the forthcoming task. Extra resources are usually requested sequentially. In general, each request is followed by a wait until sufficient of the resource is available and can be seized. This being understood, from now on we will usually use `acquire` for this joint phase instead of separating it into a `request` followed by a `seize`.

2. `hold` all resources (both those just seized and those previously acquired) while carrying out the task.

3. `release` those resources no longer required and continue with the next activity, if any.

activity	**extra resources required**	**duration**	**resources to release**
dock	1 jetty + 2 tugs	2.0	2 tugs
unload	none	14.0	none
leave	1 tug	2.0	1 tug + 1 jetty

Table 3.3: The three activity patterns in detail

Table 3.3 tabulates these activity stages in the context of our port system. If all the extra resources required for the start of an activity are free, they can be seized in turn immediately and the activity starts at once (e.g. `B1` when docking). When a resource is not available, then a request is followed by a wait, and the boat object issuing the request has to wait until any current user(s) have freed sufficient for it to proceed (e.g. at time 1.0, `B2` can seize a jetty, but must wait for 2 tugs as only 1 is then free). Again, at time 15.0, `B3` must wait for a jetty to be released. When this occurs at time 18.0 (released by `B1`), `B3` does not have to wait further to seize 2 tugs as sufficient are then free.

Notice that each boat object retains the same jetty throughout its lifespan, but releases the tugs after docking and before unloading. In general, resources may be retained through an arbitrary number of activities. Also note that if no extra resources are required, as for 'unload' (because the jetty has already been seized), the pattern simplifies as the acquire phase drops out and there can be no waiting. Such activities are sometimes called *bound activities* as they are bound to follow straight on from the previous activity.

This activity pattern of `acquire` (with the implicit wait until resources are available), `hold`, and then `release` turns out to be very common in discrete event model building—indeed, CSL (Buxton and Laski [17], ECSL (Clementson [23]), SIMON and HOCUS (Hills [38, 41]) were founded upon it. Activity patterns can be presented neatly by *activity diagrams* introduced by Tocher [99] but we can simplify them since Simula supports objects (see significant further work extending our presentation by Pooley [81, 82, 83]).

Activity diagrams are essentially high level flow charts which capture the structure of a model at a very convenient level of abstraction. They are important because they not only provide a good basis for model presentation and discussion (especially when some participants are unfamiliar with programming languages): but also because they guide the eventual coding of the model.

In activity diagrams, we depict model resources by circles labelled with the initial value of the item they represent, e.g.

jetties
2
3
tugs

Figure 3.1: Depiction of system resources

and an activity duration by a rectangular box labelled with the appropriate task, e.g.

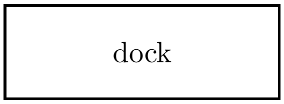


Figure 3.2: Depiction of an activity duration

To construct a diagram of an object, its activities are considered one by one. Where extra resources are required for an activity, we draw directed lines from the appropriate resource circle into the top edge of the activity box (if time is taken to flow down the page).

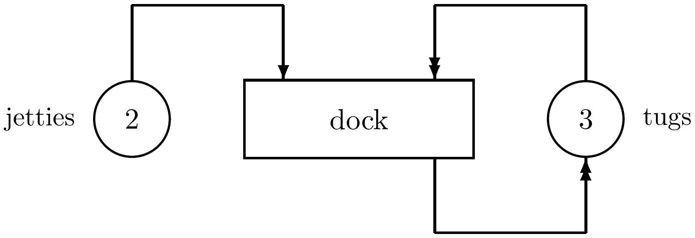


Figure 3.3: The activity dock

Directed lines leading from the bottom edge of the activity box back into resource circles represent resources no longer needed when the activity has been completed. Figure 3.3 shows the diagram for the activity *dock*. Notice that we have used double arrows from and into the resource `tugs` to indicate that two tugs are required. The full activity diagram for each boat's behaviour is obtained by stringing together the separate activity drawings in the correct time ordered sequence (see figure 3.4 overleaf).

In figure 3.4, we have included the outline of `class boat` (with a request followed by a seize shortened into acquire). Notice that the class declaration can be written down from the activity diagram line by line in a quite mechanical way. Indeed, Clementson [23] and Mathewson [57] have automated similar processes.

```
class boat;
begin
    acquire 1 jetty;
    acquire 2 tugs;

    dock;

    release 2 tugs;

    unload;

    acquire 1 tug;

    leave;

    release 1 tug;
    release 1 jetty;
end***boat***;
```

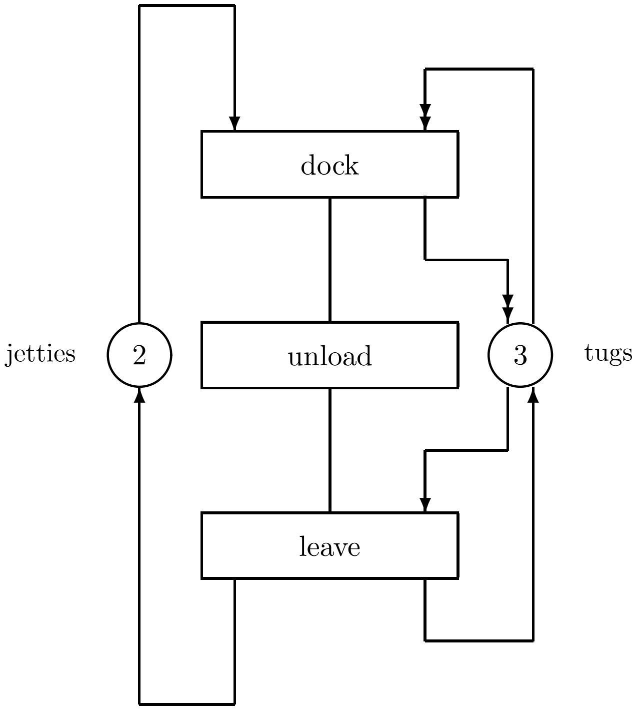


Figure 3.4: Port System activity diagram

EXERCISES 3 (continued)

Exercise 3.2 Customers arrive at a one man barber's shop for haircuts. If the barber is free, a haircut starts at once. If not, the customers wait on the first-come, first-served (henceforth *FCFS*) basis. Give a trace for the simulation in terms of the actions of the customers, assuming that four customers arrive at times 0, 20, 35, and 40 respectively and that each haircut takes 15 minutes. (Ignore the time taken to get seated, leave the chair and pay the barber.) Prepare an activity diagram for the model, and give an informal declaration for `class customer`.

HINT: let the barber be modelled as a resource, initially of size 1.

Exercise 3.3 Repeat exercise 3.2 above for a two man barber's shop using the same customer arrival sequence and the same length of time per haircut.

Exercise 3.4 A fleet of vans delivers sheet metal to a factory. When a van arrives at the one entrance, it passes overs a weighbridge which can weigh one van at a time. Then the van is driven to an unloading area where its contents are removed. Assume that each unloading takes 20 minutes and that there is always ample space for unloading so that the vans never need to queue here. Once unloaded, the vans are driven out of the factory, again passing over the weighbridge. Assume that vans arrive at times 0.0, 1.0, 24.0, and 25.0 minutes, that the weighing operation takes 3 minutes for vans going in either direction, and that the vans queue for the weighbridge in FCFS fashion. Ignore the time taken to drive between the weighbridge and the unloading area. Give a trace for this problem (stop at time = 40.0), construct the activity diagram and give a declaration for `class van`. What difference would it make to the trace if vans leaving were given priority in the weighbridge queue over vans entering?

3.2 A first look at Demos

We now outline Demos and its method of scheduling events. First note that Demos is just another Simula context, no more, no less. Demos takes its inspiration and approach from Simula's standard `class SIMULATION` (but does not use it). As with `class SIMULATION`, Demos provides various behind-the-scenes structures such as an event list and primitive routines for entering entities into it and deleting entities from it. These are not meant to be used directly. Instead Demos provides a selection of much higher-level classes and procedures for direct use. From the user's point of view, we can picture the declaration of the Demos context by[1]

```
class Demos;
begin
  class entity(title); value title; text title;
  begin
    procedure schedule(t); real t;...........;
    procedure cancel;........................;
  end***entity***;

  class res(title, avail);...................;

  ref(entity)procedure current;..............;

  real procedure time;.......................;

  procedure hold(t); real t;.................;
  ............................................
  actions to set up the event list at clock time 0.0;
  ............................................
end***demos***;
```

Demos is used as a context in the usual manner:

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    .....
  end;
```

NB. In the code presented throughout this text

```
external class Demos = "/usr/local/simulabin/demos.atr";
```

gives the path to where the externally compiled class Demos is located on *my* system at The University of Calgary. Expect the path to be different on your system.

[1] This skeleton suffices for the time being—other Demos facilities will be introduced as we go along. A more complete outline is given in appendix B.

3.2.1 Entities

`class entity` is used as prefix to declarations of major simulation components in Demos. Entity objects are given full life histories and are the only objects that can be scheduled in the event list. The event list is ordered according to the time of an entity's next scheduled event, with the smallest at the front. The scheduling strategy guarantees that the (single) active entity is the one at the head of the event list. `ref(entity)current` returns a reference to the entity that is now active, and `real procedure time` returns the simulation clock time (the event time associated with current).

`procedure schedule` is used to enter an unscheduled (or *passive*) entity into the event list. For example, `new boat("boat").schedule(15.0)` enters a new boat object into the event list at *the current clock time + 15.0.* A call `X.schedule(dt)` has no effect if `X` is already in the event list, i.e. is already scheduled.

`procedure hold` always operates on the entity at the head of the event list and is used to represent the duration of an activity. Seen from inside the calling object itself, it represents a period of time in the same state and holding the same resources until it takes up its actions again. `hold` is global rather than local to `class entity` to prevent an arbitrary object from being "held".

Note that the argument to `hold` and `schedule` represents a delay from the simulation clock time: it is not an absolute time.

`cancel` is used to remove a scheduled entity from the event list, e.g. a call `RIVAL.cancel` suffices to take the `RIVAL` entity out of the event list, should it be in there.

3.2.2 Resources

`class res` is used to model minor simulation elements, such as `tugs` and `jetties`, where we don't need to give full role descriptions, but essentially record how much of the modelled resource is currently available. In the port example, we declare `ref(res)tugs, jetties` and create appropriate objects by

```
jetties   :-   new res("jetties",   2);
tugs      :-   new res("tugs",      3);
```

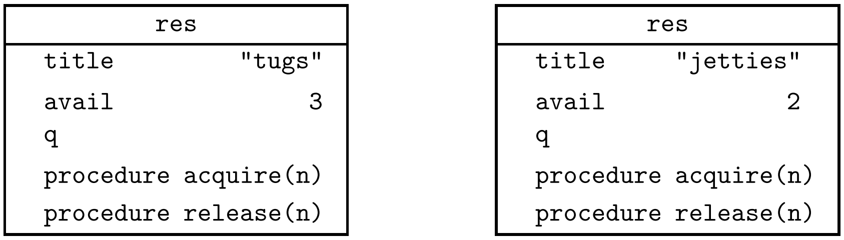


Figure 3.5: The two res objects

`res` objects require two parameters: the first is `text title` (enclosed in double quotes `"`) which is used in reports and traces; the second is `integer avail` which is used to fix the initial size of the resource pool. Thereafter, `avail` is maintained by calls on `acquire` and `release` to record the current level of the resource pool. Calls on `acquire` and `release` maintain the invariant `0 <= avail <=` *initial size.*

After object creation, portions of a resource may be acquired in integer chunks, e.g. `jetties.acquire(1); tugs.acquire(2);` Requests are considered on the first-come, first-served basis (this can be altered by use of `priority`, see chapter 4). A request is granted immediately if sufficient of the resource is free and no other entity is blocked. Otherwise the requester is itself blocked and is held in a hidden resource queue `q` (`ref(queue)q`, see chapter 5) local to the `res` object. There it remains until it is first in the queue and there is sufficient of the resource available for it to proceed.

When a current user releases its share, it sends a signal to the resource, e.g. `tugs.release(2); jetties.release(1);` A call on `release` not only increments the resource pool, but also unblocks any waiting entities whose request can be granted. Each unblocked entity leaves the resource queue `q` and enters the event list behind its unblocker and at the same clock time (see figure 3.3) at times 2.0 and 18.0).

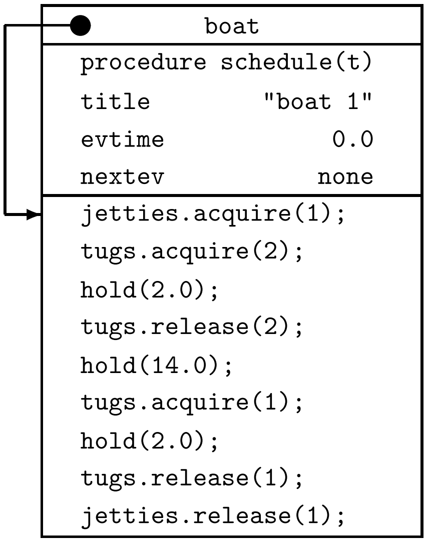
More detail on `acquire` and `release` is supplied when priority queueing is introduced in chapter 4.

We can now complete our model description in Demos:

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(res)tugs, jetties;

    entity class boat;
    begin
      jetties.acquire(1);
        tugs.acquire(2);
        hold(2.0);
        tugs.release(2);

        hold(14.0);

        tugs.acquire(1);
        hold(2.0);
        tugs.release(1);
      jetties.release(1);
    end***boat***;

    jetties :- new res("jetties", 2);
    tugs    :- new res("tugs"   , 3);
    new boat("boat").schedule(0.0);
    new boat("boat").schedule(1.0);
    new boat("boat").schedule(15.0);
    hold(36.0);
  end;
```

3.2.3 Remarks on Example 1

We create the first boat object by executing `new boat("boat")`. This gives us a carbon copy of the class declaration as depicted in figure 3.6.



```
boat
procedure schedule(t)
title          "boat 1"
evtime              0.0
nextev             none
jetties.acquire(1);
tugs.acquire(2);
hold(2.0);
tugs.release(2);
hold(14.0);
tugs.acquire(1);
hold(2.0);
tugs.release(1);
jetties.release(1);
```

Figure 3.6: The first boat object

The `entity` prefix contains the *user-accessible* procedure `schedule` and `text` title. You supply the class name when the object is created (`new boat("boat")`) and the Demos system automatically gives each fresh boat object its serial number, as shown in figure 3.4 below. Also shown in figure 3.4 are the extra attributes `evtime` and `nextev` which we will not need to use in any of our examples in the text, but are essential to our full trace in section 3.4. `evtime` records the time of this entity's next scheduled event and `nextev` the next entity (if any) after this one in the event list.

Because of a command within its `entity` prefix, a freshly created `boat` object is frozen with its LSC referring to the first user-written action. It will not start executing this action until it is scheduled. In our program, all three boat objects are scheduled as soon as they are created by the calls on `schedule`. Thus they enter their user-written action sequences at simulation clock times 0.0, 1.0, and 15.0 respectively. Thereafter, their lives are sequences of resource requests with implied waits of locally unpredictable (but possibly zero) length and holds of known duration.

Objects which have been scheduled by a `schedule` or `hold` are chained together behind the scenes in an *event list*. Associated with each such object is the known time of its next event—the time at which it is due to be first released into the simulation, or the time at which its current activity is due to finish. (Objects awaiting the availability of resources cannot be members of the event list.) Scheduled objects are ranked according to the `real` value of their next event time. Scheduling is framed so that the object at the head of the event list (the one with the least event time) is active. It has the standard reference `ref(entity)procedure current` and its event time is always available through a call on the standard `real procedure time`.

3.3 Trace of the Port System

We now illustrate how Demos's scheduling operates by tracing out the changes in the event list as the program is executed. The important points to watch out for are:

1. there are 4 entities in this model. Only one will be executed at any given time.

2. the entity being executed is always the one at the head of the event list, named *current*.

3. the next command to be executed is always `current.LSC` and the time at which it is executed is `current.evtime` also available as `time` for short.

4. a *X.schedule(dt)* schedules object `X` in the event at the *current clock time + dt*. If there are any other objects in the event list scheduled at this same time, `X` will be placed after them.

5. a *hold(dt)* command reschedules *current* in the event list delayed by *dt* (at *current.evtime + dt*), again placed after any object in the event list already scheduled for that time.

6. an *acquire* command will be carried out in zero time if enough of the resource is free. If not, *current* will be blocked. It is removed from the event list and queued on the resource until its request can be satisfied.

7. a *release* command always takes zero time and always leaves *current* active. The release command will awaken any blocked entities that can satisfy their acquisition request. This will move them from the resource queue and into the event list at the current clock time, but after *current*.

8. When an object has exhausted its useful actions and is no longer required, it will be automatically deleted. *Automatic garbage collection* has always been a positive feature of Simula; the manual deletion of objects is far too error prone to be left to humans. Avoid languages that let you do it like the plague.

3.3.1 Notation and layout

Space does not permit us to draw out each object in full so we have to make some compromises. Here is how we would represent the the boat object `B1` in the event list scheduled to carry out its next action at clock time 2.0.

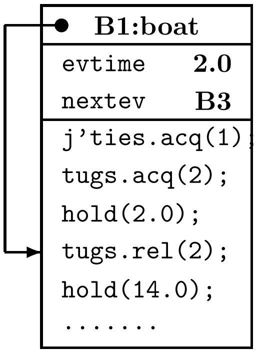


Figure 3.7: B1 scheduled to release 2 tugs at clock time = 2.0

We fudge the heading by giving the object reference and the class name, e.g. **B1:boat**, **D:Demos**; contract inconveniently long names (*j’ties* for jetties, *acq* for acquire, *rel* for release); and use dots ...... to de-emphasize uninteresting code.

To gain more insight into the behind-the-scenes-mechanisms, for each object we also note its

- *LSC* gives the next action to be carried out by B1 when active, here *tugs.release(1);*.
- *evtime* gives the time of an entity’s next scheduled event, (here 2.0) when in the event list
- *nextev* is used to chain entities scheduled in the event list and when blocked waiting on a resource. Here B3 will be scheduled behind B1 in the event list.

  When drawing such figures by hand, you will probably put in the *nextev*s as arrows. It looks a bit too cluttered if we do it here. Sorry.

Snapshots of the system state follow a fixed format, typified by figure 3.8 below, which show the entities scheduled for action in the event list and those blocked queued on the appropriate resource.

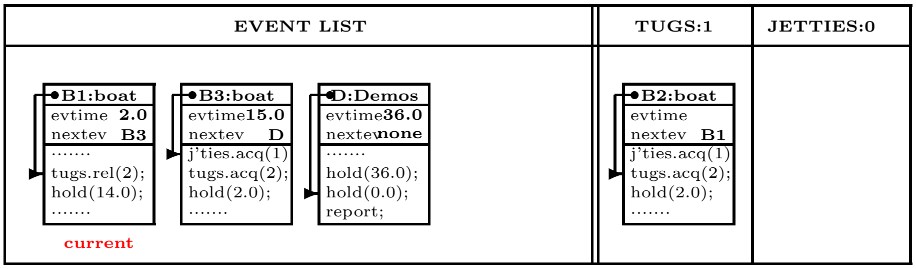


Figure 3.8: Snapshot layout: event list plus resource queues

In figure 3.8, `B1`, `B3` and `D` are scheduled to carry out their next events at times 2.0, 15.0 and 36.0 respectively; and `B2` has blocked as 2 tugs are not available. The entities in the event list are sorted by their `evtime`s and the object with the smallest evtime, known as `current`, is always at its front end. Scheduling is so framed that the object active now is *always* current. The simulation clock time is always taken to be `current.evtime`.

In figure 3.8,

- `B1` is `current` and is about to carry out the action referenced by its LSC and release two tugs. The simulation clock time is 2.0.

  To avoid clutter, we will not specially mark the current entity nor the simulation clock time but pick them up implicitly from the object at the front of the event list.

- `B3` is scheduled to start its actions at time 15.0.

- The `Demos` block is scheduled to resume its actions at time 36.0 when it will carry out a standard default coda: first a `hold(0.0)` which will give the opportunity for any other events scheduled at that time to complete. It will then issue a standard final report and shut down the simulation.

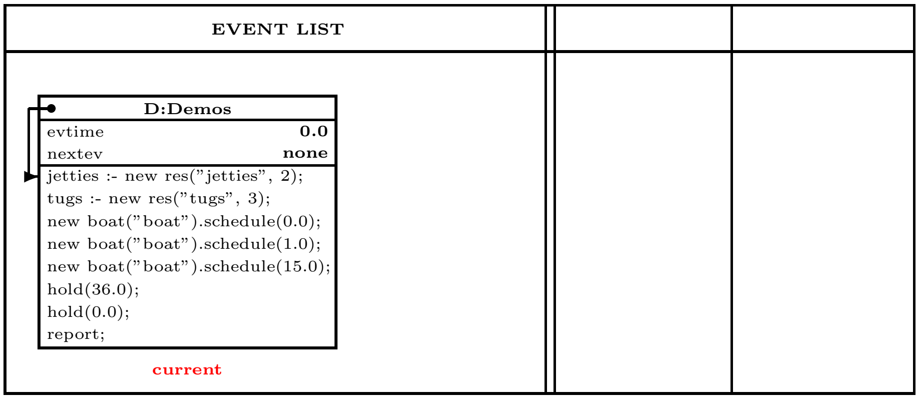
- `B2` is blocked awaiting the availability of 2 tugs.

In the headings to the resource columns, it is handy to note how much of each is currently free, here 1 tug and 0 jetties. `B1` owns 2 tugs and one jetty; `B2` owns 1 jetty and is blocked awaiting the availability of 2 tugs; `B3` has not yet started.

In the next section we give a full trace of the port system. The trace is a sequence of snapshots taken at those critical times when current changes or when a blocked entity is awakened.

3.3.2 A full trace of the Port System

Snapshot 1: simulation start up. Once the Simula system is loaded, it enters our program at the Demos level. There standard initialisation code sets up the event list and enters the Demos object, `D`, into it with its evtime set to 0.0.

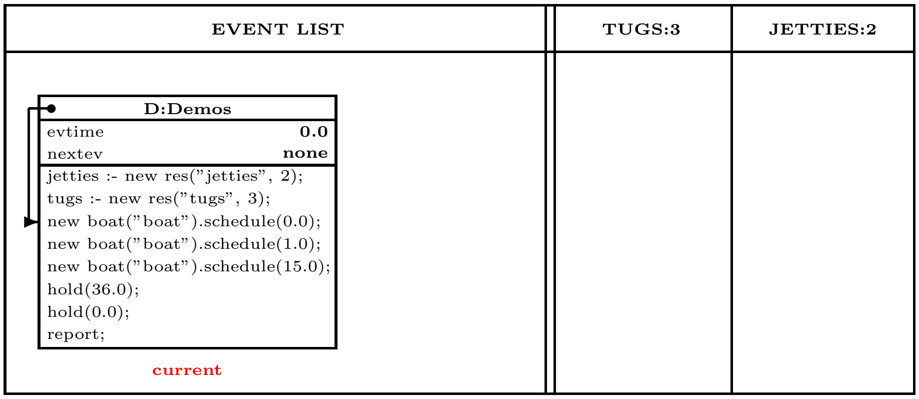


```
jetties :- new res("jetties", 2);
tugs :- new res("tugs", 3);
new boat("boat").schedule(0.0);
new boat("boat").schedule(1.0);
new boat("boat").schedule(15.0);
hold(36.0);
hold(0.0);
report;
```

Figure 3.9: Clock time = 0.0; current == Demos; initialisation commences

That done, we are ready to execute the user-written code prefixed by Demos. Snapshot 1 shows the Demos block with its LSC pointing to its first command `jetties :- new res("jetties", 2);` Control enters current (the first entity in the event list) and starts executing from its LSC.

Snapshot 2: the Demos block about to create B1. After executing the first two commands, our next snapshot is:



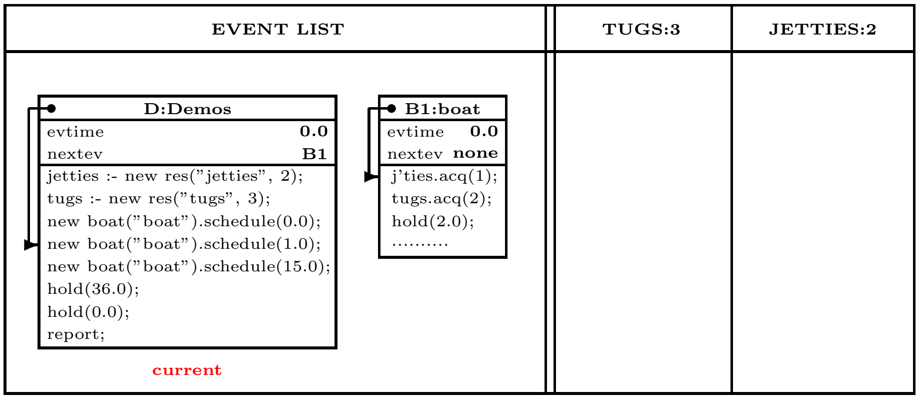
```
jetties :- new res("jetties", 2);
tugs :- new res("tugs", 3);
new boat("boat").schedule(0.0);
new boat("boat").schedule(1.0);
new boat("boat").schedule(15.0);
hold(36.0);
hold(0.0);
report;
```

Figure 3.10: Clock time = 0.0; current == Demos; resources established

in which the the LSC of the Demos block has moved on past the creation of the resources `jetties` and `tugs`. We fill out the resource columns in the tableau and note how much of each resource is currently free. `D` is still current and the clock time is still 0.0.

The next command to be executed is `new boat("boat").schedule(0.0);` which enters a fresh boat object into the event list at the current clock time delayed by 0.0. The entity will be given a unique serial number automatically. In practice this would be `"boat 1"`, which is here shortened to `"B1"`. `B1` is inserted into the event list at clock time 0.0 but behind the `Demos` block.

Snapshot 3: D about to create B2; and then B3. `D` is still current and the clock time is still 0.0.



```
jetties :- new res("jetties", 2);
tugs :- new res("tugs", 3);
new boat("boat").schedule(0.0);
new boat("boat").schedule(1.0);
new boat("boat").schedule(15.0);
hold(36.0);
hold(0.0);
report;
```

```
j'ties.acq(1);
tugs.acq(2);
hold(2.0);
..........
```

Figure 3.11: Clock time = 0.0; current == Demos; B1 now scheduled

The Demos block continues by scheduling `B2` at clock time 1.0 and `B3` at clock time 15.0 in turn.

Snapshot 4: D about to set the run length. `D` is still current and the clock time is still 0.0.

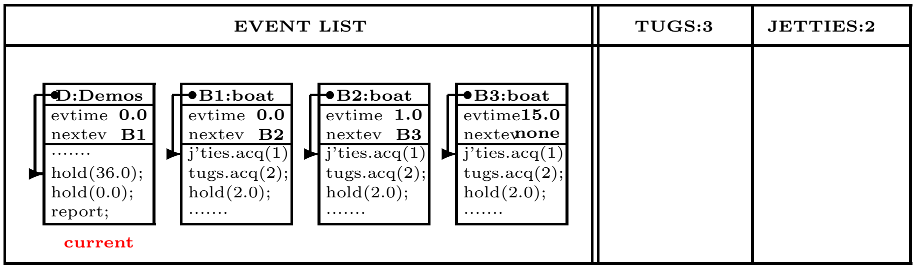


Figure 3.12: Clock time = 0.0; current == Demos; initialisation complete

With all the resources and entities in place, the Demos block is about to delay until the run is to terminate.

Note the "slimline" version of the Demos object which enables us to jam all future snapshots in one page width.

Do it now exercise: What would the snapshot be had we scheduled the 3 boats in reverse order ? i.e. by

```
new boat("boat").schedule(15.0);
new boat("boat").schedule( 1.0);
new boat("boat").schedule( 0.0);
```

Answer: just a renaming of the boats `B3` and `B1`.

The next action by `D` is `hold(36.0)` which removes it from the head of the event list and inserts it at the end of the event list.

Snapshot 5: B1 starts to dock. `B1` becomes the new `current` and we continue from its LSC. The simulation clock time becomes the evtime of B1, i.e. the clock time remains at 0.0.

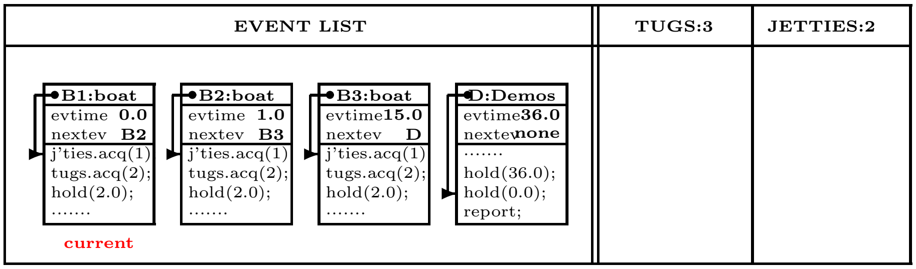


Figure 3.13: Clock time = 0.0; current == B1; B1 about to dock

`B1` seizes 1 jetty and 2 tugs and then executes `hold(2.0)`. This causes `B1` to be re-inserted in the event list at time 2.0.

Snapshot 6: B2 starts to dock. `B2` becomes the new `current` and the simulation time moves up to B2.evtime, which is 1.0. We continue from the LSC of `B2`.

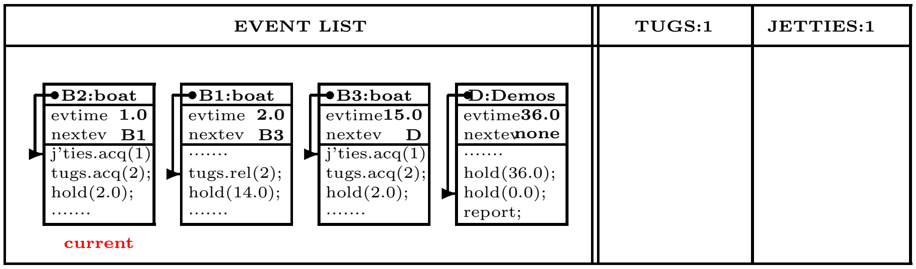


Figure 3.14: Clock time = 1.0; current == B2; B2 starts to dock

`B2` seizes a jetty, but is then blocked as 2 tugs are not currently available.

Snapshot 7: B1 completes docking. B2 has been removed from the event list is blocked on resource tugs and until (at least) 2 are available. B1 is the new current and the simulation clock time moves up to 2.0.

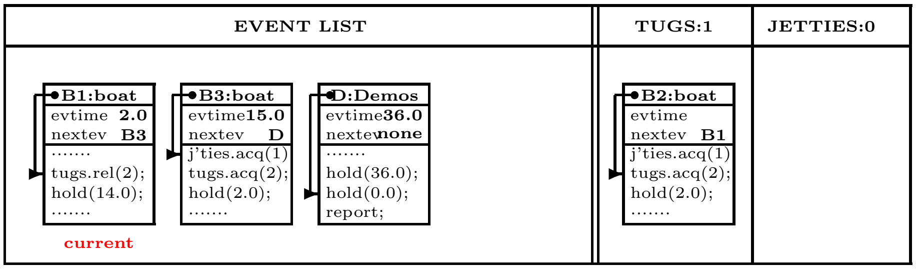


Figure 3.15: Clock time = 2.0; current == B1; B1 completes entering

B1 now releases 2 tugs. The release command will automatically awaken B2.

Snapshot 8: B1 starts to unload. B2 returns to the event list at once, but behind B1. Notice that the number of available tugs is still 1, and that the awakening handshake has nudged the LSC of B2 past the completed `tugs.acquire(2);` onto its next command `hold(2.0)`. B1 is still current and the simulation clock time remains at 2.0.

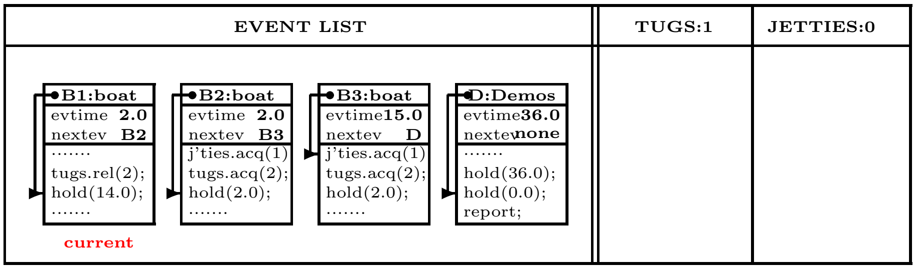


Figure 3.16: Clock time = 2.0; current == B1; B1 awakens B2

B1 continues by entering its docking phase (`hold(14.0)`) and is rescheduled in the event list at time 16.0.

Snapshot 9: B2 continues docking. `B2` is the new current and the simulation clock time remains at 2.0.

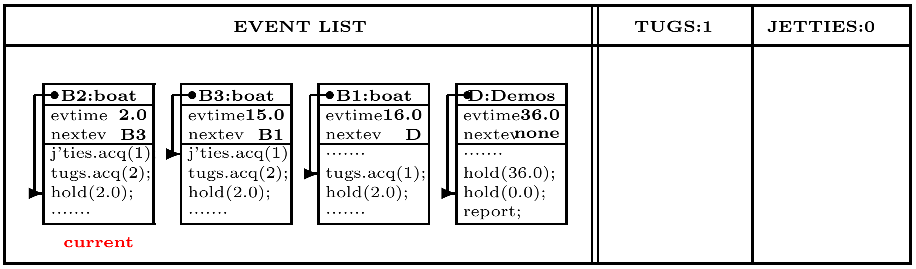


Figure 3.17: Clock time = 2.0; current == B2; B2 enters

`B2` has seized 2 tugs and starts to dock by executing `hold(2.0)`.

Snapshot 10: B2 completes docking and starts to unload. `B2` remains current but the simulation clock time moves up to 4.0.

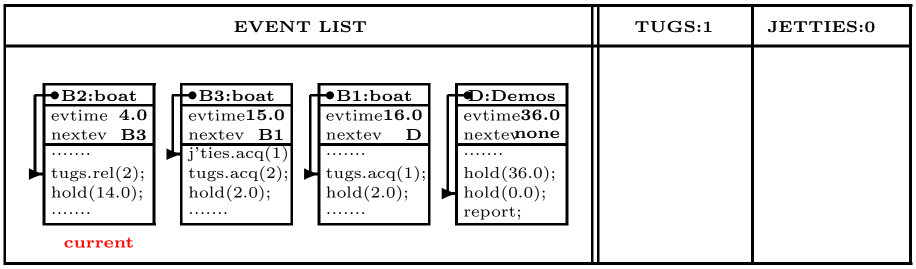


Figure 3.18: Clock time = 4.0; current == B2; B2 completes entry

`B2` releases 2 tugs and then starts to unload by executing `hold(14.0)`.

Snapshot 11: B3 tries to dock. `B2` is entered into the event list at time 18.0. `B3` becomes the new current and the simulation clock moves up to 15.0. All 3 tugs are now available.

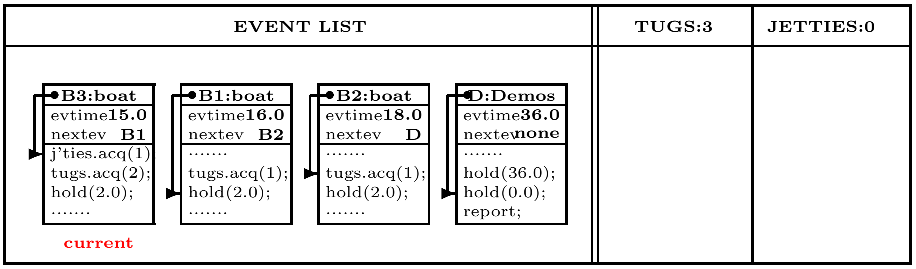


Figure 3.19: Clock time = 15.0; current == B3; B3 starts to enter

Poor old `B3` is immediately blocked as no jetties are free. It is removed from the event list and queued on the `jetties` resource.

Snapshot 12: B1 starts to leave. `B1` becomes the new `current` and the clock time advances to 16.0.

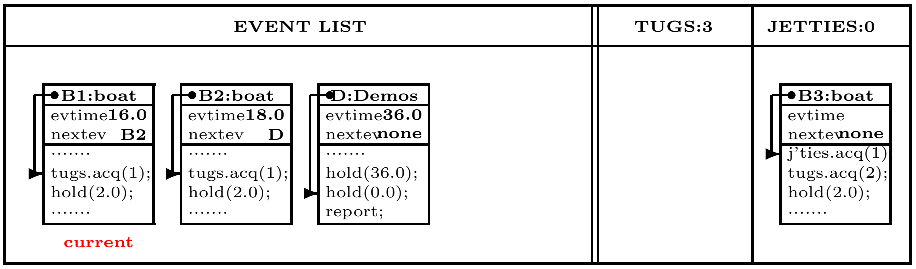


Figure 3.20: Clock time = 16.0; current == B1; B1 starts to leave

`B1` seizes one tug and then starts to leave (`hold(2.0);`).

Snapshot 13: B2 starts to leave. B1 is rescheduled in the event list at time 18.0, but behind B2. N.B. In general, tie breaks in the event list are resolved on the 'first scheduled for that event time, first taken' principle. A hold thus re-inserts the caller in the event list behind all other entities with events due to take place at the same time (if any). B2 is now current and the simulation clock time moves up to 18.0.

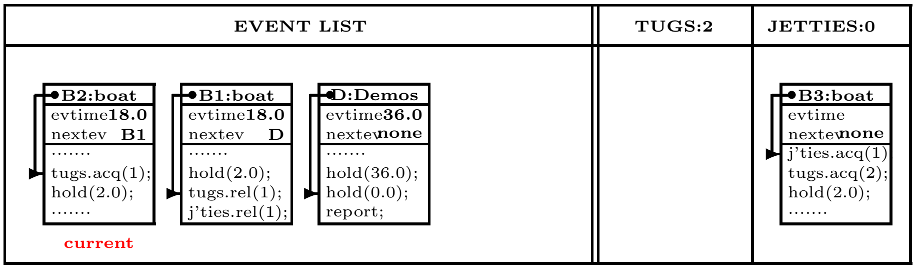


Figure 3.21: Clock time = 18.0; current == B2; B2 starts to leave

B2 seizes 1 tug and starts leaving by executing (hold(2.0);).

Snapshot 14: B1 releases its resources. B2 is re-entered into the event list at time 20.0. B1 becomes the new current and the simulation clock time remains at 18.0.

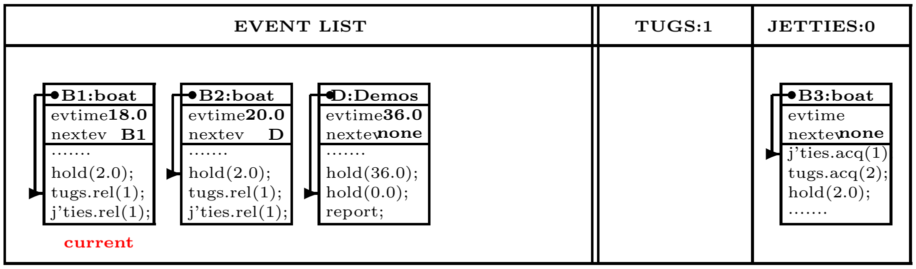


Figure 3.22: Clock time = 18.0; current == B1; B1 starts to leave

B1 now releases 1 tug and then 1 jetty. This last action awakens B3.

Snapshot 15: B1 is about to complete. Having seized a jetty, B3 joins the event list behind B1 and at the current simulation clock time. Notice that the LSC of B3 has been nudged past `jetties.acquire(1)` and that the number of free jetties is zero (B3 already owns a share). B1 remains current and the simulation clock time remains at 18.0.

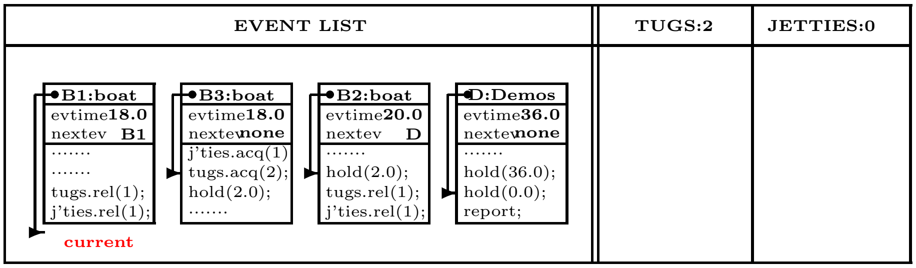


Figure 3.23: Clock time = 18.0; current == B1; B1 awakens B3

B1 has completed its actions and is automatically deleted from the simulation run.

Snapshot 16: B3 enters. The event list is now down to 3 live entities. B3 becomes the new `current` and the simulation clock time remains at 18.0.

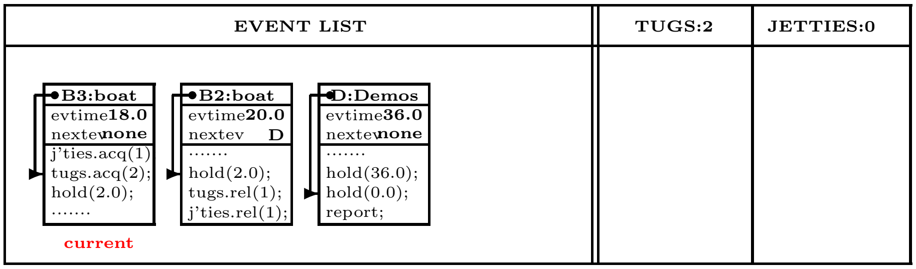


Figure 3.24: Clock time = 18.0; current == B3; B3 enters

B3 now seizes 2 tugs and executes a `hold(2.0)` simulating the docking activity.

Snapshot 17: B2 completes. `B3` is rescheduled in the event list at time 20.0 but after `B2`. `B2` becomes the new `current`.

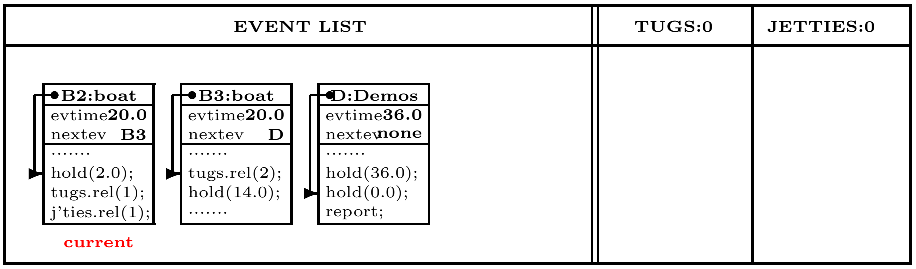


Figure 3.25: Clock time = 20.0; current == B2; B2 leaves

`B2` releases 1 tug and 1 jetty, and is deleted from the event list as its actions are exhausted.

Snapshot 18: B3 completes docking and starts to unload. The event list is now down to two objects. `B3` is the new current and the simulation clock time is 20.0;

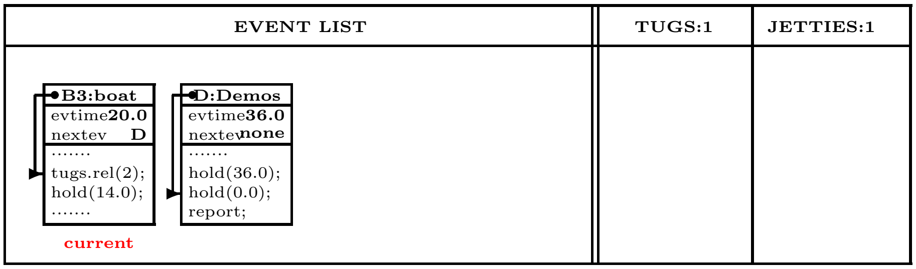


Figure 3.26: Clock time = 20.0; current == B3; B2 unloads

`B3` now releases 2 tugs and enters its unloading phase by executing `hold(14.0)`.

Snapshot 19: B3 completes unloading and starts to leave. `B3` remains `current` but the clock time advances to 34.0.

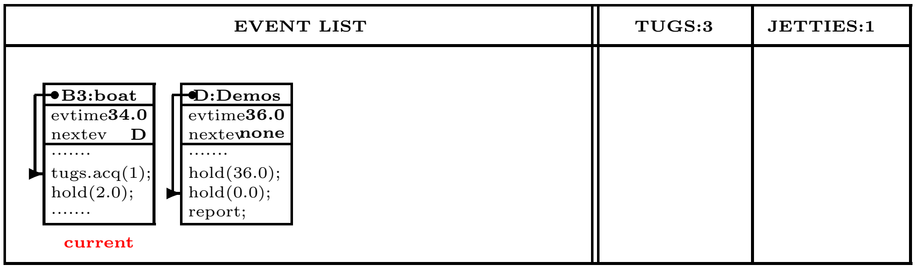


Figure 3.27: Clock time = 20.0; current == B3; B3 leaves

`B3` now seizes 1 tug and starts leaving, rescheduling itself for clock time 36.0.

Snapshot 20: Making sure that all boats complete. `B3` is entered into the event list at clock time 36.0 but is behind the Demos block `D`.

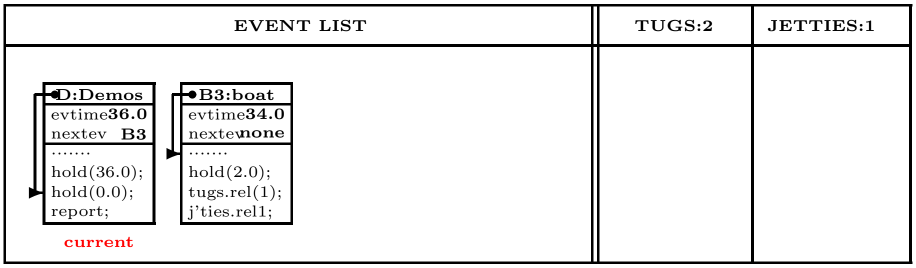


Figure 3.28: Clock time = 36.0; current == Demos; Demos pauses

If the simulation were to end now, it would not be quite complete as `B3` has not yet released its resources. To allow for this circumstance, the Demos block has been coded to execute an extra hidden `hold(0.0)` before it closes down the simulation. This simple trick causes the simulation run to end at the expected simulation clock time, but in effect gives all other events scheduled for that time precedence.

Snapshot 21: now B3 completes. `B3` becomes current again.

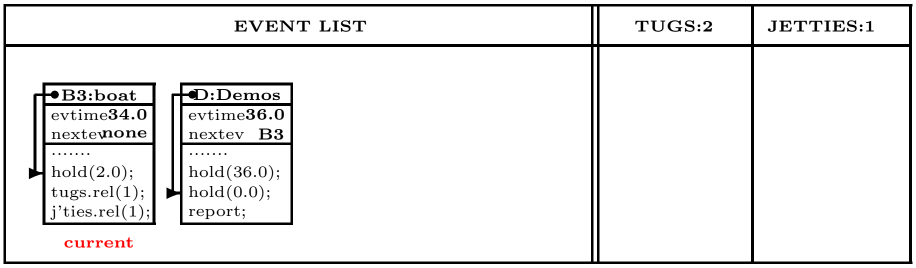


Figure 3.29: Clock time = 36.0; current == B3; B3 leaves

`B3` continues its actions by releasing 1 tug and 1 jetty. It is then deleted from the event list.

Snapshot 22: issuing a final report. Now `D` is current and the simulation clock time is 36.0.

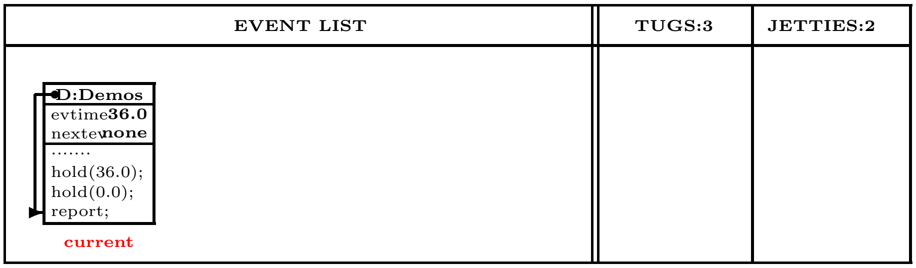


Figure 3.30: Clock time = 36.0; current == Demos; final report

Finally then, the Demos block is re-entered. It automatically issues a final report before closing down the simulation.

Coda. Once an entity has been created and scheduled, it decides for itself what to do next until its action pattern has been exhausted. It is then said to be *terminated* and can never be scheduled again. If not referenced (the attributes of referenced terminated objects may be accessed, e.g. `tugs` and `jetties`), the object is automatically deleted from the model. Thus boat objects are automatically deleted at clock times 18.0, 20.0, and 36.0.

Notice that the scheduling style, which is inherited from Simula, focusses our attention on `current`. We have not needed to reference any of the boat objects in the model explicitly. They are found implicitly either in the event list as `current`, or at the head of a resource queue.

EXERCISES 3 (continued)

Exercise 3.5 Give Demos code for the barber’s shop model given as exercise 3.2.

Exercise 3.6 Give Demos code for the factory model given as exercise 3.4.

Exercise 3.7 Give a frame by frame snapshot of the changes in the event list as your program for exercise 3.6 above is executed.

Exercise 3.8 Notice that all the boat objects in the port system (also customer objects in exercise 3.5 and van objects in exercise 3.6) are created at clock time zero. This may be tolerable in a small simulation involving only a few entities, but in larger models this method of dealing with transient entities is not acceptable. As an example, consider running our model of the port system over 10,000–20,000 time units. Perhaps 1000 boats will pass through, but no more than 5 or 6 may be in contention at any one time. To create all the boat objects at time zero wastes much space in the computer and clutters up the event list. What can be done to improve matters if we know the distribution of inter-arrival times of the boats?

Example 2: Port system revisited

Model data

Timings in hours:	
docking	`constant`:2.0
unloading	`normal`:mean=14.0,st.dev.=3.0
leaving	`constant`:2.0
boat inter-arrival	`negexp`:mean=0.1/hour
Resources:	
tugs	`res`:limit=3
jetties	`res`:limit=2

We have followed through the construction of one model in Demos and its execution by hand. The task was simplified by our choice of ’easy’ numbers. We now repeat the exercise with more realistic data, and use standard Demos mechanisms to introduce random behaviour into the simulation. We also take the opportunity to show a standard method of dynamically generating a stream of transient entities whose inter-arrival pattern is known (exercise 3.8), and introduce event tracing.

As before, docking and leaving are fixed length activities of duration 2.0 hours. Unloading is a sample from a `normal` distribution with mean 14.0 and standard deviation 3.0. The inter-arrival times are drawings from a negative exponential (negexp) distribution with a mean inter-arrival time of one boat every 10 hours. The simulation model is run for 28 days.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(res)tugs, jetties;
    ref(rdist)next, discharge;

    entity class boat;
    begin
      new boat("boat").schedule(next.sample);

      jetties.acquire(1);
        tugs.acquire(2);
          hold(2.0);
        tugs.release(2);

        hold(discharge.sample);

        tugs.acquire(1);
          hold(2.0);
        tugs.release(1);
      jetties.release(1);
    end***boat***;

    tugs      :- new res("tugs",    3);
    jetties   :- new res("jetties", 2);
    next      :- new negexp("next boat", 0.1);
    discharge :- new normal("discharge", 14.0, 3.0);

    trace;

    new boat("boat").schedule(0.0);
    hold(28.0*24.0);
  end;
```

OUTPUT

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                  *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

     0.000 demos        schedules boat 1 now
                        holds for 672.000, until 672.000
           boat 1       schedules boat 2 at 26.574
                        seizes 1 of jetties
                        seizes 2 of tugs
                        holds for 2.000, until 2.000
     2.000              releases 2 to tugs
                        holds for 16.385, until 18.385
    18.385              seizes 1 of tugs
                        holds for 2.000, until 20.385
    20.385              releases 1 to tugs
```

```
                        releases 1 to jetties
                        ***terminates
    26.574 boat 2       schedules boat 3 at 33.116
                        seizes 1 of jetties
                        seizes 2 of tugs
                        holds for 2.000, until 28.574
    28.574              releases 2 to tugs
                        holds for 14.724, until 43.298
    33.116 boat 3       schedules boat 4 at 41.836
                        seizes 1 of jetties
                        seizes 2 of tugs
                        holds for 2.000, until 35.116
    35.116              releases 2 to tugs
                        holds for 16.287, until 51.403
    41.836 boat 4       schedules boat 5 at 56.849
                        awaits 1 of jetties
    43.298 boat 2       seizes 1 of tugs
                        holds for 2.000, until 45.298
    45.298              releases 1 to tugs
                        releases 1 to jetties
                        ***terminates
           boat 4       seizes 1 of jetties
                        seizes 2 of tugs
                        holds for 2.000, until 47.298


......................................................................


                       clock time =    672.000
**********************************************************************
*                                                                    *
*                             r e p o r t                            *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
next boat         0.000     64 negexp        0.100            33427485
discharge         0.000     58 normal       14.000     3.000  22276755


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/ % usage/  av. wait/qmax
tugs              0.000    114    3    0    3   17.063 2.854&-002    1
jetties           0.000     56    2    0    0   78.566      5.498    6
```

3.3.3 Remarks on Example 2

The input to the program is the specification of the two random streams. The output from the program is an event trace followed by a standard report on the use of user-created Demos facilities. In the report, two distributions (`next` and `discharge`) and two resources (`tugs` and `jetties`) are detailed.

The report first details the distributions used in the simulation:

- `title`, `type`, and `a` and `b` echo back the user given name for the distribution, its type and argument(s)
- `obs` records the number of calls on the distribution
- `(re)set` gives the simulation time at which the object was created or reset (resetting is described in chapter 8)
- `seed` quotes the automatically determined value for each distribution. These come out in a mechanically predetermined sequence which can be overridden (see further remarks in chapter 8).

Then resource usages are detailed.

- `title` and `(re)set` are obvious
- `obs` gives the number of calls on `release`
- `lim` echoes the initial value of the resource
- `min` the minimum level of the resource reached in the observation period—`0 <= min <= lim`
- `now` is the currently available level of the resource
- `% usage` is the time weighted average of the portions of the resource seized expressed as a percentage of the maximum possible usage
- `av. wait` is the average wait time of entities which have completed calls on `acquire`. It includes zero waits.
- `qmax` is the maximum attained length of the queue of blocked requesters. Instant seizes—zero waits—ARE included).

N.B. Real values in reports are usually printed to 3 decimal places, but any real values which are too large to fit their allotted field or give only two significant places or less are printed floating point (see the `av. wait` column for `tugs`. 2.854&-02 is interpreted as 0.02854 (2.854×10^{-2}).

3.4 Dynamic entity generation

Notice how the boats are generated and scheduled (see in the trace at clock times 0.000 (twice), 26.574, 33.116, ...) and that they are automatically numbered sequentially. The first boat object is generated in the Demos block; thereafter, the first action of each boat object is to generate the next in sequence delay T (T = 26.574, 6.542, 8.720, ... in turn) before continuing with its own actions. The values in the sequence T (the boat inter-arrival times) are drawings from a negative exponential distribution with a mean rate of 1 boat every 10 hours. Using this device, we need keep only one as yet unentered boat object in the event list at a time.

3.5 Event tracing

Event tracing is initially off. It is switched on by a call `trace`, and continues to be on until switched off by a call `notrace`. `trace` and `notrace` are global routines which can be called from within entities as well as from the Demos block. Notice that the Demos system itself numbers the entities in the order of their creation. The user supplies a text (here ”boat”) and the Demos system appends to that text its serial number (taken modulo 100). In programs with several different entity classes, Demos will give each class its own sequence of serial numbers.

Because the Demos block itself behaves very much like an entity, it is simple to trace over a selected period. For example, should we wish to trace only the first 24 hours of the simulation, we replace the coding sequence

```
trace;
hold(28.0*24.0);
```

by the sequence

```
trace;
hold(24.0);
notrace;
hold(27.0*24.0);
```

With this coding, the Demos block sets tracing on after initialising the system, places itself into the event list at the close of the first day. When it becomes current again, it switches tracing off, and then places itself in the event list 27 days later. We will take full advantage of this flexibility in several later examples.

3.6 Pseudo-random number generation

The program contains two of Demos's random sampling mechanisms; several more will be met later[2].

Any simulation based on random behaviour naturally requires mechanisms for generating sequences of random numbers from various probability distributions. It is sufficient to have a sequence of random numbers from a uniform distribution available; for from it, by suitable mathematical transformations, it is possible to generate sequences of random numbers for other distributions. In turn, to obtain a sequence of uniform random numbers, it is sufficient to be able to generate a sequence X_k of integers in the range [1, M-1] as the sequence X_k/M is approximately uniformly distributed over (0,1). The simple examples below indicate how this can be done in Simula. Consider the sequences *s1 = 7, 3, 6, 1, 2,....* and *s2 = 4, 8, 5,10, 9,.....* Can you work out which number comes next in either sequence? In fact, both sequences have been produced mechanically by repeated application of the formula

$$\begin{aligned}\mathrm{X}_0 &= \text{any number in the range 1 through 10}\\ \mathrm{X}_{k+1} &= 2\times\mathrm{X}_k \text{ modulo 11 (k = 0, 1, 2, ...)}\end{aligned}$$

Clearly, X_0 = 9 for s1 and X_0 = 2 for s2. The full sequence starting with X_0 = 2 is *4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4 (repeats)*. The sequence is said to have a *cycle* of length 10. It contains all the integers in the range 1 through 10, although in "random" order.

The program segment below implements a uniform distribution based on this generator.

```
class random(u); integer u;
begin
  real procedure next;
  begin
    u := 2*u;
    if u > 11 then u := u-11;
    next := u/11;
  end***next***;

  if u < 1 or u > 10 then ERROR;  ! *** check the argument ***;
end***random***;

ref(random)s1, s2;

s1 :- new random(9);
s2 :- new random(2);
```

Successive calls on `s1.next` produce the numbers 0.636, 0.273, 0.545,...; and successive calls on `s2.next` the numbers 0.364, 0.727, 0.455,...

[2] A proper account of random number generation techniques is beyond the scope of this little book—see instead Fishman [30, 31], Knuth [44], or Shannon [90].

Notice that if by ill luck we had chosen 7 as the starting value for `s2`, calls on `s2.next` would have produced the sequence 0.273, 0.545, 0.091 .. which are too closely related to the outputs from `s1.next` for comfort. The problem is less acute in proper generators as their cycle lengths are enormous; but it should not be ignored.

Variates for a wide variety of theoretical and empirical distributions can be generated by building on the output from `next`. We give three examples to illustrate the method:

1. `randint` which generates `integer` values
2. `negexp` which generates `real` values, and
3. `draw` which generates `boolean` values

3.6.1 class randint

To generate random integers in the range [A, B], we declare

```
random class randint(a, b); integer a, b;
begin
  integer procedure sample;
  begin
    sample := a + entier((b-a+1)*next);
  end***sample***;

   if a > b then ERROR;
end***randint***;

ref(randint)r;

r :- new randint(9, 2, 5);
```

The first argument (9, above) is inherited from the prefix `random` and supplies the start seed., The second two arguments (2 and 5) fix the range of the drawings. Successive calls `r.sample` produce the sequence—4, 3, 4, 2, ... —randomly distributed integers in the range [2, 5].

3.6.2 class negexp

A negative exponential distribution may be implemented by

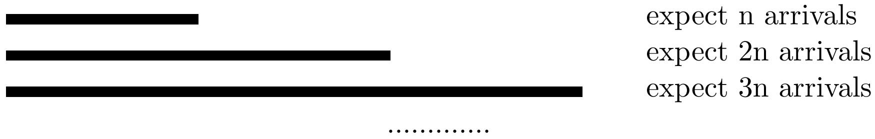
```
random class negexp(m); real m;
begin
  real procedure sample;
  begin
    sample := -ln(next)/m;
  end***sample***;

  if m <= 0.0 then ERROR;
```

```
end***negexp***;

ref(negexp)n;

n :- new negexp(2, 0.1);
```

Successive calls on `n.sample` produce the sequence of values 10.116, 3.185, 7.885, ... — a negative exponential distribution with a mean of 10.0. Following the precedent set by the Simula host, we supply *1/mean* as the actual argument.

Aside on the exponential nature of random arrivals

If arrival and service patterns do not depend upon the state of the system (i.e. they remain the same whether the system is heavily loaded or lightly loaded), then it is reasonable to assumpte that the number of arrivals in a time slot (period of observation) depends directly on the length of the time slot.



In otherwords, if the arrival rate is λ then we expect to average λ.t arrivals in a time slot of width t.

Now let's look at the probability of NO arrivals in a time slot of width t. We call this probability P(t), and by definition the probability of no arrivals in a time slot of width t+δt is P(t+δt).

Now we get no arrivals in a time slot of width t+δt by having no arrival in the slot of width t (and the probability of this happening is P(t)) and having no arrival in the next period of length δt. And since the probability of an arrival in a period of length δt is $\lambda.\delta$t (our starting assumption), then the probability of no arrival in the period δt is (1- $\lambda.\delta$t). So that

$$\mathrm{P(t+\delta t) = P(t) \times (1 - \lambda.\delta t)}$$

But the Taylor expansion of P(t+δt) is

$$\mathrm{P(t+\delta t) = P(t) + \delta t.P'(t) + O(\delta t^2) \; ....}$$

Combining these last two formulae we get

$$\mathrm{P(t) + \delta t.P'(t) \; .... = P(t) - \lambda.\delta t.P(t)}$$

As δt gets vanishingly small, we may neglect second order terms, so

$$\delta t.P'(t) = -\lambda.\delta t.P(t)$$

i.e. P'(t) = –λ.P(t), the solution to which is

$$\boxed{P(t) \quad = \quad e^{-\lambda t}}$$

Remember that P(t) is is the probability of no arrivals in a time period of length t, so the probability of some arrivals in this time period is $(1 - e^{-\lambda t})$.

3.6.3 class draw

Finally, we define a simple truth distribution

```
random class draw(p); real p;
begin
  boolean procedure sample;
  begin
    sample := p > next;
  end***sample***;
end***draw***;

ref(draw)heads, sixthrown;

heads     :- new draw(9, 0.5);
sixthrown :- new draw(9, 0.1667);

while not sixthrown.sample do
  .........
```

Successive calls on `heads.sample` produce the sequence of values `false`, `true`, `false`, `true`, `true`, ... —`boolean` values with a 50% chance of being `true`. Successive calls on `sixthrown.sample` produce the sequence of values `false`, `false`, `false`, `true`, `false`, ... —`boolean` values with a 16.67% chance of being `true`.

3.6.4 The Demos random number generators

The basic random number generator used in Demos is a Lehmer generator published by Downham and Roberts [31]. It is

$$\begin{aligned} Xo &= \text{some seed generated by Demos} \\ X_{k+1} &= 8192 \times X_k \text{ modulo } 67099547 \end{aligned}$$

and has a cycle length of 67099546. By noting that 8192 = 32×32×8, the generator can be coded in Simula in such a way as not to overflow on a 32 bit (or longer) word computer since $67099547 < 2^{26}$. Such random number generation algorithms produce predictable sequences of numbers since each later number is fixed by its predecessor. Thus all numbers in a sequence are completely determined by the initial value. To emphasise their predictable character, they are called *pseudo-random numbers*. But if the basic algorithm is carefully chosen, the numbers possess sufficient of the properties of random sequences to be capable of use as random sequences for many practical purposes.

Three groupings of distribution are defined in Demos.

- six return `real` values and are sub-classes of `rdist`:

  `constant`, `empirical`, `erlang`, `negexp`, `normal`, `uniform`,

- two return `integer` values and are sub-classes of `idist`:

  `poisson`, `randint`

- one returns `boolean` values and is a sub-class of `bdist`:

  `draw`

More details are found in appendix C.

To assist the inexperienced, the selection of appropriate well-separated random seeds is done automatically by a special routine in the Demos system. The routine is based on work by Mats Ohlin [72]; for the theory behind the method, see Fuller [33]. As noted before, these default values may be overridden.

It can be shown that $8192^{120633} = 36855$ modulo 67099547 (and $36855 = 3^4 \times 5 \times 7 \times 13$), and so we can derive a stream of seeds from the generator

$$\begin{array}{lcl} \mathrm{U}_0 & = & 907 \\ \mathrm{U}_{k+1} & = & 36855 \times \mathrm{U}_k \text{ modulo } 67099547 \end{array}$$

which has cycle of length 33549773 = 67099546/2. We can thus take over 120000 drawings before one stream starts to overlap with its successor, and this separation holds for more than 300 distributions.

3.7 Deadlock

> **NB** This section might have made sense in 1978, but we can do much better now. See the final section of Chapter 8 and appendices E and F for ideas on using process algebras to check out models for deadlock and safety and livelock properties.

One has to be careful about the order in which entities are allowed to request resources. For example, if we have two resources A and B each of limit 1, and entities P and Q which execute

```
a)  P:  a.acquire(1);      Q:  b.acquire(1);
          hold(t1);              hold(t2);
        b.acquire(1);          a.acquire(1);
```

then the following sequence of actions could occur: `P` seizes `A` and holds; then `Q` seizes `B` and also holds. Now both `P` and `Q` are blocked forever as each has the very resource the other one needs in order to continue. This situation is known as *deadlock*. It can be induced into Example 2 (page 40) if instead of the correct

```
b)   jetties.acquire(1);  tugs.acquire(2);
```

we code

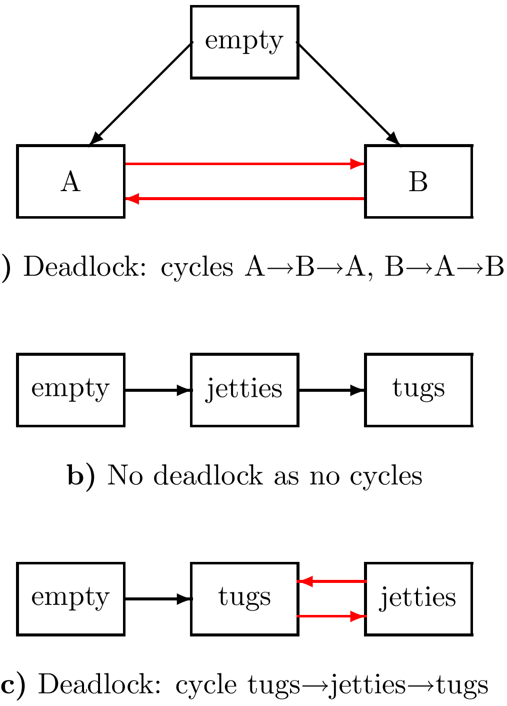
```
c)   tugs.acquire(2);  jetties.acquire(1);
```

The distribution report at time 672.0 then reads

```
                     d i s t r i b u t i o n s
                     *************************

title       /   (re)set/   obs/type     /         a/         b/     seed
next boat         0.000     64 negexp        0.100              33427485
discharge         0.000     10 normal       14.000      3.000   22276755
```

which shows that while 64 boats have arrived only 10 have discharged. The resource report reads

```
                         r e s o u r c e s
                         *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
tugs              0.000     18    3    0    1    60.654     0.111   55
jetties           0.000      8    2    0    0    96.787     0.862    1
```



a) Deadlock: cycles A→B→A, B→A→B

b) No deadlock as no cycles

c) Deadlock: cycle tugs→jetties→tugs

Figure 3.31: Deadlock: resource graphs

and reveals that many boats are blocked waiting for a tug. Deadlock has been caused by boats which own jetties and wish to leave blocking boats which own tugs (or have a prior claim) and want jetties.

A good account of deadlock, containing several examples, is found in Shaw [91]. The Demos system does not prevent deadlock occurring nor attempt to recover from it if it sets in. Deadlocked entities just grind to a halt while other entities continue. In certain elementary situations, the possibility of deadlock can be checked in advance using a simple graphical technique (see [91, chapter 8] for a more general method based on Petri nets). We represent the state of the system by a graph in which the nodes are resources and an arc from node A to node B implies that an entity which holds resource A can request node B. The node `empty` represents the initial empty state of an entity. Figure 3.31 gives resource graphs for the situations labelled a), b), and c) above. The deadlock condition is revealed by a closed loop in the resource graphs. Thus possible deadlocks can be detected quickly and corrected prior to coding a program. There is no way a context can do this for you, but either a compiler or a pre-processor for Demos could.

EXERCISES 3 (continued)

Exercise 3.9 Write a sub-class to `class random` which implements the normal distribution. For suggestions on the algorithm, see Fishman [30, 31], Knuth [44], Pritsker and Kiviat [85], and/or Shannon [90].

Exercise 3.10 Two types of customer arrive at a one chair barber's shop. Customers wanting a haircut only arrive at a mean rate of one every 40 minutes (`negexp` distributed), and customers wanting both a haircut and a shave arrive at a mean rate of one every 60 minutes (again `negexp` distributed). The barber serves on the first-come, first-served principle (FCFS). It takes him from 12-24 minutes (`uniform`ly distributed) to give a haircut only, and from 20-36 minutes (`uniform`ly distributed) to give both a haircut and a shave. The first haircut-only customer arrives at opening time, the first customer who wants both arrives 10 minutes later. Model the barber's shop and run it for 8 hours of simulated time.

HINT: model the barber as a `res`ource, and declare separate entity classes for the two types of customer.

Exercise 3.11 A tool crib is manned by two clerks who check out tools to mechanics. Mechanics use the tools to repair failed machines. The time to process a tool request depends on the type of tool. Requests fall into two categories

Type	Mechanic inter-arrival time	Service times in seconds
1	`negexp`:mean=0.005/sec	`uniform`:100.0→200.0
2	`negexp`:mean=0.008/sec	`uniform`: 75.0→150.0

At time 0.0, there is one request of each type pending. The clerks serve the mechanics in FCFS fashion independent of the type of request. Run the simulation model for 8 hours.

Exercise 3.12 A small grocery store has three aisles and two checkout counters. Shoppers arrive at the store with a mean inter-arrival time of 100 seconds, `negexp` distributed. on arrival, each takes a basket and may go down one or more of the three aisles selecting items for purchase as she/he proceeds. The probability of going down an aisle, the time required to shop an aisle and the number of items selected for purchase in the process are

Aisle	Prob	Time in seconds	No. of items
1	0.75	`uniform`: 60.0→180.0	`randint`:2→4
2	0.55	`uniform`:120.0→180.0	`randint`:3→5
3	0.82	`uniform`: 75.0→165.0	`randint`:6→8

When shopping has been completed, the customers queue up FCFS fashion at one of two checkout counters. Here each chooses another 1→3 impulse items (`randint` distributed). A customer's checkout time depends on the number of items she/he has bought and is 10 seconds per item, plus 15→35 seconds (`uniform`ly distributed) to pay and get the change. Run the model for 8 hours.

HINTS: the ALGOL 60 construction

`if` *condition* `then begin` *statements;* `end`

is also a part of Simula and hence Demos.

Also, you may care to use the global `text procedure edit` which accepts a `text t` and an `integer n` as actual parameters, and combines them into a single text (e.g. `edit("aisle", 17)` returns `"aisle17"`. If the text `t` is more than 10 characters long then it is stripped down to the first 10; if the integer value of N is not in the range 0 through 99 then `abs(n)//100` is accepted. It is commonly used with `res` etc. arrays which share the same text as title.

Exercise 3.13 Consider the program below which models a doctor's surgery.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(res)doctor;

    entity class patient;
    begin
      new patient("p").schedule(NEXT);

      doctor.acquire(1);
         hold(CONSULTATION);
      doctor.release(1);
    end***patient***;

    hold(540.0);  comment***start at 9 o'clock***;
    doctor :- new res("doctor", 1);
    new patient("p").schedule(0.0);
    hold(90.0);
  end;
```

The program has the first patient arriving at 9.00, and the doctor starting work at the same time. It closes abruptly at 10.30 whether the doctor is engaged in a consultation or not. Modify the program so that the surgery doors are locked at 10.30 (no more patients can then be admitted. Arrange to cut off the arrival stream at this time), and let the doctor finish off his current consultation (if any) and also consult with any patients who are waiting but arrived before 10.30.

Exercise 3.14 Suppose in exercise 3.13 above we have as initial conditions that `tt` patients are already waiting at time 9.00 when the doctor begins his work. It is not correct to replace the single statement

```
new patient("p").schedule(0.0)
```

in the Demos block by

```
for k := 1 step 1 until n do                          [ integer k ]
  new patient("p").schedule(0.0);
```

because then *each* of the n `patient`s has as its first action the generation of another. Thus this code would model n separate streams of patients instead of the one stream required. Give a correct solution.

Chapter 4

Entity-resource synchronisations

The main task of this chapter is to show how minor resources are handled in Demos. Resource synchronisations are classified into one or other of two types: mutual exclusion and producer/consumer. In mutual exclusion synchronisations, items from a pool of resources are requested and released by the same entity. In producer/consumer synchronisations, producer entities make resources available to consumer entities (in the manner of relay runners handing on a baton).

Separate classes `res` and `bin` are defined in Demos to handle these cases. Although not absolutely necessary, this is desirable as it allows better error control and more explicit reporting. These two classes have much in common and we will often call items modelled by `res` or `bin` objects 'resources'.

4.1 class res

We have already seen how to use `res`ources. Here we go straight into a second example intended to reinforce our understanding of mutual exclusion.

We tackle this and the remaining problems in this text with a standard divide-and-conquer methodology. We first present the problem and its data, then show a top-level decomposition into entities. We then decide how the entities are to interact. Once this has been decided, they can be developed separately.

Example 3: Readers and writers

A file is used to record the current status of elements in a dynamic system. It could, for example, be flight records for an airport. The file is periodically updated by writer processes, each of which must have sole access to the file when carrying out an update. The file is also read from time to time by reader processes, any number of which may access the file at the same time.

This model reinforces our newly acquired knowledge of Demos. It contains two `entity` classes contending for a resource and introduces priority queuing.

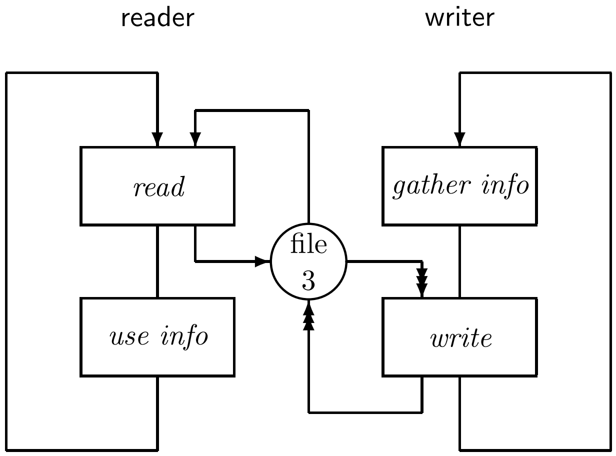


Figure 4.1: Readers and writers activity diagram

Structure of the model

We describe this model with two entity classes, `reader` and `writer`, and abstract the file away to a `res`.

The structure of the reader objects is a loop comprising a period of access to the file to read the latest information, followed by a sequence of actions based on that information.

Writers have a similar structure: within their behavioural loop, they first gather information, and then write it to the file.

These lead us to initial class outlines of:

$$\text{reader} = \textit{read};\ \textit{use info};\ \texttt{repeat};$$

$$\text{writer} = \textit{gather info};\ \textit{write};\ \texttt{repeat}$$

The activities *read* and *write* are critical code sections — we want to prevent several writers writing at the same time, and readers trying to read when a writer is writing. As a first attempt we protect file access by a resource `file` of size 1 and enforce the protocol that readers may not read and writers may not write without first acquiring the resource. It is easy to see that reader and writer code

$$\text{reader} = \texttt{file.acquire(1);}\ \textit{read};\ \texttt{file.release(1);}\ \textit{use info};\ \texttt{repeat;}$$

$$\text{writer} = \textit{gather info};\ \texttt{file.acquire(1);}\ \textit{write};\ \texttt{file.release(1);}\ \texttt{repeat;}$$

is sufficient but too stringent in that it permits only one reader to read at a time. Given that there are r readers and w writers, we can solve the file accessing problem very simply by means of a `res` object of limit `r`, say

```
  file :- new res("file", r);
```

As before a reader object gains access to the file by

```
  file.acquire(1); read; file.release(1);
```

which permits several readers at a time, but a writer object gains access to the file by acquiring all of the resource

```
  file.acquire(r); write; file.release(r);
```

Thus reader objects do not block each other, but block any writer object; a writer object will not only block other writer objects, but reader objects too.

The activity diagram for the model is given in figure 4.1 (notice how we represent the cyclic nature of these objects by a simple loop).

A complete program for 3 readers and 2 writers is listed below. The *read*, *use info*, *gather info*, and *write* timings are explicitly listed in the program. The program introduces a new data collection device — the `count`. `count` objects are used to record incidences.

```
C.update(n);                                     [ref(count) C ;]
```

increments a counter local to `C` by `n`. As can be seen from the program text, `count` objects have but one parameter, a `text title`. This title, the reset time, and the sum of the `n`'s are recorded in the final report.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(res)file;
    ref(count)reads, writes;

    entity class reader;
    begin
      file.acquire(1);
        hold(2.0);
      file.release(1);
      reads.update(1);

      hold(5.0);
      repeat;
    end***reader***;

    entity class writer;
    begin
      hold(5.0);

      file.acquire(3);
        hold(3.0);
      file.release(3);
      writes.update(1);
      repeat;
    end***writer***;
    reads  :- new count("reads");
    writes :- new count("writes");
    file   :- new res("file", 3);

    trace;

    new reader("r").schedule(0.0);
    new writer("w").schedule(0.0);
    new reader("r").schedule(0.0);
    new reader("r").schedule(2.0);
    new writer("w").schedule(1.0);
    hold(25.0);
  end;
```

OUTPUT

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                  *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

     0.000 demos        schedules r 1 now
                        schedules w 1 now
                        schedules r 2 now
                        schedules r 3 at 2.000
                        schedules w 2 at 1.000
                        holds for 25.000, until 25.000
           r 1          seizes 1 of file
                        holds for 2.000, until 2.000
           w 1          holds for 5.000, until 5.000
           r 2          seizes 1 of file
                        holds for 2.000, until 2.000
     1.000 w 2          holds for 5.000, until 6.000
     2.000 r 3          seizes 1 of file
                        holds for 2.000, until 4.000
           r 1          releases 1 to file
                        holds for 5.000, until 7.000
           r 2          releases 1 to file
                        holds for 5.000, until 7.000
     4.000 r 3          releases 1 to file
                        holds for 5.000, until 9.000
     5.000 w 1          seizes 3 of file
                        holds for 3.000, until 8.000
     6.000 w 2          awaits 3 of file
     7.000 r 1          awaits 1 of file
           r 2          awaits 1 of file
     8.000 w 1          releases 3 to file
                        holds for 5.000, until 13.000
           w 2          seizes 3 of file
                        holds for 3.000, until 11.000
     9.000 r 3          awaits 1 of file
    11.000 w 2          releases 3 to file
```

```
                        holds for 5.000, until 16.000
           r 1          seizes 1 of file
                        holds for 2.000, until 13.000
           r 2          seizes 1 of file
                        holds for 2.000, until 13.000
           r 3          seizes 1 of file
                        holds for 2.000, until 13.000
    13.000 w 1          awaits 3 of file
           r 1          releases 1 to file
                        holds for 5.000, until 18.000
           r 2          releases 1 to file
                        holds for 5.000, until 18.000
           r 3          releases 1 to file
                        holds for 5.000, until 18.000
           w 1          seizes 3 of file
                        holds for 3.000, until 16.000
    16.000 w 2          awaits 3 of file
           w 1          releases 3 to file
                        holds for 5.000, until 21.000
           w 2          seizes 3 of file
                        holds for 3.000, until 19.000
    18.000 r 1          awaits 1 of file
           r 2          awaits 1 of file
           r 3          awaits 1 of file
    19.000 w 2          releases 3 to file
                        holds for 5.000, until 24.000
           r 1          seizes 1 of file
                        holds for 2.000, until 21.000
           r 2          seizes 1 of file
                        holds for 2.000, until 21.000
           r 3          seizes 1 of file
                        holds for 2.000, until 21.000
           ..............................
    24.000 w 2          awaits 3 of file
           w 1          releases 3 to file
                        holds for 5.000, until 29.000
           w 2          seizes 3 of file
                        holds for 3.000, until 27.000


                       clock time =     25.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************



                             c o u n t s
                             ***********

                    title       /   (re)set/   obs
                    reads             0.000      9
                    writes            0.000      5


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
```

```
file              0.000     14    3    0    0    88.000     1.000    3
```

Remarks on Example 3

In the report on `file`, the statistic `% USAGE = 88.000` is made up of nine completed reads of chunk size 1 and duration 2 plus five completed writes of chunk size 3 and duration 3 (the fourteen completed usages recorded under `OBS`) plus one part write (`w 2` at time 24.0) of duration 1. The total number of space×time units comes to be $9{\times}1{\times}2 + 5{\times}3{\times}3 + 1{\times}3{\times}1 = 66$ which is 88% of the maximum possible usage $3{\times}25$.

A call on `repeat` (a procedure local to `class entity`) causes all the user written actions of the calling object to be repeated. The procedure has to be more subtle than revealed in this example, for very often the actions of an entity body take the form

```
begin
   initialising actions;
   repeated actions;
end;
```

A call on `repeat` in an entity body must not cause *all* the actions of such a body to be repeated: we want to exclude repetitions of the initialising actions. To avoid this, place a label `LOOP` on the first statement of the actions to be repeated.

```
begin
   initialising actions;
LOOP:
   repeated actions;
   repeat;
end;
```

Note that the label identifier *must* be `LOOP`, and `LOOP` can have no other meaning within an entity body. Should the entity body contain no initialising actions, no explicit occurrence of `LOOP` is needed.

Example 4: Readers and writers with priority

Example 3 gave neither readers nor writers priority. Should this be required, we can make use of a hitherto unmentioned attribute of `class entity`, namely `integer priority`. A local variable, it is initially zero. When an entity enters any queue, it is always ranked according to its current value of `priority` (larger values in front of smaller values, but after all other entities in the same queue with the same value of `priority`). In the first version of the readers and writers problem, readers and writers were queued on the FCFS principle as each and every `priority` was zero.

We can give reader objects priority by altering the actions in the body of their declaration to (recording essential actions only)

```
entity class reader;
begin
  priority := 1;
LOOP:
  file.acquire(1);
    hold(2.0);
  file.release(1);
  hold(5.0);
  repeat;
end***reader***;
```

and leaving the declaration of `class writer` unaltered. The following segment from the trace shows its effect. There is no change until time = 8.000 when `w 2`, `r 1`, and `r 2` are blocked in the resource `file`, `w 2` since 6.000, `r 1` since 7.000, and `r 2` since 7.000. At time 8.000, `w 1` releases 3 units back to the file. In example 3, `w 2` was promoted as it had been waiting longest. In this example, `r 1` and then `r 2` are promoted since they have higher priority.

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                  *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

     ....................................
     6.000 w 2          awaits 3 of file

     7.000 r 1          awaits 1 of file
           r 2          awaits 1 of file
     8.000 w 1          releases 3 to file
                        holds for 5.000, until 13.000
           r 1          seizes 1 of file
                        holds for 2.000, until 10.000
           r 2          seizes 1 of file
                        holds for 2.000, until 10.000
     9.000 r 3          seizes 1 of file
                        holds for 2.000, until 11.000
    10.000 r 1          releases 1 to file
                        holds for 5.000, until 15.000
           r 2          releases 1 to file
                        holds for 5.000, until 15.000
    11.000 r 3          releases 1 to file
                        holds for 5.000, until 16.000
           w 2          seizes 3 of file
                        holds for 3.000, until 14.000
```

The report the counts remains the same; the report on the resource `file` is

```
                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
file              0.000     14    3    0    0   84.000     0.941    3
```

Remarks on Example 4

As expected, with the shortest jobs getting priority (reading is faster than writing), the average wait time has decreased. Priority can be dynamically reassigned as often as desired. For example, we could give boat objects in Example 2 priority when entering the port by writing (informally)

```
entity class boat;
begin
  priority := 1;
  dock;
  unload;
  priority := 0;
  leave;
end***boat***;
```

Now that we have met `priority`, we can present a more complete picture of `release` and `acquire`. (See Appendix B for their semi-formal algorithms.)

acquire

A call `r.acquire(n)` does not delay `current` should sufficient of resource `r` be available (`r.avail` >= `n`) *and* `current` (who is making the request) have greater priority than any entity awaiting a share in `r`. In this case, `r.avail` is decremented by `n` and `current` continues on. Otherwise, `current` is deleted from the event list and enters the queue for `r` (`r.q` for short) in priority order.

release

A call on `r.release(n)` increments `r.avail` by `n` and enters `E1` the entity (if any) at the head of `r.q` (`E1 == r.q.first`) into the event list at the current clock time, but as last entity scheduled for that time. When `E1` becomes `current`, it tests to see if it can proceed (`current == r.q.first and r.avail` >= `n`). If not, it is deleted from the event list and remains blocked in `r.q`. Otherwise, `E1` leaves `r.q`, decrements `r.avail`, and promotes `E2` (the new first entity in `r.q`, if any) into the event list, but *after* itself. `E1` then continues on as `current`. When `E2` becomes `current`, it goes through the same exercise. Thus, waiting entities which can now proceed are peeled off the front end of `r.q` in priority order after `E1`. Note that a call on `release` does not delay `current` nor cause it to lose its position at the head of the event list.

EXERCISES 4

Exercise 4.1 Manufacturing widgets involves a relatively lengthy assembly process followed by a short fixing time in an oven.

Model data

Timings in minutes:

Assemble widget	`uniform` : 25.0→35.0
Fire in oven	`normal` : mean=8.0,st.dev.=2.0

Several assemblers share a single oven which can hold only one widget at a time. An assembler cannot begin assembling a new widget until she/he has removed the old one from the oven. Assume that there are 3 assemblers and that there is an infinite supply of raw widgets.

Run your model for a 40 hour week assuming no discontinuities within a day or in moving between consecutive 8 hour days.

Exercise 4.2 A machine is used to polish castings. The steps required to polish a casting are shown below (the timings are in minutes and all distributions are `uniform`).

1. fetch a raw casting from the storage area (9.0 → 15.0). Assume an infinite supply of unpolished castings.
2. load raw casting on to the polishing machine (6.0 → 14.0).
3. polish the raw casting (60.0 → 100.0).
4. reposition the casting on the machine for a final polishing (8.0 → 22.0).
5. carry out the final polishing (80.0 → 140.0).
6. unload and store the finished casting (15.0 → 30.0).
7. Repeat from 1).

The castings are too heavy to be handled by an operator. He requires the use of an overhead crane for each of the steps (1), (2), (4), (6) above. There is but one overhead crane which is also used to perform other tasks. Such tasks occur in the mean every 50 minutes (`negexp` distributed), and the time taken to service each call is `normal`ly distributed with a mean of 25.0 and a standard deviation of 5.0. Run your model for 400 hours of simulated time assuming no discontinuities, that the one polisher starts work at step (1) at time 0.0, and that the first *other task* for the overhead crane occurs at time 20.0.

Exercise 4.3 Assembled TV sets move through a number of testing stations in the final stages of their production. At the last of these, the vertical control is tested. If it is wrong, the offending set is rerouted for adjustment (the setting is modified).

After adjustment, the set is returned to the inspection station where it queues for retesting (with increased priority each time it fails, if more than once). After passing the test, the sets move on to a packing area.

Model data

Timings in minutes:

set arrivel rate	`negexp` : mean=0.2/minute
inspection time	`uniform` : 6.0 → 10.0
readjustment time	`normal` : mean=30.0,st.dev.=5.0

Resources:

inspectors	res:limit=2
adjusters	res:limit=1

The chance of a set passing the inspection is 90% whether it be for the first, second, ... attempt. Run your model for 40 hours and estimate the staging space (space for waiting sets) required ahead of both the stations.

Exercise 4.4 Faulty units are sent for repair to a special section in a factory. Repairs are carried out in two stages - first the unit is stripped down, and then it is rebuilt. Each operation has its own work station. Work station 1 (stripping) can work on two units at a time, work station 2 (rebuilding) on one unit at a time. But storage is limited, and at most four units can be queued in front of work station 1, and at most two in front of work station 2. If four units are already queued in front of work station 1, a newly arrived faulty unit is subcontracted. When a strip job is completed, the unit is automatically moved to the area in front of work station 2 when there is room (it takes 0.2 hours should the area be empty, and 0.1 hours should there be one unit already there) and a new strip job is started. Should the storage area in front of work station 2 be full, work station 1 is blocked until a space is freed.

Model data

Timings in hours:

unit arrival rate	`negexp` : mean=4/hour
strip down	`normal` : mean=0.50,st.dev.=0.05
rebuild	`normal` : mean=0.25,st.dev.=0.1
between stations	`constant` : 0.2 if area empty, 0.1 if not

At the start of the day, both work stations are idle, and two repair jobs are waiting. The next unit arrives at 0.5 hours. Run the model for a working week of 136 hours assuming no discontinuities and report on how many units were subcontracted.

Exercise 4.5 Repeat exercise 4.4 above with the following twist. Arrange for units arriving at the end of the week (after 134 hours) to be blocked. They are not subcontracted, but left as 'starters' for the following week. At this time, any other work in hand or pending is completed.

Exercise 4.6 A production line involves 5 servers stationed along a conveyor belt. Items to be serviced arrive at a mean rate of 4 per minute (`negexp` distributed). If unserviced they are carried along the conveyor passing a server every minute. If an item reaches an idle server, the item is picked off the conveyor, serviced (which takes `uniform` $0.8 \rightarrow 1.2$ minutes) and stored away. If an item passes all the servers, then it is recirculated and reappears in front of server 1 after a delay of 5 minutes. Run the simulation for 480 minutes, and note the work rates of the servers and the number of recirculated items.

Exercise 4.7 Repeat exercise 4.6 with the following change. Items are not recirculated. There is sufficient storage space allocated in front of server 5. Server 5 thus services all items that get past the other servers. Give an estimate of the storage space necessary in front of server 5.

4.2 class bin

We now introduce the second basic synchronisation, commonly called the *producer/consumer* synchronisation. A simple manifestation occurs when we have two cooperating entities, the first of which produces items for the second one to consume. Typical code skeletons for a producer and a consumer are

producer = *make item*; *give item*; `repeat;`

consumer = *take item*; *use item*; `repeat;`

Note that

- the producer is blocked if there is no space to put the item it has just made, i.e, it is producing items faster then they are being consumed
- and the consumer is blocked if no item is currently available when one is needed, i.e. it is consuming items faster than they are being produced.

Normally the producer will be (at least a little) faster then the consumer, but extra buffer spaces can help if there is a sudden flurry of new items.

In Demos, we represent an unbounded pool of slots for available items by a `bin` object (see figure 4.2). For this example, we could create an initial pool of with two slots already filled by

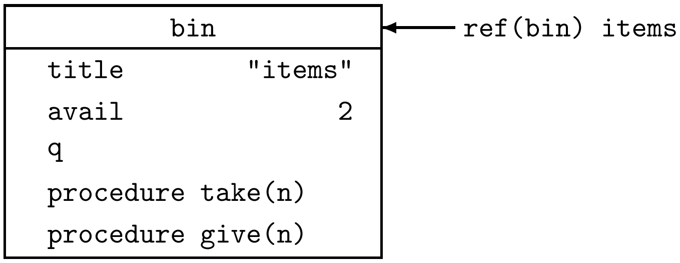


Figure 4.2: A bin object

```
items :- new bin("items",2);                    [ref(bin) items;]
```

The first parameter `text title` (passed as `"items"`) is used in reports and in traces; the second parameter `integer avail` gives the initial size of the pool (here 2, as shown in figure 4.2). Thereafter, as with `class res`, `avail` is maintained to record the current level of the available pool. Note however that this `avail` has no upper limit (`0` <= `avail`).

The consumer obtains an item from the pool by `items.take(1)`. If the pool is empty (`avail` = 0), the consumer waits in a hidden queue `q` local to `items`. When permission is granted, the consumer reduces the pool size by the amount requested, leaves the queue and is entered into the event list at the current clock time, but behind `current` (the producer).

When the producer has completed an item, it updates the size of the pool by `items.give(1)`. This command also awakens the consumer should it be blocked and allows it to continue now (at the current clock time, but behind `current`). Then the producer makes a start on the next item.

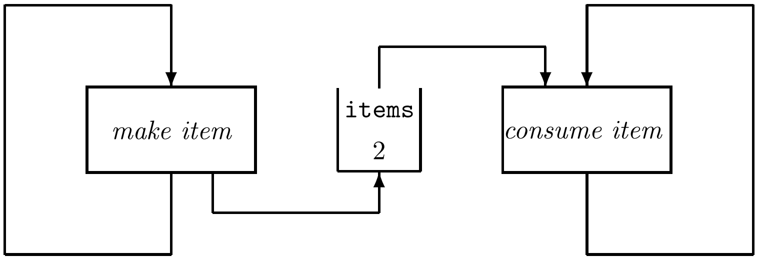
N.B. Of course `bin` portions larger than unity may be taken and given. The rules for `take` and `give` follow the pattern of those for `acquire` and `release`. Their semi-formal algorithms are given in appendix B.

The skeleton outlines for the producer and the consumer entities become simply:

producer = *make item*; `items.give(1); repeat;`

consumer = `items.take(1);` *use item*; `repeat;`

The activity diagram for this simple model together with corresponding Demos code is given in figure 4.3. Notice how the `bin` object has been represented in that figure: by convention, its initial value is drawn inside a “bucket”.



```
entity class consumer;
begin
  hold(make item);
  items.give(1);
  repeat;
end;
```

```
entity class producer;
begin
  items.take(1);
  hold(process time);
  repeat;
end;
```

Figure 4.3: Producer/consumer activity diagram.

Example 5: Car ferry

Motorists wishing to cross the strait between the mainland and a small island have to use a ferry. The ferry ties up on the mainland overnight and starts work promptly each day at 7.00. It shuttles to and fro between the mainland and the island until approximately 22.00 hours when the service closes down for the night. The ferry has a capacity limit of six cars. When the ferry arrives at a quay, the cars on the ferry are driven off, and then any waiting cars (up to the maximum of six) are driven on. When the ferry is fully loaded, or the quay queue is empty, the ferry leaves that side of the strait and starts another crossing. When the ferry has completed a round trip and deposited any passengers on the mainland, the captain checks the time. If it is 21.45 hours or later, the captain down the service for the night.

Model data

Timings in minutes:	
car inter-arrival:	
on mainland	`negexp` : mean = 0.15/min
on island	`negexp` : mean = 0.15/min
crossing time	`normal` : mean = 8.0, st.dev. = 0.5
drive on	`constant` : 0.5 (per car)
drive off	`constant` : 0.5 (per car)

Run the model for 1 working day under the following initial conditions: at time 7.00 there are 3 cars waiting on the mainland and 1 on the island.

Compute the average number of cars per trip, the number of fruitless (empty) crossings, and the average waiting time per car on each side.

Structure of the model

We exploit the symetry in the model by using arrays. Let the quantities on the mainland side have index 1, and corresponding quantities on the island side have index 2.

Since we are interested in how many cars are waiting, and not in how individual cars arrive, cross, and continue, we need not model cars as entities Instead we can use objects to record the current queue length on each side of the strait. Thus we can model this problem with one main entity representing the car ferry. Besides the ferry, we need minor entities to generate car arrivals at appropriate times. An outline of these entities is:

```
ferry      =  $load_1$; cross; $unload_2$;
              $load_2$; cross; $unload_1$;
              if ok then repeat;
              shutdown;

arrival 1  =  hold(car interarrival); generate car on mainland side; repeat

arrival 2  =  hold(car interarrival); generate car on island side; repeat
```

Now that we have the program decomposition, we examine the details of the rôles of the various entities separately.

car arrivals

Taking `NEXT[1]` and `NEXT[2]` as the car inter-arrival distributions, and `Q[1]` and `Q[2]` as `bins` for the car queues, car arrivals can be taken care of by a pair of objects (one for each side) of `class arrival`.

```
entity class arrival(side); integer side;
begin
  hold(NEXT[side].sample);
  Q[side].give(1);
  repeat;
end***arrival***;
```

class ferry

The ferry starts from the mainland and sails to and fro until it is time to shut down. A skeleton of `class ferry` is expressed neatly using a for-loop

```
entity class ferry;
begin
  integer side, c;

  for side := 1, 2 do
  begin
    load;
    cross;
    unload;
  end;
  if time < 21.45 o'clock then repeat;
  shutdown;
end***ferry***;
```

We now examine in turn the activitities *load* and *unload*, and the shutting down of the simulation model.

load

We consider a number of options on our way to acceptable coding.

- First we reject

  ```
  Q[side].take(6);
  hold(6*0.5);
  ```

  since it waits until the queue length is 6 before permitting a loading.

- As a second attempt,

  ```
  for k := 1 step 1 until 6 do
  begin
    Q[side].take(1);
    hold(0.5);
  end;
  ```

  loads the cars as they arrive but still waits for a full load before crossing.

- Our third attempt maintains a count of the number of cars onboad in `c` and uses the `bin` attribute `avail` to quit loading if the queue is empty. We assume a variable `integer c` local to `class ferry` (we may wish to extend the model to several ferries, and then each ferry will keep its own count). Loading takes place while there is at least one car in the queue and the ferry capacity limit of 6 has not been reached. Assuming that `c = 0` initially, this is

  ```
  while c < 6 and Q]side].avail > 0 do
  begin
    Q[side].take(1);
    hold(0.5);
    c := c + 1;
  end;
  ```

  which we accept.

- Aside: A fourth real-life strategy might be to wait, say 30 minutes, before loading. This too is easy to express (assuming `c = 0` initially):

```
hold(30.0);
while c < 6 and Q[side].avail > 0 do
begin
  Q[side].take(1);
  hold(0.5);
  c := c + 1;
end;
```

unload

Unloading is easier. It merely consists of letting all the cars (and there are `c` of them) drive off. This is the appropriate place to set `c`, the count of cars on board the ferry, to zero.

```
hold(c*0.5);
c := 0;
```

Shutting down the simulation

This is the second model (see also exercise 3.13) in which the closing down time of the simulation cannot be predicted in advance. We synchronise via another `bin` object, named `shutdown` which is initially zero. After initialising the system, the Demos block calls `shutdown.take(1)` and is blocked. It is left to the `ferry` object to decide when to shut down the system. This it does by executing `shutdown.give(1)` as its last action. This awakens the Demos block at precisely the right instant.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
begin
  ref(bin)array Q [ 1:2 ];
  ref(rdist)array NEXT [ 1:2 ];
  ref(bin) shutdown;
  ref(rdist)cross;
  ref(tally)load;
  ref(count)trips, empties;

  entity class ferry;
  begin
    integer side, c;

    for side := 1, 2 do
    begin
      while c < 6 and Q[side].avail > 0 do
      begin
        Q[side].take(1);
        hold(0.5);
        c := c + 1;
      end;
      load.update(c);
      if c = 0 then empties.update(1);
      hold(cross.sample);
      hold(c*0.5);
      c := 0;
    end;
    trips.update(1);

    if time < 1305.0 then repeat;
    shutdown.give(1);
  end***ferry***;

  entity class arrival(side); integer side;
  begin
    hold(NEXT[side].sample);
    Q[side].give(1);
    repeat;
  end***arrival***;

  hold(420.0);   comment***start at 7.00;

  Q [ 1 ]    :- new bin("mainland", 3);
  Q [ 2 ]    :- new bin("island",   1);
  shutdown   :- new bin("shutdown", 0);
  NEXT [ 1 ] :- new negexp("mainland", 0.15);
  NEXT [ 2 ] :- new negexp("island",   0.15);
  cross      :- new normal("crossing", 8.0, 0.5);
  trips      :- new count("trips");
  empties    :- new count("empty trips");
  load       :- new tally("av. load");

  new arrival("arr", 1).schedule(0.0);
  new arrival("arr", 2).schedule(0.0);
  new ferry("ferry").schedule(0.0);
  shutdown.take(1);
end;
```

OUTPUT

```
                       clock time =   1315.296
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                             c o u n t s
                             ***********

                    title       /   (re)set/   obs
                    trips           420.000     39
                    empty trips     420.000      2


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
mainland        420.000    133 negexp        0.150            33427485
island          420.000    146 negexp        0.150            22276755
crossing        420.000     78 normal        8.000     0.500  46847980
```

```
                            t a l l i e s
                            *************

title       /   (re)set/   obs/  average/est.st.dv/  minimum/  maximum
av. load        420.000     78     3.487     1.634     0.000     6.000

                               b i n s
                               *******

title       /   (re)set/   obs/init/  max/  now/  av. free/  av. wait/qmax
mainland        420.000    132    3     6     2     1.678      0.000    1
island          420.000    145    1     7     7     1.705      0.000    1
shutdown        420.000      1    0     1     0     0.000    895.296    1
```

Remarks on Example 5

The output echoes the distribution definitions, then the counts and tallies, and finally the BIN usages are listed. The `bin` report columns are headed

- `title` and `(re)set` are both obvious
- `obs` records the number of completed calls on `GIVE`
- `init` records the initial level of this `bin`
- `max` records the maximum level of this `bin`,
- `now` records the current level of items available for taking
- `av. free` records the time-weighted average number of items available since the creation of this `bin`
- `av. wait` records the average time spent queueing by entities blocked on this `bin` including zero waits, and finally
- `qmax` records maximum number of blocked entities including zero waits

The program introduces another data collection device — the `tally`. `tally` objects are used to record time independent variables. Observations are recorded by

```
    T.update(x);                                    [ ref(tally) T;]
```

Various statistics are maintained over the readings x_1, x_2, ..., x_n, as the report `"t a l l i e s"` shows. They include the number of observations (`OBS`), their mean, their estimated standard deviation, their minimum, and their maximum.

EXERCISE 4 (continued)

Exercise 4.8 This simulation follows the (much simplified) progress of a billet through a steel mill. Billets are long, thickly sectioned steel bars which arrive at the mill from another factory (we assume an inexhaustible stockpile). The job of the mill is to convert each billet into steel plate. To accomplish this, the billets are heated one at a time in a furnace until they have reached a 'suitable' temperature. Each billet is then transported on a railway bogie to a soaking pit area (if no bogies are available, the current billet is kept inside the furnace). The billet is unloaded with the help of a crane. This frees the bogie which is then shunted back to the furnace. Note that a billet will be loaded straight from its bogie into a soaking pit should one be free; otherwise the crane dumps it in the soaking pit area and loads it later when a pit is free.

The billet is left in the soaking pit until it has reached a uniform temperature throughout. It then becomes eligible to be rolled, but is kept in the soaking pit until a rolling mill is free. A crane is also needed for the unloading operation. The billet passes through the rolling mill several times and is shaped a little more on each pass. Eventually it is squeezed into the desired shape of a flat plate.

Use lower case letters to denote informally the activity durations. Assume that bogie movements do not interfere with each other and take a negligible time. Simulate with 12 soaking pits, 2 cranes, 9 bogies and 1 rolling mill.

Exercise 4.9 A small production line has three stages: the first assembles the inner and outer rings of bearings, the second greases the assemblage, the third packs them two to a box (the packers take two greased assemblages at a time). There are 3 assemblers, 1 greaser, and 2 packers.

Model data

Timings in minutes:

inner arrival rate	`negexp` : mean=6/min
outer arrival rate	`negexp` : mean=6/min
assembling	`normal` : mean=0.5,st.dev.=0.1
greasing	`constant` : 0.15
packing	`normal` : mean=0.6,st.dev.=0.1

Initially there are 10 inners and 10 outers. Run the model for 8 hours.

Exercise 4.10 A machine uses a type of part which fails periodically. Whenever this happens, the machine must be switched off. The faulty part is then removed by its operator and wheeled by him to the repair shop. The operator then takes a replacement part (queueing if none are there) and wheels it back to his machine. The operator then installs the replacement part, and starts a fresh run.

Faulty parts are repaired by a repairman. He also has a never failing supply of other jobs which occupy him when there are no faulty parts to repair. Although these other jobs have a lower priority, once started they cannot be interrupted.

Model data

Timings in hours:	
part life time	`normal` : mean=36.0,st.dev.=7.0
removal time	`constant` : 0.4
repair time	`normal` : mean=2.0,st.dev.=0.5
replacement time	`constant` : 0.4
other job time	`uniform` : 0.5→1.5

The removal and replacement times include the time spent wheeling the parts to and from the repairman. There are sufficient wheelbarrows in the factory for them not to cause delays.

Run the model for a four week month assuming that there are no discontinuities between shifts. Let there be three machines (initially all in working order). At time 0.0, there is one faulty part awaiting repair. Arrange for the machines to break down at approximately 6 hours, 18 hours and 30 hours respectively.

Exercise 4.11 Two processes communicate with each other via a buffer of capacity L. The sender process, S, deposits messages of uniform size into one of the L buffer slots. The receiver process, R, extracts the messages one by one and decodes them. R and S may not access the buffer at the same time.

Model data

Timings in minutes:	
message arrival rate	`negexp` : mean=1/min
deposit a message	`constant` : 0.05
extract a message	`constant` : 0.05
decode a message	`uniform` : 0.6→1.4

Write Demos programs to model this system under the assumptions

1. the buffer has an infinite capacity. S puts the messages into buffer slots 1, 2, 3, ... etc.

2. the buffer has a finite capacity and is organised cyclically. S puts messages into buffer slots 1, 2, ..., L, 1, 2, ..., L, 1, etc. Be careful to ensure that S does not overwrite a previous message before R has extracted it.

Exercise 4.12 A garage is open from 9.00 until (about) 17.00 weekdays and from 9.00 until (about) 13.00 on Saturdays for the maintenance and repair of motor cars. The garage has 5 service bays each of which can deal with one car at a time. Two classes of car are serviced by the garage

1. private cars which are booked-in in advance and left by their owners outside the garage on the appointed day at or before 9.00.

2. police cars which are repaired by the garage under a special contract. Police cars are in use 24 hours a day. When one is in trouble it is brought to the garage at once for an unscheduled but high priority repair.

The simulation runs over a four week period. Each weekday the garage tries to shut at 17.00 hours. Any work then in progress is completed, but work not yet started is suspended until 9.00 the following day (and this includes work on police cars). On saturdays, the garage completes outstanding services on all vehicles booked in before 13.00 before closing for the weekend. Any police cars arriving while the garage is still open servicing this backlog will also be serviced. Notice that if a private car is kept overnight it takes precedence over the next day's intake of private cars, but it may be overtaken the following workday morning by a police car which has arrived overnight.

Model data

Timings in hours:	
Scheduled maintenance	`uniform` : 1.5→2.5
Police car inter-arrival	`negexp` : mean=1/12 per hour
Police car repair	`normal` : mean=2.5,st.dev.=1
Resources:	
Bays	`res` : limit=5
Scheduled bookings:	
Private car group size (halved on saturdays)	`randint` : 12→20 weekdays

Chapter 5

Entity-entity cooperations

In the simulations we have described so far, an activity has always involved one entity and one or more minor components modelled by resources (`res` or `bin` objects). Sometimes this is not possible and an activity must be described as the coming together of two or more entities. We illustrate the problem with a sequence of examples ranging from a simple modification to the car ferry example to some which are quite hard.

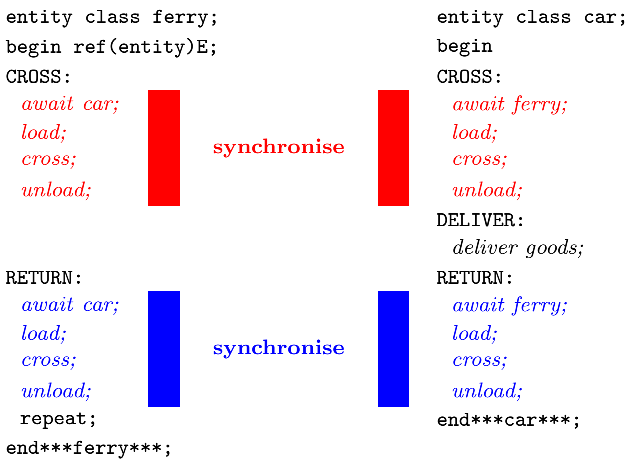
5.1 Master/slave synchronisations

Consider a ferry service similar to that of Example 5, except that this time we are interested in modelling the behaviour of the cars as they cross to the island, deliver goods, and finally return to the mainland. So that we can focus upon this new issue, we introduce some simplifications. Specifically, we allow only one car per trip on the ferry, we let the ferry operate a 24 hour service round the clock, and insist that the ferry wait for a car should now be queued for a loading. The crux of the problem is that we cannot now model the cars as resources; they must be modelled as entities. Informal declarations of `class car` and `class ferry` are taken as

`car` = *load*; *cross*; *unload*; *deliver*; *load*; *cross*; *unload*

`ferry` = *load*; *cross*; *unload*; *load*; *cross*; *unload*; `repeat`

The sequence of activities *load*; *cross*; *unload*; (which occurs twice) cannot take place without both a car and a ferry and the question is how to make two (or more) entities do — or at least *think* they are doing — the same thing at the same time.

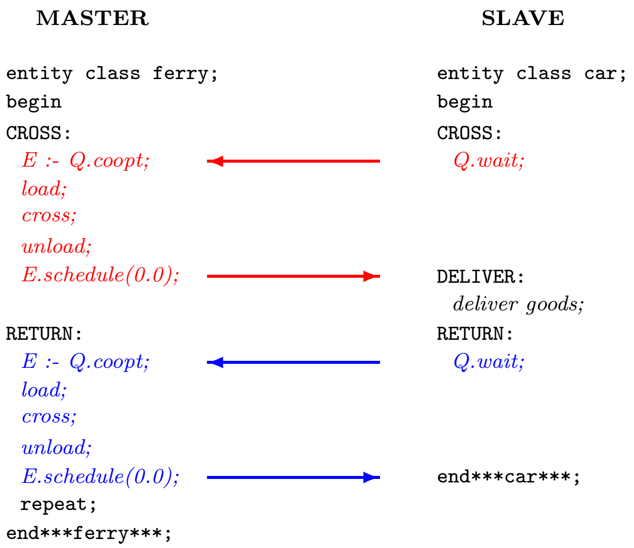


```
entity class ferry;
begin ref(entity)E;
CROSS:
  await car;
  load;
  cross;
  unload;

RETURN:
  await car;
  load;
  cross;
  unload;
 repeat;
end***ferry***;
```

```
entity class car;
begin
CROSS:
  await ferry;
  load;
  cross;
  unload;
DELIVER:
  deliver goods;
RETURN:
  await ferry;
  load;
  cross;
  unload;
end***car***;
```

Figure 5.1: Entity-entity synchronisation

Representing two entities doing the same thing at the same time by writing code which has both of them moving down the event list is rather prone to error. Instead, we arrange for one of the entities to dominate and let it treat the other as an item to be coopted, retained as a passive slave throughout the period of cooperation, and then be scheduled for independent progress at the end of this period of cooperation. In figure 5.2 we have arbitrarily made the ferry the master and the cars slaves.



```
entity class ferry;
begin
CROSS:
  E :- Q.coopt;
  load;
  cross;
  unload;
  E.schedule(0.0);

RETURN:
  E :- Q.coopt;
  load;
  cross;
  unload;
  E.schedule(0.0);
 repeat;
end***ferry***;
```

```
entity class car;
begin
CROSS:
  Q.wait;




DELIVER:
  deliver goods;
RETURN:
  Q.wait;




end***car***;
```

Figure 5.2: Master/slave descriptions

During the synchronisation, only the master entity appears in the event list, and there will be a corresponding hole in the life history of the slave entity for each such period of cooperation. Thus there are “holes” in the life history of a car for both the task sequence labelled `CROSS` and the task sequence labelled `RETURN`.

5.2 class waitq

A new synchronisation mechanism is supplied — the `waitq`. The synchronisation is interesting in that not only are slave entities blocked until after the common work period but master entities are also blocked if no slaves are available at the time of the request. Thus each `waitq` object has two local queues: a `masterq` which hold blocked masters and a `slaveq` which holds blocked slaves.

A `waitq` (see figure 5.3) has the following attributes:

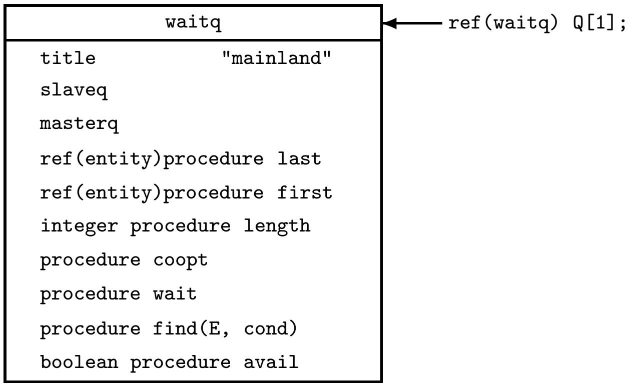


Figure 5.3: Result of Q[1] :- new waitq(”mainland”);

- two subsidiary queues — `slaveq` holds potential victims (here, cars), `masterq` holds potential masters when the slaveq is empty. Both queues contain entities which are ordered according to their `priority`.
- `length` returns the current length of the queue of slaves
- `first` and `last` return references to the first and last slave entities respectively. If the value of `length` is zero, `first` and `last` return `none`;
  if the value of `length` is 1, then they return the same value.

  `first`, `last` and `length` are short for `slaveq.first`, `slaveq.last`, and `slaveq.length` respectively. There is no short access to the the corresponding attributes of the `masterq`: they are available as `masterq.first`, `masterq.last`, and `masterq.length` respectively.

- `coopt` and `wait` are reviewed at length below. See also Appendix B.
- `find` is a synchronisation is which the master is blocked until a slave with specific characteristics is available. It is dealt with fully in section 5.4.
- `avail` is the means of checking the `slaveq` for slaves with specific characteristics. Its usage is deferred until chapter 6.

Structure of the model

In our model, we synchronise affairs by means of two `waitq` objects, `Q[1]` representing the mainland car queue and `Q[2]` the car queue on the island. We depict `waitq`'s in our activity diagrams by solid black boxes. Figure 5.4 shows that part of the activity diagram concerned with the ferry loading a car on the mainland side, crossing the strait, and then unloading the car on the island side.

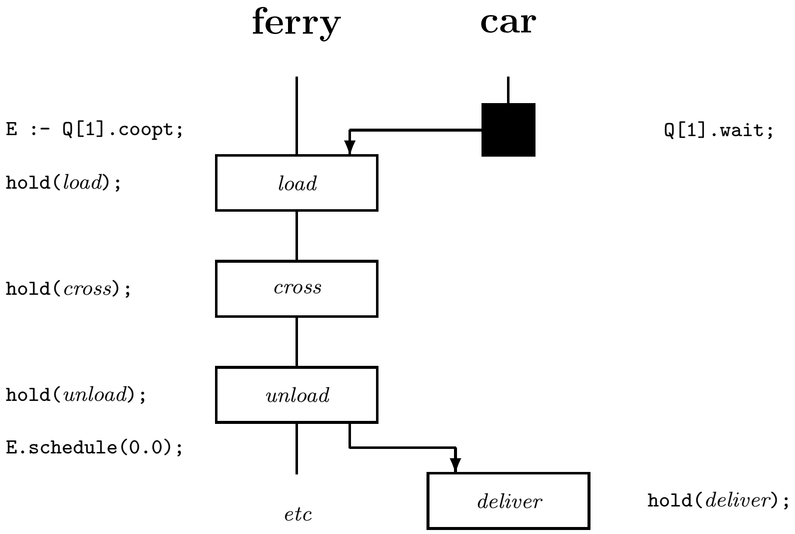


Figure 5.4: Mainland ferry/car synchronisation

coopt

The ferry awaits on the mainland side by calling `E :- Q[1].coopt`. There are two cases to consider:

1. the `slaveq` of `Q[1]` is not empty.
   The call on `Q[1].coopt` removes the first entity from `Q[1].slaveq` and returns a reference to it in `E`. The master entity is not delayed.

2. the `slaveq` of `Q[1]` is empty.
   The would-be coopter (the ferry) is blocked in `Q[1].masterq` until such a time as a slave enters `Q[1].slaveq`. When this happens, the ferry is awakened and re-enters the event list, removing as it does so the slave entity, and naming it `E`. Notice that repeated calls on `coopt` do not return the same victim, since each completed call on `coopt` removes some slave object from the `slaveq`.

wait

A car signals its readiness for this crossing by a call `Q[1].wait` which puts it to sleep (out of the event list) in `Q[1].slaveq`. Again there are two cases to consider:

1. the `masterq` of `Q[1]` is empty.
   The car waits passively (out of the event list) in `Q[1].slaveq`.

2. the `masterq` of `Q[1]` is not empty.
   The car leaves the event list, enters `Q[1].slaveq`, and awakens the first entity in `Q[1].masterq` (which will now complete its `coopt` action).

Reawakening a slave

The car `E` remains passive during the ferry's subsequent

`hold(`*load*`); hold(`*crossing*`); hold(`*unload*`)`

actions. The ferry's next action, `E.schedule(0.0)`, causes `E` to be placed in the event list, at the current clock time, but as the last entity scheduled for that time. Notice that the simulation clock time is where it should be for both the car and the ferry. Thus as seen by the car, it will then have crossed the strait, and its actions are re-entered at the action `hold(`*deliver*`)`. Thus the ferry will now start a fresh *load* and the car is about to start delivering goods. The synchronisation on the island side is, of course, very similar.

More complete declarations for `car` and `ferry` are:

```
entity class ferry;
begin
 integer side;
 ref(car)L;
 for size := 1, 2 do
 begin
  L :- Q[side].coopt.
  hold(load);
  hold(cross);
  hold(unload);
  L.schedule(0.0);
 end;
 repeat;
end***ferry***;
```

```
entity class car;
begin
 Q[1].wait;

 hold(deliver);

 Q[2].wait;
end***car***;
```

Notice that should `Q[n]` be empty — n = 1 or 2 — the ferry awaits the arrival of a car as the request was made as `Q[n].coopt` and a call on `coopt` implies a delay if no victim is waiting in the `Q[n].slaveq`. Should the ferry be required to leave at once if no cars are waiting, we have to put in a test before calling `coopt`, e.g.

```
if Q[n].length = 0 then hold(crossing) else
begin
  E :- Q[n].coopt;
  hold(load + cross + unload);
  E.schedule(0.0);
end;
```

5.3 class queue

Consider a revamping of the ferry problem in which instead of allowing just one car per crossing, we may accommodate up to six. We now have an instance of the general case of one master and several slaves. Coopting several slaves one by one is no trouble, the question is how can we keep tabs them all. It is most convenient to place them in a `queue` (see figure 5.5).

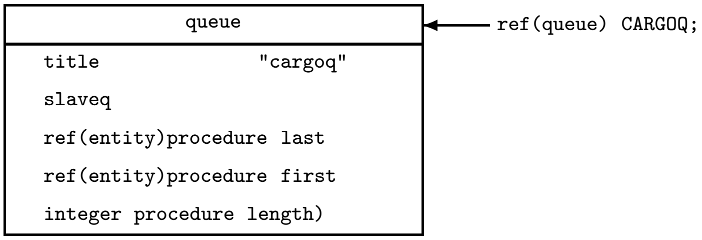

queue
ref(queue) CARGOQ;
title
"cargoq"
slaveq
ref(entity)procedure last
ref(entity)procedure first
integer procedure length)

Figure 5.5: Result of CARGOQ :- new queue(”cargoq”).

Some of the attributes of `class queue` are shared with `class waitq`. As might be expected, `class queue` is used as prefix to `class waitq`.

In this case, we keep the cars in `ref(queue)CARGOQ` declared local to `class ferry`. (Should we have several ferries in an extended model, we would want each ferry to have its own `queue`.) We create the queue by

```
CARGOQ :- new queue("cargoq");
```

Any coopted `car` object `E` can be placed in the queue (in priority order, of course) by `E.into(CARGOQ)` and removed from it and scheduled by `E.out; E.schedule(`*delay*`)`. `into` and `out` are attributes of `class entity`. We let the cars queue for the ferry in `waitq`s `Q[1]` and `Q[2]`. Thus the `car` and `ferry` class outlines are

```
ref(queue) Q [ 1:2 ];

entity class car;
begin
  Q[1].wait;
  hold(tour island);
  Q[2].wait;
end***car***;

entity class ferry;
begin
  integer side;
  ref(queue) CARGOQ;
  ref(entity) E;

  CARGOQ :- new queue("cargoq");
LOOP:
  for side := 1, 2 do
  begin
    while CARGOQ.length < 6 and Q[side].length > 0 do
    begin
      E :- Q[side].coopt;
      E.into(CARGOQ);
      hold(load);
    end;

    hold(crossing);

    while CARGO.length > 0 do
    begin
      hold(unload);
      E :- CARGO.first;
      E.out;
      E.schedule(0.0);
    end;
  end;
  .....repeat;
end***ferry***;
```

Since we can get the current length of a `queue` object at any time through its attribute `integer procedure length`, we have no need to maintain an explicit count of the current number of cars on board. Should the cars be unloaded according to the last-on-board, first-off rule we would replace the seventh last line above by `E :- CARGOQ.last`.

Example 6: Information system

This problem has been used in several papers and books and so provides an interesting comparison (see for example, Knuth and McNeley [46, 45], Pritsker and Kiviat [85]). The model represents an information retrieval system with a number of remote terminals each capable of interrogating a single processor (`cpu`). A customer with a query arrives at one or other of the terminals. It may be necessary to queue for a terminal — the terminals are far apart physically and no queue jumping is possible. When the terminal is free, the request is keyed in, and its presence signalled to the system. The customer then awaits a reply.

Queries are detected by a scanner which looks at each terminal in turn. If there is no query outstanding, the scanner rotates on to the next terminal. If there is a query, the scanner locks on to that terminal and does not rotate further the query has been forwarded to a buffer unit capable of holding three queries at a time. Copying is blocked if no buffer slot is available. When the copying has been completed, the scanner starts to rotate again leaving a cpu to deal with the query.

The `cpu` processes the query and places the answer in the same buffer slot (overwriting the query). The answer is returned to the terminal by the buffer unit (without using the scanner) and then that buffer slot is freed. The customer reads the reply and then releases the terminal. Conveniently, the distribution given for the processing time `ref(rdist)process` takes account of how the requests share the cpu (it is not our data) and we can abstract away the need for a cpu. That is instead of the (perhaps) expected

```
[ ref(res}cpu; ]

cpu.acquire(1);
hold(process.sample);
cpu.release(1);
```

we need not model the cpu at all and our description of this phase simplifies to

```
hold(process.sample);
```

Structure of the model

By dispensing with a separate class describing customers we may describe the model in terms of just two entity classes.

`class query` describes the history of a query as it is posed by a customer, passed to the buffer, processed, returned to the customer, and scrutinised. Its outline is:

keyin; transfer; process; reply; read;

`class scanner` describes the actions of the scanner as it rotates from terminal to terminal. If the current terminal has no request pending, the scanner moves on. If a query is detected, it locks on and does not rotate on until the query has been transferred to the buffer. Its outline is:

rotate; `if` *request* `then` *transfer*; `repeat`;

What remains for us to settle is which process is to be the master and which the slave in for the *transfer* task. We have arbitrarily chosen to make the `query` the master and use `ref(waitq)RQ[k]` to synchronise a query at terminal `k` with the `scanner`. The class outlines become:

```
entity class query;                   entity class scanner;
begin                                 begin
  keyin;                                rotate;
  S :- RQ[k].coopt;         ▬▬▬         if request then RQ[k].wait;
  transfer;
  S.schedule(0.0);          ▬▬▬         repeat;
  process;                            end***scanner***;
  reply;
  read;
end***query***;
```

Now that we have outlined the roles to be played by scanner and query objects and decided upon their interactions, we can tackle their declarations separately once given the model data.

Model data

Timings in minutes:	
inter-customer arrivals	`negexp`:mean=5/min
keyin a query	`uniform`:0.3→0.5
transfer query to buffer	`constant`:0.0117
process a query on CPU	`uniform`:0.05→0.10
transfer reply back	`constant`:0.0397
scanner rotation	`constant`:0.0027
scanner to test a terminal	`constant`:0.0027
customer to read reply	`uniform`:0.6→0.8
Resources:	
buffers	`res`:initially 3
TERM [1:6]	`res`:limit=1,terminals
RQ [1:6]	`waitq`:scanner requests

Assuming no blocking, the expected average terminal occupation by a customer will be the mean time for each task plus the time average time for the scanner to rotate to the terminal, i.e.

$$0.400 + 0.0117 + 0.0750 + 0.0397 + 0.7000 + (0.0027{+}0.0027)\times 3$$
$$= 1.2426 \text{ minutes}$$

As we expect to average 50 users per terminal per hour (1.2 minutes between arrivals) this shows that the original system design is inadequate and we must expect queues to build up—as indeed they do. Such rough and ready analyses should always be performed on simulation models to give an idea of through times etc. (or at least rough bounds for them) for they pin point expected bottlenecks and, as in this case, may even obviate the need to run this configuration of the model.

class query

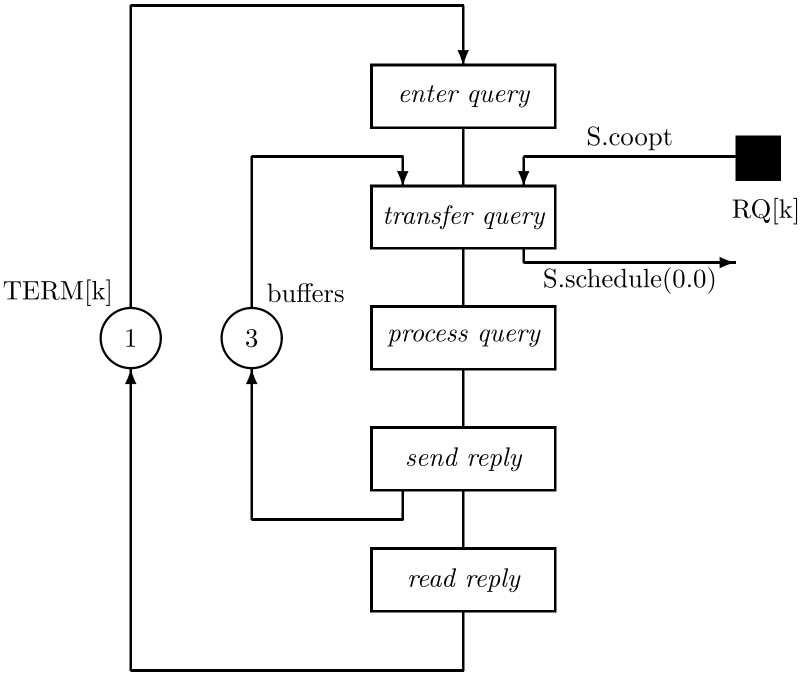


Figure 5.6: Activity diagram: class query

In our formulation, a query object:

1. first generates the next query object, notes its arrival time (`t := time`), and then chooses its terminal (`k`). That terminal is then seized by the

action TERM[k].acquire(1), perhaps after a wait. The query is entered (hold(keyin.sample)) and the query waits for the scanner to turn up by requesting S :- RQ[k].coopt.

2. the scanner locks on to that query by RQ[k].wait. Then a buffer is acquired by buffers.acquire(1) (which may imply a delay), and the query is transferred to the buffer (hold(0.0117)). After the transfer has been completed, the scanner freed to rotate on by S.schedule(0.0).
3. the query is processed by hold(process.sample)
4. after processing the reply is returned to the appropriate terminal by hold(transfer.sample) and the buffer slot freed by buffers.release(1).
5. the reply is read (hold(read.sample)), and then the terminal is vacated (TERM[k].release(1)), which allows in the next query, if any. Finally, a histogram of through times (THRU) is updated by the elapsed time of this query through the system.

Here is the complete Demos code for class query:

```
entity class query;
begin
  integer k;
  real t;
  ref(scanner) S;

  new query("query").schedule(arrivals.sample);
  t := time;
  k := terminals.sample;
  TERM[k].acquire(1);
    hold(keyin.sample);

    S :- RQ[k].coopt;
    buffers.acquire(1);
      hold(0.0117);
      S.schedule(0.0);
      hold(process.sample);
      hold(0.0397);
    buffers.release(1);

    hold(read.sample);
  TERM[k].release(1);
  THRU.update(time-t);
end***query***;
```

class scanner

We now turn our attention to the slave entity, the scanner. The scanner rotates from terminal 1 to terminal 6, and then repeats. These actions are captured in figure 5.7

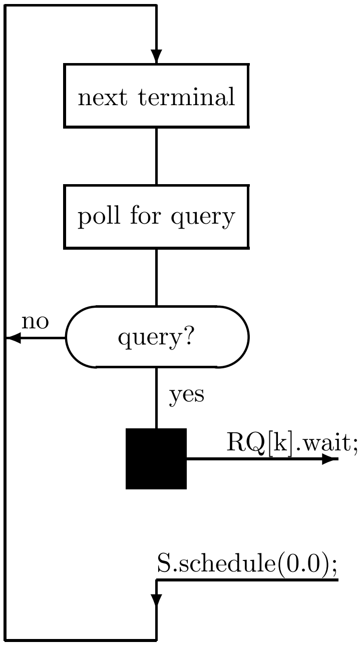


Figure 5.7: Activity diagram: class scanner

1. `hold(0.0027)` captures the rotatation to the next terminal `k`
2. a test (`b := RQ[k].masterq.length` > `0`) to see if a query is pending takes a further `hold(0.0027)`. ASIDE: `RQ[k].masterq.length` (or `RQ[k].slaveq.length`) is a test on the length of the slave queue of `RQ`.
3. if a request is pending, the scanner locks on by `RQ[k].wait`. When awakened the query will be safely in the buffer.

The full declaration of `class scanner` is:

```
entity class scanner;
begin
  integer k;  boolean b;

  for k := 1 step 1 until 6 do
  begin
    hold(0.0027);
    b := RQ[k].masterq.length > 0;
    hold(0.0027);
    if b then RQ[k].wait;
  end;
  repeat;
end***scanner***;
```

The driving program

The driving program contains these definitions plus the various resources, queues and distributions and one histogram. It generates the scanner and the first query and runs the model for 60 simulated minutes.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
begin
  ref(waitq)array RQ [ 1:6 ];
  ref(res)array TERM [ 1:6 ];
  ref(res) buffers;
  ref(rdist) arrivals, keyin, process, read;
  ref(idist) terminals;
  ref(histogram) THRU;
  integer k;

  entity class query........;
  entity class scanner......;

  arrivals  :- new negexp("arr", 5.0);
  terminals :- new randint("terminals", 1, 6);
  keyin     :- new uniform("keyin", 0.3, 0.5);
  process   :- new uniform("process", 0.05, 0.10);
  read      :- new uniform("read", 0.6, 0.8);
  THRU      :- new histogram("thru", 1.0, 11.0, 10);
  for k := 1 step 1 until 6 do
  begin
    RQ[k]   :- new waitq(edit("request",k));
    TERM[k] :- new res(edit("terminal", k), 1);
  end;

  buffers :- new res("buffers", 3);
  new scanner("scanner").schedule(0.0);
  new query("q").schedule(0.0);
  hold(60.0);
end;
```

OUTPUT

```
                       clock time =     60.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /         a/         b/      seed
arr               0.000    300 negexp        5.000             33427485
terminals         0.000    300 randint           1         6  22276755
keyin             0.000    259 uniform       0.300     0.500  46847980
process           0.000    257 uniform  5.000&-002     0.100  43859043
read              0.000    257 uniform       0.600     0.800  64042082


                         h i s t o g r a m s
                         *******************

                            s u m m a r y

title       /   (re)set/   obs/  average/est.st.dv/  minimum/  maximum
thru              0.000    253     4.459     3.469     1.131    18.529

cell/lower lim/    n/   freq/  cum :
                                       I------------------------------
   0 -infinity     0    0.00    0.00   I
   1     1.000    72    0.28   28.46   I******************************
   2     2.000    44    0.17   45.85   I******************
   3     3.000    30    0.12   57.71   I*************
   4     4.000    24    0.09   67.19   I**********
   5     5.000    20    0.08   75.10   I********
   6     6.000    15    0.06   81.03   I******
   7     7.000    11    0.04   85.38   I*****
   8     8.000     7    0.03   88.14   I***
   9     9.000    11    0.04   92.49   I*****
  10    10.000     3    0.01   93.68   I*
  11    11.000    16    0.06  100.00   I*******
                                       I------------------------------


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
terminal 1        0.000     40    1    0    0    83.553     4.345   15
terminal 2        0.000     46    1    0    0    96.128     6.928   18
terminal 3        0.000     45    1    0    0    94.196     1.828    4
terminal 4        0.000     42    1    0    0    88.847     2.135    6
terminal 5        0.000     37    1    0    0    78.770     0.647    3
terminal 6        0.000     43    1    0    0    89.603     3.315    7
buffers           0.000    257    3    0    3    18.084     0.000    1


                        w a i t   q u e u e s
                        *********************
```

```
title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
request 1         0.000     41     1     0 1.205&-002     0 1.763&-002
request 1   *     0.000     41     1     0      0.000     41      0.000

request 2         0.000     46     1     0 1.674&-002     0 2.183&-002
request 2   *     0.000     46     1     0      0.000     46      0.000

request 3         0.000     45     1     0 1.642&-002     0 2.190&-002
request 3   *     0.000     45     1     0      0.000     45      0.000

request 4         0.000     43     1     0 1.293&-002     0 1.804&-002
request 4   *     0.000     43     1     0      0.000     43      0.000

request 5         0.000     38     1     0 1.338&-002     0 2.113&-002
request 5   *     0.000     38     1     0      0.000     38      0.000

request 6         0.000     44     1     0 1.291&-002     0 1.760&-002
request 6   *     0.000     44     1     0      0.000     44      0.000
```

Remarks on Example 6

The report on each wait queue details the delays caused to the masters wishing to coopt victims (line 1) and to the victims (in the starred line, line 2). In line 1 (referring to the `masterq`:

1. `title` and `(re)set` are obvious
2. `obs` gives the number of completed `wait/coopt` handshakes for the `waitq`
3. `qmax` gives the maximum length of the masterq (which includes zero waits)
4. `qnow` gives the current length of the masterq
5. `q average` gives the time weighted average length of the masterq
6. `zeros` gives the number of zero waits (instant coopts) in the masterq
7. `av.  wait` gives the average wait time of each master including zero waits

In the same manner, the second line reports on the way slaves are delayed in the `slaveq`. `OBS` must be the same for both lines.

As expected, the scanner is never delayed in a `RQ` (see line 1 of the `waitq` reports) as it makes sure a query is ready before calling `wait`.

We have used a histogram — `THRU` — to collect and display the elapsed through times for each query. Having entered the system, each query makes a local note of the current clock time by `t := time`. The last action of each query object is to update `THRU` by a call `THRU.update(time - t)`. As can be seen from the report on `THRU`, a summary of the update readings is printed followed by the histogram itself. Each histogram object requires 4 parameters:

1. a `text title`
2. a `real` lower bound for the update values
3. a `real` upper bound for the update values, and
4. an `integer` giving the number of recording cells.

Each cell has the same width = (upper bound – lower bound)/number of cells. Thus

```
THRU :- new histogram("thru", 1.0, 11.0, 10);
```

establishes a histogram entitled `THRU` with 10 cells for recording values in the ranges

[1.0→2.0), [2.0→3.0), ..., [10.0→11.0)

There are also two extra cells for recording *underflow* (here updates less than 1.0) and *overflow* (here 11.0 or greater). In this case, the underflow cell has no entries recorded and the overflow cell 15 entries. The summary records the minimum (here 1.080) and the maximum (here 17.292) through times and these could be used in later runs to reset the histogram bounds should this be desired.

Example 7: Aluminium plant

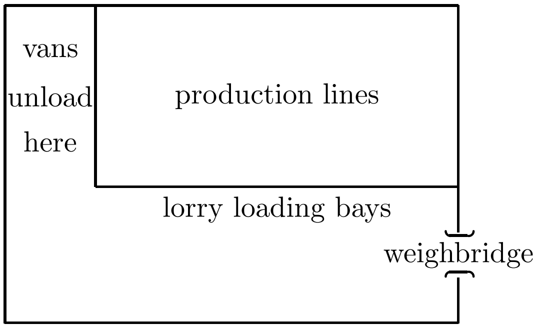


Figure 5.8: Factory layout

A factory has one entrance guarded by a weighbridge over which all incoming and outgoing vehicles must pass. Only one vehicle can move in this area at a time. Aluminium sheets are delivered to the factory in vans and loaded into hoppers. The hoppers are fed onto production lines, their contents formed into cans, filled, capped and then placed in containers. The containers are removed by lorries. Van and lorry movements within the plant are of comparatively short duration and are omitted from the model.

Model data

Timings in minutes:

van	
inter-arrival	98 + `negexp`:mean=0.1
at weighbridge	`constant`:2
fill a hopper	`normal`:mean= 5.0, st.dev.=1.0
lorry	
inter-arrival	`negexp`:mean=0.1/min
at weighbridge	`constant`:3
production-line	
fill a hopper	`constant`:25

Resources:

weighbridge	`res`:limit=1
vanspaces	`res`:limit=4
crane	`res`:limit=1
bays	`res`:limit=6
full hoppers	`bin`:initially 3
empty hoppers	`bin`:initially 5

Structure of the model

We model the plant with three classes of entity — `van`, `production`, and `lorry` with life cycles:

```
van         =   enter; unload; leave; reload; repeat

lorry       =   enter; load; leave

production  =   fst batch; snd batch; repeat
```

Since there is interplay amongst all three entities, we present the process interactions in two gentle iterations.

The vans and lorries interact only at the plant entrance where they compete for the weighbridge on the way in and on the way out. We ignore any time spent by van and lorry movements within the plant. As shown below, this interaction is modelled by making the weighbridge a resource of size 1.

```
van     =   w'bridge.acquire(1); enter; w'bridge.release(1);
            unload;
            w'bridge.acquire(1); leave; w'bridge.release(1);
            reload;
            repeat

lorry   =   w'bridge.acquire(1); enter; w'bridge.release(1);
            load;
            w'bridge.acquire(1); leave; w'bridge.release(1);
```

Van/production line interplay is also straightforward. Van require empty hoppers to unload (each van fills three hoppers) full hoppers are passed to the production lines. The production line empties a full hopper per batch and returns it back to the van unloading area. We assume instantaneous hopper movements between vans and production lines. Below we show *just* this interaction which is achieved through two `bins`, `emptyhopper` and `fullhoppers`.

```
van          =   enter;
                 emptyhoppers.take(3); unload; fullhoppers.give(3);
                 leave; reload;
                 repeat

production   =   emptyhoppers.take(1); fst batch; fullhoppers.give(1);
                 emptyhoppers.take(1); snd batch; fullhoppers.give(1);
                 repeat
```

The production line/lorry interplay is the hardest to understand. Once started each batch takes 25 minutes to complete, but it takes 10 minutes before the first crate emerges. There is no storage space at the end of a production line — crates are put straight onto lorries. Thus production is halted unless a lorry is there. Once a lorry has been "claimed" it is filled with two hoppers worth of crates. We use the master/slave synchronisation with production lines as masters and lorries as slaves, cooperating through `ref(waitq)BAYQ`. Here is a sketch of *just* this interaction:

```
lorry        =   enter; BAYQ.wait; leave

production   =   hold(10.0); L :- BAYQ.coopt; hold(15.0);
                 hold(25.0); L.schedule(10.0);
                 repeat
```

Once started, each batch takes 25 minutes. We delay the coopting of a lorry until one is necessary, which is 10 minutes later. Once coopted, the lorry is kept as a slave until the second batch has been completed which is 35 minutes after it started (25 minutes to unload a full hopper and 10 minutes for the last crate to clear). If there are free empty hoppers, a production line is permitted to start another run whilst the last one is clearing. Notice that we are neglecting lorry movement times by assuming zero interference as one lorry clears the bays and another moves in.

We are now in position to refine these outlines and we develop the entity descriptions one by one.

class van

The vans arrive at the plant periodically. Once across the weighbridge (which takes two minutes in or out), each van goes to the rear of the factory to an unloading area where its load is removed with the assistance of a crane. The load of aluminium sheets fills three empty hoppers one by one. Full hoppers are then fitted onto the production lines. Each van then leaves, again passing over the weighbridge. To prevent congestion, at most four vans are allowed in the factory grounds at a time.

A pool of seven vans serves the factory. A `res vanspaces` is used to limit the number in the factory grounds to 4 at any one time.

Unloading takes place when the crane and empty hoppers are available. An unloading, which fills three hoppers, may start even if only one or two are free; but the crane is only released by its owning van when three hoppers have been filled.

The filling of each hopper takes about 5 minutes (`normal`, mean = 5, standard deviation = 1).

After exiting, the van returns with a new load in about 108 minutes (98 + `negexp`(0.1)).

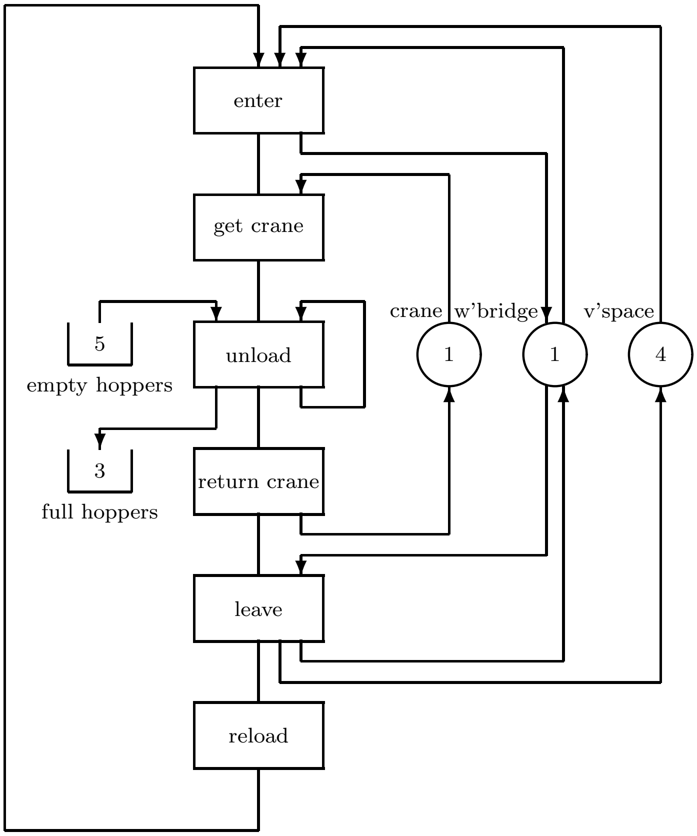


Figure 5.9: Van activity diagram

```
entity class van;
begin
  integer k;

  vanspaces.acquire(1);
  weighbridge.acquire(1);
  hold(2.0);
  weighbridge.release(1);

  crane.acquire(1);
  for k := 1 step 1 until 3 do
  begin
    emptyhoppers.take(1);
    hold(fill.sample);
    fullhoppers.give(1);
  end;
  crane.release(1);

  weighbridge.acquire(1);
  hold(2.0);
```

```
    weighbridge.release(1);
    vanspaces.release(1);

    hold(98.0 + nexttrip.sample);
    repeat;
  end***van***;
```

class lorry

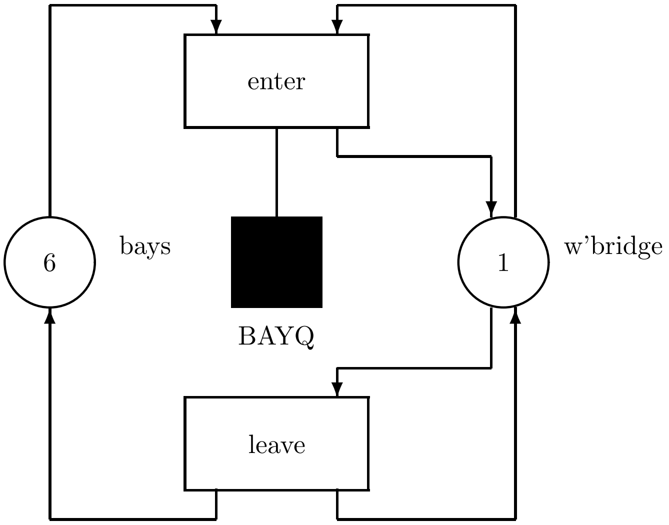


Figure 5.10: Lorry activity diagram

A full hopper fits onto a production line (of which there are five). The aluminium sheets are removed from the hopper and processed one by one. As the sheets pass down the line, they are formed into cans, filled with liquid XXXXX and capped. It takes two hoppers to fill one container. If all goes smoothly, the processing time per hopper is 25 minutes.

The containers are loaded onto articulated trucks. The trucks wait outside the factory until a loading bay is free. They take three minutes to cross the weighbridge (in and out) and then manoeuvre into a loading bay. When the lorry is loaded, it departs via the weighbridge.

Lorries arrive roughly every 10 minutes (`negexp`(1/10)). They enter the factory grounds when they have a bay (there are 6 bays in the model) and the weighbridge. Once in, they accept two containers and then leave.

```
entity class lorry;
```

```
begin
  new lorry("lorry").schedule(nextlorry.sample);

  bays.acquire(1);
  weighbridge.acquire(1);
  hold(3.0);
  weighbridge.release(1);

  BAYQ.wait;

  weighbridge.acquire(1);
  hold(3.0);
  weighbridge.release(1);
  bays.release(1);
end***lorry***;
```

class production

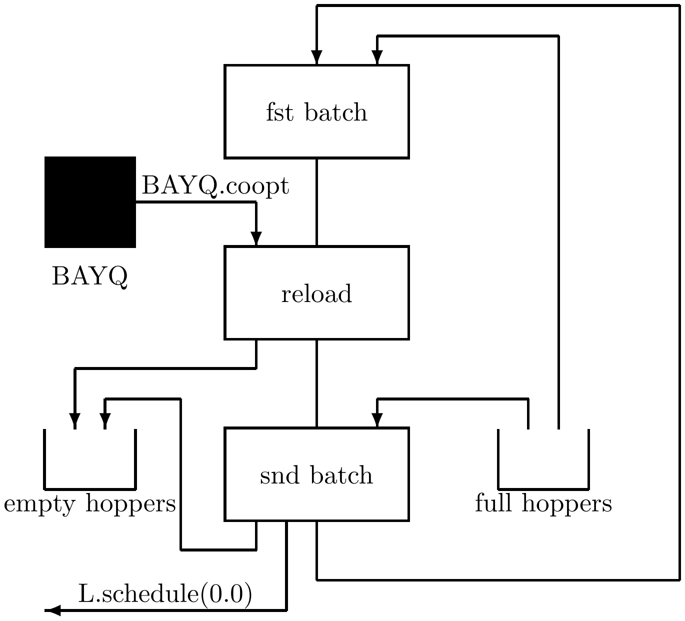


Figure 5.11: Production line activity diagram

When a hopper is put on the line, the plant starts producing cans. The first can is ready ten minutes later. If there is no waiting container, production is halted but can continue without penalty when one arrives. After a further fifteen minutes, the hopper has to be replaced with another possible production line halt. Twenty five minutes later the second hopper will have been emptied, but the last can will not arrive at the end of the production line until another five minutes have elapsed.

A second container load can be started on the production line immediately after the first if required, there being no need to wait for the final can of the first load to be ready before the first can of the second can be started.

```
entity class production;
begin
  ref(lorry) L;

  fullhoppers.take(1);
  hold(10.0);

  L :- BAYQ.coopt;
  hold(15.0);
  emptyhoppers.give(1);

  fullhoppers.take(1);
  hold(25.0);
  emptyhoppers.give(1);
  L.schedule(10.0);
  repeat;
end***production line***;
```

The driving program

The driving program includes distribution, resource and entity declarations and initialisations. Initially there are five empty hoppers and three full hoppers. The first lorry arrives at time zero; each lorry schedules the next. A seven-strong van fleet is established, weakly spaced in time, and five production lines are ready for action. The model is run for 8 hours with a cold start and an abrupt end.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(rdist) nextlorry, fill, nexttrip;
    ref(res) weighbridge, crane, bays, vanspaces;
    ref(bin) fullhoppers, emptyhoppers;
    ref(waitq) BAYQ;

    entity class van............;
    entity class lorry..........;
    entity class production.....;

    integer k;


    nextlorry    :- new negexp("next lorry", 0.1);
    fill         :- new normal("fill hopper", 5.0, 1.0);
    nexttrip     :- new negexp("van return", 0.1);

    weighbridge  :- new res("weighbridge", 1);
    vanspaces    :- new res("van spaces", 4);
    crane        :- new res("crane", 1);
    bays         :- new res("bays", 6);
    fullhoppers  :- new bin("full hoppers",  3);
    emptyhoppers :- new bin("empty hoppers", 5);
    BAYQ         :- new waitq("await container");

    new lorry("l").schedule(0.0);

    for k := 1 step 1 until 7 do
      new van("v").schedule((k-1)*14.0);

    for k := 1 step 1 until 5 do
      new production("P-line").schedule(0.0);

    hold(480.0);
  end;
```

OUTPUT

```
                       clock time =    480.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
next lorry        0.000     41 negexp        0.100            33427485
fill hopper       0.000     74 normal        5.000     1.000  22276755
van return        0.000     24 negexp        0.100            46847980


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
weighbridge       0.000    121    1    0    1    65.417     1.168    4
van spaces        0.000     24    4    0    3    40.525     0.000    1
crane             0.000     24    1    0    0    81.393     8.943    2
bays              0.000     33    6    0    0    88.570    12.900    4


                               b i n s
                               *******

title       /   (re)set/   obs/init/ max/ now/ av. free/ av. wait/qmax
full hoppers      0.000     73    3    3    0     0.516     5.477    5
empty hopper      0.000     71    5    8    2     2.571     0.219    1


                        w a i t   q u e u e s
                        *********************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
await contai      0.000     38     4     0      0.315    27      3.977
await contai*     0.000     38     3     1      0.578    12      6.743
```

5.4 Find

As yet a master entity attempting to locate a victim from a `waitq` by a call on `coopt` is always allotted the first available entity without discrimination. Sometimes we would like the master entity to be able to select and coopt a victim with specific characteristics. For example, we may wish to select a car whose external dimensions fit the space left on the ferry. To do this, we need a more subtle routine than `coopt`.

This is provided by the routine `find`, also local to `class waitq` (see figure 5.3, section 5.2). `find` parallels `coopt` in that it locates a suitable victim and blocks the caller if need be until one is located. `find` has the heading

```
procedure find(E, c); name E, c; ref(entity) E; boolean c;
```

where the arguments are a reference variable `E` and a condition (boolean expression) `C` which usually involves `E`. This combination enables arbitrarily complicated choices to be expressed, a point which we now illustrate by an example. Note that both `E` and `c` are called by `name` and are dynamically evaluated each time they are referenced within the body of `find`, a trick known as Jensen's device (if this is unknown to you, seek out Knuth and Merners' wonderful account in section 9 of [47]).

We illustrate the use of `find` informally with a small example. Suppose we are modelling scheduling policies in a computer system model. Jobs are graded into high and low priorities. The job of the scheduler is to load external jobs into main memory. Jobs are not taken FCFS but according to a more complicated formula which favours high priority jobs over low priority jobs. The scheduler also makes sure that the next job selected will "fit" into main memory and within a priority group favours small jobs over large jobs. In addition, the scheduler ensures that at most `N` jobs are loaded at any given time.

We model the scheduler as a master entity and programs as slaves. Jobs that are ready to run wait in the `READYQ`. Once selected, they are loaded into main memory and then wait in the `CPUQ` for service. Various times are recorded for statistical analysis (time of entry into the model, time when loaded into main memory, completion time). Jobs are entered into `READYQ` in priority order, with high priority jobs at the front. The number of free memory slots is maintained in `ref(bin) slots`.

```
ref(waitq) READYQ;
ref(rdist)load;
ref(bin) slots;

slots :- new bin(memory slots, N);

entity class job(jobP, size); integer jobP, size;
begin
  real entryT, startT, exitT;

  priority := jobP;
  entryT   := time;
  READYQ.wait;

  startT := time;
  hold(load.sample);
  CPUQ.wait;

  exitT := time;
  slots.give(1);
end***job***;
```

It is up to the scheduler to select the next job for loading. This time it is not a matter of taking the first job in `READYQ`, but of finding the first job, if any, that will fit, but favouring high priority jobs. Instead of using `coopt` we use `find`:

```
entity class scheduler;
begin
  ref(job)J;

  slots.take(1);
  READYQ.find(J, J.size <= memFree);
  memFree := memFree - J.size;
  J.schedule(0.0);
  repeat;
end***scheduler***;
```

Jobs are naturally queued according to their priority and arrival times, and the `slaveq` is searched from first to last. The scheduler will only exit from the `READYQ.masterq` of when a suitable `job` has been located. Then it decrements free memory and schedules the job `J` to load itself into main memory.

Mechanics of find

A call on `find`, say

```
Q.find(E, condition);                        [ref(waitq) Q;]
```

operates as follows. First the `slaveq` of `Q` is inspected.

1. `Q.slaveq` is empty.
   The caller (`current`) is put to sleep in the `masterq` of `Q`. It will be rewawkened and test again each time a new slave entity enters `Q.slaveq`.

2. `Q.slaveq` is not empty.
   `V` is set to reference the entities in it in turn and the `condition` is tested against each. If an `E` is found for which the condition holds, then that `E` is extracted from `Q.slaveq`, coopted by the caller and control remains with the caller (`current`).

Should no victim be found, the caller is put to sleep in `Q.masterq` in order of its priority. Each new slave arriving via a call on `Q.wait` awakens the masters in turn who test to see if the newcomer satisfies their condition. The first master with a true `find` condition seizes the new slave and becomes unblocked. If no such master can be found, the new slave waits in the `Q.slaveq` in its priority order. Notice that `E :- Q.coopt` is equivalent to `Q.find(E, true)`.

Example 8: Tanker simulation

Tankers arrive periodically at a harbour and discharge their cargo into shore tanks. When a shore tank is full, or nearly so, its contents are automatically transferred to the refinery. While this transfer is taking place, a shore tank may not be filled by a tanker.

Model data

Timings in hours:	
Tanker arrival rate	`negexp`:0.125/hour
Setup time for pump	`constant`:0.5 hours
Pumping rate	`constant`:1000 tons/hour
Discharge rate	`constant`:4000 tons/hour
Capacities in 1000 ton units:	
Tanker loads	15,20,25 equally likely
Shore tank volume	70

Run the simulation for 1000 continuous hours with 5 shore tanks. Take as initial conditions that two shore tanks are empty and free, one is currently discharging and will be free at 8 hours, and that the other two are currently being loaded and will be freed at times 12 (with 45 units still free) and 3.5 (with 25 units free) respectively. The first tanker arrives at time 0.0.

Structure of the model

We work with 1000 tons as the basic capacity unit.

We split the description of the model into two components requiring entity declarations — `class tanker` and `class shoretank` with life cycles:

```
tanker      =   load shore tank;

shoretank   =   while not full do load ; discharge; repeat
```

We have to synchronize the activity `load`. Since we have cooperation between entities, we use the master/slave synchronisation and let `tanker`s be the masters with `TANKQ` as the named `waitq`. Our outline unfolds to:

```
tanker      =   TANKQ.find(ST, ...); load shore tank; ST.schedule(0.0);

shoretank   =   while not full do
                begin
                  TANKQ.wait; load;
                end;
                discharge;
                repeat
```

class tanker

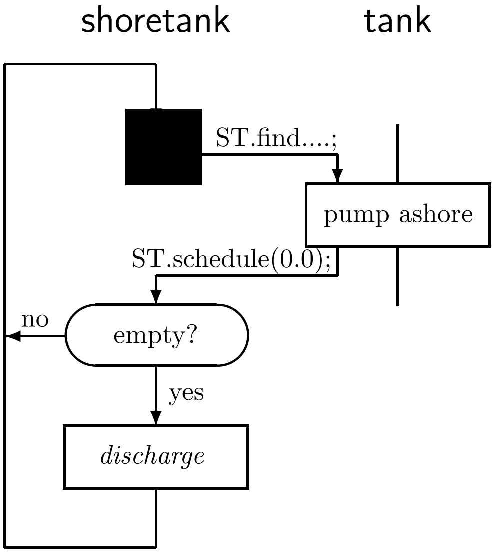


Figure 5.12: Tanker simulation activity diagram

Tankers arrive, find a suitable shore tank (one with enough capacity left to take their load), pump their load into that shore tank and then depart.

```
entity class tanker;
begin
  ref(shoretank) ST;
  integer load;

  load := ...;
  TANKQ.find(ST, ST.free >= load);
  hold(pumping time);
  ST.schedule(0.0);
end***tanker***;
```

class shoretank

Each shore tank waits passively in `TANKQ` until selected by a tanker, and is then its slave whilst being loaded. After each loading, a shore tank decides for itself what to do next. If it has too little capacity left (taken fairly arbitrarily as less than 20,000 tons) either to take another load or to make it worthwhile waiting for another load, it discharges its contents into the refinery. Then it returns empty to `TANKQ`. If it has sufficient capacity left for another load, it returns directly to `TANKQ`.

```
entity class shoretank;
begin
  ..........
  LOOP:
  while room for another load do
    TANKQ.WAIT;
  DISCHARGE:
  hold(time to empty);
  repeat;
end***shoretank***;
```

The complete program

The complete program fills out the entity sketches above with “suitable” data. By using `priority`, we let the shore tanks wait in `TANKQ` ordered according to their capacity remaining — least capacity at the front. The priority is recomputed after each interaction with a tanker.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(waitq) TANKQ;
    ref(rdist) arr; ref(idist) size;
    real pumprate, drate, setuptime;

    entity class tanker;
    begin
      ref(shoretank)ST;
      integer load;

      new tanker("t").schedule(arr.sample);
      load := 5*size.sample;
      TANKQ.find(ST, ST.free >= load);
      hold(setuptime + load*pumprate);
      ST.free     := ST.free-load;
      ST.priority := -ST.free;
      ST.schedule(0.0);
    end***tanker***;

    entity class shoretank(free); integer free;
    begin
      integer max;
      max := 70;
    LOOP:
      priority := -free;
      while free >= 20 do
      begin
        TANKQ.WAIT;
        priority := free;
      end;
    DISCHARGE:
      hold((max-free)*drate);
      free := max;
      repeat;
    end***shoretank***;

    trace;
    setuptime := 0.5;
    pumprate  := 1.0;
    drate     := 0.25;
    arr       :- new negexp("arrivals", 0.125);
    size      :- new randint("load", 3, 5);
    TANKQ     :- new waitq("shoretanks");
    new shoretank("s", 70).schedule(0.0);
    new shoretank("s", 70).schedule(0.0);
    new shoretank("s", 45).schedule(12.0);
    new shoretank("s", 25).schedule(3.5);
    new shoretank("s", 70).schedule(8.0);
    new tanker("t").schedule(0.0);
    hold(1000.0);
  end;
```

OUTPUT: Partial trace and full report.

We record a section of the trace recording a wait, a find, and a schedule.

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                 *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

    ..............................................
    21.259 t 2          schedules t 3 at 26.493
                        coopts s 4 from shoretanks
                        finds s 4 in shoretanks
                        holds for 15.500, until 36.759
    25.500 t 1          schedules s 1 now
                        ***terminates
```

```
           s 1          waits in shoretanks
    26.493 t 3          schedules t 4 at 33.469
                        coopts s 1 from shoretanks
                        finds s 1 in shoretanks
                        holds for 25.500, until 51.993
    33.469 t 4          schedules t 5 at 45.479
                        coopts s 3 from shoretanks
                        finds s 3 in shoretanks
                        holds for 15.500, until 48.969
    36.759 t 2          schedules s 4 now
                        ***terminates
           s 4          holds for 15.000, until 51.759
    45.479 t 5          schedules t 6 at 49.491
                        coopts s 2 from shoretanks
                        finds s 2 in shoretanks
                        holds for 20.500, until 65.979
    48.969 t 4          schedules s 3 now
                        ***terminates
           s 3          waits in shoretanks
    49.491 t 6          schedules t 7 at 58.747
                        coopts s 3 from shoretanks
                        finds s 3 in shoretanks
                        holds for 20.500, until 69.991
    51.759 s 4          waits in shoretanks
    .......................................

                       clock time =   1000.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
arrivals          0.000    128 negexp        0.125            33427485
load              0.000    128 randint           3         5  22276755


                        w a i t   q u e u e s
                        *********************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
shoretanks        0.000    128     5     0      0.210    97      1.638
shoretanks  *     0.000    128     5     2      1.838    32     14.105
```

Remarks on Example 8

There is a little interplay between a tanker and the shore tank after the tanker has discharged its cargo. The tanker sees to the updating of the shore tank's current contents by `ST.free := ST.free-load` before scheduling it (the tanker knows how much has been pumped). The shoretank resets its priority to the new value (`priority := -free`) before re-entering `TANKQ`.

Exercises 5

Exercise 5.1 A library has an archive section containing specialist books. Anyone requesting such a book must first fill out a request slip and then present it to a librarian. The librarian then goes into the archive stacks to locate the book and return with it. The book is then checked out and handed over to the reader. Assume that all requests are found in the stack, and that each reader makes one request at a time. If several readers are waiting, a librarian can pick up several request slips at a time, up to a maximum of five. The librarians are quite democratic and if more than one is free, they divide the work amongst themselves as equally as possible.

Model data

Timings in minutes:	
request rate	`negexp`:mean=0.5/min
time to check request	`constant`:0.1
walk to stack	`uniform`:0.5→1.5
locate n books	`normal`:mean=n,st.dev.=n/5
return from stack	`uniform`:0.5→2.0
check out each book	`constant`:0.5

Assume that there are three librarians and that each can handle up to five requests at a time. Assume that the first request arrives at time 0.0. Run your model for 8 hours.

HINT: If X is a sample from a `normal` distribution with mean 0 and standard deviation 1, then `Y = m + sX` is a sample from a `normal` distribution with mean `m` and standard deviation `s`.

Exercise 5.2 Rewrite exercise 5.1 above with the following new strategy for the librarians when collecting slips. The librarians are queued for work on the longest-idle, first-back-to-work principle. When they begin to accept requests, they take as many as they can up to the maximum of five before allowing the next free librarian (if any) to accept any remaining requests.

Exercise 5.3 A steel mill furnace melts a load of steel, and then pours it into batches of moulds. Then the furnace is reloaded and its work cycle repeated.

The molten steel in the moulds is allowed to set (form a solid crust on the outside so that it is self-supporting). Then the moulds are stripped away and the ingots removed. The batch is then loaded into a soaking pit where it is heated until it

Model data

Timings in minutes:	
furnace	
load and smelt	`normal`:mean=165.0,st.dev.=20.0
pour	`constant`:20 per batch
batch of ingots	
set	`constant`:75
load into pit	`constant`:15 per batch
soak	`normal`:mean=160.0,st.dev.=30.0
unload from pit	`constant`:1 per ingot
roll	`constant`:3 per ingot
strippers	
strip batch	`uniform`:10.0→16.0
clean moulds	`uniform`:10.0→12.0
reassemble moulds	`uniform`:10.0→12.0
Resources:	
BOGIES	`bin`:initially 8
CRANES	`res`:limit=3
PITS	`res`:limit=10 batches
MILL	`res`:limit=2

has achieved a certain uniform temperature. Meanwhile, the moulds are cleaned, reassembled for further use, and returned to the furnace area. When a batch has reached the requisite temperature, it is noted as ready for rolling. Rolling turns the ingots into slabs, the end product of the mill. The furnace has a capacity of 300 tons which is enough to fill 2 batches of moulds, one after the other. Each batch of moulds is transported on its own railway bogie (there are always 15 moulds to a batch). After a pouring, each batch of moulds is shunted into a siding to set. After setting, the batch can be moved from the sidings. A team of strippers take the bogie to the soaking pit area where, with the help of a crane (there is one reserved for each team), they remove the moulds and dump the ingots. The ingots await placement in a soaking pit. Meanwhile the strippers clean the moulds, reassemble them and put them back on their bogie. The bogie is then shunted back to the furnace area, and the team of strippers looks for more work. The batch of ingots is loaded into a soaking pit when one becomes free. The loading requires use of one of three overhead cranes. Unloading also requires use of one of these cranes, but in order to maintain their temperature, individual ingots are left in their pit as long as possible. Thus once a crane has been acquired for unloading, it is retained and is used to unload the ingots one at a time at a pace dictated by the rolling mill. The crane is released only when the last ingot in the batch has been unloaded.

Assume that their are 4 furnaces and two teams of strippers. Assume further that all bogie movements take a negligible time. The furnaces start up at times 0, 40, 80, and 120 minutes respectively. Run your model for 1500 time units assuming a *cold* start. Investigate the effect of priorities in the use of the soaking pit cranes and estimate a maximum value for the number of setting places required (the capacity of the siding).

Exercise 5.4 Change the work cycle of a furnace in exercise 5.3 to the one detailed below. The furnace goes through the cycle

load; *melt*; *refine*; *tap*; *clean*; `repeat`

The loading of scrap metal requires the use of a crane, `C1`. When loaded, the furnace melts its load using 3 units of electric power. Once melted, two units of electricity are returned, and one is retained. After melting, the metal is refined. Then the furnace is tapped (its contents are poured out). A tapping requires a set of moulds and another crane, `C2`. In this case assume that the furnace discharges all of its load in one go. After being tapped, the furnace relinquishes its last unit of electric power. Every ten such cycles, the furnace lining is inspected by a group of asbestos clad brickies who repair any cracks or faults. Use italics to denote the activity durations informally.

Exercise 5.5 A newspaper has an office for receiving advertisements placed by telephone. There are `n` telephone trunks, and `m` telephone operators. A call is accepted at once should an operator be free. Otherwise, an incoming call is kept in a queuing system (FCFS). This consists of two arrays each with a capacity of `k` calls. The calls are always entered into a background queue, `Q1`. Whenever the foreground queue, `Q2`, is empty all the entries in `Q1` are automatically transferred into `Q2` (assume this takes zero time). When an incoming call has been accepted and completed, an operator spends a little time completing notes about it before looking for a fresh task. Then he/she is free to accept a call from `Q2`. The operator continues taking calls from `Q2` in this manner until `Q2` (and hence `Q1`) is empty. Not all calls can be accepted. If all `n` trunks are engaged, then an incoming call is rejected. A call must also be rejected if `Q1` is full. Run your model for 8 simulated hours.

Model data

Timings in minutes:

call inter-arrival	`negexp`:mean=1/min
advert placing	`normal`:mean=4.0,st.dev.=1.0
complete notes	`normal`:mean=1.25,st.dev.=0.5

System sizing:

`n` the number of trunks	15
`k` the capacity of `Q1` and `Q2`	9
`m` the number of operators	6

Exercise 5.6 The model of example 6 would be badly behaved if the request rate were low (it isn't in this case except right at the start). For then the scanner, which has a fine grain of time compared to other entities, would do much fruitless rotating and testing. It is instructive to modify the program in such a way that the scanner will go to sleep if there are no requests.

HINT: You may wish to use the scheduling routine `cancel` which is an attribute of `class entity`. A call `E.cancel` (`ref(entity)E`) removes `E` from the event list if there, and has no effect if `E` is not in the event list. `current.cancel` (of course `passivate` suffices — make sure you understand why) puts `current` to sleep out of the event list and resumes the actions of the new entity at the head of the event list. If, because of this, the event list becomes empty, then the call on `cancel` causes a run time error.

`Demos.cancel` removes the main program (the `Demos` block) from the event list. This may be useful in situations where the length of the simulation run is to be determined from internal conditions rather than predicted in advance.

Chapter 6

Waits until

In the models we have examined so far, we have been able to express the action histories of entities as sequences of activities, usually of the form

```
acquire R1; acquire R2; ... acquire Rn;
   hold(activity duration);
release R'1; release R'2; ... release R'm;
```

where the extra resources required (R_1, R_2, ..., Rn), be they modelled as `res`, `bin` or `entity` objects, have been requested and acquired one at a time. In this chapter, we consider models in which wanted resources must be acquired at the same instant so the requesting object is blocked until that is the case. Such situations arise frequently in the real world and it is important that a simulation language can handle them. We illustrate two classes of problem informally and sketch their style of solution in Demos.

1. An entity competes with other entities from a pool of resources, and is not allowed to start its next activity unless *all* the resources required for its commencement are available. For example, given resources R_1, R_2 and R_3, and several entities E_k which use one or more of these resources to carry out a tasks. Suppose E_1, requires R_1 and R_2 to start a task. The coding

   ```
   R1.acquire(1);
   R2.acquire(1);
   hold(task time);
   ........
   ```

   is manifestly undesirable as E_1 may seize R_1 and wait a long time before R_2 is available. Further, whilst E_1 holds resource R_1, it is preventing other entities which require R_1 but not R_2 from progressing. What we need is a synchronisation which lets E_1 know when all the resources it requires for its next activity are available and allows E_1 to seize them all at once. In Demos this is the `condq`. Informal code for E_1 takes the shape:

   ```
   Q.waituntil(R1 available and R2 available);
   R1.acquire(1); R2.acquire(1);
   hold(task time);
   ........
   ```

   in which we have used `ref(condq) Q` to delay E_1 until all the resources it requires are available until it commits to seizing them.

Notice that the wait until testing and the actual seizing are coded above as separate statements:

```
Q.waituntil(R1 available and R2 available);
R1.acquire(1); R2.acquire(1);
```

In Demos (and Simula) models, control always remains with `current` until `current` itself relinquishes it. So `current` cannot be interrupted between `Q.waituntil...;` and `R1.acquire(1); R2.acquire(1);` or between `R1.acquire(1)` and `R2.acquire(1)`.

So our wait until then acquire coding is safe within its context.

But take care if you are using Demos to develop (say) real network software where such wait until and seize synchronisations must be atomic (at the lowest level, wrap then in some sort of semaphore).

It *would* have been nice to develop a general synchronisation that waited until and seize as a single indivisible atomic action akin to an operating systems monitor. But we wanted our wait until to handle very general conditions some of which are not expressible in terms of resources, so something had to give[1].

2. An entity can handle several types of request and waits to see what turns up next before committing to a task and acquiring the appropriate resources. Again we use a `condq` and informal code takes the form:

```
Q.waituntil(all resources for task$_1$ available
        or all resources for task$_2$ available);
if all resources for task$_1$ available then
begin
  seize all resources for task$_1$;
  hold(task$_1$ time);
  release unwanted resources$_1$;
end else
begin
  seize all resources for task$_2$;
  hold(task$_2$ time);
  release unwanted resources$_2$;
end;
```

Notice that we can make use of existing Demos facilities to express many of the waituntil conditions (using attributes like `avail`, `find`, and `length`).

[1] It is an ongoing concern however, and pehaps one day ....

6.1 Condition queues

Example 9: Port system with tides

Consider an extension to the port system of Example 2 which takes account of the state of the tide. Boats arrive laden and depart empty. We now place the extra constraint that boats may only dock if the tide is not low. As before, they may leave whatever the state of the tide. It is still fair for a boat to request a jetty on arrival, but the following partial coding

```
entity class boat;
begin
  jetties.acquire(1);
  tugs.acquire(2);
  wait until the tide is not low;
  hold(2.0);
  tugs.release(2);
  ...............
end***boat***;
```

is not satisfactory because a significant period of time may elapse before a boat actually uses the tugs if it acquires them while the tide is low. During this interval, one or other or both these tugs could perhaps be gainfully employed by boats wishing to leave the port.

Notice that reversing the order of the tug and tide requests does not help because the tide may have gone out before two tugs are available. Boats wishing to leave their jetty must wait until such a time as two tugs are available *and* it is not low tide.

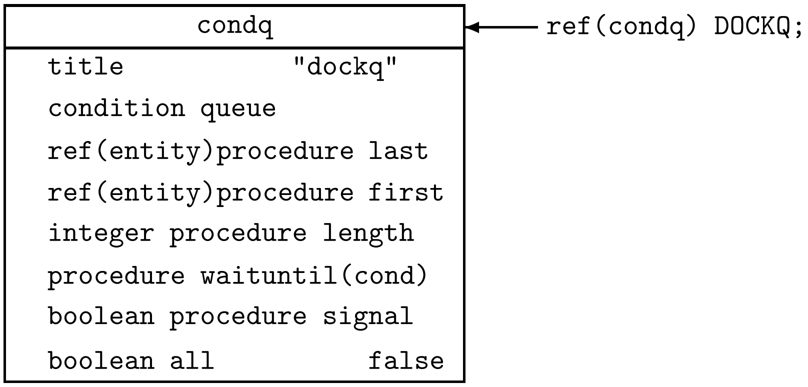
Dealing with the periodic setting and resetting of the state of the tide is quite straightforward. Suppose low tides occur every 13 hours and last for 4 hours. We can represent the state of the tide by a global `boolean lowtide` (initially `false`) and arrange to set its value appropriately by an object of `class tide`

```
[ boolean lowtide; ]

entity class tide;
begin
  lowtide := true;
  hold(4.0);

  lowtide := false;
  hold(9.0);
  repeat;
end***tide***;
```

The statement `new tide("tide").schedule(1.0);` (when executed at simulation time zero) corresponds to low tides starting at `time` = 1.00, 14.00, 27.00, ... We can now express the condition for docking in Demos by

```
tugs.avail >= 2 and not lowtide
```

We have not modelled low tide as a resource: in no sense should a `boat` be able to seize the tide.



```
condq                                  <--- ref(condq) DOCKQ;
  title             "dockq"
  condition queue
  ref(entity)procedure last
  ref(entity)procedure first
  integer procedure length
  procedure waituntil(cond)
  boolean procedure signal
  boolean all           false
```

Figure 6.1: Result of DOCKQ :- new condq(”dockq”)

Demos allows entities awaiting a particular condition to arise to be detained in a condition queue (of `class condq`, see figure 6.1) by a `waituntil` command. For example, the docking activity of `class boat` will now read

```
[ ref(condq) DOCKQ;]

jetties.acquire(1);
DOCKQ.waituntil(tugs.avail >= 2 and not lowtide);
tugs.acquire(2);
  hold(2.0);
tugs.release(2);
```

The synchronisation works as follows: let entity `E` (or indeed, the Demos block itself) issue the request

```
Q.WAITUNTIL(condition);          [ ref(condq) Q;]
```

The condition can be any (arbitrarily complicated) boolean valued expression The condition is dynamically re-evaluated each time it is tested—it is called by `name`—and is best considered as being local to `E` the maker of the request rather than the condition queue `Q`. The caller `E` continues straight on as `current` without delay should the condition evaluate to `true` *and* `E` have priority over any entities waiting in the `Q.condition_queue`. If the condition evaluates to `false`, then `E` is removed from the event list and passivated (put to sleep) in `Q.condition_queue` (in priority order, of course). There it remains until it is awakened, tested and its condition is found to be `true`.

Arranging to reawaken such dormant entities at the appropriate moment can be implemented in several ways and is the subject of much debate in the discrete event world. In Demos, the responsibility is put squarely on the shoulders of the programmer himself (an approach which follows the philosophy of the host language Simula. See Nygaard and Dahl [41, section 2.3.5]).

In this example, several boats may be blocked at a given time due to insufficient tugs and/or the state of the tide. The only possible times at which they can be unblocked are a) when another boat releases some tugs, and b) when the tide turns from being low. Thus the programmer has to ensure that `DOCKQ` is signalled (by a call `DOCKQ.signal`) whenever tugs are released or `lowtide` is reset to `false`. The intent of a call on `signal`, e.g.

```
Q.signal;                                   [ ref(condq) Q;]
```

is to unblock those entities waiting until at the front end of `Q.condition_queue` that can now go (there may be several). It operates as follows.

1. if the condition queue of Q is empty, it has no effect.

2. Otherwise, denote the entities waiting until in `Q.condition_queue` `E1`(== `Q.FIRST`), `E2`, ..., `En`. Then `Q.signal` enters `E1` into the event list at the current clock time, but as last entity at that time. When `E1` becomes `current`, it tests its own condition. If this evaluates to `false`, `E1` drops out of the event list and falls asleep again at the head of `Q.condition_queue`. If the condition of `E1` evaluates to `true`, then `E1` leaves the condition queue **and** promotes `E2` into the event list *immediately* behind itself. `E1` now continues on as `current`, usually acquiring resources whose availability was tested in the condition, and thus diminishing the total pool of resources. E.g.

   ```
   DOCKQ.waituntil(tugs.avail >= 2 and not lowtide);
   tugs.acquire(2);
   ```

   When `E1` steps down as `current`, the new first entity in the event list is `E2`. If `E2`'s condition evaluates to `true`, then `E3` is promoted into the event list directly behind `E2`, and `E2` acquires the resources it wants; otherwise `E2` is dropped from the event list back to the head of the condition queue and `E3` is not tested at all. Thus `Q.signal` activates entities at the head of `Q.condition_queue` in turn either until the queue is exhausted or else an entity condition fails. See appendix B for semi-formal algorithms for `waituntil` and `signal`.

Complete program

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(res) tugs, jetties;
    ref(condq) DOCKQ;
    ref(rdist)next, discharge;
    boolean lowtide;

    entity class boat;
    begin
      new boat("boat").schedule(next.sample);
      jetties.acquire(1);
      DOCKQ.waituntil(tugs.avail >= 2 and not lowtide);
      tugs.acquire(2);
        hold(2.0);
      tugs.release(2);
      DOCKQ.signal;

      hold(discharge.sample);

      tugs.acquire(1);
        hold(2.0);
      tugs.release(1);
      jetties.release(1);
      DOCKQ.signal;
    end***boat***;

    entity class tide;
    begin
      lowtide := true;
      hold(4.0);

      lowtide := false;
      dockq.signal;
      hold(9.0);
      repeat;
    end***tide***;

    trace;
    tugs      :- new res("tugs", 3);
    jetties   :- new res("jetties", 2);
    DOCKQ     :- new condq("dockq");
    next      :- new negexp("next boat", 0.1);
    discharge :- new normal("discharge",14.0,3.0);
    new tide("tide").schedule(1.0);
    new boat("boat").schedule(0.0);
    hold(28.0*24.0);
  end;
```

OUTPUT: Partial Trace and the final report.

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                  *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

     ...................................

    53.000 tide 1       holds for 4.000, until 57.000
    53.403 boat 3       releases 1 to tugs
                        releases 1 to jetties
                        signals dockq
                        ***terminates
    56.849 boat 5       schedules boat 6 at 61.864
                        seizes 1 of jetties
                        w'until in dockq
    57.000 tide 1       signals dockq
                        holds for 9.000, until 66.000
           boat 5       leaves dockq
                        seizes 2 of tugs
                        holds for 2.000, until 59.000

     ...................................



                       clock time =    672.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
next boat         0.000     64 negexp        0.100            33427485
discharge         0.000     58 normal       14.000     3.000  22276755


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
tugs              0.000    114    3    0    3    17.064     0.000    1
jetties           0.000     56    2    0    0    81.396     6.343    7


                   c o n d i t i o n   q u e u e s
                   *******************************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
dockq             0.000     58     2     0 7.683&-002    36      0.890
```

Remarks on Example 9

We pick up the trace at time 53.0, when the fifth low tide period is setting in. The trace shows `lowtide` being set at time 53.0 and reset at time 57.0. At time 56.849, `boat 5` seizes 1 jetty and then enters `DOCKQ` to await its condition being set. At time 57.0, the tide is reset and a signal is sent to `DOCKQ`. This awakens `boat 5` which continues on its way seizing two tugs.

The partial report shows that the extra constraint has caused a little more congestion (the average wait for a jetty is up to 6.343 from 5.498, and for a tug is 1.006 instead of 0.0285; compare with the results for Example 2). The column headings in the condition queue reports have been explained in connection with `waitqs` (see section 5.2) but this time we require only one line per queue as the master/slave situation does not obtain.

In this example, boats leaving and boats arriving compete for tugs. Boats arriving never actually queue for tugs on the resource `tugs` itself: when they escape from `DOCKQ` two tugs are available. But boats leaving do queue on the resource `tugs` and it is important to realise that the calls which release tugs and signal `DOCKQ`, e.g.

```
tugs.release(2);    DOCKQ.signal;
```

in effect give boats leaving priority because boats queueing on the resource `tugs` (all of which are waiting to leave) are tested before `DOCKQ.signal` has a chance to test any wait until conditions. Perhaps this is as it should be, but if not the priority can be changed simply by reversing the order of the calls

```
 DOCKQ.signal;    tugs.release(2);
```

Make sure you understand why (follow through the examples using the semi-formal algorithms in appendix B).

Another way of doing the same sort of thing is to introduce a second `condq` (here `OUTQ`) for boats leaving. We create it by

```
OUTQ  :- new condq("LEAVING");   [ ref(condq) OUTQ;]
```

The condition for boats leaving could be quite simple as

```
OUTQ.waituntil(tugs.avail > 0);
```

We also need to signal `OUTQ` at appropriate times: times at which tugs are released. Thus the lines `DOCKQ.signal`; inside `class boat` are to be replaced by

```
DOCKQ.signal; OUTQ.signal;
```

(reverse the order of the calls if you wish boats leaving to have priority). Note that there is no point at all in signalling OUTQ when the tide turns—it is of no concern to entities waiting in that condq. In general, when resources are released, some, but not all, condqs need to be be signalled. It is enough to signal only those condqs containing entities waiting on the freshly released resources.

Example 10: Tanker simulation revisited

As an extension to the basic model of Example 7, we assume that tankers now carry one of five types of oil. The grade of oil is indicated by integer type which takes values in the range 1 through 5, a higher value indicating a better grade of oil. In order to unload when docked, they need a shore tank which (in order of preference):

1. has the same type of fuel and sufficient capacity to take all that the tanker is carrying
2. has the same type of oil but not enough capacity to accept the full load
3. is empty and previously held either the same or a better type of fuel
4. is empty and previously held an inferior type of oil. In this case, the shore tank must first be cleaned. This takes a long time and is the last resort.

In this new situation it is much harder to express the search by using find, although it could perhaps be done by devious use of priority within class shoretank. Even so it would lead to a non-transparent program text and in cases like this we really need another mechanism so that we can express the program logic clearly. It is perhaps best to have all the decisions at one point in the program instead of scattered over several entities. This is where the hitherto unexplained boolean procedure avail local to waitq (see figure 5.3) comes into its own. A call

```
        Q.avail(E, condition);            [ ref(waitq) Q;]
```

tries to locate an entity E satisfying the condition (just like find), but does *not* coopt the entity E if found nor block the caller if an entity E satisfying the condition cannot be found. It returns true and sets E to refer to the found entity if one can be located; if not, it returns false and sets E to none.

In this variation of Example 7, it is convenient to use two waitqs to hold available shore tanks. We place completely empty shore tanks in EMPTYQ and partly full shore tanks in TANKQ. Note that in this description, a shore tank will discharge its contents into the refinery when less than 5 units of volume are available.

```
[ ref(waitq) EMPTYQ,TANKQ; ]

entity class shoretank(free); integer free;
begin
   integer type, max;
   type := an appropriate type;
   max := 70;

   if free = max then goto EMPTY;
LOOP:
   while free >= 5 do
   begin
     Q.signal;
     TANKQ.wait;
   end;
   hold(time to empty);
EMPTY:
   priority := -type;
   Q.signal;
   EMPTYQ.wait;
   priority := 0;
   repeat;
end***shoretank***;
```

The body of `class shoretank` consists of an initialisation (setting `type` and `max`: `free`, as a parameter, is already set) and then the main loop is entered at `EMPTY` if the initial value of `free = max`; otherwise at `PART_FULL`. We use `priority` only in `EMPTYQ` where shore tanks are ranked with those containing the least quality oil at the front. A shore tank is coopted by several tankers in turn who part fill it (then it returns to `TANKQ`). When the shore tank is full, or very nearly so (less than 5 free units of capacity remain), then it does not return to `TANKQ`, but discharges its contents into the refinery and joins `EMPTYQ`. A shore tank coopted from `EMPTYQ` under conditions (3) or (4) may have its TYPE reset; under condition (4), it will also have to be cleaned.

A tanker looks first for a shore tank in `TANKQ` of the same type and with sufficient capacity

1. `TANKQ.avail(ST, ST.type = type and ST.free >= load)`

   if this search fails, it then tries to locate a shore tank which is available for loading and contains the same type of oil

2. `TANKQ.avail(ST, ST.type = type) and ST.free < load)`

   notice that testing first on condition a) and then on condition b) implies two sweeps through `TANKQ` unless the shore tanks have been ideally sorted first (and they haven't—try to do so as an extra exercise).

   If a) fails then we could express condition (2) by just `ST.type = type`. Perhaps its clearer to write the condition in full?

If this second search fails, then the tanker turns its attention to EMPTYQ. It now tries to locate a shore tank in EMPTYQ which contained oil of the same or of a better quality.

3. EMPTYQ.avail(ST, ST.type >= type)

   Since we have queued shore tanks in EMPTYQ ranked according to type, we automatically return the shore tank of least quality passing the test. Finally if all else fails, then any shore tank in EMPTYQ will do.

4. EMPTYQ.avail(ST, true)

   If all of these tests fail, then a tanker must wait until a suitable shore tank does appear in either TANKQ or EMPTYQ.

We thus require a condq (here Q) and make the request via a wait until statement of the form

```
Q.waituntil(or4(a, b, c, d));
```

where we have used our own boolean procedure or4

```
boolean procedure or4(a,b,c,d); name a,b,c,d; boolean a,b,c,d;
  or4 := if a then true else
         if b then true else
         if c then true else
            d;
```

The reader should be able to convince his- or her-self that the formulation

```
Q.waituntil(a or b or c or d);
```

would not not work in this case because each of the part conditions a, b, c, and d is called in turn before any or operation is carried out. Each one not only returns true or false but also sets ST. Thus the test on b overwrites the assignment to ST in a, the test on c that in b, and the test on d that in c.

Notice that since find is restricted to searching down just one waitq it is inappropriate here. This is unfortunate in that find/wait are self-reawakening and obviates the need for the calls on signal.

class tanker now has the form:

```
entity class tanker;
begin
  integer type, load, l;
  ref(shoretank) ST;

  new tanker("t").schedule(arr.sample);
  type := Ttype.sample;
  load := Tload.sample;

  while load > 0 do
  begin !*** get shore tank ***;
```

```
    Q.waituntil
       (or4 ( TANKQ.avail(ST, ST.type = type and ST.free >= load),
             TANKQ.avail(ST, ST.type = type and ST.free <  load),
            EMPTYQ.avail(ST, ST.type >=type),
            EMPTYQ.avail(ST, true)
       )    );
    ST.coopt;
    if ST.type < type then hold(clean.sample);
    ST.type := type;

    !*** pump ***;

    l := if ST.free >= load then load else ST.free;
    hold(setuptime + l*pumprate);

    ST.free := ST.free - l;
    ST.schedule(0.0);
    load := load - l;
    priority := priority + 1;
  end;
end***tanker***;
```

The main body of `class tanker` consists of a loop in which it remains until it has discharged all of its cargo (until `LOAD = 0`). While a tanker does try to discharge all of its cargo in one go (conditions a), c) and d)), a shore tank seized under condition b) has not enough capacity left and in this case the tanker will have to queue again in `Q` (it is given increased priority each time). When its wait until condition is `true`, a tanker leaves `Q` with its local variable `ST` referencing the most suitable storage tank thanks to the call on `avail`. But `ST` is still in its `waitq` and hasn't yet been coopted. To cater for this case (coopting an explicitly named entity) there is a `procedure coopt` local to `class entity`. A call `ST.coopt` removes `ST` from its current queue (if any) and places it under the bondage of `current` (in this case, a tanker). The tanker then causes it to be cleaned if it was empty and contained inferior quality oil is transferred. Then the type of `ST` is reset, a quantity `l` of oil is transferred from the tanker to the shore tank and `ST` is rescheduled. `ST` decides for itself what to do next (join `TANKQ` or discharge). Finally the tanker's own priority is incremented (it may loop again).

EXERCISES 6

Exercise 6.1 In Simula and ALGOL 60, compound conditions, such as `a and b` and `a or b`, are evaluated by first finding the value of `a`, then the value of `b`, and then performing the `and` or the `or` operation. If we wish to drop out of a compound evaluation with a minimum of testing, we can code

`if a then b else false` instead of `a and b`
`if a then true else b` instead of `a or b`

Write `boolean procedure and2` and `boolean procedure or2` to perform these optimisations.

Exercise 6.2 Customers with a predetermined thirst—taken as 1 to 6 pints—arrive at a pub, queue for a beer, drink it, and then either rejoin the beer queue or else (if their thirst has been assuaged) leave the pub.

Model data

Timings in minutes:

Customer arrival	`negexp`:mean=0.2/min
Waiter (re)entry	`constant`:30
Pouring	`constant`:1
Drinking	`uniform`:15.0→25.0
Washing	`constant`:0.5

A barman serves the customers giving each a clean glass for their beer every time. His/her other task is to wash empty glasses (these are collected from the customer and placed within her orbit by a waiter). While neither of these tasks is interruptible once started, the task of serving a customer naturally takes precedence. The waiter enters the bar every 30 minutes, collects all the empties, and places them on the bar top for the barmaid. She/he then retires to perform other duties. Run your model for 4 hours with 15 glasses, all initially clean.

Exercise 6.3 Repeat the Port System of Example 8 with the following different tidal constraints: boats can dock only at high tide and may leave only if it is not low tide.

Exercise 6.4 Consider the following (very simplified) model of a single lane of cars at a four way traffic intersection in which we ignore possible interference from other lanes. Cars arriving at the junction can turn left, continue straight on or turn right when the lights ahead are green and the car ahead is sufficiently clear.

Model data

Timings in seconds:

Car inter-arrival	`negexp`:0.15/second
Lights green	`constant`:20
Lights red	`constant`:24
Time to clear	`normal`:mean=2.0,st.dev.=0.5

Simulate for two hours under the initial conditions that at time 0.0 the lights are just turning green and the first car arrives.

Exercise 6.5 Sketch a solution to the following problem. Consider a junction consisting of a single lane side road joining a main road. The main road has one lane in each direction. Traffic is not allowed entry from the main road into the side road. Traffic moving from the side road on to the main road may filter right when the near lane is clear; or may turn left on to the main road only when both the near lane and the far lane of the main road are free.

Exercise 6.6 Boats arrive periodically at both ends of a long, narrow canal. For this exercise, we ignore such complications as tugs, storms, etc. (which can easily be added) and concentrate on deciding which direction should have control of the canal. The canal is narrow (so boats may not pass each other in either direction), and long (so that several boats are allowed to travel along it in the same direction). Let the time taken to pass through the canal be `ctime`. To reduce the risk of collision, boats travelling in the same direction must be well separated. Once a boat has entered the canal, no other boat may follow it until at least `ctime/3` has elapsed. Model the system when the canal direction is switched according to the fixed time slot rule. Here each direction has a time slot of fixed length in rotation. When the time slot is up, any boats currently in the canal are cleared, but boats not actually in the canal are blocked. When all the boats are clear, the direction of allowed travel is switched.

Exercise 6.7 Repeat exercise 6.6 switching canal direction when the length of the blocked queue has reached a certain limit, say L. N.B. in conditions of overload, it is possible to get at least L boats in queueing for each direction. To prevent the switching mechanism from looping, give each direction a certain minimum burst, say `l*ctime/3`.

Exercise 6.8 Repeat exercise 6.6 switching control of the canal only when the queue for the direction in control is empty. N.B. the switching mechanism may have to go into “neutral” if both queues are empty.

Exercise 6.9 Repeat exercise 6.6 switching when the blocked queue is longer than the queue in control.

Exercise 6.10 Remind yourself of exercise 4.8, page 72. As billets queue for a soaking pit they lose heat. Newcomers are clearly warmer than those blocked earlier. Modify exercise 4.8 so that when a fourth billet joins the queue for the pits, the *first* (and hence the coldest) billet in the queue is removed and deposited outside. It will only be brought back when there are no billets waiting for a pit and 5 pits are empty. Assume that such billet movements also require the use of a soaking pit crane.

6.2 Condition queues with all set

This section deals with the case when the entities in a condition queue are waiting on different conditions. In this case we want a call on SIGNAL to test each and every member of the queue and not to drop out if one entity condition fails.

Example 11: Dining philosophers

Five philosophers are seated round a circular table which contains an inexhaustible supply of spaghetti within easy reach of all at its centre. Between each pair of adjacent philosophers is a fork. The philosophers have a simple life style

```
LOOP:
  think;
  eat;
  REPEAT;
```

In order to eat, a philosopher requires both the fork on his left and the fork on his right. Each thus competes for resources with his immediate neighbours. The orgy is to last for 3 hours.

Model data

Timings in minutes:

think	`randint`:20→30
eat	`constant`:10→20

Resources:

fork	`ref(res)array fork[ 1:5 ]`:limit=1 each

In this model we represent the forks by `ref(res)array fork [ 1:5 ]` and initialise them by

```
for k := 1 step 1 until 5 do          [ integer k; ]
  fork[k] :- new res(edit("fork", k), 1);
```

We number the forks so that philosopher n finds `fork[n]` on his or her left and `fork[n+1]` on his or her right (`fork[1]` in the case of philosopher 5). Then `class philosopher` can be written

```
entity class philosopher(n); integer n;
begin
  ref(res) L, R;

  L :- fork[n];
  R :- fork[if n=5 then 1 else n+1];
LOOP:
  hold(think.sample);
  Q.waituntil(L.avail > 0 and R.avail > 0);
  L.acquire(1);  R.acquire(1);
    hold(eat.sample);
  L.release(1);  R.release(1);
  Q.signal;
  repeat;
end***philosopher***;
```

The new facet of this example lies in the fact that each philosopher has a different wait until condition (no two philosophers await the availability of the same pair of forks) and this causes some problems. If we let them all wait in the same queue, then we must remember that `signal` is coded (as so far revealed) to stop testing queue members when the condition of one member (the current `first`) fails. Thus if the condition of an earlier queue member is `false`, the remainder will not be tested even though their conditions, being different, may yield `true`. Giving each philosopher his own queue doesn't really help in this case either (although in general it may be a useful idea) as we then have to decide which queue gets priority.

To solve this and similar problems, we have given `condq` another attribute—`boolean all`—which is initially `false`. When `all` is `false`, `signal` operates as previously explained. When `all` is set to `true` by such an assignment as

```
Q.all := true;                      [ ref(condq) Q;]
```

a call on `Q.signal` will test the condition of each and every entity waiting until in the `condq`. This gives us what we want: by using a single `condq` with `all` set, philosophers are queued ranked according to their time of entry (their priorities are all zero) and every member of the condition queue will be tested. The complete program reads

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin integer k;
    ref(res)array fork[1:5];
    ref(idist) think, eat;
    ref(condq) Q;


    entity class philosopher(n); integer n;
    begin
      ref(res) L, R;

      L :- fork[n];
      R :- fork[if n=5 then 1 else n+1];
    LOOP:
      hold(think.sample);
      Q.waituntil(L.avail > 0 and R.avail > 0);
      L.acquire(1);  R.acquire(1);
      hold(eat.sample);
      L.release(1);  R.release(1);
      Q.signal;
      repeat;
    end***philosopher***;

    Q     :- new condq("await eat");
    Q.all := true;
    think :- new randint("think", 20, 30);
    eat   :- new randint("eat"  , 10, 20);
    for k := 1 step 1 until 5 do
      fork[k] :- new res(edit("fork", k), 1);
    trace;
    for k := 1 step 1 until 5 do
      new philosopher("p", k).schedule(0.0);
    hold(180.0);
  end;
```

OUTPUT. Partial trace and full report.

Starting at time 24.0 when P1 and P4 are eating, and P2, P3, and P5 are thinking.

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                  *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

      ...................................
    24.000 p 3          w'until in await eat
    25.000 p 2          w'until in await eat
    26.000 p 5          w'until in await eat
    34.000 p 4          releases 1 to fork 4
                        releases 1 to fork 5
                        signals await eat
                        holds for 23.000, until 57.000
           p 3          leaves await eat
                        seizes 1 of fork 3
                        seizes 1 of fork 4
                        holds for 18.000, until 52.000
    37.000 p 1          releases 1 to fork 1
                        releases 1 to fork 2
                        signals await eat
                        holds for 27.000, until 64.000
           p 5          leaves await eat
                        seizes 1 of fork 5
                        seizes 1 of fork 1
                        holds for 12.000, until 49.000
    49.000              releases 1 to fork 5
                        releases 1 to fork 1
                        signals await eat
                        holds for 23.000, until 72.000
    52.000 p 3          releases 1 to fork 3
                        releases 1 to fork 4
                        signals await eat
                        holds for 25.000, until 77.000
           p 2          leaves await eat
                        seizes 1 of fork 2
                        seizes 1 of fork 3
                        holds for 15.000, until 67.000
      ...................................


                       clock time =    180.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************



                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
think             0.000     23 randint          20        30  33427485
eat               0.000     20 randint          10        20  22276755


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
fork 1            0.000      7    1    0    0    62.778     0.000    1
fork 2            0.000      7    1    0    0    60.556     0.000    1
fork 3            0.000      7    1    0    0    67.222     0.000    1
fork 4            0.000      8    1    0    1    68.333     0.000    1
fork 5            0.000      7    1    0    0    62.222     0.000    1


                   c o n d i t i o n   q u e u e s
                   *******************************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
await eat   *     0.000     20     3     1      0.489     8      4.400
```

Remarks on Example 11

At time 26.0, P3, P2, and P5 (in that order) are waiting until in Q. When P4 releases his forks at time 34.0, P3 can proceed. When P1 releases his forks at time 37.000, P2 is tested first but is still blocked. P5 is allowed to proceed.

The only other thing worth remarking on is that the condition queues with `all` set are specially marked in reports with an asterisk following their `title` (see `await eat` above).

EXERCISES 6 (continued)

Exercise 6.11 Unsealed containers are placed on a conveyor belt every `cycle` seconds. The belt moves with a regular but jerky action past a row of N sealing machines. The movement of the belt is such that the containers are stationary for `pause` seconds in front of each machine. The movement into the next position takes `move` seconds (`cycle = move+pause`). During each pause, a fresh container is added at one end of the belt, and the 'oldest' container is removed from the other. It is also during part of this time interval that an unsealed container may be sealed. Sealing takes `seal` seconds and a sealing is not allowed to be started if the belt is moving or if less than `seal` seconds remain of the current pause period.

The sealing is done by a complicated machine which has two parts: a picker which dips into an inexhaustible supply of seals and returns with one, and a capper which accepts a seal from its picker partner (thus sending it off for another seal) and then waits its chance to seal. It will seal the next unsealed container that pauses before it and is given enough time to perform the operation. Then it collects a new seal from its picker.

Assume that each picker starts primed with a seal and at time 0.0 the first unsealed container is placed on the belt (it will be in position in front of the first sealing machine at `time = cycle`). Run your model for 8 simulated hours with n = 6 sealing machines, pause = 5 seconds, move = 3 seconds, seal = 2 seconds,and the time for a picker to fetch a seal at 7.0→11.0 seconds, `uniform`ly distributed. Report on the number of unsealed containers that get through. Experiment further with `n`.

Exercise 6.12 Metal plates arrive periodically at a plate cutting yard to be cut into shape. The yard's cutting shop contains two cutters `C1` and `C2`, each of which has its own buffer area for up to three uncut plates. Plates are typed on arrival into those requiring cutting by `C1` and those by `C2` (equally likely). If the appropriate buffer is full, freshly arriving plates are dumped on one side in the yard. They are brought in from the cold only when there is buffer space and the arrival area is empty. Plate movements are carried out by a crane which can handle one at a time.

Model data

Timings in minutes:

Plate inter-arrival	negexp:0.1 per minute
Cutting time	normal:mean=8.0,st.dev.=2.0
Crane movement times	
arrival-buffer	constant:1
arrival-dump	constant:0.5
dump-buffer	constant:1

N.B. wherever plates are stacked they are picked up in last-come, first-served order since they are piled one on top of the other. Assume that when dumped outside the plates are sorted into 2 piles - one awaiting C1, the other C2.

6.3 Waits until: signal versus snoopy

There are two main approaches for dealing with entities which are waiting until. One places the responsibility on the programmer; the other leaves it to the system itself to do the reawakenings. The second method was used in SIMON 75 (Hills and Birtwistle [41]) and used a special entity object named snoopy. Transferred to the context of Demos, a rough outline of the SIMON 75 snoopy is

```
[ref(watchdog) snoopy; ]

entity class watchdog;
begin
  ref(condq) Q;
  for each condq Q do
    Q.signal;
  hold until next event time;
  repeat;
end***watchdog***;
```

snoopy is meant to operate as follows. When it becomes current, it signals the user-created condqs in some predetermined order: in Demos this is the order in which they were created by new condq ... calls. When all the condqs have been tested in this manner, snoopy holds until the next event time, reappearing as the last entry scheduled for that time. snoopy thus becomes current again when all the resources to be freed at that clock time have been released and can now set about testing the conditions of any blocked entities.

Notice that the wait until routine associated with snoopy differs from the one used in Demos in some details. An entity E executing a wait until should now always be put to sleep in the appropriate condq and left to be awakened by snoopy. Otherwise, some waiting entities may be bypassed; for example, those already waiting on the same condition as E. Notice also that snoopy need only be active when it has work to do (not all the `condq`s are empty). If `snoopy` is passive when a wait until call is made, then that call is responsible for activating it (a technique similar to that used in awakening the scanner in exercise 5.6.

However this rough outline for `snoopy` is certainly not fool proof. For an entity unblocked from a later `condq`, Qm say, may alter conditions which enable entities blocked in one or more earlier `condq`s to go; indeed, these entities may even test the value of $\texttt{Q}_m\texttt{.length}$ as a part of their condition. A way round this is to force `snoopy` out of its depicted for-loop and make it repeat again from the beginning whenever it unblocks one entity from the current `condq` that it is testing (there are many simple variations on this theme). This is clearly a slower algorithm than the original one. Its is also less transparent.

Vaucher [105] discusses a *generalised wait until statement* (read Franta [32, pages 188-194] for a more accessible account). Instead of having user-defined `condqs`, Vaucher's algorithm makes use of a single system defined queue in which all entities waiting until are queued. Vaucher's `snoopy` also reappears alternately in the event list which can give rise to certain subtle differences in performance. It is instructive to compare Vaucher's algorithm and its corresponding wait until routine with the ones sketched in this chapter and find ways in which they can break.

The decision to implement wait until with `signal` rather than `snoopy` was based on the following points:

1. waits until usually cover complicated situations. In order to show that a program is correct, it is desirable to be able to argue from the program text exactly what should happen next. The algorithm for `signal` is very simple and is under direct user control; `snoopy` has a rather more complicated algorithm and operates more remotely behind the scenes. It is correspondingly more difficult to ensure that the correct synchronisations take place. See Palme [73] for an intertaining paper on a similar theme.

2. because Demos contains `res`, `bin`, and `waitq` facilities which automatically test and promote any waiting entities when incremented, the number of `condq`s in Demos programs is quite small. (In SIMON 75 and ECSL programs, every non-bound activity has an associated `condq`.) Further the `condq`s to test when resources are released, etc., are directly available from the activity diagrams and so the calls on `signal` are not all that difficult to get right. Remember that when a condition is reset, only those `condq`s containing entities waiting on that condition need be tested, and not all of them. It is surprising that most compilers for activity based languages simply throw away this information; otherwise they could be quite competitive with event or process based discrete event simulation languages.

3. `signal` is more efficient than `snoopy`; usually of the order of 2 or 3 times, but in pathological cases this factor can increase unboundedly.

4. not least, it follows the approach of the host language Simula (see Nygaard and Dahl [25, 71]).

However consider the following: based upon personal experience `waituntil/signal` synchronisations are hard to get right.

Practically every error noted in developing the
examples for this book was a missing call on signal.

So unless you have an automated front end or follow the techniques of appendices D and E, you may be worthwhile implementing and using a version of `snoopy` until satisfied that the model is working correctly, and only then remove it and convert to in-line signal calls. Check that you get the same results.

So it is not just an argument about ease of use (`snoopy`) against efficiency (`signal`). It should also be borne in mind that waits until are often quite complicated and in these situations there is no substitute for clear thought and well-defined tools. One of the dangers is that it is all too easy to write down and accept a model "works" (in the sense that it is not obviously wrong), and accept it without rigorous, proper checking.

EXERCISES 6 (continued)

Exercise 6.13 Why is it usually preferable to code

```
hold(t-time);
Q.waituntil(condition);
```

instead of

```
Q.waituntil(time >= t and condition);
```

Chapter 7

Breakdowns and interrupts

In this chapter, we investigate a few models in which the expected action history of an entity is not followed due, perhaps, to an equipment break down failure or to an interruption by another entity.

1. **breakdown:** in the very simplest cases, the disturbance merely imposes an unexpected delay on the victim. For example, should a machine part fail, then the machine operator may well be able to continue on with his current task as soon as a repair has been carried out.
2. **interrupt:** ome forms of disturbance cannot be cast into such a simple mould. An interrupt may well insist that we stop what we are doing now and get on with something else. For example, a hospital doctor doing his/her morning rounds may be interrupted to deal with an emergency case. While the doctor is away, the rounds must go on under some one else's supervision, and may well have been completed before the doctor has finished dealing with the emergency. So there is no compunction for the doctor to return to his/her previous task.

The possibilities are many and various and instead of trying to be all things to all men, Demos provides a few simple tools which can be applied to cover a wide range of problems. As usual, these tools are motivated by particular examples.

7.1 Simple breakdowns

A stream of orders is processed on a lathe L. If we assume that there are always orders waiting, then we can sketch the actions of the lathe by

```
lathe  =  hold(process time); done.update(1); repeat;
```

We now throw in the complication that the lathe is subject to periodic breakdowns. Each breakdown requires a halt in the current job for a spare part to be fitted (by the lathe operator himself). Then the lathe continues on with the same order and from where it left off.

As seen by the lathe, a breakdown is an unpredictable event. Accordingly we prepare a separate entity description for each type of breakdown along the lines of:

```
breakdown   =   hold(time to next breakdown);
                stop lathe;
                repair lathe;
                restart lathe;
                repeat
```

Thus we view a breakdown as the coming together of two entities in the style of `coopt/wait` except that the slave (here the lathe) is located in the event list.

Strategy 1

As a first try we might try the strategy

1. the breakdown agent, say `B`, schedules itself for the next breakdown
2. `B` notes how much time the lathe `L` has left in its current activity and then removes `L` from the event list
3. `B` then re-schedules `L` in the event list delayed by the time of the repair plus what was left of its original activity plus perhaps an extra delay for setting-up.
4. `B` itself holds for the repair time and then repeats its actions.

This interplay may be summarized by:

```
entity class lathe;         entity class breakdown(L); ref(lathe) L;
begin                       begin
  hold(process time);         hold(time to next breakdown);
  repeat;                     L.cancel;
end***lathe***;               L.schedule(repair time + time left + setup);
                              hold(repair time);
                              repeat;
                            end***breakdown***;
```

When cancelled the LSC of `L` will be pointing to the statement `hold(`*process time*`)` so that when it again becomes current it will execute the statement after. What we have to ensure is that this statement is not completed at time `T`, but at time `T` `+` *repair time* `+` *setup*, hence the arithmetic involved in the scheduling statement.

Strategy 2

But there is a weakness with the style of strategy 1. When a breakdown occurs, we will usually not be able to predict how long the repair will take as the resources required to effect the repair may not be available (although that is not the case here). Even worse, perhaps the repairman's gear may itself need a sudden repair!

Thus we prefer to arrange things so that the breakdown agent keeps the victim until the repair has been carried out before rescheduling it. This also makes it very much easier to deal with more complicated breakdowns. The new strategy is:

1. `B` schedules itself for the next breakdwon
2. `B` notes how much time the victim `L` has left in its current activity and then removes `L` from the event list
3. `B` carries out the repair
4. `B` re-schedules `L` in the event list delayed by what was left of its original activity plus perhaps an extra delay for setting-up.
5. `B` repeats its actions.

which is summarized by:

```
entity class lathe;          entity class breakdown(L); ref(lathe)L;
begin                        begin
  hold(process time);          hold(time to next breakdown);
  repeat;                      L.cancel;
end***lathe***;                hold(repair time);
                               L.schedule(time left + setup);
                               repeat;
                             end***breakdown***;
```

The arithmetic computations are straightforward. Let a breakdown occur at time `t` and the lathe's current job be timed to finish at time `T (T >= t)`. At time `t`, `L` will be scheduled in the event list with an associated event time of `T`. Now the event time of a scheduled entity `E` is accessible via a call `E.evtime` (`real procedure evtime` is an attribute of all entity objects.) It follows that that the amount of time left in the lathe's current task is just `L.evtime - t`).

A semi-formal outline of `class breakdown` is now:

```
entity class breakdown(L); ref(lathe) L;
begin
  real tleft;

  hold(time to next breakdown);

  tleft := L.evtime - time;
  L.cancel;
  hold(repair time for L);
  L.schedule(tleft + setup);
  repeat;
end***breakdown***;
```

EXERCISES 7

Exercise 7.1 Write a complete Demos program for the model described in this section.

Model data

Timings in minutes:

lathe processing time	`normal`:mean=15.0,st.dev.=3.0
between breakdowns	`negexp`:mean=1/300 mins
repair time	`normal`:mean=30.0,st.dev.=5.0

This time, after the lathe has been repaired it must be reset before it can continue. Let the reset time be a constant 5 minutes. Run your model for four weeks of simulated time assuming no discontinuities.

Exercise 7.2 Repeat exercise 7.1 above with the following twist: when a breakdown occurs, the current order is spoiled. Keep track in another `count spoiled`. Assume that it takes 6 minutes to reset the lathe and discard a spoiled order.

HINT: You may care to send a signal to the lathe and schedule it immediately the repair has been carried out. The signal is to indicate which alternative has cropped up and we then let the lathe itself sort out which `count` to update and which extra delay is needed over and above `tleft`.

Exercise 7.3 A machine shop contains six identical lathes. A continuous stream of orders arrives at the machine shop carefully timed to ensure that there are always orders waiting. Each lathe is subject to periodic breakdown, but this time the repairs are to be carried out by a specialist repairman. The solitary repairman has other duties to perform when not repairing a lathe. Although these other duties are of a lower priority, they cannot be interrupted once started.

Model data

Timings in minutes:

lathe processing time	`normal`:mean=10.0,st.dev.=2.0
between breakdowns	`negexp`:mean=1/300 mins
lathe reset time	`constant`:5
repair time	`constant`:30
other duty	`constant`:15

Run your model for 4 weeks assuming no discontinuities between shifts.

7.2 Interrupts

It doesn't require many extra complications before the `cancel/schedule` mechanism of section 7.1 proves to be inadequate. But we have learned one important lesson — that of writing the interrupting agent as a separate entity.

In the next example, the victim has to drop its resources when interrupted and try to regain control of them in competition with other entities. For security reasons, a share in a resource can only be released by the entity holding it, i.e. an interrupted victim has to be activated in order to relinquish its own resources. Thus we require a different technique from the `cancel/schedule` of section 7.1 where the victim lay passively out of the event list throughout the breakdown. It turns out to be easiest to let the interrupt be sent across just as a signal and to let the victim sort out its own reaction to it (as in exercise 7.2) instead of it being imposed upon from without.

Example 12: Coal hopper

Consider a coal depot where coal is loaded by gravity from a hopper into one truck at a time.

Model data

Timings in minutes:	
truck arrival rate	`negexp`:mean=1/12 per minute
loading rate	`constant`:0.1 tons/min
Capacities:	
truck load	`5,10,15 tons,equally likely`
priority	`randint`:1→4

If the trucks are served FCFS, then we can model `class truck` by

```
[ ref(res) hopper; ]

entity class truck;
begin
  hopper.acquire(1);
  hold(load time);
  hopper.release(1);
end***truck***;
```

We now alter the rule for who is to be loaded. Trucks are now allocated a priority in the range 1 through 4. A freshly arriving truck can interrupt the truck currently being loaded if it has a greater priority than the latter. The interrupted entity has its priority increased by one each time it is interrupted—an attempt to see that a truck with a low initial priority cannot be delayed too long. Then it rejoins the queue for the hopper and tries again.

Notice that this is not just a simple pushdown effect. For if $\mathtt{L}_p$ displaces $\mathtt{L}_r$, $\mathtt{L}_r$ does not necessarily acquire the hopper again when $\mathtt{L}_p$ quits. For in the mean time, a third truck $\mathtt{L}_q$ may arrive with priority in between those of the other two ($\mathtt{priority}_p$ $>=$ $\mathtt{priority}_q$ $>$ $\mathtt{priority}_r$). When $\mathtt{L}_p$ quits, $\mathtt{L}_q$ seizes the hopper. When $\mathtt{L}_r$ is displaced, its incremented priority ensures that it becomes the first entity queueing for the hopper. But it will be pushed back down the queue by a later arrival with higher priority or should its own interrupter be interrupted. Notice that a truck may be interrupted several times before being fully loaded.

The synchronisation is achieved in Demos as follows: at time `t`, let $\mathtt{L}_r$ be loading and $\mathtt{L}_p$ deliver an an interrupt. The key statements executed by $\mathtt{L}_p$ and $\mathtt{L}_r$ at time `t` are

```
           Lp                        Lr

1   Lr.interrupt(1);
2   hopper.acquire(1);
3                             hopper.release(1);
4                             priority:=priority+1;
5                             interrupted := 0;
6                             hopper.acquire(1);
7   hold(load time);
```

1. `procedure interrupt` is local to `class entity` and takes an `integer` parameter n (should there be several interrupts, each can be given a distinguishing number). A call `E.interrupt(n)` operates as follows:
   (a) the routine saves the value of `n` in an `integer` variable `interrupted` local to `E` (`integer interrupted` is another `entity` attribute)
   (b) `E` is placed in the event list, at the current clock time but as last entity scheduled for this time.
   (c) when the ongoing active phase of `current` finishes (here with a `hopper.acquire(1)`), `E` is promoted to be the new `current`.
2. $\mathtt{L}_p$ attempts to gain control of the hopper, but must wait; by the nature of the problem, it will be inserted at the head of the queue for the hopper.
3. $\mathtt{L}_r$ is now `current` and finds its own local `integer` variable `interrupted` set to `n`. It releases the hopper (which will be seized by $\mathtt{L}_p$), and then
4. $\mathtt{L}_r$ increases its own priority
5. sets its own `interrupted` to zero, and then

6. attempts to regain control of the hopper. It joins the hopper queue behind L_p.
7. L_p now gains control of the hopper and a fresh loading starts.

The complete program for an eight hour run (in simulation time!) reads:

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    real rate;
    ref(entity) USER;
    ref(res) hopper;
    ref(idist) p, vol;
    ref(rdist) next;

    boolean procedure and2(a, b); name a, b; boolean a, b;
      and2 := if a then b else false;

    entity class truck;
    begin
      real tleft, load, start;

      new truck("l").schedule(next.sample);
      priority := p.sample;
      load     := 5*vol.sample;
      tleft    := load/rate;
      if and2(USER =/= none, priority > USER.priority)
        then USER.interrupt(1);
      while tleft > 0.0 do
      begin
        hopper.acquire(1);
        !*** loading_starts ***;
        USER  :- current;
        start := time;
        hold(tleft);
        !*** done_or_interrupted ***;
        USER  :- none;
        hopper.release(1);
        if interrupted = 0 then tleft := 0.0 else
        begin
          interrupted := 0;
          tleft       := tleft - (time-start);
          priority    := priority + 1;
        end;
      end***of while loop***;
    end***truck***;

    trace;
    rate   := 1.0;
    p      :- new randint("priority", 1, 4);
    vol    :- new randint("truck load", 1, 3);
    next   :- new negexp("next truck", 1/12);
    hopper :- new res("hopper", 1);
    new truck("l").schedule(0.0);
    hold(480.0);
  end;
```

OUTPUT

```
                       clock time =      0.000
**********************************************************************
*                                                                    *
*                 t r a c i n g   c o m m e n c e s                 *
*                                                                    *
**********************************************************************

      time/ current     and its action(s)

     0.000 demos        schedules l 1 now
                        holds for 480.000, until 480.000
           l 1          schedules l 2 at 7.358
                        seizes 1 of hopper
                        holds for 15.000, until 15.000
     7.358 l 2          schedules l 3 at 88.690
                        interrupts l 1, with power = 1
                        cancels l 1
                        awaits 1 of hopper
           l 1          releases 1 to hopper
                        awaits 1 of hopper
           l 2          seizes 1 of hopper
                        holds for 5.000, until 12.358
    12.358              releases 1 to hopper
                        ***terminates
           l 1          seizes 1 of hopper
                        holds for 7.642, until 20.000
    20.000              releases 1 to hopper
                        ***terminates
    ................................


                       clock time =    480.000
**********************************************************************
*                                                                    *
*                            r e p o r t                            *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
priority          0.000     44 randint           1         4  33427485
truck load        0.000     44 randint           1         3  22276755
next truck        0.000     44 negexp   8.333&-002            46847980


                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
hopper            0.000     51    1    0    0    82.252    17.978    8
```

Remarks on Example 12

The trace shows `L 1` starting an expected 15 minute load at time 0.0. When interrupted at time 7.358 by `L 2`, `L 1` relinquishes the hopper and tries again. When `L`

1 eventually regains control of the hopper at time 12.358, it has 15.0-7.358 = 7.642 minutes worth of loading left. This time no interruptions are made (L 3 does not enter the system until well after L 2 has quit).

As to the program itself, the global variable USER always references the current occupier of the hopper (or none). A freshly arriving truck sends an interrupt only if the hopper is occupied (user =/= none) and its priority is greater than that of the hopper (priority > USER.priority). By writing the interrupt in the form if *condition* then interrupt(1) we only disturb the occupier when it has to give up the hopper.

The victim is at once rescheduled in the event list with its local variable interrupted set to 1 signifying that an interrupt has been made. N.B. After an interrupt has been dealt with, remember to reset interrupted to zero (or some other suitable neutral value) to signify that no interrupt is now pending.

It is up to the truck concerned to remember how much loading remains. This has been done in a rather special way by computing the length of the loading time straight away and recording it in tleft. Each time the truck has a spell on the hopper and is interrupted, tleft is decremented by the time just spent (time - start). If no interrupt occurs, tleft is set straight to zero. The next time the truck gains control of the hopper, it will try to retain it for the full expected period, namely tleft.

Example 13: Quarry

A quarry contains a narrow seam of high quality stone on the edge of a broad front of average quality stone. Early each day the site is blasted producing sufficient rock of both qualities to meet the days demand. Two types of truck arrive: large trucks which take the average quality stone, and small trucks for the high quality stone. The site has three mechanical diggers: two large diggers, L1 and L2, are used to load average quality stone on to the large trucks. They are too large to manoeuvre near the narrow seam and can thus never be used to load the high quality stone. The site geography demands that the high quality stone be loaded by a third, smaller digger S.

The large diggers can load at a considerably higher rate than the small digger S. When S has no customer for high quality stone it is allowed to load a large truck should L1 and L2 be busy. But should a high quality customer then arrive, S stops loading average quality stone and loads high quality stone. Meanwhile its previous customer does not necessarily have to wait for the small digger to become free again—it can be loaded by a large digger if one becomes free first. S may also be interrupted if it is loading a large truck with average quality stone when a large digger becomes free. The large digger should take over the job freeing the small digger as it loads at a faster rate.

Model data

Timings in hours:	
Inter-arrival times	
large trucks	`negexp`:mean=22/hour
small trucks	`negexp`:mean=10/hour
Loading rates	
large digger	`constant`:240 tons/hour
small digger	`constant`: 60 tons/hour
Capacities:	
large truck load	20 tons
small truck load	5 tons

Structure of the model I

We first sketch a solution ignoring interrupts and then add in the code for the interrupts. We model small trucks, large trucks, small diggers, and large diggers as entities. Our starting point has large and small trucks arriving at their own rates. The small digger may only load small trucks and the large digger may only load large trucks. We choose to let the diggers be the masters and the trucks be the slaves. On arrival, a truck waits for attention in `waitq` `STQ` or `LTQ` according to its type. At this level of abstraction, the entity outlines are:

```
Struck   =   STQ.wait
Ltruck   =   LTQ.wait

Sdigger  =   T :- STQ.coopt; hold(load); T.schedule(0.0); repeat
Ldigger  =   T :- LTQ.coopt; hold(load); T.schedule(0.0); repeat
```

Structure of the model II

The small digger can load trucks from either `STQ` or `LTQ`. We use `ref(condq) Q` suspend the small digger until the appropriate choice can be made, which in turn means that the trucks must signal `Q` when they arrive in the system to make sure that if `S` is idle it can start loading. The system description unfolds to

```
Struck   =  Q.signal; STQ.wait
Ltruck   =  Q.signal; LTQ.wait

Sdigger  =  Q.waituntil(STQ.length > 0 or LTQ.length > 0);
            if STQ.length > 0
              then service small truck T in STQ
              else service large truck T in LTQ;
            T :- none; repeat
Ldigger  =  T :- LTQ.coopt; hold(load); T.schedule(0.0); repeat
```

When loading, the small digger refers to its companion truck by `T`. When `T` has been loaded and sent on its way (by `T.schedule(0.0)`), the local variable `T` of `S` is reset to `none` prior to `S` re-entering the `condq Q`. Thus `S.T == none` is a sign that `S` is idle.

Structure of the model III

As yet we have made no attempt to incorporate interrupts — once started a digger will complete a loading. We now turn our attention to handling the interrupts.

First consider the possible interrupt when a small truck arrives. If the small digger `S` is idle or dealing with another small truck, then no interrupt is necessary. But if the small truck finds the small digger loading a large truck (expressed by `S.T is Ltruck`) then is has to force `S` to stop loading its large truck at once. `S` can then allow `S.T` to rejoin `LTQ` partly filled (and with higher priority?), and then `S` can start loading the interrupting small truck. The interruption is achieved by coding small truck as

```
truck class Struck;
begin
  schedule small next;
  if S.T is Ltruck
    then S.interrupt(1)
    else Q.signal;
  STQ.wait;
end***struck***;
```

and expanding the code for the small digger to

```
entity class Sdigger;
begin
  ref(truck) T;

  Q.waituntil(STQ.length > 0 or LTQ.length > 0);

  if STQ.length > 0 then
  begin !*** load small truck ***;
   T :- STQ.coopt;
   hold(load time);
   T.schedule(0.0);
  end else

  begin !*** load large truck ***;
   T :- LTQ.coopt;
   hold(load time);
   !*** possible interrupt ***;
   if interrupted = 0 then T.schedule(0.0) else
   begin
     note that T is partially filled;
     let T rejoin LTQ;
     interrupted := 0;
   end;
 end;
  T :- none;
  repeat;
end***Sdigger***;
```

Since it may be loaded in stages instead of all at once, we have to cast the body of `Ltruck` in the form of a loop. We assume that the local variable `load` keeps track of how much loading there is left to do.

```
truck class Ltruck;
begin
  schedule next;
  load := 20.0;
  while load > 0.0 do
  begin
    Q.signal;
    LTQ.wait;
  end;
end***Ltruck***;
```

When rescheduled, a large truck quits if its `LOAD = 0.0`; or else it re-enters `LTQ` for another load.

Structure of the model IV

The second type of interrupt occurs when a large digger becomes free and the small digger is loading a large truck. Since a large digger is so much faster than the small digger, it seems only sensible that it should take over. Accordingly, a large digger will issue an interrupt when that situation arises. We tentatively code

```
entity class Ldigger;
begin
  ref(truck) T;

  T :- LTQ.coopt;
  hold(load time);
  T.load := 0.0;
  T.schedule(0.0);
  if LTQ.length = 0 and S.T is Ltruck
    then S.interrupt(2);
  repeat;
end***Ltruck***;
```

On the interrupt call, the issuer of the call continues straight on and awaits the arrival of the large truck `T` in `LTQ`. The interrupt stops `S` from working on its current large truck. `S` resets the current load of the truck `T` and sends it off to `LTQ` where it will at once get service (a large digger is waiting). `S` then re-enters the **`condq Q`**.

The complete program

The complete program gives some hierarchy to the trucks and diggers (hiving off their common parts) and details the loading rates. The model is run for 10 hours.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
 begin
   ref(sdigger) S;
   ref(condq) Q;
   ref(waitq) LTQ, STQ;
   ref(rdist) nextl, nexts;

   entity class truck;
   begin
     real load;
   end***truck***;

   truck class Ltruck;
   begin
     load := 20.0;
     new Ltruck("Ltruck").schedule(nextl.sample);
     while load > 0.0 do
     begin
       Q.signal;
       LTQ.wait;
     end;
   end***Ltruck***;

   truck class Struck;
   begin
     load := 5.0;
     new Struck("Struck").schedule(nexts.sample);
     if S.T is Ltruck
       then S.interrupt(1)
       else Q.signal;
     STQ.wait;
   end***struck***;
```

```
  entity class digger;
  begin
    ref(truck) T;
    real rate;
  end***digger**;

  digger class Ldigger;
  begin
    rate := 240.0;
  LOOP:
    T :- LTQ.coopt;
    hold(T.load/rate);
    T.load := 0.0;
    T.schedule(0.0);
    T :- none;
    if LTQ.length = 0 and S.T is Ltruck
      then S.interrupt(2);
    repeat;
  end***Ldigger***;

  digger class Sdigger;
  begin
    real start;
    rate := 60.0;
  LOOP:
    Q.waituntil(STQ.length > 0 or LTQ.length > 0);
    if STQ.length > 0 then
    begin  ! ***  load small truck ***;
      T :- STQ.coopt;
      hold(T.load/rate);
    end else
    begin  ! ***  load large truck ***;
      start := time;
      T :- LTQ.coopt;
      hold(T.load/rate);
      if interrupted = 0 then T.load := 0.0 else
      begin
        T.load   := T.load-(time-start)*rate;
        T.priority  := 1;
        interrupted := 0;
      end;
    end;
    T.schedule(0.0);
    T :- none;
    repeat;
  end***Sdigger***;

  nextl :- new negexp("next large", 22.0);
  nexts :- new negexp("next small", 10.0);
  Q     :- new condq("Sq");
  STQ   :- new waitq("Struckq");
  LTQ   :- new waitq("Ltruckq");

  S     :- new Sdigger("S");
  S.schedule(0.0);
  new Ldigger("Ldigger").schedule(0.0);
  new Ldigger("Ldigger").schedule(0.0);
  new Ltruck("Ltruck").schedule(0.0);
  new Struck("Struck").schedule(0.0);
  hold(10.0);
end;
```

OUTPUT

```
                       clock time =     10.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************


                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /         a/        b/     seed
next large        0.000    215 negexp       22.000           33427485
next small        0.000    100 negexp       10.000           22276755


                   c o n d i t i o n   q u e u e s
                   *******************************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
Sq                0.000    117     1     0 1.103&-002   113 9.430&-004


                        w a i t   q u e u e s
                        *********************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
Struckq           0.000     97     1     0      0.000    97      0.000
Struckq     *     0.000     97     5     3      1.110    13      0.113

Ltruckq           0.000    227     3     0      0.372   180 1.640&-002
Ltruckq     *     0.000    227    13     8      3.349    55      0.137
```

Remarks on Example 13

In the final listing, just as `entity class truck` was used to define the common attribute `load`, so have we also included `entity class digger` which contains attributes common to `Ldigger` and `Sdigger`. `real rate` is set to the appropriate loading rate for each type of digger at the next level; `ref(truck) T` references a digger's current customer or is `none`.

Besides interrupting an active entity (i.e. one in the event list), one may also interrupt an entity which is passive (i.e. waiting (until) in a queue, blocked on a resource, or passivated). In the latter cases, the interrupted entity will depart from its current explicit or implicit queue (if any) and be placed in the event list immediately behind its interrupter `current` with `interrupted` set.

When a loading by `S` is interrupted we have to be able to compute how much `S` has managed to load before the interrupt. This we do by noting when an interruptible loading operation starts in `start`. When an interrupt occurs, the amount loaded during this operation is given by `(time-start)*rate`.

7.3 Scheduling with now

The programmed strategy in Example 13 is unfair in one respect. Suppose that `S` is currently loading a large truck $\mathtt{T}_n$, another large truck $\mathtt{T}_m$ (with m > n) waits in `LTQ` and then a large digger, `L 1` say, becomes free. Our coding directs `L 1` to coopt $\mathtt{T}_m$. As `L 1` loads faster than `S`, $\mathtt{T}_m$ could well be loaded and away before $\mathtt{T}_n$. It would be fairer to let `L 1` interrupt `S` and take over responsibility for loading $\mathtt{T}_n$. `S` would then naturally get on with $\mathtt{T}_m$. If we alter the coding of the interrupt inside `class Ldigger` to

```
if S.T is Ltruck then S.interrupt(2);
```

then `L 1` will still coopt $\mathtt{T}_m$ and `S` picks up $\mathtt{T}_n$ again! This is because the actions of `L 1`, S and $\mathtt{T}_n$, although all executed at the same clock time, are threaded so

```
          L 1                 S                Tn

 1  S.interrupt(2);
 2  T :- LTQ.coopt;
 3  hold(........);
 4                    T.load := ..;
 5                    T.priority := ..;
 6                    T.schedule(0.0);
 7                    T :- none;
 8                    interrupted := 0;
 9                    Q.waituntil(...);
10                                        LTQ.wait;
```

Action 2) means that `L 1` coopts $\mathtt{T}_m$ since $\mathtt{T}_n$ does not enter LTQ until action 10). It is essential that `S` be given the opportunity to release $\mathtt{T}_n$ and $\mathtt{T}_n$ be allowed to enter `LTQ` (with its increased priority taking it to the front) before `L 1` attempts a coopt. This we can arrange by the insertion of appropriate `hold`s and by using the rather special `real procedure now`. `now` is intended to express urgency. Whereas `E.schedule(0.0)` schedules `E` at the current clock time but as the *last* entity at that time, `E.schedule(now)` is treated as a priority request and `E` actually *preempts* `current`. When the next phase of `E` is over, then the pushed-down previous `current` will be restored. Thus the following alterations do the trick.

```
if S.T is Ltruck then S.interrupt(2);
hold(0.0);
repeat;
```

inside `class Ldigger`, and

```
T.schedule(now);
T :- none;
T.interrupted := 0;
hold(0.0);
Q.waituntil(.....);
```

inside `class Sdigger`. These extra `hold`s produce the following new threading of code

```
          L 1                S               Tn

 1  S.interrupt(2);
 2  hold(........);
 3                   T.load := ..;
 4                   T.priority := ..;
 5                   T.schedule(now);
 6                                      LTQ.wait;
 7                   T :- none;
 8                   interrupted := 0;
 9                   hold(0.0);
10  T :- LTQ.coopt;
11  hold(........);
12                   Q.waituntil(...);
```

EXERCISES 7 (continued)

Exercise 7.4 Cars are transported from a harbour and over the water by a fleet of ferries each with a capacity of 20 cars. The ferry service runs continuously round the clock and a departure is scheduled every hour on the hour. The ferry due to depart at n o'clock arrives at `normal` (mean=20, standard deviation=5) minutes past (n-1) o'clock, unloads, and then takes on board as many cars as it can up until its scheduled departure time. At that time, the ferry will continue to load while the queue is not empty and it is not yet full.

Car arrivals at the ferry point are distributed with different arrival rates according to the time of day. In daytime [06.00 → 18.00), they arrive at a mean rate of 15 per hour; at night time [18.00 → 06.00), they arrive at a mean rate of 9 per hour. Season ticket holders get priority in the queue. The percentage of season ticket holders is 40 in daytime and 25 at night. Find the mean waiting times of season ticket holders and other cars and also the mean delay of the ferry by running your model over a 28 day month. Take as your initial conditions that the first car arrives at time 5 minutes and the first ferry at time 20 minutes.

Model data

Timings in minutes:

unload ferry	`randint`:6→12
load each car	`constant`:1

Exercise 7.5 Customers queueing for service in a shop are characterised by their own individual impatience. They leave the queue after a certain time if not already being served or placed first in the queue. Model the system given that customers arrive at a mean rate of one per minute (`negexp` distributed) and that each service lasts for 40→60 seconds (`uniform`ly distributed). A customer quits the shop after `randint` 120→300 seconds if not already being served, or else he is not the first in the queue.

Run your model for 4 simulated hours assuming that the first customer arrives at time 0.0.

Exercise 7.6 In Example 12, the validity of an interrupt was tested by the interrupter instead of by the recipient. Another way is to make the interrupt and let the recipient decide for itself whether to accept it, ignore it, or reserve it for later attention. For example, in our formulation of the small digger `S` if we altered the interrupt calls to `interrupt(n)` instead of `if` *condition*`then interrupt(n)` then S would receive interrupts when

1. waiting until in `Q` (always accepted),
2. loading a small truck (always ignored),
3. loading a large truck (always accepted if n = 1, always ignored if n = 2).

Recode the whole problem according to this suggestion.

Chapter 8

Summing up

8.1 Some loose ends

This primer has introduced the Demos facilities (resource types, queues, distributions, etc.) by a sequence of examples which usually illustrate the point at hand but not much more. Here we spend a little extra time on periodic reporting, starting up simulations and closing them down.

Periodic reports can be issued rather neatly using an object of such a class as `reporter` below.

```
entity class reporter(t); real t;
begin
  hold(t);
  report;
  reset;
  repeat;
end***reporter***;

ref(reporter) R;

R :- new reporter("daily report", 24.0);
R.schedule(0.0);
```

Working in hourly units, `R` will issue a full report on the last day's facility usage every 24 hours. `report` is a global Demos procedure which prints standard reports on the usage of all the Demos facilities created by the user. The report covers their usage since the object's creation (if not reset) or since its last reset. (The Demos system itself calls this procedure at the end of each simulation run.) `reset` is another global Demos procedure which resets all Demos facilities created by the user so that they now collect afresh over the next time period.

`procedure reset` is also useful in the warm start situation. We have usually started up our models by letting the first arrivals fall due at time 0.0. It takes some time before enough entities have worked their ways through the system for it to have settled down to approximately normal working conditions. The cold start naturally biases the facility reports as initially there will be little interference between entities. Using `reset` it is easy to let the system settle down and then gather data over the desired time slot. We merely change code in the Demos block from the usual

```
hold(simulation period);
```

into

```
hold(warm up period);
reset;
hold(time slot);
```

The final report covers a period of duration 'time slot'. The length of the warm up period is usually chosen by some rule of thumb rather than by a precise method. Tocher advocated twice the expected length of the longest periodic entity in more complex models—which is suitably vague about both expected and complex—but he was so experienced and so filled with insight that he would always get it right. However Shannon [90, pages 183-186] is particularly informative on this topic and gives several further references.

Closing down a simulation is less of a problem in Demos as the Demos block can itself be treated as an entity and acquire resources, wait until in queues, etc. This neat and very effective idea was borrowed directly from Simula; it was used in example 5 and exercises 3.13, 4.5, and 4.12. See further remarks in Appendix B.

8.2 Demos facilities not covered

Besides the global procedures `report` and `reset` mentioned above, there are also procedures `report` and `reset` local to classes `res`, `bin`, and the various queues — `condq`, `queue`, and `waitq`. We are able to report and reset all user created facilities, select those of one class (e.g. all `res` objects) and even select single items (e.g. a single `bin`).

There is a class `empirical` which can be used to represent empirical data tabulated as a cumulative probability function.

In connection with the design of experiments, the default values for distribution seeds can be overridden (useful for antithetic drawings) and the Demos-defined first seed value (33427485 — and hence all the other seeds) can be changed. In fact, even the random number generators can be replaced.

There are also several snapshot routines which can be called to detail individual entities either waiting (until) in queues, awaiting resources, or scheduled in the event list.

There is a `class accumulate` which parallels `class tally`, but for time dependent variables. And there is a `class regression` too.

Finally, `class entity` contains certain additional attributes.

1. `boolean procedure avail` which returns `true` if the entity is not coopted

2. `boolean procedure idle` which returns `true` if the entity is not in the event list

3. `ref(entity)procedure nextev` which returns `none` if the entity is idle or last in the event list, otherwise a reference to the next entity in the event list, and

4. `real procedure evtime` (which returns the entity's scheduled time if it lies in the event list. Otherwise, a call on `evtime` causes a run time error: if in doubt, check using `if not E.idle then .....`).

8.3 The Simula implementation

It is worth remarking on how much Demos owes to Simula. Although we have certainly borrowed ideas from other languages (resource types and reporting from GPSS, conditions and activity diagrams from CSL and ECSL), Demos would not exist without the inspiration of its host language which positively invites the user to write programs in process (= entity) style. All we have done is add a few user-oriented bells and whistles.

Demos is implemented as a Simula context (prefixed by `SIMSET`, but not by `SIMULATION`) and extends over roughly 2000 lines. Along with language design, detailed documentation, and testing, Demos was approximately a 9 person-month project. The resulting system is portable and has proved easy to experiment with, extend, alter and maintain.

Implementing a Demos compiler from scratch was out of the question because Demos is Simula plus, and Simula is itself a 10 person-year project (or rather was in the 1970's). However a Demos compiler would have certain advantages. It could, for example, give error messages written in Demos (rather than Simula) terminology (although it must be admitted that Simula compilers are pretty good in this respect anyway), detect the possibility of deadlock at compile time, accept a Demos program written with waits until and itself insert the correct calls on `Q.signal`, give resource usage cross-reference lists, do away with the need for explicitly titling every Demos object (as in `new boat("boat")`, etc.). Without a compiler, the same effect can be achieved by using a pre-processor, which is much easier to write but more expensive to run.

Clementson [23, 22] and Mathewson [57] have gone further than this and written *program generators*; inter-active programs which ask the user about his model and then actually write the code. Clementson's system produces code for ECSL, but Mathewson's stores the structure of the model and can then be persuaded to produce code for several different languages—FORTRAN, GASP, ECSL, Simula, Demos ... —thus confirming that there is indeed a central notation capable of capturing different styles of modelling[1]. Thus he could develop a model in Simula, say,

[1] This was very perceptive for its time. For example although Tocher decomposed his models as interacting objects, he then implemented them by splitting them up into individual activities together with a controller

on a large machine, and run it in GASP on a small machine. It also reinforces the point that activity diagrams can be used as a basis for writing simulations in any simulation programming language.

There are many advantages to coding Demos in Simula. Firstly, Simula is widely implemented and to a good standard. The implementors of Simula systems meet regularly in the SSG (Simula Standards Group). The hope being that this would ensure that Simula systems are compatible now and will remain so in the future. Experience with porting Demos has been fairly trouble free, only the CDC implementations [26] giving rise to non-trivial problems. Demos was implemented superbly on DEC System 10 hardware [28] and has been ported to DEC System 20, IBM 360/370 [68], and ICL System 4 [78], and ICL 2900 hardware with no modifications at all. The UNIVAC 1100 implementation [69] does not yet[2] support virtual labels, so that version of Demos has to make do without REPEAT. Both the NDRE [70] and CDC [26] Simula implementations quote key words. Once that hurdle has been accounted for, the NDRE version supports neither virtual labels nor functions returning references, so that `repeat` is out, and the function `ref(entity) procedure nextev` has had to be rewritten as a procedure. The CDC [26] Simula compiler still does not treat virtual quantities correctly, and the Demos code for the reset and report routines had to be “bent” to fit. Altogether, the experience has not been too bad (have you tried porting FORTRAN programs from one machine to another?). The situation improved in the 1980’s when Lund Software released their brilliantly engineered Simula implementations [54] making identical Simula compilers available on an even wider range of hardware.

Secondly, Simula’s object and context features are considerably ahead of anything offered by other languages [3]. It is easy to ‘see’ how to implement Demos facilities as Simula objects. The context feature enables an implementation to progress in an orderly fashion layer by layer, each new step adding in a few new interrelated ideas. Simula has very strict security and consistency checks so that many mistakes are picked up as soon as possible at compile time.

Thirdly, any ideas not built into Demos can be added straightforwardly (which eases the designer’s dilemma — if in doubt, leave it out!). One easy addition (should the arguments of chapter 6.3 fail to be convincing) would be to put in one’s own `snoopy`. But the area in which advanced users will most probably want to make changes is in the `report` routines. Here it is sufficient to extend the facility concerned and define one’s own `report` routine, e.g.

```
    res class my_res;
    begin
```

(this was pre-assembler, he coded in octal). When I knew him (19670’s and onwards, he considered objects to be beyond the pale and I could not convince him that he had been using them for 20 or so years.

[2] That was 1978.

[3] That too wwas 1978—not much has changed, has it.

```
    procedure report;.........;
  end***my_report***;
```

and then work with my_res objects instead of res objects. The newly written report replaces the standard Demos report thanks to Simula's virtual mechanism. For further detail on virtual, see Birtwistle et al [6, chapter 4] or [5].

Finally, Demos is not the end of the road. A user should proceed in standard Simula fashion to develop a more specialised contexts for areas of particular interest, e.g. here is a the skeleton of a link level context for simulating the X25 long haul network protocol. It contains definitions for frames, and sender, receiver, and retransmitter processes (whose common characteristics have been gathered together in a Real Time process declaration). We also predefine a network node each of which will have n duplex links to other network nodes. Each node has a trio of processes looking after each network link. Finally we declare a network composer which automatically buids a specific configuration.

```
  Demos class X25_link_level;
  begin
    entity class frame; ................ ;

    entity class RT_process; ........... ;

    RT_process class sender; ........... ;

    RT_process class receiver; ......... ;

    RT_process class retransmitter ; .... ;

    entity class node(nLinks); ......... ;
    begin
      ref(qeueue) array RTQ, INQ, OUTQ (1:nLinks);
      ref(RT_process) S, R, RT (1:nLinks);
      integer k;

      for k := 1 step 1 until n Links do
      begin
        RTQ(k)  :- new queue("retransmit queue");
        INQ(k)  :- new waitq("frames in");
        OUTQ(k) :- new waitq("frames out");

        S(k)    :- new sender("S");
        R(k)    :- new receiver("R");
        RT(k)   :- new retransmitter("RT");

        S(k).schedule(0.0);
        R(k).schedule(0.0);
        RT(k).schedule(0.0);
      end;
    end***node***;

    procedure read_network; ............ ;

  end***X25_link_level***;
```

For examples of well-used contexts on Simula, study Lie's SIMWAP [51], Rogeberg's TETRASIM [86]; and not least Vaucher's GPSSS [104] which paved the way for Demos. Merci beaucoup, Jean.

Several other simulation packages have been implemented as extensions to general programming languages. The two best known to the author are SIMONE and ALGOLSIM. SIMONE (see Kaubisch et al. [42]) is an extension to PASCAL produced by extending a PASCAL compiler. Its design was heavily influenced by Simula. ALGOLSIM (Shearn [92]) is an extension to ALGOL 68. Both papers are well written, and it is interesting to compare ALGOL 68, PASCAL and Simula as extendible languages.

Virjo [106] gives a very thorough comparison of GPSS, SIMSCRIPT, and Simula. The paper includes several examples coded in all three languages and run on a variety of machines.

8.4 The distribution of Demos

As stated in the preface, Demos is an ordinary Simula program and will run on any computer that supports Simula (see references [26, 28, 68, 69, 78, 87, 54, 27]). The Demos Reference Manual ([5]) gives full documentation of the complete Demos system, and includes a Simula source listing of Demos as an appendix. It is the author's hope that this will lead to Demos being read and improved by others, and to its being used in education to show what the components of a discrete event simulation language are and how they fit together. The Demos system is maintained by the author who will be pleased (?) to receive constructive criticisms and error reports.

The Demos system (both source code and reference manual) are available from the author:

Graham Birtwistle,
School of Computer Science,
University of Sheffield,
Regent Court, 211 Portobello Street,
Sheffield S1 4DP, England.
Tel: (+44) 114 222 1842
Net: graham@dcs.shef.ac.uk Web: www.dcs.shef.ac.uk/∼graham

8.5 Coda

There is still much to learn and this primer has merely scratched the surface. Fishman [30] and/or Shannon [90] (which both contain many further references) are ways into the important topics of experimental design and input/output analysis. Franta [32] combines an explanation of Simula together with an introduction to the statistical aspects of simulation. Note especially his treatment of the regenerative method. Kreutzer [49], Poole and Szymankiewicz [79], Pritsker and Kiviat [85] and Schriber [88, 89] contain many simulation examples which can be used to build up modelling experience if you are not on an actual project. References [88, 89] in particular contain a good selection of straightforward examples and exercises.

Finally, note that when you have managed to work your way through this book and complete the exercises then ... then you can keep it on your bookshelf. For it is intended as an introductory teaching text and not as a reference manual. Once a certain competence in Demos has been reached, then the Demos Reference Manual [5] and the appropriate Simula User's Manual [26, 28, 68, 69, 78, 87, 54, 27] should prove more useful.

Bibliography

[1] J. C. M. Baeten. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, Cambridge, 1990.

[2] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, Cambridge, 1990.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, Hemel Hempstead, UK, 1988.

[4] G. Birtwistle. *DEMOS — a system for discrete event modelling on Simula.* Macmillen, London, 1979.

[5] G. Birtwistle. The Demos Implementation Guide and Reference Manual. Technical Report, 260 pages, Computer Science Department, University of Calgary, 1983.

[6] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula* **begin**. Studentlitteratur, Lund, Sweden, 1973.

[7] G. Birtwistle, P. A. Luker, G. Lomow, and B. Unger. Process style packages for discrete event modelling: Experience from the transaction, activity, and event approaches. *Transactions of The Society for Computer Simulation*, 2(1):25–56, 1985.

[8] G. Birtwistle, R. Pooley, and C. Tofts. Characterising the Structure of Simulation Models in CCS. *Transactions of the Society for Computer Simulation*, 10(3):205–236, 1993.

[9] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 1: πDemos. *Transactions of the Society for Computer Simulation*, 10(4):299–333, 1993.

[10] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 2: μDemos. *Transactions of the Society for Computer Simulation*, 11(4):303–336, 1994.

[11] G. Birtwistle and C. Tofts. Relating Operational and Denotational Descriptions of πDemos. *Simulation Practice and Theory*, 5(1):1–33, 1997.

[12] G. Birtwistle and C. Tofts. Getting Demos Models Right. Part I: Practice. *Simulation Practice and Theory*, 8, 2001.

[13] G. Birtwistle and C. Tofts. Getting Demos Models Right. Part II: ... and Theory. *Simulation Practice and Theory*, 8, 2001.

[14] P. Bratley, B. Fox, and L. Schrage. *A Guide to Simulation*. Springer Verlag, New York, 1987.

[15] A. Cave Brown. *C*. Macmillan, New York, 1987.

[16] W. Burge. *Recursive Programming Techniques*. Addison-Wesley, New York, 1975.

[17] J. N. Buxton and J. G. Laski. Control and Simulation Language. *Computer Journal*, 5(3), 1962.

[18] CACI. *SimscriptII.5*. CACI International Inc, Web: `www.caciasl.com`, 1963.

[19] A. S. Carrie. *Simulation of Manufacturing Systems*. J. Wiley and Sons, 1988.

[20] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[21] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[22] A. Clementson. Extended Control and Simulation Language. Technical Report 94, University of Birmingham, 1973.

[23] A. T. Clementson. Extended Control and Simulation Language. *The Computer Journal*, 9(3):215–220, 1985.

[24] O-J. Dahl. Hierarchical program structures. In E. W. Dijkstra O-J. Dahl and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[25] O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula 67 Common Base Language*. Norwegian Computing Center, Oslo, Norway, 1970.

[26] Control Data. Control Data 6400/6500/6600 Computer Systems Simula Reference Manual. Technical report, Control Data Corporation, 1969.

[27] Norsk Data. TPH Simula Reference Manual (NORD 10). Technical report, Norsk Data AS, Oslo, Norway, 1977.

[28] DEC. DEC System 10 Simula Language Handbook. Part 1: User's Guide. (Implementation modified to run on the DEC 20), Swedish Defense Research Establishment, Stockholm, 1975.

[29] J. B. Evans. *Structures of Discrete Event Simulation.* Ellis Horwood, London, 1988.

[30] G. S. Fishman. *Concepts and Methods in Digital Discrete Event Simulation.* John Wiley and Sons, New York, 1973.

[31] G. S. Fishman. *Principles of Discrete Event Simulation.* Wiley Interscience, New York, 1978.

[32] W. R. Franta. *The Process View of Simulation.* North Holland, 1978.

[33] A. T. Fuller. The period of pseudo-random numbers generated by Lehmer's congruential method. *Computer Journal*, 19(2):173–177, 1976.

[34] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[35] H. Rohlfing. Simula: Eine Einfuhrung. Technical report, Bibliographisches Institutt, Mannheim, 1973.

[36] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Mass, 1988.

[37] M. Hennessy. *The Semantics of Programming Languages.* John Wiley, Chichester, England. Now out of print but available via Matthew's home page under Teaching — The Semantics of Programming Languages — but minus chapter 6 on CSP semantics at `www.cogs.susx.ac.uk/users/matthewh/teaching.html`, 1990.

[38] P. Hills. SIMON—a Simulation Language in ALGOL. In S. M. Hollingdale, editor, *Simulation in OR*. English Universities Press, London, 1965.

[39] P. Hills. An Introduction to Simulation using Simula. NCC Publication S-55, Norwegian Computing Center, Oslo, Norway, 1972.

[40] P. Hills. HOCUS—Basic and Advanced Manuals. Technical report, PE Group, Egham, Surrey, England, 1976.

[41] P. Hills and G. Birtwistle. SIMON75 Reference Manual. Technical report, P. Hills, Shaldon, Devon, England, 1976.

[42] W. H. Kaubusch, R. H. Perrott, and C. A. Hoare. Quasi-parallel programming. *Software Practice and Experience*, 6(4):341–356, 1976.

[43] P. J. Kiviat, R. Villanueva, and H. M. Markovitz. *The SIMSCRIPT II Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1968.

[44] D. E. Knuth. *The Art of Computer Programming: Volume 2*. Addison Wesley, Reading, Massachusetts, 1969.

[45] D. E. Knuth and J. L. McNeley. A Formal Definition of SOL. *IEEE Transactions of Electronic Computers*, EC-13:409–14, 1964.

[46] D. E. Knuth and J. L. McNeley. SOL—A Symbolic Language for General Purpose Simulation. *IEEE Transactions of Electronic Computers*, EC-13:401–08, 1964.

[47] D. E. Knuth and J. N. Merner. ALGOL 60 Confidential. *CACM*, 4:268–272, 1961.

[48] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[49] W. Kreutzer. *System simulation — programming styles and languages*. Addison Wesley, 1986.

[50] A. M. Law and J. S. Kelton. *Simulation Modelling and Analysis*. McGraw Hill, New York, 1982.

[51] A. Lie, K. Elgsaas, and J. Evansmo. SIMWAP—a Computer Package for Warehouse Planning. NCC Publication S-42, Norwegian Computing Center, Oslo, Norway, 1973.

[52] N. Lynch. Simulation Techniques for Proving Properties of Real-Time Systems. In S. H. Son, editor, *Advances in Real Time Systems*, pages 299–322, Englewood Cliffs, New Jersey, 1995. Prentice Hall.

[53] N. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[54] B. Magnusson and P. Holm. Using Lund Simula on Unix Systems. Technical report, Lund Software House, University of Lund, Sweden, 1995.

[55] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

[56] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[57] S. Mathewson. Simulation Program Generators. *Simulation*, 23(6):181–189, 1974.

[58] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[59] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[60] R. Milner. Elements of Interaction (Turing Award lecture). *Communications of tne ACM*, 16(1):78–89, January, 1993.

[61] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes: Part I. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, 1989.

[62] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes: Part II. Technical Report ECS-LFCS-89-86, Computer Science Department, University of Edinburgh, 1989.

[63] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Mass., 1991.

[64] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[65] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In J. W. Klop J. C. M. Baeten, editor, *CONCUR '90*, number 458 in LNCS. Springer-Verlag, 1990.

[66] F. G. Moller and P. Stevens. The Edinburgh Concurrency Workbench, Version 7. Technical Report, Computer Science Department, University of Edinburgh, 1991.

[67] R. E. Nance. A History of Discrete Event Simulation Programming Languages. In T. J. Bergin and R. G. Gibons, editors, *History of Programming Languages*, pages 369–427. ACM Press and Addison-Wesley Publishing Company, 1996.

[68] NCC. Simula for IBM 360/370: User's Guide. NCC Publication S-24, Norwegian Computing Center, Oslo, Norway, 1971.

[69] NCC. UNIVAC EXEC-8 Simula User's Guide. NCC Publication S-36, Norwegian Computing Center, Oslo, Norway, 1971.

[70] NDRE. NDRE Simula Implementation User's Manual. NDRE Publication S-370 (3rd Edition), Norwegian Defence Research Establishment, Kjeller, 1977.

[71] K. Nygaard and O-J. Dahl. The Development of the Simula Languages. In A. Wexelblatt, editor, *ACM Conference* A History of Programming Languages. Academic Press, 1981.

[72] M. Ohlin. Next Random—a Method of Fast Access to Any Number in the Random Generator Cycle. *Simula Newsletter*, 6(2):18–20, 1977.

[73] J. Palme. How I fought with Software and Hardware and Succeeded. *Software Practice and Experience*, 8(1):77–81, 1978.

[74] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK, 1991.

[75] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, London, 1986.

[76] G. D. Plotkin. A Structural Approach to Operational Semantics. Research Report DAIMI FN–19, Computer Science Department, Aarhus University, Denmark. Available on the web in postscript from Gordon's Publications (ranked at number 2 (number 1 must be good!)) `www.dcs.ed.ac.uk/home/gdp/publications/`, 1981.

[77] G. D. Plotkin. An Operational Semantics for CSP. Research Report CSR–114–82, Computer Science Department, Edinburgh University, Scotland, 1982.

[78] PLU. System 4 Multijob Simula Programmer's Manual (ICL Systems 4). Program Library Unit, Edinburgh University, 1980.

[79] G. T. Poole and J. Z. Szymankiewicz. *Using Simulation to Solve Problems*. McGraw Hill, London, 1977.

[80] R. Pooley. An Introduction to Programming in Simula. Originally published by Blackwell but now out of print, Now available from Rob's web site at Herriott Watt University, 1990.

[81] R. Pooley. Towards a standard for hierachical process oriented discrete event simulations. *Transactions of the Society for Computer Simulation*, 8(1):1–20,21–32,33–42, 1991.

[82] R. Pooley. Formalising the Description of Process Based Simulation Models. In P. Luker, editor, *Proceedings of the Summer Computer Simulation Conference, Reno, Nevada*, July, 1992.

[83] R. Pooley. The Derivation of Functional Properties of Process Based Simulation Models. In P. Maceri, editor, *Proceedings of the EUROSIM Simulation Congress, Capri*, September, 1992.

[84] A. A. B. Pritzker. *Introduction to Simulation and SLAMII.* Halstead Book Press, New York, 1984a.

[85] A. A. B. Pritzker and P. J. Kiviat. *Simulation with GASP II.* Prentice-Hall, New York, 1969.

[86] T. Rogeberg. Simulation and Simula as applied to the design and analysis of telephone systems. NCC Publication S-30, Norwegian Computing Center, Oslo, Norway, 1970.

[87] S-PORT. S-PORT—the portable Simula Project. Implementations based on the Norwegian Computing Center Program Portable Core for Honeywell, Prime, Univac, and VAX computers, Norwegian Computing Center, 1982.

[88] T. J. Schriber. *Simulation using GPSS.* John Wiley and Sons, New York, 1974.

[89] T. J. Schriber. *Simulation using GPSS/H.* John Wiley and Sons, New York, 1991.

[90] R. E. Shannon. *Systems Simulation: the art and science.* Prentice Hall, Englewood Cliffs, 1975.

[91] A. C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 1974.

[92] D. C. S. Shearn. Discrete Event Simulation in ALGOL 68. *Software Practice and Experience*, 5(4):279–293, 1975.

[93] C. Stirling. Modal Logics for Communicating Systems. *Theoretical Computer Science*, 49:311–347, 1987.

[94] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, number 491 in LNCS, pages 2–20. Springer-Verlag, 1991.

[95] C. Stirling. Modal and Temporal Logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, vol 2*, pages 477–563. Oxford University Press, 1992.

[96] C. Stirling. Modal and Temporal Logics for Processes. Tech Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1992.

[97] C. Stirling. Modal and Temporal Logics for Processes. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, Berlin, 1996. Springer Verlag.

[98] C. Stirling and D. Walker. CCS, Liveness, and Local Model Checking in the Linear Time Mu-Calculus. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 166 – 178. Springer-Verlag, 1990.

[99] K. Tocher. *The Art of Simulation.* The English Universities Press, London, 1963.

[100] C. Tofts. A Synchronous Calculus of Relative Frequency. In J. W. Klop J. C. M. Baeten, editor, *CONCUR '90*, number 458 in LNCS. Springer-Verlag, 1990.

[101] C. Tofts. Process Semantics for Simulation. Technical Report, Department of Mathematics and Computer Science, University of Swansa, Swansea, Wales, 1993.

[102] C. Tofts. Processes with Probability, Priority and Time. *Formal Aspects of Computer Science*, 6(5):536–564, 1993.

[103] C. Tofts and G. Birtwistle. A denotational semantics for a process-based simulation language. *ACM Transactions on Modelling and Simulation*, 8(3):281–305, 1998.

[104] J. Vaucher. Simulation Data Structures using Simula 67. *Proc Winter Simulation Conference*, pages 255–260, 1971.

[105] J. Vaucher. A Generalised Wait-until Algorithm for General Purpose Simulation Languages. *Proc Winter Simulation Conference*, pages 177–183, 1973.

[106] A. Virjo. A Comparison of some Discrete Event Simulation Languages. *NORDDATA 72 Conference, Helsinki*, 1972.

[107] D. Walker. Introduction to a Calculus of Communicating Systems. Technical Report ECS-LFCS-87-22, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1987.

[108] J. M. Wing and C. Gong. Testing and Verifying Concurrent Objects. *Journal of Parallel and Distributed Computing*, 17:164–182, 1993.

[109] B. Ziegler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, New York, 1976.

[110] B. Ziegler. *Theory of Modelling and Simulation.* J. Wiley and Sons, New York, 1976.

Appendix A

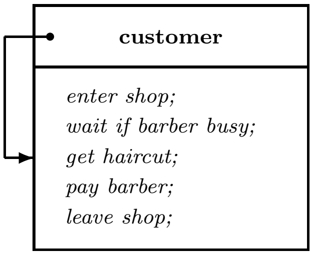
Answers to exercises

Answers to exercises 2

2.1 Customer declaration and object

These objects have no parameters and no local declarations.

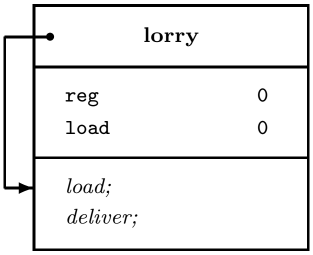
```
class customer;
begin
  enter shop;
  wait if barber busy;
  get haircut;
  pay barber;
  leave shop;
end***customer***
```



2.2 Lorry declaration and object

These objects have local data values which are automatically set to zero (if arithmetic) or `false` if boolean.

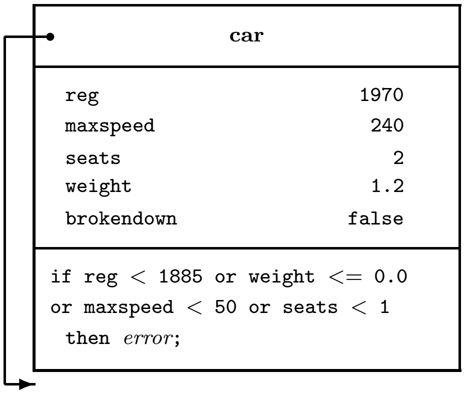
```
class lorry;
begin
  integer reg, load;
  load;
  deliver;
end***lorry***
```



2.3 Car declaration and object

These objects have parameters (which are initialised on object creation) and local variables.

```
class car(reg,maxspeed,seats,weight);
 integer reg,maxspeed,seats;
 real weight;
begin
 boolean brokendown;
 if reg < 1885 or weight <= 0.0
  or maxspeed < 50 or seats < 1
 then error;
end***car***
```



car

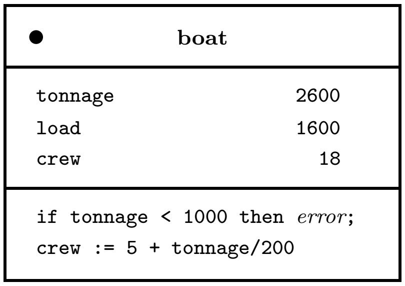
```
reg             1970
maxspeed         240
seats              2
weight           1.2
brokendown     false
```

```
if reg < 1885 or weight <= 0.0
or maxspeed < 50 or seats < 1
 then error;
```

The object is depicted after its body actions have been completed, with its LSC “beyond range”. In this circumstance, it is usual to show the LSC as just a (large) dot as below.

2.4 Boat declaration and object

```
class boat(tonnage); integer tonnage;
begin
 integer load, crew;
 if tonnage < 1000 then error;
 crew := 5 + tonnage/200
end***boat***
```



● **boat**

```
tonnage         2600
load            1600
crew              18
```

```
if tonnage < 1000 then error;
crew := 5 + tonnage/200
```

2.5 Remote accessing analogue

A straightforward example is one’s address. When asked the question “Where do you live?”, “At number 12” is an adequate answer if already on that particular street; “Number 12, Main Street”, if in one’s home town, “Number 12, Main Street, Ugglebarnby’ if out of town, etc., etc.

2.6 Class order and its sub-classes

```
class order;
begin
  integer number, arrival;
  real setup_time, processing_time;
end***order***;

order class batch;
begin
  integer size;
end***batch***;

order class single;
begin
  real weight, finishing_time;
end***single***;

single class plate;
begin
  real length, width;
end***plate***;
```

2.7 Harbour context

There are, of course, several “solutions” to this problem depending upon what you have in mind. A typical skeleton context could be

```
class Harbour;
begin
  class crane............;

  class boat.............;
  boat class cargo........;
  boat class passenger....;
  boat class tanker.......;
  boat class tug..........;

  class container.........;
  class tide.............;
  .........................
end***harbour***;
```

which we would use to prefix a user program thus

```
external class Harbour;

Harbour
  begin
    ref(crane) c1, c2;
    ref(tug) t1, t2, t3;
    ..................
  end;
```

if `Harbour` is externally compiled in the local directory, and by something like

```
external class Harbour = "/usr/profs/graham/book/examples/Harbour.atr";

Harbour
  begin
    ref(crane) c1, c2;
    ref(tug) t1, t2, t3;
    ...................
  end;
```

when the externally compiled version lies elsewhere.

Answers to exercises 3

3.1 Decomposition analogy

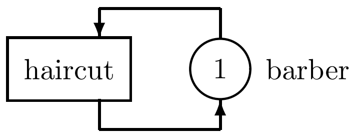
The Macbeth analogy allows one player per role, but in a play with a chorus we can have several actors with the same lines.

In the world of music, only the conductor has the complete score for a symphony (which corresponds to a full Demos trace). Different part scores are produced for 1st. violins, 2nd. violins, etc. Each such score gives only the one part and a count of the bars when its players are silent (which corresponds to a `hold`). Note that in this case, we have several players per role playing in unison. In simulation models, several objects may well have the same class declaration, but they can be executed at different speeds and start at different times.

3.2 Barber's shop trace

time	customer	current action	time of next event
0.0	C1	arrive	
	C1	request 1 barber	
	C1	seize 1 barber	
	C1	start haircut	15.0
15.0	C1	release 1 barber	
	C1	quit	****
20.0	C2	arrive	
	C2	request 1 barber	
	C2	seize 1 barber	
	C2	start haircut	35.0
35.0	C3	arrive	
	C3	request 1 barber	
	C2	release 1 barber	
	C2	quit	****
	C3	seize 1 barber	
	C3	start haircut	50.0
40.0	C4	arrive	
	C4	request 1 barber	
50.0	C3	release 1 barber	
	C3	quit	****
	C4	seize 1 barber	
	C4	start haircut	65.0
65.0	C4	release 1 barber	
	C4	quit	****

Customer activity diagram



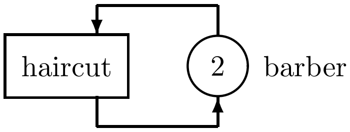
Customer declaration

```
class customer;
begin
   acquire 1 barber;
   get haircut;
   release 1 barber;
end***customer***;
```

3.3 2-man barber's shop trace

time	customer	current action	time of next event
0.0	C1	arrive	
	C1	request 1 barber	
	C1	seize 1 barber	
	C1	start haircut	15.0
15.0	C1	release 1 barber	
	C1	quit	****
20.0	C2	arrive	
	C2	request 1 barber	
	C2	seize 1 barber	
	C2	start haircut	35.0
35.0	C3	arrive	
	C3	request 1 barber	
	C3	seize 1 barber	
	C3	start haircut	50.0
	C2	release 1 barber	
	C2	quit	****
40.0	C4	arrive	
	C4	request 1 barber	
	C4	seize 1 barber	
	C4	start haircut	55.0
50.0	C3	release 1 barber	
	C3	quit	****
55.0	C4	release 1 barber	
	C4	quit	****

Customer activity diagram



Customer declaration

Exactly as in exercise 3.2 above.

3.4 Factory trace

time	van	current action	time of next event
0.0	V1	arrive	
	V1	request w'bridge	
	V1	seize w'bridge	3.0
	V1	start weighing	
1.0	V2	arrive	
	V2	request w'bridge	
3.0	V1	release w'bridge	
	V1	start unloading	23.0
	V2	seize w'bridge	
	V2	start weighing	6.0
6.0	V2	release w'bridge	
	V2	start unloading	26.0
23.0	V1	request w'bridge	
	V1	seize w'bridge	
	V1	start weighing	26.0
24.0	V3	arrive	
	V3	request w'bridge	
25.0	V4	arrive	
	V4	request w'bridge	
26.0	V2	request w'bridge	
	V1	release w'bridge	
	V1	quit	****
	V3	seize w'bridge	
	V3	start weighing	29.0
29.0	V3	release w'bridge	
	V3	start unloading	49.0
	V4	seize w'bridge	
	V4	start weighing	32.0
32.0	V4	release w'bridge	
	V4	start unloading	52.0
	V2	seize w'bridge	
	V2	start weighing	35.0
35.0	V2	release w'bridge	
	V2	quit	****

Note: V3 and V4 are currently unloading.

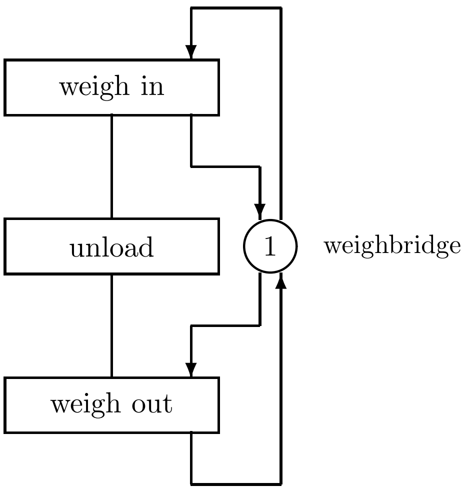
Van activity diagram

In our model, vans are served on the first-come, first-served principle (FCFS). See particularly at time = 26.0 when `V3` heads the waiting queue for the weighbridge. `V2` and `V4` are also blocked at the weighbridge when `V1` releases it. If vans moving out were given priority, `V2` would have been the next to acquire the weighbridge.

3.5 Barber's shop program

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) barbers;
```

```
class van;
begin
  acquire 1 weighbridge;
  weigh in;
  release 1 weighbridge;
  unload;
  acquire 1 weighbridge;
  weigh out;
  release 1 weighbridge;
end***van***;
```



```
    entity class customer;
    begin
      barbers.acquire(1);
        hold(15.0);
      barbers.release(1);
    end***customer***;


    barbers :- new res("barbers", 1);

    new customer("c").schedule( 0.0);
    new customer("c").schedule(20.0);
    new customer("c").schedule(35.0);
    new customer("c").schedule(40.0);

    hold(65.0);
  end;
```

3.6 Factory program

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res)weighbridge;
    entity class van;
    begin
      weighbridge.acquire(1);
        hold(3.0);
      weighbridge.release(1);

      hold(20.0);

      weighbridge.acquire(1);
        hold(3.0);
      weighbridge.release(1);
    end***van***;

    weighbridge :- new res("weighbridge", 1);
    new van("V").schedule(0.0);
    new van("V").schedule(1.0);
    new van("V").schedule(24.0);
    new van("V").schedule(25.0);

    hold(40.0);
  end;
```

3.7 Event list trace of factory program

time	event list					weighbridge
0.0	D(0.0)					
0.0	D(0.0)	V1(0.0)				
0.0	D(0.0)	V1(0.0)	V2(1.0)			
0.0	D(0.0)	V1(0.0)	V2(1.0)	V3(24.0)		
0.0	D(0.0)	V1(0.0)	V2(1.0)	V3(24.0)	V4(25.0)	
0.0	V1(0.0)	V2(1.0)	V3(24.0)	V4(25.0)	D(40.0)	
1.0	V2(1.0)	V1(3.0)	V3(24.0)	V4(25.0)	D(40.0)	
3.0	V1(3.0)	V3(24.0)	V4(25.0)	D(40.0)		V2
3.0	V1(3.0)	V2(3.0)	V3(24.0)	V4(25.0)	D(40.0)	
3.0	V2(3.0)	V1(23.0)	V3(24.0)	V4(25.0)	D(40.0)	
6.0	V2(6.0)	V1(23.0)	V3(24.0)	V4(25.0)	D(40.0)	
23.0	V1(23.0)	V3(24.0)	V4(25.0)	V2(26.0)	D(40.0)	
24.0	V3(24.0)	V4(25.0)	V2(26.0)	V1(26.0)	D(40.0)	
25.0	V4(25.0)	V2(26.0)	V1(26.0)	D(40.0)		V3
26.0	V2(26.0)	V1(26.0)	D(40.0)			V3, V4
26.0	V1(26.0)	D(40.0)				V3, V4, V2
26.0	V1(26.0)	V3(26.0)	D(40.0)			V4, V2
26.0	V3(26.0)	D(40.0)				V4, V2
29.0	V3(29.0)	D(40.0)				V4, V2
29.0	V3(29.0)	V4(29.0)	D(40.0)			V2
29.0	V4(29.0)	D(40.0)	V3(49.0)			V2
32.0	V4(32.0)	D(40.0)	V3(49.0)			V2
32.0	V4(32.0)	V2(32.0)	D(40.0)	V3(49.0)		
32.0	V2(32.0)	D(40.0)	V3(49.0)	V4(52.0)		
35.0	V2(35.0)	D(40.0)	V3(49.0)	V4(52.0)		
40.0	D(40.0)	V3(49.0)	V4(52.0)			
.....	....	....	....	....	....	.....

3.8 Entity generation

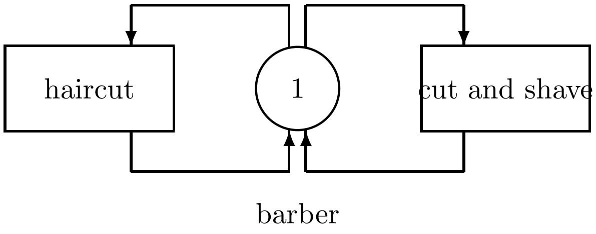
See example 2, page 40.

3.9 Normal distribution

```
random class normal(mean, sig); real mean, sig;
begin
  real procedure sample;
  begin
    real sum;
    integer k;

    for k := 1 step 1 until 12 do
      sum := sum + next;
    sample := mean + (sum-6.0)*sig;
  end***sample**;

  if sig < 0.0 then error;
end***normal***;
```

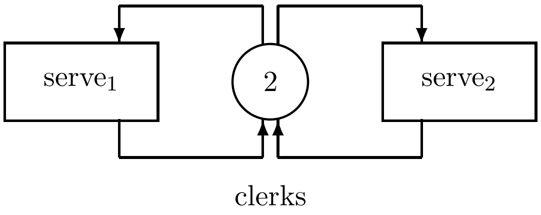
This algorithm is based upon the Central Limit Theorem. For better methods, see Fishman [33, p.128] or Shannon [37, p.362].

3.10 Barber's shop: two customer types



```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) barbers;
    ref(rdist) c1, c2, s1, s2;

    entity class cut;
    begin
      new cut("cut").schedule(c1.sample);
      barbers.acquire(1);
        hold(s1.sample);
      barbers.release(1);
    end***cut***;

    entity class both;
    begin
      new both("both").schedule(c2.sample);
      barbers.acquire(1);
        hold(s2.sample);
      barbers.release(1);
    end***both***;

    c1       :- new negexp("c1", 0.025);
    c2       :- new negexp("c2", 0.01666667);
    s1       :- new uniform("s1", 12.0, 24.0);
    s2       :- new uniform("s2", 20.0, 36.0);
    barbers  :- new res("barbers", 1);

    new cut("cut").schedule(0.0);
    new both("both").schedule(10.0);

    hold(480.0);
  end;
```

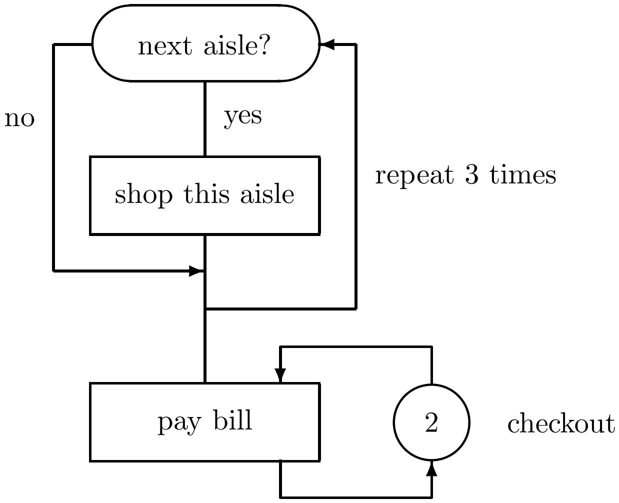
3.11 Tool check out



```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) clerks;
    ref(rdist) nextm1, nextm2, serve1, serve2;

    entity class mech1;
    begin
      new mech1("m1:").schedule(nextm1.sample);
      clerks.acquire(1);
        hold(serve1.sample);
      clerks.release(1);
    end***mech1***;


    entity class mech2;
    begin
      new mech2("m2:").schedule(nextm2.sample);
      clerks.acquire(1);
        hold(serve2.sample);
      clerks.release(1);
    end***mech2***;

    nextm1   :- new negexp("nextm1", 0.005);
    nextm2   :- new negexp("nextm2", 0.008);
    serve1   :- new uniform("serve1", 200.0, 400.0);
    serve2   :- new uniform("serve2",  75.0, 150.0);
    clerks   :- new res("clerks", 2);

    new mech1("m1:").schedule(0.0);
    new mech2("m2:").schedule(0.0);

    hold(480.0*60.0);
  end;
```

Notice that this model is structurally identical to that of exercise 3.10.

3.12 Grocery store program



```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res)checkout;
    ref(rdist)array shop (1:3);
    ref(bdist)array aisle(1:3);
    ref(idist)array items(1:3);
    ref(rdist)next, overhead;
    ref(idist)impulse;
    integer k;

    entity class customer;
    begin
      integer k, tot;

      new customer("c").schedule(next.sample);
      for k := 1 step 1 until 3 do
      begin
        if aisle(k).sample then
        begin
          tot := tot+items(k).sample;
          hold(shop(k).sample);
        end;
      end;
      tot := tot+impulse.sample;
      checkout.acquire(1);
      hold(overhead.sample + 10*tot);
      checkout.release(1);
    end***customer***;

    next     :- new negexp("next", 0.0125);
    aisle(1) :- new draw("aisle 1", 0.75);
    aisle(2) :- new draw("aisle 2", 0.55);
    aisle(3) :- new draw("aisle 3", 0.82);
    items(1) :- new randint("items 1", 2, 4);
    items(2) :- new randint("items 2", 3, 5);
    items(3) :- new randint("items 3", 6, 8);
    shop (1) :- new uniform("shop 1",  60.0, 180.0);
    shop (2) :- new uniform("shop 2", 120.0, 180.0);
    shop (3) :- new uniform("shop 3",  75.0, 165.0);
    overhead :- new uniform("overhead",15.0,  35.0);
    impulse  :- new randint("impulse", 1, 3);
    checkout :- new res("check out", 2);
    new customer("c").schedule(0.0);
    hold(8*3600);
  end;
```

3.13 Closing down the surgery

First cut off the arrival stream by an `if ...  then` test. `class patient` becomes

```
entity class patient;
begin
  if time <= 630.0 then
  begin
    new patient("p").schedule(next);
    doctor.acquire(1);
    hold(consultation);
    doctor.release(1);
  end;
end***patient***;
```

Each patient thus checks for itself whether or not the door is closed (`time <=` *10.30 o'clock*). The first arrival after that time exits at once and no further patients are generated. To make sure that the doctor sees all the patients, we can queue the Demos block itself behind the last patient (if any) by replacing the statement `hold(90.0)` in the main program block by

```
hold(90.0);
doctor.acquire(1);
```

When the Demos block is re-entered, all waiting patients have been consulted.

3.14 Entity generation

In this situation, we cannot allow the n'th. patient object to generate patient the n+1'st. regardless. A simple, yet powerful, way out is to declare a completely separate object whose sole purpose is to generate patient objects, and remove the generating statement `new patient...` from the body of `class patient`. The new entity declaration is:

```
entity class gen;
begin
  hold(next);
  new patient("P").schedule(0.0);
  repeat;
end***gen***;
```

We also alter `class patient` by deleting `new patient("P").schedule(`*next*`)` and the statement generating the first patient object (in the Demos block) is replaced by

```
[integer k;]

for k := 1 step 1 until n do
  new patient("P").schedule(0.0);
new gen("Pgen").schedule(0.0);
```

Answers to exercises 4

4.1 Widget making

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    integer k;
    ref(res) oven;
    ref(rdist) assembly, fire;
    ref(count) widgets;

    entity class assembler;
    begin
      hold(assembly.sample);
      oven.acquire(1);
        hold(fire.sample);
      oven.release(1);
      widgets.update(1);
      repeat;
    end***assembler***;

    assembly :- new uniform("assembly", 25.0, 35.0);
    fire     :- new normal("fire", 8, 2);
    widgets  :- new count("widgets");
    oven     :- new res("oven", 1);

    for k := 1 step 1 until 3 do
      new assembler("assembler").schedule(0.0);

    hold(40.0*60.0);
  end;
```

4.2 Polishing castings

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) crane;  ref(count)jobs;
    ref(rdist) load, polish1, reposition, polish2, remove, next, use;

    entity class machine;
    begin
      crane.acquire(1);
    LOOP:
        hold(load.sample);
      crane.release(1);
      hold(polish1.sample);
      crane.acquire(1);
```

```
        hold(reposition.sample);
      crane.release(1);
      hold(polish2.sample);
      crane.acquire(1);
        hold(remove.sample);
      jobs.update(1);
      repeat;
    end***machine***;

    entity class other;
    begin
      new other("other").schedule(next.sample);
      crane.acquire(1);
        hold(use.sample);
      crane.release(1);
    end***other***;

    crane      :-  new res("crane", 1);
    jobs       :-  new count("jobs done");
    load       :-  new uniform("load", 15.0, 29.0);
    polish1    :-  new uniform("polish 1", 60.0, 100.0);
    reposition :-  new uniform("reposition", 8.0, 22.0);
    polish2    :-  new uniform("polish 2", 80.0, 140.0);
    remove     :-  new uniform("remove", 15.0, 30.0);
    next       :-  new negexp("next o'job", 0.020);
    use        :-  new normal("use", 25.0, 5.0);
    new other("other").schedule(20.0);
    new machine("m").schedule(0.0);
    hold(24000.0);
  end;
```

Notice how we let the machine retain the crane through the remove and then the fetch and lift phases.

4.3 TV set assembly

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) adjusters, inspectors;
    ref(rdist) next, inspection, readjust;
    ref(bdist) faulty;

    entity class tvset;
    begin
      new tvset("TV").schedule(next.sample);
    LOOP:
      inspectors.acquire(1);
        hold(inspection.sample);
      inspectors.release(1);

      if faulty.sample then
      begin
        adjusters.acquire(1);
          hold(readjust.sample);
        adjusters.release(1);
        priority := priority + 1;
        repeat;
      end;
    end***tvset***;

    adjusters  :- new res("adjusters", 1);
    inspectors :- new res("inspectors", 2);
    next       :- new negexp("next TV", 0.2);
    inspection :- new uniform("inspection", 6.0, 10.0);
    faulty     :- new draw("faulty", 0.10);
    readjust   :- new normal("readjust", 30.0, 5.0);
    new tvset("TV").schedule(0.0);
    hold(5*8*60);
  end;
```

The staging spaces may be roughly estimated from the QMAX value reported for the res objects.

4.4 Unit repairs

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) ws1, ws2, sp2;
    ref(rdist) arrivals, strip, rebuild;
    ref(count) sub;
    integer k;

    entity class unit;
    begin
      if w1.length = 4 then sub.update(1) else
      begin
        ws1.acquire(1);
        hold(strip.sample);
        sp2.acquire(1);
        ws1.release(1);
          hold(if sp2.avail = 1 then 0.2 else 0.1);
        ws2.acquire(1);
        sp2.release(1);
          hold(rebuild.sample);
        ws2.release(1);
      end;
    end***unit***;

    entity class next;
    begin
      new unit("unit").schedule(0.0);
      hold(arrivals.sample);
      repeat;
    end***next***;

    arrivals :- new negexp("arr", 4.0);
    strip    :- new normal("strip", 0.50, 0.05);
    rebuild  :- new normal("rebuild", 0.25, 0.1);
    ws1      :- new res("work st. 1", 2);
    ws2      :- new res("work st. 2", 1);
    sp2      :- new res("area 2", 2);
    sub      :- new count("subcontracts");

    for  k := 1 step 1 until 2 do
```

```
      new unit("unit").schedule(0.0);
    new next("next").schedule(0.5);
    hold(136.0);
  end;
```

4.5 Unit repairs II

To stop the run appropriately, replace `hold(136.0)` in the Demos block by

```
    hold(134.0);
    ws1.acquire(2);
    sp2.acquire(2);
    ws2.acquire(1);
```

Any units arriving after time 134.0 will be blocked requesting `ws1`.

4.6 Production line

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res)array server(1:5);
    ref(rdist) arr, serve;
    ref(count) again, done;
    integer k;

    entity class item;
    begin
      integer k;
      new item("item").schedule(arr.sample);
    LOOP:
      for k := 1 step 1 until 5 do
      begin
        hold(1.0);
        if server(k).avail = 1 then
        begin
          server(k).acquire(1);
            hold(serve.sample);
          server(k).release(1);
          done.update(1);
          goto L;
        end;
      end;
      hold(4.0);
      again.update(1);
      repeat;
  L:end***item***;

    arr   :- new negexp("arrivals", 4.0);
    serve :- new uniform("service", 0.8, 1.2);
    again :- new count("re-cycles");
    done  :- new count("items done");
    for k := 1 step 1 until 5 do
      server(k) :- new res(edit("server",k),1);
    new item("item").schedule(0.0);
    hold(480.0);
  end;
```

4.7 Production line II

As in exercise 4.6 above, except delete all references to the `ref(count) again` and alter the declaration of `class item` to

```
entity class item;
begin
   integer k;

   new item("item").schedule(arr.sample);
   for k := 1 step 1 until 4 do
   begin
     hold(1.0);
     if server(k).avail = 1 then
     begin
        server(k).acquire(1);
          hold(servel.sample);
        server(k).release(1);
        goto L;
     end;
   end;
   hold(1.0);
   server(5).acquire(1);
     hold(serve.sample);
   server(5).release(1);
 L:done.update(1);
end***item***;
```

The storage space required in front of server 5 can be roughly estimated from the `QMAX` statistic of the corresponding `res` object. By declaring an appropriate procedure local to `class item`, the declaration can be made somewhat neater as below

```
entity class item;
begin
  integer k;

  procedure service(n); integer n;
  begin
    server(n).acquire(1);
      hold(serve.sample);
    server(n).release(1);
    goto L;
  end***service***;

  new item("item").schedule(arr.sample);
  for k := 1 step 1 until 4 do
    if server(k).avail then service(k);
  service(5);
L:done.update(1);
end***item***;
```

4.8 Steel billets

```
external class Demos = "/usr/local/simulabin/Demos.atr";
```

```
Demos
  begin
    ref(bin) bogies;
    ref(res) mills, pits, cranes;

    entity class furnace;
    begin
      hold(heat_billet_time.sample);
      bogies.take(1);
      new billet("billet").schedule(0.0);
      repeat;
    end***furnace***;

    entity class billet;
    begin
    UNLOAD:
      if pits.avail = 0 then
      begin
    NO_PITS_FREE:
        cranes.acquire(1);
          hold(unload_from_bogie_time.sample);
        cranes.release(1);
        bogies.give(1);
    AWAIT_PIT:
        pits.acquire(1);
        cranes.acquire(1);
          hold(load_into_pit_time.sample);
        cranes.release(1);
      end else
      begin
    STRAIGHT_IN:
        pits.acquire(1);
        cranes.acquire(1);
          hold(from_bogie_into_pit_time.sample);
        cranes.release(1);
        bogies.give(1);
      end;

      hold(soak_time.sample);
    ROLLING:
      mills.acquire(1);
      cranes.acquire(1);
        hold(unload_from_pit_time.sample);
      cranes.release(1);
      pits.release(1);
      hold(roll_time.sample);
      mills.release(1);
    end***billet***;

    cranes :- new res("cranes", 2);
    mills  :- new res("mills",  1);
    pits   :- new res("pits",  12);
    bogies :- new bin("bogies", 9);

    new furnace("furnace").schedule(0.0);
    hold(simulation_period.sample);
  end;
```

4.9 3-stage assembly line

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(count) done;
    ref(bin) assembled, greased, packed, inners, outers;
    ref(rdist) nexti, nexto, assemble, grease, pack;
    integer k;


    entity class iring;
    begin
      inners.give(1);
        hold(nexti.sample);
      repeat;
    end***iring***;

    entity class oring;
    begin
      outers.give(1);
        hold(nexto.sample);
      repeat;
    end***outer rings***;

    entity class assembler;
    begin
      inners.take(1);
      outers.take(1);
        hold(assemble.sample);
      assembled.give(1);
      repeat;
    end***assembler***;

    entity class greaser;
    begin
      assembled.take(1);
        hold(grease.sample);
      greased.give(1);
      repeat;
    end***greaser***;

    entity class packer;
    begin
      greased.take(2);
        hold(pack.sample);
      done.update(1);
      repeat;
    end***packer***;

    assembled :-  new bin("assembled", 0);
    greased   :-  new bin("greased",   0);
    packed    :-  new bin("packed",    0);
    inners    :-  new bin("inners",   10);
    outers    :-  new bin("outers",   10);

    done      :-  new count("jobs done");
    nexti     :-  new negexp("inner", 6.0);
    nexto     :-  new negexp("outer", 6.0);
    assemble  :-  new normal("assemble", 0.5, 0.1);
    grease    :-  new constant("grease", 0.16);
    pack      :-  new normal("pack", 0.6, 0.1);

    new iring("i-ring").schedule(0.0);
    new oring("o-ring").schedule(0.0);

    for k := 1 step 1 until 3 do
     new assembler("assembler").schedule(0.0);
    new greaser("greaser").schedule(0.0);
    for k := 1 step 1 until 2 do
     new packer("packer").schedule(0.0);

    hold(480.0);
  end;
```

4.10 Faulty part

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(bin)faulty, good;
    ref(rdist)run, repair, other;
    integer k;

    entity class operator;
    begin
      hold(0.4);
      faulty.give(1);
    REPLACE:
      good.take(1);
      hold(0.4);
    OK_TO_RUN:
      hold(run.sample);
      repeat;
    end***operator***;

    entity class repairman;
    begin
    do_repairs:
      while faulty.avail > 0 do
      begin
        faulty.take(1);
          hold(repair.sample);
        good.give(1);
      end;
    other_work:
      hold(other.sample);
      repeat;
    end***repair***;

    run    :- new normal("run", 36.0, 7.0);
    repair :- new normal("repair", 2.0, 0.5);
    other  :- new uniform("other", 0.5, 1.5);
    faulty :- new bin("faulty", 1);
    good   :- new bin("good", 0);

    for k := 1 step 1 until 3 do
      new operator("o").schedule((2*k-1)*run.sample/6);
    new repairman("r").schedule(0.0);
    hold(672.0);
  end;
```

4.11a — infinite buffer

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) access;
    ref(bin) messages;
    ref(rdist) nextm, decode;

    entity class sender;
    begin
      hold(nextm.sample);
      access.acquire(1);
        hold(0.05);
      access.releasel(1);
      messages.give(1);
      repeat;
    end***sender***;

    entity class receiver;
    begin
      messages.take(1);
      access.acquire(1);
        hold(0.05);
      access.release(1);
      hold(decode.sample);
      repeat;
    end***receiver***;

    nextm    :- new negexp("nextm", 1.0);
    decode   :- new uniform("decode", 0.6, 1.4);
    access   :- new res("access", 1);
    messages :- new bin("messages", 0);

    new sender("s").schedule(0.0);
    new receiver("r").schedule(0.0);

    hold(100.0);
  end;
```

4.11b — buffer of capacity L

In 4.11a above, `access` is used to guarantee single access to the buffer slots, and `messages` holds the number of messages sent by `S` but not yet extracted by `R`. In this problem, `S` may not be more than `L` slots ahead of `R` or else it starts overwriting a previous message. This can be controlled by a further `bin lead` which is used to block `S` should `R` be `L` messages behind. To 4.11a we add the declaration `ref(bin)lead` and the initialising statement

```
        lead :- new bin("lead", L);
```

The synchronisation is completed by altering the sequence

```
hold(nextm.sample);
access.acquire(1);
```

in `class sender` to

```
hold(nextm.sample);
lead.take(1);
access.acquire(1);
```

which makes sure that `S` is not too far ahead before attempting to place the next message in the buffer; and by altering the sequence

```
access.release(1);
```

inside `class receiver` to

```
access.release(1);
lead.give(1);
```

This lets `S` know each time a slot has been freed. We must, of course, also declare and initialise `L` or else use a constant.

4.12 Garage

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) bays;
    ref(rdist) pservice,cservice,nextp;
    ref(idist) group;
    integer week, day, k, n;
    real t;

    entity class pcar;
    begin
      priority := 1;
      new pcar("p").schedule(nextp.sample);
      bays.acquire(1);
        hold(pservice.sample);
      bays.release(1);
    end***police car***;

    entity class car;
    begin
      bays.acquire(1);
        hold(cservice.sample);
      bays.release(1);
    end***car***;
    bays     :- new res("bays", 5);
    pservice :- new normal("pservice", 2.5, 1.0);
    nextp    :- new negexp("next p", 0.08333333);
    group    :- new randint("group", 12, 20);
    cservice :- new uniform("cservice", 1.5, 2.5);

    Demos.priority := 2;
    bays.acquire(5);
    new pcar("p").schedule(nextp.sample);

    for week := 1 step 1 until 4 do
    begin
      for day := 1 step 1 until 5 do
      begin
        hold(9.0);
        n := group.sample;
        for k := 1 step 1 until n do
          new car("c").schedule(0.0);
        bays.release(5);
        hold(8.0);
        t := time;
        bays.acquire(5);
        hold(7.0 - (time - t));
      end;

   SATURDAY:
      hold(9.0);
      n := group.sample/2;
      for k := 1 step 1 until n do
        new car("c").schedule(0.0);
      bays.release(5);
      hold(4.0);
      t := time;
      Demos.priority := 0;
      bays.acquire(5);
      Demos.priority := 2;
      hold(11.0 - (time - t));

    SUNDAY:
      hold(24.0);
    end;
  end;
```

Answers to exercises 5

5.1 Library archive

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(waitq) DESK;
    ref(rdist) nextr, there, back, st;
    ref(histogram) thru;
    integer k;

    entity class librarian(n); integer n;
    begin
      procedure customer_req;
      begin
        C :- DESK.coopt;
        r := r + 1;
        C.into(Q);
        hold(0.1);
      end***customer_req***;;

      ref(queue) Q;
      ref(entity) C;
      ref(count) slips;
      integer r;

      Q     :- new queue(edit("Q", n));
      slips :- new count(edit("slips", n));
    LOOP:
      r := 0;
      customer_req;
    ANY_MORE:
      while desk.length > 0 and r < 5 do
        customer_req;
      slips.update(r);
    GET_REQUESTS:
      hold(there.sample);
      hold(r*(1.0+st.sample/5.0));
      hold(back.sample);
    SIGN_OUT:
      while Q.length > 0 do
      begin
        hold(0.5);
        Q.first.schedule(0.0);
      end;
      repeat;
    end***librarian***;

    entity class request;
    begin
      real arrtime;

      arrtime := time;
      new request("r").schedule(nextr.sample);
      DESK.wait;

    quit:
      thru.update(time - arrtime);
    end***request***;

    nextr :- new negexp("next req", 0.5);
    there :- new uniform("there", 0.5, 1.5);
    st    :- new normal("st", 0.0, 1.0);
    back  :- new uniform("back", 0.5, 2.0);
    desk  :- new waitq("desk");
    thru  :- new histogram("thru times", 0, 3, 10);

    for k := 1 step 1 until 3 do
      new librarian("l", k).schedule(0.0);
    new request("req").schedule(0.0);

    hold(480.0);
  end;
```

5.2 Library archive II

The essence is to allow only one librarian to be attending to the desk queue at once. In our solution to Exercise 5.1 whilst one librarian is signing in a request, another may poach the next in line. This we can avoid by using a `res` object `access` of limit 1 and inserting `access.acquire(1);` after the label `LOOP` and `access.release(1);` immediately before the label `GET_REQUESTS` in the body of `class librarian`.

5.3 Steel I

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(rdist) load_smelt, pour, strip, clean_assemble, set, load_pit, soak;
    ref(res) pits, cranes, mills;
    ref(bin) bogies;
    ref(waitq) STRIPQ;
    integer k;

    entity class furnace;
    begin
      integer k;

      hold(load_smelt.sample);

      for k := 1 step 1 until 2 do
      begin
        bogies.take(1);
        hold(pour.sample);
        new batch("b").schedule(0.0);
      end;
      repeat;
    end***furnace***;

    entity class strippers;
    begin
      ref(batch)B;

      B :- STRIPQ.coopt;
      hold(strip.sample);
      B.schedule(0.0);

      hold(clean_assemble.sample);
      bogies.give(1);
      repeat;
    end***strippers***;

    entity class batch;
    begin
      hold(set.sample);
      STRIPQ.wait;

      pits.acquire(1);
      cranes.acquire(1);
      hold(load_pit,.sample);
      cranes.release(1);
      hold(soak.sample);

    UNLOAD_15_AND_ROLL_14:
      mills.acquire(1);
      cranes.acquire(1);
      hold(1.0 + 14*3.0);
      cranes.release(1);
      pits.release(1);
    ROLL_THE_LAST:
      hold(3.0);
      mills.release(1);
    end***batch***;

    load_smelt     :- new normal("load_and_smelt", 165.0, 20.0);
    pour           :- new constant("pour", 20.0);
    strip          :- new uniform("strip", 10.0, 16.0);
    clean_assemble :- new uniform("clean_assemble", 20.0, 24.0);
    set            :- new constant("set", 75.0);
    load_pit       :- new_constnt("load pit", 15.0);
    soak           :- new normal("soak", 160.0, 30.0);

    pits           :- new res("pits",  10);
    cranes         :- new res("cranes", 3);
    mills          :- new res("mills",  2);
    bogies         :- new bin("bogies", 8);
    STRIPQ         :- new waitq("await strip");

    for k := 1 step 1 until 4 do
      new furnace("f").schedule(40*(k-1));

    for k := 1 step 1 until 2 do
      new strippers("s").schedule(0.0);

    hold(1500.0);
  end;
```

5.4 Steel II

```
ref(res) power, brickies, c1, c2;
ref(bin) bogies;

entity class furnace;
begin
  integer k;

  for k := 1 step 1 until 10 do
  begin
    c1.acquire(1);
      hold(load.sample);
    c1.release(1);

    power.acquire(3);
      hold(melt.sample);
    power.release(2);

    hold(refine.sample);

    bogies.take(1);
    c2.acquire(1);
```

```
        hold(tap.sample);
      new batch("B").schedule(0.0).
      c2.release(1);
      power.release(1);
    end;

    brickies.acquire(1);
      hold(clean.sample);
    brickies.release(1);

    repeat;
  end***furnace***;
```

5.5 Newspaper adverts

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(waitq) Q1, Q2;
    ref(res) trunks;
    ref(count) calls, accepted, rej, completed, overflows, direct, indirect;
    ref(rdist) arr, notes, advert;
    ref(histogram) waittimes, thrutimes;

    entity class call;
    begin
      real arrtime;

      new call("call").schedule(arr.sample);
      arrtime := time;
      calls.update(1);
      if trunks.avail = 0 then rej.update(1)       else
      if Q1.length = k    then overflows.update(1) else
      begin
        accepted.update(1);
        trunks.acquire(1);
        if Q2.masterq.length > 0 then
        begin
          direct.update(1);
          Q2.wait;
        end else
        begin
          indirect.update(1);
          if Q2.length = 0 then Q2.wait else Q1.wait;
        end;

      AWAIT_END_OF_CONVERSATION:
        trunks.release(1);
        thrutimes.update(time - arrtime);
        completed.update(1);
      end;
    end***call***;

    entity class operator;
    begin
      procedure Q1intoQ2;
      begin
        ref(entity) C;
        while Q1.length > 0 do
        begin
          C :- Q1.first;
          C.out;
          C.into(Q2);
        end;
      end***Q1 into Q2***;

      ref(call) C;

      C :- Q2.coopt;
      if Q2.length = 0 then Q1intoQ2;
      waittimes.update(time - C.arrtime);
      hold(advert.sample);
      C.schedule(0.0);
      hold(notes.sample);
      repeat;
    end***operator***;

    integer k, m, n, j;

    k := 9;  m := 6;  n := 15;
    arr       :-  new negexp("arr", 1.0);
    notes     :-  new normal("notes", 4.0, 1.0);
    advert    :-  new normal("advert", 1.25, 0.5);
    calls     :-  new count("calls");
    rej       :-  new count("rej");
    overflows :-  new count("overflows");
    completed :-  new count("completed");
    direct    :-  new count("direct");
    indirect  :-  new count("indirect");
    accepted  :-  new count("accepted");
    waittimes :-  new histogram("waits", 0.0, 10.0, 10);
    thrutimes :-  new histogram("thru times", 0.0, 10.0, 10);
    trunks    :-  new res("trunks", n);
    Q1        :-  new waitq("Q1");
    Q2        :-  new waitq("Q 2");

    for j := 1 step 1 until m do
      new operator("O").schedule(0.0);

    new call("C").schedule(0.0);
    hold(480.0);
end;
```

5.6 Example 6 revisited

(Sketch only.) Maintain a `bin pending` on the number of untreated requests in each request queues. Now a query places itself in a request queue by:

```
      pending.give(1);
      RQ(k).wait;
```

The scanner executes a `pending.take(1)` to await the next query when the request queues are all empty. This keeps the scanner asleep while no queries are currently pending. When the scanner is woken up again, it has to compute where it should be (quite tricky), `hold` until it is time to lock onto the next station (careful as it will be in mid-rotation or in mid-test). Then we let it rotate, test and transmit while `pending.avail` > `0`. Then the scanner saves its current status (time and position) and hangs itself up with a `pending.take(1)`.

Answers to exercises 6

6.1 Lazy boolean operators

```
  boolean procedure and2(a, b); name a, b; boolean a, b;
    and2 := if a then b else false;

  boolean procedure or2(a, b); name a, b; boolean a, b;
    or2  := if a then true else b;
```

6.2 Bar

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(rdist) drink, pour, wash, next;
    ref(idist) thirst;
    ref(bin) clean, dirty, empty;
    ref(condq) IDLEQ;
    ref(waitq) BAR;

    entity class customer;
    begin
      integer k, n;

      new customer("c").schedule(next.sample);
      n := thirst.sample;
      for k := 1 step 1 until n do
      begin
        IDLEQ.signal;
        BAR.wait;

        hold(drink.sample);
        empty.give(1);
      end;
    end***customer***;

    entity class waiter;
    begin
      integer n;

      n := 0;
      while empty.avail > 0 do
      begin
        empty.take(1);
        hold(0.2);
        n := n+1;
      end;
      dirty.give(n);
      IDLEQ.signal;
      hold(30.0 - n*0.2);
      repeat;
    end***waiter***;

    entity class barman;
    begin
      ref(entity) C;

      IDLEQ.waituntil
        ( (bar.length > 0 and clean.avail > 0) or dirty.avail > 0);
      if (BAR.length > 0 and clean.avail > 0) then
      begin
        C :- BAR.coopt;
        clean.take(1);
          hold(pour.sample);
        C.schedule(0.0);
      end else
      begin
        dirty.take(1);
          hold(wash.sample);
        clean.give(1);
        IDLEQ.signal;
      end;
      repeat;
    end***barmaid***;

    clean  :- new bin("clean", 15);
    dirty  :- new bin("dirty", 0);
    empty  :- new bin("empty", 0);
    idleq  :- new condq("idle");
    bar    :- new waitq("bar");
    thirst :- new randint("thirst", 1, 6);
    drink  :- new uniform("drink", 15.0, 25.0);
    pour   :- new constant("pour", 1.0);
    wash   :- new constant("wash", 0.5);
    next   :- new negexp("next", 0.2);

    new barman("B").schedule(0.0);
    new customer("C").schedule(0.0);
    new waiter("W").schedule(0.0);

    hold(180.0);
  end;
```

6.3 Port with tides

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) tugs, jetties;
    ref(condq) DOCKQ, OUTQ;
    ref(rdist) next, discharge;
    boolean lowtide, hightide;

    entity class boat;
    begin
      new boat("b").schedule(next.sample);
      jetties.acquire(1);
      DOCKQ.waituntil(tugs.avail >= 2 and hightide);
      tugs.acquire(2);
        hold(2.0);
      tugs.release(2);
      DOCKQ.signal;
      OUTQ.signal;

      hold(discharge.sample);

      OUTQ.waituntil(tugs.avail > 0 and not lowtide);
      tugs.acquire(1);
        hold(2.0);
      tugs.release(1);
      jetties.release(1);
      DOCKQ.signal;
      OUTQ.signal;
    end***boat***;

    entity class tide;
    begin
      lowtide := true;
        hold(4.0);
      lowtide := false;
      OUTQ.signal;

      hold(2.5);

      hightide := true;
      DOCKQ.signal;
      hold(4.0);
      hightide := false;

      hold(2.5);
      repeat;
    end***tide***;

    tugs      :- new res("tugs",    3);
    jetties   :- new res("jetties", 2);
    DOCKQ     :- new condq("DOCK");
    OUTQ      :- new condq("LEAVING");
    next      :- new negexp("next boat", 0.10);
    discharge :- new normal("discharge", 14.0, 3.0);

    new tide("tide").schedule(1.0);
    new boat("b").schedule(0.0);

    hold(28.0*24.0);
  end;
```

6.4 Traffic lights

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(condq) LIGHTS;
    ref(rdist) next, clear;
    boolean ok, green;
    entity class car;
    begin
      new car("c").schedule(next.sample);

      LIGHTS.waituntil(ok and green);
      ok := false;
        hold(clear.sample);
      ok := true;
      LIGHTS.signal;
    end***car***;

    entity class tlights;
    begin
      green := true;
      LIGHTS.signal;
        hold(20.0);
      green := false;

      hold(24.0);
      repeat;
    end***tlights***;

    ok     := true;
    LIGHTS :- new condq("traffic lights");
    next   :- new negexp("next car", 0.03);
    clear  :- new normal("clear time", 2.0, 0.5);

    new tlights("lights").schedule(0.0);
    new car("c").schedule(0.0);

    hold(7200.0);
  end;
```

6.5 Traffic lights II

We use `res` objects `near` and `far` to indicate whether or not the near and far lanes are currently free. They are switched by objects of `class convoy` (representing a line of cars with no break in between). There is a convoy for each direction. We declare

```
ref(res) near, far;

entity class convoy(lane); ref(res)lane;
begin
LANE_BLOCKED:
  lane.acquire(1);
  hold(time_for_convoy_to_pass.sample);
  lane.release(1);
  LIGHTS.signal;
GAP:
  hold(safe_to_cross_time.sample);
  repeat;
end***convoy***;
```

and issue the initialising statements:

```
    near :- new res("near lane", 1);
    far  :- new res("far lane",  1);
    new convoy("near",near).schedule(...);
    new convoy("far",  far).schedule(...);
```

The cars elect to filter onto the main road (acquiring `near`) or to cross the main road (acquiring both `near` and `far`). In both cases, they need to be at the front of the queue and are delayed a little by the car in front (as in exercise 6.4).

```
  entity class car;
  begin
    boolean filter;

    new car("c").schedule(next);
    filter := probability_of_filtering.sample;
    lights.waituntil
      ( ok and near.avail > 0
        and (filter or far.avail > 0));
    ok := false;
      hold(time_to_clear.sample);
    ok := true;
    LIGHTS.signal;
  end***car***;
```

6.6 Channel I

In this answer, and in the answers to exercises 6.7–6.8 as well, we assume (without loss of generality) that the canal runs from east to west. We use booleans `sail(E)` and `sail(W)`) to indicate the prevailing direction. The prevailing direction is switched periodically by an object of `class switcher`. Notice how it acquires 3 units of the `res canal` prior to doing the direction switching. This gives any boats in the canal time to clear it.

```
  external class Demos = "/usr/local/simulabin/Demos.atr";

  Demos
    begin
      ref(rdist) next;
      boolean array sail(1:2);
      ref(res) canal;
      ref(condq)array Q(1:2);
      boolean entry;
      integer E, W;
      real timeslot, ctime;

      entity class boat(D); integer D;
      begin
        new boat("b", D).schedule(next.sample);
      AWAIT_ENTRY_PERMISSION:
        Q(D).waituntil(sail(D) and entry);
        canal.acquire(1);
        entry := false;
      FIRST_PART_OF_CANAL:
        hold(ctime/3.0);
        entry := true;
        Q(D).signal;
      REST_OF_CANAL:
        hold(2.0*ctime/3.0);
        canal.release(1);
      end***boat***;

      entity class switcher;
      begin
         integer D;

         priority := 1;
         canal.acquire(3);
      LOOP:
         for D := E, W do
         begin
           canal.release(3);
           sail(D) := true;
           Q(D).signal;
           hold(timeslot);
           sail(D) := false;
           canal.acquire(3);
         end;
         repeat;
      end***switcher***;

      ctime    := ...;
      timeslot := ...;
      E        := 1;
      W        := 2;

      Q(E) :- new condq("going east");
      Q(W) :- new condq("going west");
      next :- new normal("next", ...., ....);

      entry := true;
      canal :- new res("canal", 3);

      new switcher("s").schedule(0.0);
      new boat("W boat:", W).schedule(...);
      new boat("E boat:", E).schedule(...);
      hold(.......);
    end;
```

6.7 Channel II

Declare globally, and suitably initialise

```
ref(condq) SQ;  integer l;
```

and alter the definition of `switcher` in exercise 6.6 to

```
entity class switcher;
begin
  integer D;

  canal.acquire(3);
```

```
LOOP:
  for D := E, W do
  begin
    canal.release(3);
    sail(D) := true;
    Q(D).signal;
      hold(l*ctime/3.0);
    SQ.waituntil(Q(3-D).length = l);
    sail(D) := false;
    canal.acquire(3);
  end;
  repeat;
end***switcher***;
```

`sq` is a third `condq` specially for the `switcher` object. The value of `3 - D` is `E` if `D = W`, and `W` if `D = E`, i.e. if `D` is the prevailing direction, `3 - D` returns the blocked direction, and vice versa. Also, alter `class boat` in 6.6 by including an `SQ.signal` before the call on `Q(D).waituntil`.

6.8 Channel III

As in 6.7, except alter the declaration of `switcher` to

```
entity class switcher;
begin
  integer D;

  priority := -1;
LOOP:
  SQ.waituntil(Q((E).length > 0 or Q(W).length > 0);
  D := if Q((E).length > 0 and Q(W).length > 0
         then choose_E_or_W_with_equal_weight.sample
         else if Q(E).length > 0 then E else W;
  sail(D) := true;
  Q(D).signal;
  canal.acquire(3);
  sail(D) := false;
  canal.release(3);
  repeat;
end***switcher***;
```

By giving the `switcher` a low priority, when it queues for the canal, it can be overtaken by any boat arriving for the prevailing direction.

6.9 Channel IV

As in 6.7, but amend the declaration of `switcher` to

```
entity class switcher;
begin
  integer D;
```

```
    priority := 1;
    canal.acquire(3);
  LOOP:
    for D := E, W do
    begin
      canal.release(3);
      sail(D)   := true;
      sail(3-D) := false;
      Q(D).signal;
      SQ.waituntil(Q((3-D).length > Q(D).length);
      sail(D) := false;
      canal.acquire(3);
    end;
    repeat;
  end***switcher***;
```

6.10 Steel revisited

Add to 4.8 the declarations and appropriate initialisations of `ref(condq) PITS, OUTSIDE` and alter `class billet` to:

```
  entity class billet;
  begin
    boolean cold;

  AWAIT_PIT:
    cranes.acquire(1);
    PITQ.waituntil(pits.avail > 0 or PITQ.length >= 4);
    if pits.avail = 0
      then
        begin
          hold(move_outside.sample);
          cranes.release(1);
          OUTSIDE.signal;
          cold := true;

          OUTSIDE.waituntil
            (cranes.avail > 0 and pits.avail > 5 and PITQ.length = 0);
          cranes.acquire(1);
          pits.acquire(1);
          hold(return_inside.sample);
        end
      else pits.acquire(1);

    hold(load_into_pit.sample);
    cranes.release(1);
    OUTSIDE.signal;

    hold(if cold then soak_longer.sample else soak_shorter.sample);

    mills.acquire(1);
      cranes.acquire(1);
        hold(unload_from_pit.sample);
      cranes.release(1);
    pits.release(1);
    PITQ.signal;
    OUTSIDE.signal;

    hold(roll_in_mill.sample);
    mills.release(1);
  end***billet***;
```

6.11 Pickers and cappers

```
begin
  external class Demos = "/usr/local/simulabin/Demos.atr";
  integer n, m, k;

  n := 6;
  m := n + 1;

  Demos
    begin
      ref(count)array done(1:N);
      boolean array pos(0:M);
      real pause_time, move_time, seal_time;
      ref(count) ok, fail;
      ref(condq) CQ;
      ref(waitq) PQ;
      ref(rdist) fetch;
      boolean capping;

      entity class belt;
      begin
        integer k;

      LOOP:
        hold(move_time);
        for k := m step -1 until 1 do
          pos(K) := pos(K-1);
        if pos(M) then ok.update(1) else fail.update(1);
        capping := true;
        CQ.signal;
        hold(pause_time - seal_time);
      TOO_LATE:
        capping := false;
        hold(seal_time);
        repeat;
      end***belt***;

     entity class picker(n); integer n;
     begin
       hold(fetch.sample);
     WAIT_UNTIL_SEAL_IS_PASSED:
       PQ.wait;
       repeat;
     end***picker***;

     entity class capper(n); integer n;
     begin
       ref(picker) P;

     AWAIT_CONTAINER:
       CQ.waituntil(not pos(N) and capping);
       hold(seal_time);
       pos(N) := true;
     FIND_PARTNER:
       PQ.find(P, P.N = N);
       P.schedule(0.0);
       repeat;
     end***capper***;

     pause_time := 5.0;
     move_time  := 3.0;
     seal_time  := 2.0;
     ok    :- new count("sealed");
     fail  :- new count("not sealed");
     CQ    :- new condq("capperq");
     CQ.all:= true;
     PQ    :- new waitq("pickerq");
     fetch :- new uniform("get seal", 7.0, 11.0);

     new belt("belt").schedule(0.0);
     for k := 1 step 1 until n do
     begin
       new picker("P", k).schedule(0.0);
       new capper("C", k).schedule(0.0);
     end;

     hold(8.0*3600.0);
    end;
  end;
```

6.12 Plate cutting yard

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
   ref(idist)type;
   ref(rdist)next, cutting;
   ref(res)array c(1:2);
   ref(condq) CQ;
   ref(waitq) ARRQ;
   ref(waitq)array OUTQ(1:2);
   integer array l(1:2);

   entity class plate;
   begin
    integer n;

    new plate("p").schedule(next.sample);
    n := type.sample;
    CQ.signal;
    ARRQ.wait;

   RESUME_WHEN_JOINING_CUTTER_Q:
    l(n) := l(n)+1;
    c(n).acquire(1);
    l(n) := l(n)-1;
     hold(cutting.sample);
    c(n).release(1);
    CQ.signal;
   end***plate***;

   entity class crane;
   begin
    boolean procedure testQ(k); integer k;
    begin
```

```
      testQ := l(k) < 3 and OUTQ(k).length > 0;
    end***testQ***;

    ref(plate) P;

    CQ.waituntil(ARRQ.length > 0 or testQ(1) or testQ(2));
    while ARRQ.length > 0 do
    begin
      P :- ARRQ.last.coopt;
      if l(P.n) < 3 then
      begin
    STRAIGHT_THROUGH:
        hold(1.0);
        P.schedule(0.0);
        hold(1.0);
      end else
    STRAIGHT_OUTSIDE:
      begin
        hold(0.5);
        P.into(OUTQ(p.n));
        hold(0.5);
      end;
    end;
  MOVE_IN_FROM_OUTSIDE:
    if testQ(1) or testQ(2) then
    begin
      hold(1.0);
      P :- if testQ(1) then OUTQ(1).last else OUTQ(2).last;
      P.coopt;
        hold(1.0);
      P.schedule(0.0);
      hold(1.0);
      end;
    repeat;
  end***crane***;

  next    :- new negexp("plate", 0.1);
  cutting :- new normal("cutting", 8.0, 2.0);
  type    :- new randint("type", 1, 2);
  c(1)    :- new res("cutter", 1);
  c(2)    :- new res("cutter", 1);
  arrq    :- new waitq("arrivals");
  outq(1) :- new waitq("outside dump");
  outq(2) :- new waitq("outside dump");
  cq      :- new condq("idle crane");
  new plate("p").schedule(0.0);
  new crane("c").schedule(0.0);
  hold(480.0);
end;
```

6.13 A small optimisation

Because with option *b* you have to remember to signal `Q` at `time = t`. *a* is also more efficient. Why?

Answers to exercises 7

7.1 Lathe

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(lathe) L;
    ref(count) done;
    ref(rdist) p, up, repair;

    entity class lathe;
    begin
      hold(p.sample);
      done.update(1);
      repeat;
    end***lathe***;

    entity class breakdown;
    begin
      real tleft;

      hold(up.sample);

      L.cancel;
      tleft := L.evtime-time;
      hold(repair.sample);
      L.schedule(tleft+5.0);
      repeat;
    end***breakdown***;

    done   :- new count("done");
    p      :- new normal("process",15.0,3.0);
    up     :- new negexp("running", 1/300);
    repair :- new normal("repair", 30.0, 5.0);

    L :- new lathe("L");
    L.schedule(0.0);
    new breakdown("B_DOWN").schedule(0.0);

    hold(60*24*28);
  end;
```

7.2 Lather

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(count) done, spoiled;
    ref(rdist) p, up, repair;
    ref(lathe) L;

    entity class lathe;
    begin
      integer interrupted;

      hold(p.sample);
      if interrupted > 0 then
      begin
        interrupted := 0;
        spoiled.update(1);
        hold(6.0);
      end else done.update(1);
      repeat;
    end***lathe***;

    entity class breakdown;
    begin
      hold(up.sample);
      L.cancel;
      hold(repair.sample);
      L.interrupted := 1;
      L.schedule(0.0);
      repeat;
    end***breakdown***;

    done    :-  new count("done");
    spoiled :-  new count("repair");
    up      :-  new negexp("running", 1/300);
    repair  :-  new normal("repair", 30.0, 5.0);
    p       :-  new normal("process", 15.0, 3.0);

    L :- new lathe("L");
    L.schedule(0.0);
    new breakdown("B_DOWN").schedule(0.0);
    hold(60*24*28);
  end;
```

7.3 Lathest

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(res) r;
    integer k;
    ref(count) done;
    ref(rdist) p, up, repair;

    entity class lathe;
    begin
      new b_down("B", current).schedule(0.0);
    LOOP:
      hold(p.sample);
      done.update(1);
      repeat;
    end***lathe***;

    entity class b_down(L); ref(lathe) L;
    begin
      real tleft;

      hold(up.sample);
      tleft := L.evtime-time;
      L.cancel;

      r.acquire(1);
```

```
        hold(30.0);
      r.release(1);
      L.schedule(tleft+5.0);
      repeat;
    end***b_down***;

    entity class other;
    begin
      priority := -1;
    LOOP:
      r.acquire(1);
        hold(15.0);
      r.release(1);
      repeat;
    end***other***;

    r      :- new res("repairman", 1);
    done   :- new count("done");
    p      :- new normal("process",30.0,5.0);
    up     :- new negexp("running", 1/300);

    for k := 1 step 1 until 6 do
      new lathe("L").schedule(0.0);
    new other("o").schedule(0.0);

    hold(60*24*28);
  end;
```

7.4 Car ferry

```
external class Demos = "/usr/local/simulabin/Demos.atr";

Demos
  begin
    ref(ferry) CF;
    integer day, hour;
    ref(condq) DOCKQ;
    ref(waitq) ARRQ;
    ref(bdist) days, nights;
    ref(idist) unload;
    ref(rdist) nextf, nextd, nextn;
    ref(histogram)array thru(1:2);
    ref(res) quay;

    boolean procedure daytime;
      daytime := hour >= 6 and hour < 18;

    entity class clock;
    begin
      for day  := 1 step 1 until 28 do
      for hour := 0 step 1 until 23 do
      begin
        new ferry("f").schedule(nextf.sample);
        if CF.idle
          then CF.interrupt(1)
          else CF.interrupted := 1;
        hold(1.0);
      end;
    end***clock***;
```

```
entity class car;
begin
  ref(car) N;
  boolean season;
  real arrtime;

  N :- new car("c");
  if daytime then
  begin
    N.schedule(nextd.sample);
    season := days.sample;
  end else
  begin
    N.schedule(nextn.sample);
    season := nights.sample;
  end;
  priority := if season then 2 else 1;
  arrtime  := time;
  DOCKQ.signal;
  ARRQ.wait;
ON_BOARD:
  thru(priority).update(time-arrtime);
end***car***;

entity class ferry;
begin
  integer n;

UNLOADING:
  quay.acquire(1);
  CF :- current;
  hold(unload.sample);
LOAD:
  DOCKQ.waituntil(ARRQ.length > 0 and n > 20 or
    interrupted > 0);
  while ARRQ.length > 0 and n < 20 do
  begin
    n := n+1;
    ARRQ.first.schedule(0.0);
    hold(1/60);
  end;
  if interrupted = 0 then goto LOAD;
  CF :- none;
  quay.release(1);
end***ferry***;

unload  :- new randint("unload", 0.1, 0.2);
days    :- new draw("season:day", 0.4);
nights  :- new draw("season:night", 0.25);
nextf   :- new normal("ferry", 1/3, 1/12);
nextd   :- new negexp("day rate", 15.0);
nextn   :- new negexp("night rate", 9.0);
quay    :- new res("quay", 1);

DOCKQ   :- new condq("ferry load q");
ARRQ    :- new waitq("dockside q");
thru(1) :- new histogram("normal", 0.0, 1.0, 10);
thru(2) :- new histogram("season", 0.0, 1.0, 10);

new car("c").schedule(0.0);
new clock("clock").schedule(0.0);
```

```
      hold(24.0*28.0);
    end;
```

7.5 The last alarum

```
  external class Demos = "/usr/local/simulabin/Demos.atr";

  Demos
    begin
      ref(condq) LINE;
      ref(res) server;
      ref(rdist) next, serve, p;

      entity class cus;
      begin
        integer imp;

        new cus("c").schedule(next.sample);
        imp := p.sample;
        new alarm("a", current).schedule(imp);
        LINE.waituntil(server.avail>0 or interrupted>0);
        if interrupted = 0 then
        begin
          server.acquire(1);
            hold(serve.sample);
          server.release(1);
          LINE.signal;
        end;
      end***cus***;

      entity class alarm(E); ref(entity)E;
      begin
        if E.idle and E =/= LINE.first then E.interrupt(1);
      end***alarm***;

      LINE   :- new condq("await service");
      server :- new res("server", 1);
      next   :- new negexp("next", 1.0);
      serve  :- new uniform("service", 2/3, 1.0);
      p      :- new uniform("impat", 2.0, 5.0);

      new cus("c").schedule(0.0);

      hold(240.0);
    end;
```

Appendix B

Outline of Simula

B.1 Simula statements

- block

```
Demos
  begin
    declarations;
    statements;
  end

begin
  integer k, s;

  for k := 1 step 1 until n do
    s := s + X(k)**2;
end
```

- compound_statement

```
begin
  sum := sum + x;
  n := n + 1;
end
```

- procedure_statement

```
repeat

hold(0.0)

tugs.acquire(2)
```

- goto_statement

```
goto exit
```

- assignment_statement

```
m := m + 1

P :- Q :- none

T.load := 16.0
```

- object_generator

```
new boat("boat")
```

- for_statement

```
for k := 1 step 1 until n do
begin
  X(k) := false;
  Y(k) := false;
end;

for k := 3, 1, 4, 1, 5, 9 do
  freq(k) := freq(k)+1

for Q :- Q1, Q2, Q3, Q4 do
  Q.signal
```

- if_statement

```
if tugs.avail = 0 then TUGQ.wait

if free < 5 then
  begin
    hold(10.0);
    E.wait;
  end else T.wait
```

- while_statement

```
while Q.length > 0 do
  Q.first.schedule(now)
```

- inspect_statement (*not used in this book*)

```
inspect t when truck do load := 16.0
```

- dummy_statement

B.2 Simula declarations

- procedure_declaration

procedure parameters			
	mode		
parameter specification	value	ref	name
real/integer/character/boolean	d	*	o
ref(*any class*)	*	d	o
text	o	d	o
real/integer/character/boolean array	o	d	o
ref(*any class*)/text array	*	d	o
procedure/label/switch	*	d	o
key — d: default, *: illegal, o: optional			

```
procedure towin(v); ref(vehicle)v;
begin
  if v == none then ERROR("no vehicle") else
  if v.brokendown then FALSEALARM else
     victim :- v;
end***towin***

integer procedure sum(m, n); integer m, n;
begin
  sum := m + n;
end***sum***;
```

- class_declaration (somewhat restricted)

class parameters		
	mode	
parameter specification	value	ref
real/integer/character/boolean	d	*
ref(*any class*)	*	d
text	o	d
real/integer/character/boolean array	o	d
ref(*any class*)/text array	*	d
key — d: default, *: illegal, o: optional		

```
class point(x, y); real x, y;
begin
  real r;
  r := sqrt(x**2 + y**2);
end***point***

point class polar;
begin
  real angle;
  angle := if r = 0.0 then 0.0 else arctan(x, y);
end***polar***
```

- external_declaration

```
external class Traffic

external class Demos = "/usr/local/simulabin/demos.atr";
```

- array_declaration

```
ref(res)array INNERS, OUTERS (1:N);

integer array BOARD ( 1:8, 1:8 );
```

- type_declaration

```
integer a, b, c

ref(boat)B1, QE2
```

Appendix C

Outline of Demos

C.1 Demos signature

```
simset class Demos;
begin                Data collection facilities
 class count(title); value title; text title;              64
 begin
  procedure update(v); integer v;                          64
  procedure report;                                        67
 end***count***;

 class tally(title); value title; text title;           82,83
 begin
  procedure update(v); real v;                             83
  procedure report;                                        82
 end***tally***;

 class histogram(title,lb,ub,ncells); value title;
  text title; real lb, ub; integer ncells;                104
 begin
  procedure update(v); real v;                            104
  procedure report;                                       102
 end***histogram***;

       entity declarations

 class entity(title); value title; text title;             36
   virtual: label loop;                                    68
 begin
  integer priority;                                        70
  procedure schedule(t); real t;                           36
  procedure cancel;                                        37
  procedure into(q); ref(queue)q;                          93
  procedure out;                                           93
  procedure interrupt(n); integer n;                      161
  boolean procedure avail;                                181
  real procedure evtime;                                  181
  boolean procedure idle;                                 181
  ref(entity)procedure nextev;                            181
  procedure repeat;                                        67
 end***entity***;

      resource declarations

 class res(title,avail); value title;                      37
  text title; integer avail;
 begin
  procedure acquire(n); integer n;                         37
  procedure release(n); integer n;                         37
  procedure report;                                        49
 end***res***;
```

```
class bin(title,avail); value title;
 text title; integer avail;                               74
begin
 procedure take(n); integer n;                            75
 procedure give(n); integer n;                            75
 procedure report;                                        82
end***bin***;


       queue declarations

class queue(title); value title; text title;              93
begin
 ref(entity)procedure first;                              93
 ref(entity)procedure last;                               93
 integer length;                                          93
 procedure report;
end***queue***;

class waitq(title); value title; text title;              89
begin
 ref(queue)masterq, slaveq;                               89
 ref(entity)procedure first;                              90
 ref(entity)procedure last;                               90
 integer length;                                          89
 ref(entity)procedure coopt;                              90
 procedure find(e, cond); name e, cond;
  ref(entity)e; boolean cond;                            116
 boolean procedure avail(e,cond);name e,cond;
  ref(entity)e; boolean cond;                             90
 procedure wait;                                          91
 procedure report;                                   103,104
end***waitq***;


class condq(title); value title; text title;             134
begin
 boolean all;                                            149
 ref(entity)procedure first;                             134
 ref(entity)procedure last;                              134
 integer length;                                         134
 procedure waituntil(c); name c; boolean c;              134
 procedure signal;                                       135
 procedure report;                                       151
end***condq***;


 the main program impersonator

entity class mainprogram;
begin
  detach;
  repeat;
end***mainprogram***;

ref(entity)Demos;

  ... and the various routines
           N.B. the random number generators are
given separately in Appendix C.
```

```
 procedure trace;                                          50
 procedure notrace;                                        50
 procedure hold(t); real t;                                37
 real procedure time;                                      40
 real procedure now;                                      175
 ref(entity)procedure current;                             40
 procedure report;                                        179
 procedure reset;                                         180
 text procedure edit(t,n); value t; text t; integer n;     59

 Demos :- new mainprogram("demos");
 demos.schedule(0.0);
 inner;
 hold(0.0);
 report;
end***Demos***;
```

C.2 The MAIN program

It is desirable to have the Demos block entering into the Simulation as an entity, but this cannot be managed directly as context blocks may not be referenced. The same effect is achieved by having an entity of `class mainprogram`. `ref(entity) Demos` is a reference to a special “impersonating” object. It can be scheduled, held, and cancelled just like any other entity. But every time it becomes `current`, it transfers control back to the Demos context block (that is the effect of the Simula routine `detach` in this case) whose user-written main program actions are thereby continued. Thus a `hold(t)` written in the program block causes the Demos object to be rescheduled at `time+t` and the actions of (the new) `current` to be resumed. Thus the Demos context block will not be active again until the Demos object becomes `current`.

A aimulation run ends when an exit is made through the `end` of the user-written part of the Demos context block. It is usually desirable that the Demos context block itself decides the length of the run, but it is possible to cancel the Demos object and leave the decision to other entities.

C.3 Demos synchronisations

The Demos synchronisation pairs

acquire	release
take	give
coopt	schedule
waituntil	signal

can be expressed in terms of calls on a more primitive procedure pair `get/put`. This helps to ensure their consistency and once the underlying `get/put` pair are understood it makes it easier to write synchronisations correctly. `get` and `put` are (informally) listed below.

```
procedure get(Q, avail, seize); name avail; ref(queue) Q;
   boolean avail; statement seize;
begin
  current.into(Q);
  while not(current == Q.first and avail) do
    current.cancel;
  current.out;
  enter Q.first into event list immediately after current;
  seize;
end***get***;

procedure put(Q, return); ref(queue) Q; statement return;
begin
  if Q.first =/= none then Q.first.schedule(0.0);
  return;
end***put***;
```

In terms of this pair, the Demos routines are

ref(res) r

```
r.acquire(n);
   get(r.q, r.avail >= n, r.avail := r.avail - n);

r.release(n);
   put(r.q,  r.avail := r.avail + n);
```

ref(bin) b;

```
b.take(n);
   get(b.q, b.avail >= n, b.avail := b.avail - n);

b.give(n);
   put(b.q, b.avail := b.avail + n);
```

ref(condq) q;

```
q.waituntil(c);
   get(q.condition queue, c, null);

q.signal;
   put(q.condition queue, null);
```

ref(waitq) w;

```
w.coopt;
   get(q.masterq, q.slaveq.length > 0,
          coopt :- q.slaveq.first.coopt);

w.wait;
   put(q.masterq, begin
                    if q.masterq.length > 0
                      then q.masterq.first.schedule(0.0);
                    current.cancel;
                  end;)
```

Two cases do not quite fit into this general pattern: `find` and `signal` with `all` set. The reader is encouraged to modify `get` and `put` him- or her-self so that these routines can be accommodated.

Appendix D

Demos random number generators

There are 9 random drawing facilities in Demos which are categorised according to the type of distribution they sample from. rdists return real values (constant, empirical, negexp, normal, uniform, erlang); idists return integer values (randint, poisson); and the one bdist (draw) returns a boolean value. Their common portion is defined in class dist which is then used as prefix to the other three.

class dist

```
class dist(title); value title; text title;
begin
  integer u;

  procedure report;
  u := some suitable seed;
end***dist***;

dist class rdist; virtual: real procedure sample;;
dist class idist; virtual: integer procedure sample;;
dist class bdist; virtual: boolean procedure sample;;
```

Note that we can override the initial value given to u (a necessary feature for designed experiments. Negative u values produce *antithetic* drawings).

In practice, the virtual part means that we can reference any sub-class of rdist with a ref(rdist) variable and still access its appropriate procedure sample. Similarly with ref(idist) and ref(bdist) variables.

class constant

```
rdist class constant(a); real a;
begin
  real procedure sample;
end***constant***;
```

A call on sample *always* returns a. It is sometimes useful during model development to cast a distribution as a constant (returning the mean value on every call), and to replace it by the actual distribution prior to model validation.

class normal

```
rdist class normal(a, b); real a, b;
begin
  real procedure sample;

  if b < 0 then b := -b;
end***normal***;
```

A call on sample returns a drawing from a normal distribution with mean a and standard deviation b.

class negexp

```
rdist class negexp(a); real a;
begin
  real procedure sample;

  a := abs(a);
  if a = 0.0 then a := 0.0001;
end***negexp***;
```

A call on sample returns a drawing from a negexp distribution with mean time between arrivals of 1/a.

class uniform

```
rdist class uniform(a, b); real a, b;
begin
  real procedure sample;

  if a > b then swap a<->b;
end***uniform***;
```

A call on sample returns a drawing from a uniform distribution with lower bound a and upper bound b.

class erlang

```
rdist class erlang(a, b); real a, b;
begin
  real procedure sample;

  if a <= 0.0 then a := -a;
  if b <= 0.0 then b := -b;
end***erlang***;
```

A call on `sample` returns a drawing from an `erlang` distribution with mean `a` and standard deviation `1/(a*sqrt(b))`.

class empirical

```
rdist class empirical(n); integer n;
begin
  real array P, X [1:n];
  real procedure sample;

  read in the values $P_1$, $X_1$ : $P_2$, $X_2$: ... : $P_n$, $X_n$
  which represent a cumulative distribution.
  ($P_1$ = 0.0, $P_n$ = 1.0, $P_m > P_r$, $X_m > X_R$ for $m > r$);
end***empirical***;
```

A call on `sample` returns a drawing from the cumulative probability function represented by the `P` and `X` tables, `x`$_1$ <= `sample` <= `x`$_n$. Linear interpolation is used.

class randint

```
idist class randint(a, b); integer a, b;
begin
  integer procedure sample;

  if a > b then swap a<->b;
end***randint***;
```

A call on `sample` returns an integer randomly distributed between `a` and `b`.

class poisson

```
idist class poisson(a); real a;
begin
  integer procedure sample;
end***poisson***;
```

A call on `sample` returns a drawing from a `poisson` distribution with mean `a`.

class draw

Finally

```
bdist class draw(p); real p;
begin
  boolean procedure sample;
end***draw***;
```

A call on `sample` returns `true` with probability `p`: always `false` if `p <= 0.0`, and always `true` if `p >= 1.0`.

Appendix E

A subjective look at objects

Keywords: discrete simulation, object-oriented, semantics, system engineering, verification.

Abstract Discrete event simulations models are collections of objects whose actions can unfold in many ways, some obvious, some mysterious. We look at a way of developing a complex model by first describing it in a process algebra (here, CCS). CCS descriptions are isomorphic to the eventual program code — object for object at the structural level, and line by line internally. From a CCS description (text) it is straightforward to check out all possible ways in which a model can develop *no matter what timing distributions and queueing disciplines are selected* and ensure that deadlock and livelock can never arise, and that the system has particular mutual exclusion, fairness and progress properties. As simulation methodologies are increasingly being used to design and implement complex systems of interacting objects, the ability to perform such verifications is of increasing methodological importance. Checking for such properties by running a model is manifestly impractical.

Object oriented modelling has been around in the discrete event simulation world since the 1950's. Whilst the rest of the world was embracing FORTRAN, Tocher was already decomposing difficult steel plant scenarios into objects and coding them up in octal. The notion of coroutines came a decade later (to Conway, and also Knuth), so Tocher coded his models in the activity style[1]. The fully fledged notion of objects and how to implement them came with the appearance in the mid 1960's of Dahl and Nygaard's Simula1. The Nygaard/Dahl team was a splendid blend of simulation modelling expertise (Nygaard) and language and compiler expertise (Dahl). The former knew how to decompose and express complicated scenarios, the latter how to formalise them and implement them efficiently, although as time went by, each learned more than a few tricks from the other. It was at once apparent that the objects and library concepts of Simula1 were generally useful modelling tools and Simula67 was born. Besides the block structure of Algol60, other major influences were Hoare andWirth's ideas for record handling, although Simula67's inheritance, virtual, package, and pointer handling were, and sadly still are, big advances.

[1] Personal note: In the 1970's, he steadfastly refused to believe in objects, despite having used them for 20 years!

The basic idea when modelling with Simula objects is that major components in the real world would be mapped one for one into objects in the model, minor components into counters (with associated queues for blocked objects) and the system description was their composition in parallel. Some judgement and experience was required to know what was major and what was minor, and in many cases efficient model duals were possible: the one with many serving objects passing materiel to and fro; the other with one main materiel object routing itself through resources.

A Simula object had 4 characteristics: (i) a type leading to strict yet flexible accessing; (ii) its own properties and operators (data, functions/procedures); (iii) an action history which could be unfolded in stages; (iv) an LSC (local sequence control), one to each object, to keep tabs on what this object was doing now. Note that objects were usually built in levels, thus inheriting attributes and actions.

Simpler versions of objects have their uses and adherents: abstract data type leave out the action history and LSC, process algebras, like CCS, dispense with properties but retain actions and the LSC. Watch out for languages labelled "object-oriented" – they usually don't have them.

Object oriented discrete event models tend to have a consistent structure: (i) a MAIN program which contains the model resources and establishes the objects (or object streams) and (ii) a set of cooperating objects each of which "runs for a while" and than passes control to another object. Active objects are held in an event list, sorted on time of next event. The object which is "running" is the one at its head — when it moves on, the new head becomes active. The system shuts down with a final report when control returns to MAIN. Actually Simula permitted MAIN to behave as an object in that it too could acquire and release resources, and so on. This extra flexibility is a godsend when dealing with tricky model shutdowns.

Besides a consistent top-level model structure as a collection of objects, objects are also well-structured within themselves. Not only can their actions be viewed as a sequence of activities, but each activity also has a pattern of get extra resources; hold them for the activity duration; drop some resources. Several British simulation languages were founded on this notion. Noteworthy players include Tocher, Clementson and Mathewson. Tocher was decomposing systems into collections of objects in the 1950's, but didn't get the idea of an LSC, and so described them as collections of activities. He also introduced activity diagrams as high level model representations and these proved ideal for discussing model content with real practitioners. They were also useful for developing actual programs: so useful that Clementson and Mathewson automated the process in the early 1970's. Mathewson's DRAFT system was smart enough to generate object, event, or activity based programs from the same activity diagram, thus showing the equivalence of their descriptive powers.

To me the real power of objects comes from the leverage arising from composing their actions. Suppose we take 100 program actions, each of which makes a change in the system state. Written monolithically (imperatively), that amounts to 100 state changes. But suppose we distribute these actions amongst 10 objects, each of which contains 10 actions. And the objects can interleave unconstrainedly. We now have 10 × 10 × 10 ... × 10 possible state change interleavings. But nothing is for free. Not all (indeed many) of these possible interleavings will be required, and the problem now is how to constrain the objects to do the right thing at the right time.

The standard techniques have been well known to the simulation and operating systems communities for 20 or 30 years. For mutual exclusion, we use facilities/semaphores; for materiel passing, we use bins/buffers; for inter-object cooperation, we use some form of waitq/rendezvous; and multi-resource requests, waits until/monitors or condition queues. Some of these synchronisation techniques came from simulation languages (CSL, GPSS, Simula), some from operating systems (Brinch Hansen, Hoare), but it is sad to reflect that where once simulation languages were at the cutting edge, driven by the need to solve very hard problems, now they lag behind. Even with well known synchronisations and well understood protocols for using them, concurrent systems are hard to design and build correctly because of their stunning variety. A system may run well for many months/years before a combination of circumstances arises that it deadlocks, or race conditions make the wrong data “win”. We cannot afford to be left behind not using widely available tools and methods to characterise our models and ensure that they will behave soundly. These include formal semantics for our languages and model checking.

In this paper we present by example a methodology for developing simulation models which encourages different decompositions to be investigated early on and their key properties (deadlock, safety, progress) to be checked prior to detailed coding and lengthy runs. It is based upon work with Chris Tofts which has given operational and denotational semantics for OO simulation language constructs and a containment relation between them. Operational definitions are quite detailed and may be used to guide an implementation or argue in detail the precise way a particular program should unfold. Denotational definitions are much more abstract than operational definitions with detailed timings replaced by “now, or some arbitrary time later” and all queues (including the event list) running under random selection. Denotational definitions are thus useful for reasoning about properties that will hold under all possible runs of a model whatever the timing delays and whatever the queueing discipline imposed. A formal translation schema permits the automatic construction of a process algebraic representation of the underlying simulation model which can then be checked for freedom from deadlock and livelock, as well as system-specific safety and liveness properties.

By providing a formal representation of a system described in Demos we permit formal reasoning about and proofs of properties of those systems. Given that these properties are often used to make cost or safety critical decisions, we need to be sure that predictions are genuine and not the result of some programming artifact. CCS is well-suited to this task, as it provides a succinct formal notation within which captures the behaviour of the underlying system. Furthermore, formal description of system properties can be automatically checked against its implementation as described by CCS.

In this paper we introduce the basics of CCS and use them to develop an abstract representation of a model in section 2, and show how to check for some model properties in section 3.

E.1 The CCS notation

We introduce this OO notation in the context of a running example. In homage to Tocher, its a steel mill with furnaces heating chunky steel billets, which are then to be rolled into flat plates. To ensure that the billets are uniformly hot, they are kept in soaking pits prior to being rolled. System resources include bogies to transport the billets from the furnace area to the soaking pits, cranes to load and unload the billets. There are two furnaces, and one rolling mill.

Nil: CSS builds from the agent *0* which can do nothing. From there CCS permits several ways of building more interesting agents.

Prefixing: The agent $a.\overline{b}.0$ can do an a action, then a $\overline{b}$ action, and after that, nothing more. Recursive definitions are allowed. We can define a clock as: $Clock = \overline{tick}.\overline{tock}.Clock$ This clock ticks and then tocks forever.

Here are abstract descriptions of the furnace, and steel billets that appear in our model.

$$\begin{array}{lcl} Billet & = & load.soak.unload.roll.exit.0 \\ Furnace & = & heatBillet.Furnace \\ SYS & = & (Furnace^{f} \mid Billet^{b}) \end{array}$$

Each billet carries out the actions *load*, *soak*, *unload*, and *roll* once. Furnaces carry out the action *heatBillet* repeatedly. Each action may take an arbitrary time but for each individual object they can only occur in the stated order. The system has f furnaces and b billets.

Resources We model the crane by:

$$C \quad = \quad \overline{gC}.C' \qquad\qquad C' \quad = \quad \overline{pC}.C$$

A crane exhibits cyclic behaviour forever: think of it as a token that may first be “got”, but then must be “put” before it can be “got” again. The crane may only be “got” when in state C whereupon it moves to state C' in which it may only be “put”.

We enforce mutual exclusion in a system comprising several billets by enclosing the *load*, *unload*, and *exit* actions of these object classes in handshake calls to get and put the crane as below:

$$\begin{array}{lcl}
Billet & = & gC.load.pC.soak.gC.unload.pC \\
 & & \quad .roll.gC.exit.pC.0 \\
Furnace & = & heatBillet.Furnace \\
Crane & = & \overline{gC}.\overline{pC}.C \\
SYS & = & (Furnace^f \mid Billet^b \mid Crane) \setminus \{gC, pC\}
\end{array}$$

The rules of the handshake game are simple: we use names and co-names (same name without/with the overline) for handshakes and list the all handshakes when defining the complete system. A handshake action can only fire when its “partner” is available. Thus when a billet has executed as far as a gC action, it can only progress when the crane is in state C and can carry out a $\overline{gC}$ action — neither handshake can progress alone. The handshake operation will move a billet past its gC into its *load*, *unlaod* or *exit* action *and* flip the weighbridge into state C' where it shows $\overline{pC}$. A handshake is strictly between two objects, so the even if two or more vans and lorries are showing their individual gC actions, only one may progress at a time.

Parallel agents may also rendezvous, here a billet is fired up by a furnace. The handshaking style is retained, but this time it is directly between co-operating objects and does not require an intermediate resource.

$$\begin{array}{lcl}
Billet & = & sB.gC.load.pC.soak.gC.unload.pC \\
 & & \quad .roll.gC.exit.pC.0 \\
Furnace & = & heatBillet.\overline{sB}.Furnace \\
Crane & = & \overline{gC}.\overline{pC}.C \\
SYS & = & (Furnace^f \mid Billet^b \mid Crane) \\
 & & \quad \setminus\{gC, pC, sB\}
\end{array}$$

We now generalise the definition of a unit resource to that of a resource of size N and that may be got/put in unit chunks.

$$\begin{array}{lclcl}
R_N & = & \overline{gR}.R_{N-1} & & \\
R_{N-1} & = & \overline{gR}.R_{N-2} & + & \overline{pR}.R_N \\
 & ..... & & & \\
R_1 & = & \overline{gR}.R_0 & + & \overline{pR}.R_2 \\
R_0 & = & & + & \overline{pR}.R_1
\end{array}$$

In state R_k, k units of the resource are available and $n-k$ have been taken. In state R_N the resource may only be acquired. In the intermediate states, a chunk may be returned (through a $\overline{pR}$ action) or acquired (through a $\overline{gR}$ action). When empty (in state R_0) the resource may only be returned.

We use the notation $R(N)$ to denote such a resource of size N, initially in state R$_N$. We extend our initial description by adding in further constraints. There is space for only p billets in the soaking pit, and m rolling mills — and now c cranes! We assume a suitable renaming scheme so that the associated put/get operations are gP/pP, gM/pM, gC/pC respectively.

$$\begin{array}{lcl}
Billet & = & sB.gC.gP.load.pC.soak.gC.unload.pC \\
 & & \quad .gM.pP.roll.gC.pM.exit.pC.0 \\
Furnace & = & heatBillet.\overline{sB}.Furnace \\
Cranes & = & R(c) \\
Mill & = & R(m) \\
Pits & = & R(p) \\
SYS & = & (Furnace^f \mid Billet^n \mid Crane \mid Mill \mid Pits) \\
 & & \quad \backslash\{sB, gC, pC, gM, pM, gP, pP\}
\end{array}$$

Finally, the system pool of bogies may also be modelled by using a suitably sized resource. Notice that this time, the resource is “got” by a furnace and “put” by a billet.

$$\begin{array}{lcl}
Billet & = & sB.gC.gP.load.pC.pB.soak.gC.unload.pC \\
 & & \quad .gM.pP.roll.gC.pM.exit.pC.Billet \\
Furnace & = & heatBillet.gB.\overline{sB}.Furnace \\
Bogies & = & R(b') \\
Cranes & = & R(c) \\
Mill & = & R(m) \\
Pits & = & R(p) \\
SYS & = & (Furnace^f \mid Billet^b \mid Crane \mid Mill \mid Pits) \\
 & & \quad \backslash\{sB, gB, pB, gC, pC, gM, pM, gP, pP\}
\end{array}$$

Notice that here we have pulled the usual simulation trick of letting an exiting Billet become a fresh new one instead of garbage. Notice also that all the model structure maps straight into the OO simulation language of your choice. All of the synchronisations but $\overline{sB}/sB$ is of the resource type which all simulation languages would seem to possess, and the time-consuming actions such as *load*, ..., *heatBillet* translate to GPSS ADVANCEs or Simula HOLDs. So building a simulation model from the CCS is mechanical and trivial.

One very pleasant aspect of the CCS description is its succinctness — the very structure and essence of the model in only 9 thin lines instead of several pages. But even with only 1 furnace, 3 billets, 2 bogies, 1 crane, 2 pits and 1 mill this description has 176,871 possible states to constrain. It looks a neat decomposition, but do these components co-operate harmoniously?

E.2 Checking the model

Once we have a model description in CCS, we can check its characteristic properties using the Edinburgh Concurrency Workbench (CWB). Checks take the form

$$\text{SYS} \models \text{property}$$

which examine the SYS from its initial configuration and check to see if the logical property holds. Some of the simpler checks are

Deadlock: $SYS \models ALL < - > T$ which, by ALL, traverses every state reachable from the initial state SYS and asks "can you make a move" (expressed by $< - >$ T).

Livelock: $SYS \models PATH < \tau > T$ which traverses every state reachable from the initial state and tries to find a cyclic path (PATH) containing only handshake moves (expressed by $< \tau >$ T).

The above properties are posed the same way for all models.

Our model of course deadlocks in that we should acquire the soaking pit ahead of the crane. CWB testing ensures that rewrite

$$\begin{aligned} Billet &= sB.gP.gC.load.pC.pB.soak.gC.unload.pC \\ &\quad .gM.pP.roll.gC.pM.exit.pC.Billet \end{aligned}$$

is deadlock free, with the 176,871 possible states having been constrained down to just 94 when the system is minimised.

Safety, liveness, progress, and fairness properties are require the model to be decorated with "trace variables" at appropriate points. The ADVANCE/HOLD actions serve this purpose (we may wish to add more). Corresponding to every Simula $HOLD(D.sample)$ we write D in the CCS description, together with say $startF$, $startB$ prefixing each process description. Typical tailored checks might now be

Progress checks to make sure that each process will eventually carry out an exit action.

$$SYS \quad \models \quad ALL[startBillet]\ (EV < exit > T)$$

read as "in every state following an action startBillet (ALL [startBillet]) however the system unfolds (EV)entually one must reach a state permitting an *exit* action.

Safety which tests to see that bad things cannot happen. We may want to check that it is never possible to get two mill users at the same time:

$$SYS \quad \models \quad ALL[roll][roll]F$$

read as "in every state following an action roll (ALL [roll]) however the system unfolds, another immediate roll action is impossible.

Liveness tests to see that good things may happen (e.g. each request may be accepted).

$$\begin{array}{lll} SYS & \models & \text{ALL POSS } \langle \mathit{startFurnace} \rangle \text{ T} \\ SYS & \models & \text{ALL POSS } \langle \mathit{startBillet} \rangle \text{ T} \end{array}$$

read as "from every state (ALL) there exists a path (POSS) to some state where action *startFurnace/startBillet* is enabled ($\langle startFurnace/startBillet \rangle$ T)".

Fairness means that a system can not "spin" forever without enabling some particular input or output action. For any system *SYS*, and for a particular action a, this may be expressed as:

$$\begin{array}{lll} SYS & \models & \text{ALL EV } \langle a \rangle \text{ T} \end{array}$$

which reads as "from every state (ALL) for each path (EV)entually there is a state in which a is enabled ($\langle a \rangle$ T)". E.g.

$$\begin{array}{lll} SYS & \models & ALL(EV < startFurnace > T) \\ SYS & \models & ALL(EV < startFurnace > T) \end{array}$$

read as "in every state (ALL) however the system unfolds (EV)entually one must reach a state permitting some Furnace to start its workcycle again.

E.3 Summary

The main point of the talk is to show how standard techniques from the world of concurrency theory can be brought to bear on the simulation methodology. Because simulating for deadlock, say, is a hit-and-miss affair (the timings may have to be just right), it is usually not attempted at all by simulars. Likewise, neither safety, liveness and fairness testing are a part of the simulars' standard arsenal of techniques. As a result, many systems implemented from simulation models turn out to have undesirable properties. By adopting the technology of process algebra, we can now test for such properties and make our models that much more reliable. Further since this testing is done on the static description, it will save a considerable amount of expensive debugging time. The problem is being increasingly seen as important — see http://www.cs.clemson.edu/~steve/ivandv.

Varieties of CCS exist which can cope with hard timings and functionality (value passing), but they are harder to reason with and are correspondingly less well mechanised [36, 61, 62, 63, 65, 100, 101]. See also [1, 2].

The following books and papers are recommended for background material. Walker [107] and Milner [59] for material on CCS; Manna [55, part II, pages 177–387] for a basic understanding of safety, liveness, and fairness properties and how to express them; and Stirling [96] for the link between CCS and process logics.

Acknowledgements

Much of the work reported herein has been carried out in collaboration with Rob Pooley and Chris Tofts [8, 10, 9] who are a joy to work with.

Appendix F

Modelling a shaping plant

We present a method for translating the synchronisation behaviour of a process oriented discrete event simulation language into a process algebra. Such translations serve two purposes. The first exploits the formal structure of the target process algebraic representations to enable proofs of such properties of the source system as deadlock freedom, safety, fairness and liveness which can be very difficult to establish by simulation experiment. The second exploits the denotational semantics to better understand the language constructs as abstract entities and to facilitate reasoning about simulation models. Here we give the intuition and the basic translation mechanisms using a variety of the Demos simulation language and the CCS and SCCS process algebras. The translations have been automated as SML programs and produce CWB compatible input allowing the automated checking of formal system properties.

F.1 Introduction

Many complex problems are studied by building simulation models that are intended to replicate selected aspects of their behaviour [14, 19, 29, 31, 49, 50, 67, 79, 84, 85, 89, 109, 110]. Once we have a model, the first step is to validate it, that is show it exhibits certain expected behaviours. The standard practice of running the simulation “enough” times cannot guarantee that all possible model paths will be covered, so harmful behaviours such as deadlock may be missed by this approach. This paper works towards a more formal approach whose motivation is giving guaranteed answers to such questions as: “Can the system ever hang?” (deadlock), “Can the system spin forever” (livelock), “Can there ever be more than one train on this section of track at a time?” (safety), “Is it always possible to return to a certain state?” (liveness), “Once a request is made, will it be granted?” (fairness), etc. We are able to give guaranteed answers however the model may unfold on any run and under quite general circumstances (whatever the particular waiting time distributions and whatever the queueing disciplines).

In two previous papers [9, 10], we have given an operational semantics for the synchronisations of Demos [4], a typical process-based simulation language. These operational semantics are quite detailed and are best suited to serve as an implementation guide or for reasoning about an individual program run. In this paper we present more abstract denotational definitions of Demos. Our abstraction replaces the Demos model, say M, by a coarser generalisation, say M', in which structurally irrelevant actions (such as data collection) are omitted and the strict timings of M become arbitrary. Informally, we can think of M' as being akin to a worst case containment of M: if nothing bad can happen however M' unfolds, then our translation from M to M' guarantees that it cannot happen in M either. Once we have an algebraic characterisation of M', we can then check its characteristic properties using the modal-μ calculus [48] on the public domain Concurrency Workbench[1] (henceforth the *CWB* [21, 66]). Since M and M' are structurally the same, once M' has been corrected, it is straightforward to rectify the actual model M.

The most popular styles of programming simulation models have been the activity based approach of (E)CSL [17, 23, 29, 79, 99], the event based approach of GASP and early Simscript [85, 43], and the process based approach of GPSS, Simscript II.5 and Simula [89, 18, 6]. However it is quite straightforward to translate between these styles [7, 57]. Hence, without loss of generality, we concentrate upon the process based approach as typified by the Demos extension to Simula.

In this paper we give a formal denotational semantics for a variety of Demos by providing translation schemata from Demos down to CCS [59] and to SCCS [58], two of the simplest process algebras both of which are mathematically well-defined. These semantics provide us with a system description in CCS/SCCS that extends, but faithfully contains, their original description in Demos [11]. Both random delays and queueing disciplines are generalised in the translation. Informally we replace all Demos random drawings by "some time later" in CCS/SCCS and track all possible ways in which the actions of the constituent processes may interleave as the model unfolds.

Given such a translation schema, we can exploit the CWB to examine the properties and consequences of the underlying simulation and thus make our models that much more reliable. Translation into a formal system also permits proofs of properties of constructs within Demos (see section 4), as well as of particular Demos programs.

The disadvantage of using a coarser model is that it will permit some behaviours that are unrealisable in practice. However every possible failure in the original M will be captured by the translation M'. A more refined target process algebra should be used when errors unrealisable in M occur in M'. Richer process algebras can be used to account for timing [65] or probabilistic [100, 102] behaviour of systems, but their descriptions are usually much more complicated. In most cases, systems failures result from underlying interaction errors rather than specific problems and can be captured in the simpler process algebras we espouse [8, 9, 10, 11, 103].

[1] The Edinburgh CWB home page is `www.dcs.ed.ac.uk/home/cwb`.

The presentation is in two parts. Part I (sections 1..4) is motivational. In it we present the various notations, hand translations of the core synchronisations, and demonstrate model checking. The translation methods needed for more advanced synchronisations are technically more mature and are held over to part II (sections 5..8). We introduce the core synchronisations of Demos in section 2 through an example. Section 3 introduces CCS using the same example, shows that the model deadlocks and how to eliminate the deadlock. We then introduce our second process algebra, SCCS, and in section 4 show how to check for a variety of properties in the modal-μ calculus.

Part II (sections 5..8) formalises the hand translations exhibited in the part I for both CCS and SCCS. Section 5 gives formal translations for core Demos. Section 6 adds wait until and section 7 conditional commands. A normal form for Demos is presented as an application. Using these schemata, we can translate a complete Demos model to a process algebra and use the translation to formally derive properties of the original system. Finally we summarise the work completed.

F.2 Modelling in Demos

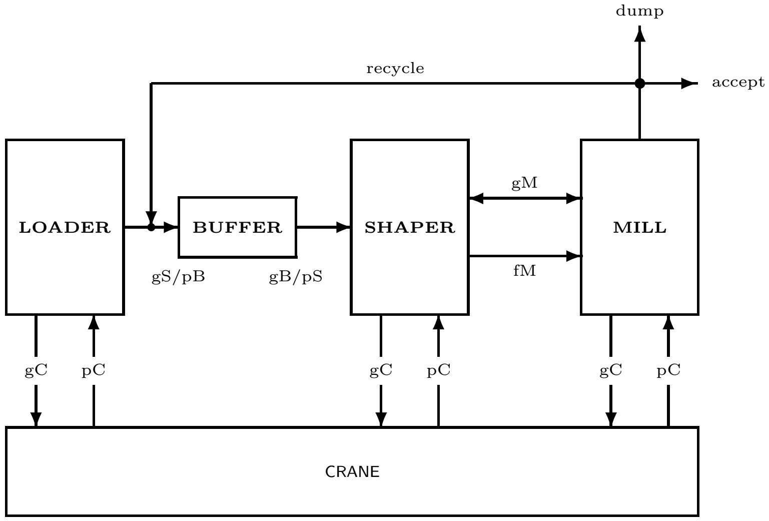


Figure F.1: System interactions

In Demos we model major components by *entities* (synonyms: agents, objects, processes, transactions) and abstract away minor components to *resources*. Demos has two types of resource: the *res* to model competition between entities and the *bin* to model co-operative (producer/consumer) entity interactions. The intention is that a res represents a fixed supply which entities fight to share, and a bin models “tokens” that can be handed on from a producing entity to consuming entity.

We illustrate the core of Demos by a small example (outlined in figure I.1) which is used as a running example throughout the paper. It is a compacted and idealised version of several models given in [4] and is sufficient to illustrate the basic synchronisations of Demos and to point out some the pitfalls of using simulation languages to model real concurrency.

A LOADER delivers scrap metal to a factory. When there is room for a load, it is placed in a 2-slot buffer. If both buffer slots are already filled, the LOADER must wait. When ready and there is (at least) one load, the SHAPER picks one up from the buffer, heats it and forms it into a hot, malleable billet. The billet is then handed over to a MILL for rolling into a flat plate. Because the billet must be at the right temperature for rolling, we cannot permit it to be dumped by the SHAPER and left to wait for the MILL to be ready — the SHAPER and MILL objects must synchronise for the handover. Once the MILL has rolled the billet into a plate, it is inspected. If deemed good it is accepted; if not it is recycled back to the buffer area for another run through the SHAPER if there is space. In later versions of the model, it may be dumped instead.

All movements of the metal — scrap, billet, or plate — require the use of a CRANE. We use the following short names for these movements:

task name	from		to	carried out by
a2b	arrival	→	buffer	LOADER
b2s	buffer	→	shaper	SHAPER
s2m	shaper	→	mill	SHAPER&MILL
m2s	mill	→	stack	MILL
m2b	mill	→	buffer	MILL
m2x	mill	→	dump	MILL

Entities. Entities are major system components whose behaviours warrant modelling in detail. In Demos, these behaviours are described by lists of time consuming tasks. (Syntactically, items in list are separated by commas and enclosed in square brackets.) Behaviours may be repeated forever. Initial (informal) definitions of LOADER, SHAPER and MILL are:

```
LOADER   =   repeat [ startL, a2b ]
SHAPER   =   repeat [ b2s, heat, s2m ]
MILL     =   repeat [ s2m, roll, m2s or m2b ]
```

The SHAPER and MILL must synchronise to perform the movement *s2m*. This we model by a Demos master/slave cooperation. We choose the MILL as slave. It signals its readiness by *wait(m)*. As master, the SHAPER *coopt*s the MILL *m*. The master or slave that signals first is blocked until its partner arrives. The master carries out the task *s2m* on behalf of both entities, and then frees up the slave to go its own way by *free(m)*. A refined definition of the system with these changes highlighted is:

```
LOADER   =   repeat [ startL, a2b ]
SHAPER   =   repeat [ b2s, heat, coopt(m), s2m, free(m) ]
MILL     =   repeat [ wait(m), roll, m2s or m2b ]
```

Initially the MILL waits passively for the next billet. When the SHAPER has heated a billet, it coopts the MILL and they carry out the transfer *s2m* together. When the MILL is freed by the SHAPER's *free(m)*, it goes its own way carrying out the remainder of its tasks before waiting again.

Bins. The drop area is best modelled in Demos by a *bin*. In fact by two Demos bins: one to count the number of unbooked slots, and the other to keep track of the number of loads actually in there. *newB(slots, 2)* creates an initial pool of two free slots where loads can be deposited; *newB(buff, 0)* creates a load pool, initially empty. The first argument to newB names the bin, the second argument gives the initial size of the pool.

To load a buffer, a producer must first book a slot and then move the load into the buffer from whence it is available to be picked up. We model the LOADER actions by:

```
LOADER   =   repeat [ startL, gB(slots, 1), a2b, pB(buff, 1) ]
```

The LOADER first checks that there is a free slot *gB(slots, 1)* (and will be blocked if the buffer is currently full), moves the load into the guaranteed slot *a2b*, and then increments the number of loads by *pB(buff, 1)*. The call on *pB* would awaken the SHAPER if it were waiting for a fresh load. The code for SHAPER, the mirrored consumer process, is refined to:

```
SHAPER = repeat [ gB(buff, 1), b2s, pB(slots, 1), heat, coopt(m), s2m, free(m) ]
```

The SHAPER makes sure there is a load with *gB(buff, 1)*, moves it out by *b2s*, and then frees up the vacated slot with *pB(slots, 1)*, which would awaken a waiting LOADER, if any.

The correspondingly refined code for the MILL is:

```
MILL   =   repeat [ wait(m), roll, m2s or gB(slots, 1), m2b, pB(buff, 1) ]
```

Res(ources). Whereas bins enable us to model co-operation, we use the *res* to model competition. Resources are created by executing *newR* whose arguments name the resource and fix its size. *newR(crane, 1)* creates a res object named *crane* with one token. The token may be acquired by *gR(crane, 1)*. Such a request is granted immediately if sufficient of the resource is free and no other entity is blocked. Otherwise the requester is itself blocked and held in a queue local to the resource. There it remains until it is first in the queue and there is sufficient of the resource available for it to proceed. When a current user releases its share by *pR(crane, 1)*, the call on *pR* not only increments the resource pool but also unblocks any waiting entity whose request can be granted. An unblocked entity leaves the resource queue and enters the event list behind its unblocker and at the same clock time. If we include competion for the crane, our next model refinement is:

```
newB(buff,   0)
newB(slots,  2)
newR(crane,  1)

LOADER  = repeat [ startL,
                   gB(slots, 1),
                   gR(crane, 1), a2b, pR(crane, 1),
                   pB(buff,  1)
                 ]

SHAPER  = repeat [ gB(buff,  1),
                   gR(crane, 1), b2s, pR(crane, 1),
                   pB(slots, 1),
                   heat,
                   coopt(m),
                     gR(crane, 1), s2m, pR(crane, 1),
                   free(m)
                 ]

MILL    = repeat [ wait(m),
                   roll,
                   if accept then
                     then gR(crane, 1), m2a, pR(crane, 1),
                     else gS(slots, 1),
                          gR(crane, 1), m2b, pR(crane, 1),
                          pB(buff,  1)
                 ]

newP(L, LOADER, 0)
newP(S, SHAPER, 0)
newP(M, MILL,   0)
```

The above model description would be completed by defining appropriate distributions for the task times and their calls (holds), initialising, and setting a run length. But when the model is run, it might grind to a halt (deadlock). Can you see why?

When a simulation run results in deadlock, there is usually no explicit indication that this has happened. One has to infer the fact from reading the trace and the final report and then set about locating the cause and rectifying it. Given a more complex scenario, detecting the precise cause of a deadlock can be extremely difficult.

It may be buried in a welter of complexity, or it may be a rare event which depends crucially upon precise timings, and if the timings are sufficiently delicate, a potential deadlock may well be missed even in a large set of runs. At best this will be irritating and costly to remove, but in safety-critical systems it is also dangerous since most systems do not have stationary fail-safe behaviour. It is important find a methodology that will cope.

F.3 Modelling in process algebras

One can detect the possibility of deadlock directly from a model structure by using the techniques of process algebra. The idea developed here is to map a Demos program into CCS or SCCS (both are object-oriented and preserve the original model structure) and then to test the translation using the modal μ-calculus. In next subsections we introduce CCS and SCCS and show how to test for properties from a static code description rather than running the model. Once we have the mathematical apparatus to discover deadlock, we can also apply it to test models for livelock, fairness, and a variety of safety and progress properties.

F.3.1 Modelling in CCS

CCS is a process algebra developed by Milner over the last 20 years [60]. CCS is a very small language (just half a dozen syntactic rules) with a clean semantics and a rich equational reasoning system for analyzing behaviours and equalities. It permits descriptions of a system by the composition of its constituent parts called *agents*. CCS scales hierarchically and is hence well suited to abstract modelling. Although its scope is limited to modelling interactions rather than functionality, it has been used successfully to describe and reason about protocols and self-timed hardware.

The simplest agent in CCS is **0** which can do nothing. CCS has three ways of building more interesting agents.

Prefixing The agent $a.\overline{b}.\mathbf{0}$ can do an a action, then a $\overline{b}$ action, and after that nothing more. By convention we overline output actions. Recursive definitions are allowed so that the agent $GFclock \;=\; \overline{tick}.\overline{tock}.GFclock$ will omit a $\overline{tick}$ and then a $\overline{tock}$ forever.

We can model a unit resource by $Res \;=\; \overline{gR}.\overline{pR}.Res$. We think of Res as possessing a token and giving permission for it to be seized a $\overline{gR}$ action, and later giving permission for it to be returned by a $\overline{pR}$ action. After the return action, the resource evolves back into a Res and is available for seizing again.

Non-deterministic choice We use + to represent choice of action. Here is a description of a Bin of size 2 which can be incremented or decremented by 1.

$$\begin{array}{lclclcl} Bin_2 & = & & & \overline{gB}.Bin_1 & + & \overline{isB2}.Bin_2 \\ Bin_1 & = & \overline{pB}.Bin_2 & + & \overline{gB}.Bin_0 & + & \overline{isB1}.Bin_1 \\ Bin_0 & = & \overline{pB}.Bin_1 & & & + & \overline{isB0}.Bin_0 \end{array}$$

When the bin is not empty, we may decrement by $\overline{gB}$. When bin spaces are free, we may increment by pB. Further we may quiz the buffer as to its current state by the $\overline{is}$ actions.

Parallel composition The | operator allows the concurrent operation of agents. Parallel agents synchronize by handshaking. A *handshake* can occur when an action is an input to one agent, an output from another agent, and both are enabled. When a signal is hidden (syntactically via $\backslash\{\}$), the communication becomes local to those agents.

Here is a small system in which two LOADERs compete for a CRANE.

$$\begin{array}{lcl} CRANE & = & \overline{gC}.\overline{pC}.CRANE \\ LOADER & = & startL.gC.a2b.pC.LOADER \\ SYS & = & (CRANE \mid LOADER \mid LOADER) \setminus \{gC, pC\} \end{array}$$

SYS has three concurrent agents: two LOADERs and a CRANE, with gC and pC as handshakes. $\backslash\{gC, pC\}$ internalises the handshakes and only the LOADERs can signal on them. In contrast $startL$ and $a2b$ are "independent" actions which may be considered as self-determined (they can fire when they want). In the simulation model, explicit timing distributions are used for $startL$ and $a2b$. In CCS these actions last for an arbitrary time and the properties we prove are valid for all interleaving possibilities.

It is worth pointing out that in a CCS handshake on x, it does not matter which agent says x and which says $\overline{x}$. They are interchangeable. We have found it worthwhile in practice to be systematic and put output handshake actions ($\overline{x}$) in the resources and their corresponding input handshake actions in the entities.

Renaming We are permitted to reuse a definition. For example, our Demos model contains two resources of size 1, $PERMIT$ and $CRANE$. We may either define them from first principles as

$$\begin{array}{lcl} CRANE & = & \overline{gC}.\overline{pC}.CRANE \\ PERMIT & = & \overline{gP}.\overline{pP}.PERMIT \end{array}$$

or *rename* the existing standard template for a unit sized *Res*:

$$\begin{array}{rcl} CRANE & = & Res[gC/gR, pC/pR] \\ PERMIT & = & Res[gP/gR, pP/pR] \end{array}$$

in which the construction a/b means replace action b with action a.

Later on we shall exploit renaming to define *generic* objects and instantiate them by appropriate renaming functions. To ease its initial reading, we do not rename in the following CCS descriptions of our Demos model.

Version 1.

$$\begin{array}{rclclcl}
CRANE & = & \overline{gC}.\overline{pC}.CRANE & & & & \\
BUFF0 & = & \overline{isB0}.BUFF0 & + & \overline{pB}.BUFF1 & & \\
BUFF1 & = & \overline{isB1}.BUFF1 & + & \overline{pB}.BUFF2 & + & \overline{gB}.BUFF0 \\
BUFF2 & = & \overline{isB2}.BUFF2 & & & + & \overline{gB}.BUFF1 \\
SLOTS2 & = & \overline{isS2}.SLOTS2 & & & + & \overline{gS}.SLOTS1 \\
SLOTS1 & = & \overline{isS1}.SLOTS1 & + & \overline{pS}.SLOTS2 & + & \overline{gS}.SLOTS0 \\
SLOTS0 & = & \overline{isS0}.SLOTS0 & + & \overline{pS}.SLOTS1 & & \\
LOADER & = & \multicolumn{5}{l}{startL.gS.gC.a2b.pC.pB.LOADER} \\
SHAPER & = & \multicolumn{5}{l}{startS.gB.gC.b2s.pC.pS.heat.S2M} \\
S2M & = & \multicolumn{5}{l}{\overline{gM}.gC.s2m.pC.\overline{fM}.SHAPER} \\
MILL & = & \multicolumn{5}{l}{startM.gM.fM.roll.(M2S + M2B)} \\
M2S & = & \multicolumn{5}{l}{ok.gC.m2s.pC.MILL} \\
M2B & = & \multicolumn{5}{l}{no.gS.gC.m2b.pC.pB.MILL} \\
RESOURCES & = & \multicolumn{5}{l}{CRANE \mid BUFF0 \mid SLOTS2} \\
ENTITIES & = & \multicolumn{5}{l}{LOADER \mid SHAPER \mid MILL} \\
SYS & = & \multicolumn{5}{l}{(RESOURCES \mid ENTITIES)} \\
 & & \multicolumn{5}{l}{\quad \backslash \; \{gM, fM, gB, pB, gC, pC, gS, pS,} \\
 & & \multicolumn{5}{l}{\qquad isB0, isB1, isB2, isS0, isS1, isS2\}}
\end{array}$$

Table F.1: Model (version 1) in CCS

We enter the model description (table I.1) into the CWB and try the built-in test for finding deadlock *fd*.

```
fd SYS;
--- startS startL tau<gS> tau<gC> a2b tau<pC> tau<pB> tau<gB>
                          tau<gC> b2s tau<pC> tau<putS> heat
```

```
startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
startL startM tau<gM> tau<gC> s2m tau<pC> tau<fM> roll no
startS tau<gB> tau<gC> b2s tau<pC> tau<putS> tau<gS>
                           tau<gC> a2b tau<pC> tau<pB> heat
startL ---> 0
```

The traces show how the system evolves (by the shortest route) into its final deadlocked state. Handshakes are shown as `tau<gS>` — the `tau` saying "handshake" and `<gS>` indicating which one.

By following the trace, we see that the LOADER has started five times; two loads have worked their way through the buffer, one is in the MILL, and one is in the SHAPER. Two others are in the buffer, which thus has no free slots. The 5th load is waiting on gS to gain entry to the buffer but is stuck. The SHAPER has completed one cycle, and handed over one billet to the MILL. The SHAPER is stuck on $\overline{gM}$ — the MILL is not ready. Why? Well it has rolled, rejected the result and is waiting on a buffer space (gS).

Version 2. The only definition to change is that of the MILL (table I.2) in which we use the $is?$ check on the slots to dump a plate if it is faulty and no slot is available (M2X); otherwise (M2B), we have tested positively for a slot by $(isS2 + isS1)$, claim the slot and move the faulty plate into it.

$$\begin{array}{rcl}
MILL & = & startM.gM.fM.roll.(M2S + M2BX) \\
M2S & = & ok.gC.m2s.pC.MILL \\
M2BX & = & no.((isS2 + isS1).M2B + is0.M2X) \\
M2B & = & gS.gC.m2b.pC.pB.MILL \\
M2X & = & gC.m2x.pC.MILL
\end{array}$$

Table F.2: Model (version 2) in CCS

But this system too deadlocks. Since the claim and the seizure (either $isS2.gS$ or $isS1.gS$) are not atomic, one parallel interleaving will permit the MILL to make the latter check and then interleave a gS from the LOADER. The MILL is now blocked on its gS action, but the LOADER can cycle round to deadlock itself on another gS action.

As with many process-oriented simulators, the semantics of Demos keep an entity active as current until it itself decides to move on. Thus interleaving is entirely dictated by the current object. The above potential deadlock would not be picked up by Demos. But CCS examines every possible interleaving and does pick up this one:

```
--- startS startL tau<gS> tau<gC> a2b tau<pC> tau<pB> tau<gB>
                      tau<gC> b2s tau<pC> tau<putS> heat
    startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
    startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
    startL startM tau<gM> tau<gC> s2m tau<pC> tau<fM> roll no
    startS tau<gB> tau<gC> b2s tau<pC> tau<putS> tau<isS1>
           tau<gS> tau<gC> a2b tau<pC> tau<pB> heat
   startL ---> 0
```

with the entities no further on than last time.

Version 3. The usual way to make sure that a sequence of actions is not interruptible, is to wrap them inside calls to a semaphore (the $PERMIT$ res). The fresh solution (we note only changed definitions) is given in table I.3.

$$\begin{array}{rcl}
PERMIT & = & \overline{gP}.\overline{pP}.PERMIT \\
 & & \\
LOADER & = & startL.gP.(A2B + LWAIT) \\
A2B & = & (isS2 + isS1).gS.gC.a2b.pC.pB.pP.LOADER \\
LWAIT & = & isS0.pP.LOADER \\
 & & \\
MILL & = & startM.gM.fM.roll.(M2S + M2BX) \\
M2S & = & ok.gC.m2s.pC.MILL \\
M2BX & = & no.gP.((isS2 + isS1).M2B + is0.M2X) \\
M2B & = & gS.gC.m2b.pC.pB.pP.MILL \\
M2X & = & gC.m2x.pC.pP.MILL \\
 & & \\
RESOURCES & = & CRANE \mid BUFF0 \mid SLOTS2 \mid PERMIT \\
SYS & = & (RESOURCES \mid ENTITIES) \\
 & & \quad \setminus \{gM, fM, gB, pB, gC, pC, gP, pP, gS, pS, \\
 & & \qquad isB0, isB1, isB2, isS0, isS1, isS2\}
\end{array}$$

Table F.3: Model (version 3) in CCS

Having acquired permission, the LOADER makes sure there is a slot before calling gS and buffering its load. Only then does it release the semaphore. If there is no slot currently free, it releases the semaphore and tries again later. Similarly, the MILL only goes ahead with gS if there is a free slot; if not it dumps the load before releasing the semaphore. Testing on the CWB confirms that this model is deadlock free. Once that is established, a variety of other property checks can be carried out (as in section 4).

Version 4. Since the original Demos was embedded in Simula, it was easy to extend the language by tailor-made constructs. We follow that lead in table I.4 and modify the definition of $SLOTS$ in such a way as to permit the MILL to request and seize a free slot ($hasS2$ or $hasS1$) in one action; on the other hand $hasS0$ returns at once so that dumping can go ahead.

$$
\begin{array}{rcl}
SLOTS2 & = & \overline{hasS2}.SLOTS1 \qquad\qquad\qquad\quad + \ \overline{gS}.SLOTS1 \\
SLOTS1 & = & \overline{hasS1}.SLOTS0 \ + \ \overline{pS}.SLOTS2 \ + \ \overline{gS}.SLOTS0 \\
SLOTS0 & = & \overline{hasS0}.SLOTS0 \ + \ \overline{pS}.SLOTS1 \\
LOADER & = & startL.gS.gC.a2b.pC.pB.LOADER \\
 & & \\
MILL & = & startM.gM.fM.roll.(M2S + M2BX) \\
M2S & = & ok.gC.m2s.pC.MILL \\
M2BX & = & no.((hasS2 + hasS1).M2B + has0.M2X) \\
M2B & = & gC.m2b.pC.pB.MILL \\
M2X & = & gC.m2x.pC.MILL \\
 & & \\
RESOURCES & = & CRANE \mid BUFF0 \mid SLOTS2 \\
SYS & = & (RESOURCES \mid ENTITIES) \\
 & & \quad \backslash\ \{gM, fM, gB, pB, gC, pC, gS, pS, \\
 & & \qquad isB0, isB1, isB2, hasS0, hasS1, hasS2\}
\end{array}
$$

Table F.4: Model (version 4) in CCS

The CWB reports no deadlock on this model. Whilst it is easy to mimic such a tailored Demos monitor in CCS, this is usually left as a last resort as it would also imply tailored extensions to the translation toolkit.

F.4 Checking the translated model

Once we have a model description in CCS, we can check its characteristic properties using the CWB. Claims take the form *SYS* $\models$ *property.* The CWB examines SYS from its initial state and checks to see if the logical property holds. The first step on the CWB is to minimise the CCS definition down to the equivalent state machine with the least number of states, henceforth SYS'. The theory behind CCS assures us that there is but one such machine and (loosely speaking) that its non-τ properties will be preserved. The minimised machines are often orders of magnitude smaller than the composed definitions (about 10× in our case) so minimisation speeds up property checking. Typical simple "one-move" checks are *SYS'* $\models \langle startL \rangle\, T$ which asks "Can SYS' make an initial move of startL?" which would return true, as would *SYS'* $\models \langle - \rangle\, T$ which asks "Can SYS' make any initial move at all?". The notation extends in an obvious way so that we may make two move checks such as *SYS'* $\models \langle startL \rangle \langle startS \rangle\, T$ which asks "Having made a startL move, can SYS' then make a startS move?" (true) and *SYS'* $\models \langle startL \rangle \langle startL \rangle\, T$ which returns false since SYS' cannot make successive startL moves. Whereas $\langle a\rangle$ tracks one a move from a state (one path), its dual [a] follows all a moves from a state.

It is possible (see [94, 97]) to define macros in modal-μ which traverse all the reachable states of a CCS agent. The most used are:

S $\models$ BOX p returns true only if S and every state reachable from S has property p. Our check for deadlock is simply *SYS'* $\models$ $BOX\langle-\rangle T$ — can each and every reachable state make a move?

S $\models$ POSS p returns true only if S or at least one state reachable from S has property p. Thus *SYS'* $\models$ $POSS\langle roll\rangle T$ asks if it is possible under some system unfolding to execute a roll action.

Note that BOX and POSS are duals; any property formulated with one operator can be reformulated in terms of the other. The dual check for deadlock is *SYS'* $\models$ $\neg POSS(\neg\langle-\rangle T)$ or *SYS'* $\models$ $\neg$*POSS([-]F).*

S $\models$ EVENT p returns true only if S has property p or else p holds at some state on every path from S. In otherwords, however the system unfolds, you must eventually reach a state with property p. *SYS'* $\models$ $EVENT\langle roll\rangle T$ asks whether SYS' always reaches a state capable of a roll action.

S $\models$ PATH p returns true there is a loop of actions with property p reachable from S. Thus *SYS* $\models$ $PATH\langle\tau\rangle T$ returns true if SYS is livelocked (has a cycle of τ moves). Since minimisation mainly removes chains of internal handshakes, we have to be careful when checking for livelock to operate with the unminimised definition, since the livelock check explicitly looks for τ operations.

PATH is actually more general than stated — there may also be a chain of states with property p leading to a deadlocked state. Note that EVENT and PATH are duals.

It is possible to combine the above macros for extra expressiveness. Safety, liveness, progress, and fairness properties require the model to be decorated with trace actions at appropriate points. ADVANCE and HOLD actions of GPSS and Demos models serve this purpose. Corresponding to every Demos $hold(T.sample)$, we write T in the CCS description, together with say $startP$ prefixing each agent definition P. Typical tailored checks might now be:

Progress checks to make sure that each process will eventually carry out a certain action.

$$\begin{array}{lll} SYS & \models & BOX[startL](EVENT\langle a2b\rangle T) \\ SYS & \models & BOX[startL](EVENT\langle startL\rangle T) \end{array}$$

read as “in every state following an action $startL$ (BOX[startL]), (EVENT)ually one must reach a state permitting an $a2b/startL$ action however the system unfolds”.

Safety tests to see that bad things cannot happen. We may want to check that it is never possible to get two crane movements to the buffer as consecutive moves:

$$SYS \quad \models \quad BOX[a2b][m2b]F$$

read as “however the system unfolds, in every state following an $a2b$ action (BOX[a2b]) an immediate $m2b$ action is impossible”.

Liveness tests to see that good things may happen (e.g. each movement from the MILL may be accepted).

$$\begin{array}{lll} SYS & \models & \mathrm{BOX(POSS}\langle m2s\rangle \mathrm{T)} \\ SYS & \models & \mathrm{BOX(POSS}\langle m2b\rangle \mathrm{T)} \\ SYS & \models & \mathrm{BOX(POSS}\langle m2x\rangle \mathrm{T)} \end{array}$$

read as “from every state (BOX) there exists a path (POSS) to some state where action *m2s*, or *m2b*, or *m2x* is enabled ($\langle m2s/m2b/m2x\rangle$ T)”.

Fairness means that a system can not “spin” forever without enabling some particular input or output action. For any system S, and for a particular action a, this may be expressed as S $\models$ BOX EVENT$\langle a\rangle$T which reads “from every state (BOX) for each path (EVENT)ually there is a state in which a is enabled ($\langle a\rangle$T)”. E.g.

$$\begin{array}{lll} \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startL\rangle \mathrm{T)} \\ \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startB\rangle \mathrm{T)} \\ \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startM\rangle \mathrm{T)} \end{array}$$

read as “in every state (BOX) however the system unfolds (EVENT)ually one must reach a state permitting the LOADER/SHAPER/MILL to start its workcycle again.

Although the above tests may be combined into one large statement of the system properties, it is much clearer and more readable to display them one at time, as above. The following books and papers are recommended for background material on property checking. Walker [107] and Milner [59] for material on CCS; Clarke et.al [20], and Manna and Pnueli [55, especially part II, pages 177–387] and [56] for a basic understanding of safety, liveness, and fairness properties and how to express them; Stirling [93, 94, 95, 97, 98] for links between CCS and process logics; Lynch and Wing [53, 52, 108] for some practical applications.

Summary

We have demonstrated the core of Demos, a typical process oriented discrete event simulation language, and given informal translations to two different process calculi, CCS and SCCS. Such translations can be used to reason formally about the properties of original model on CWB and ensure that deadlock and livelock can never arise, and that specific safety (e.g. mutual exclusion), fairness and progress properties hold. Typical templates were exhibited for carrying out these checks.

Appendix G

Yet another factory

Discrete event simulations models are collections of objects whose actions can unfold in many ways, some obvious, some mysterious. In this paper we look at a way of developing a complex model by first describing it in a process algebra (here, CCS). CCS descriptions are isomorphic to the eventual program code — object for object at the structural level, and line by line internally. From a CCS description (text) it is straightforward to check out all possible ways in which a model can develop *no matter what timing distributions and queueing disciplines are selected* and ensure that deadlock and livelock can never arise, and that the system has particular mutual exclusion, fairness and progress properties. As simulation methodologies are increasingly being used to design and implement complex systems of interacting objects, the ability to perform such verifications is of increasing methodological importance. Checking for such properties by running a model is manifestly impractical.

G.1 Introduction

Simulation is widely used for studying the behaviour and performance of complex systems, and even developing software for them. One of the underlying problems of the simulation methodology is that of ensuring the correctness of the representation of the system under study, since, in general, simulation systems do not admit formal proofs of their properties. In this paper we present by example a methodology for developing simulation models which encourages different decompositions to be investigated early on and their key properties (deadlock, safety, progress) to be checked prior to detailed coding and lengthy runs. It is based upon a series of papers with Chris Tofts which have given operational and denotational semantics for OO simulation language constructs and a containment relation between them. Operational definitions are quite detailed and may be used to guide an implementation or argue in detail the precise way a particular program should unfold. Denotational definitions are much more abstract than operational definitions with detailed timings replaced by “now, or some arbitrary time later” and all queues (including the event list) running under random selection. Denotational definitions are thus useful for reasoning about properties that will hold under all possible runs of a model whatever the timing delays and whatever the queueing discipline imposed. A formal translation schema permits the automatic construction of a process algebraic representation of the underlying simulation model which can then be checked for freedom from deadlock and livelock, as well as system-specific safety and liveness properties.

By providing a formal representation of a system described in Demos we permit formal reasoning about and proofs of properties of those systems. Given that these properties are often used to make cost or safety critical decisions, we need to be sure that predictions are genuine and not the result of some programming artifact. CCS is well-suited to this task, as it provides a succinct formal notation within which captures the behaviour of the underlying system. Furthermore, formal description of system properties can be automatically checked against its implementation as described by CCS.

In this paper we introduce the basics of CCS in section 2, use them to develop an abstract representation of a model in sections 3 and 4, show how to check for some model properties in section 5. Demos code for the model is given in an appendix.

G.2 CCS

We introduce the CCS notation in the context of the running example sketched below.

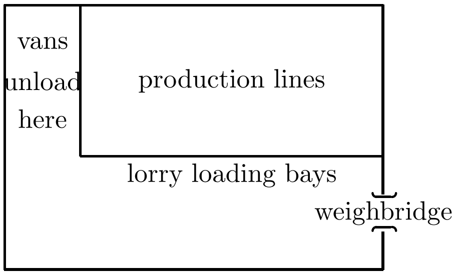


Figure G.1: Factory layout

A factory has one entrance guarded by a weighbridge over which all incoming and outgoing vehicles must pass. Only one vehicle can move in this area at a time. Aluminium sheets are delivered to the factory in vans and loaded into hoppers. The hoppers are fed onto production lines, their contents formed into cans, filled, capped and then placed in containers. The containers are removed by lorries.

The system decomposes naturally into three major object (process, agent) types: vans, lorries, and production lines. Simpler components are modelled as resources.

Nil: CSS builds from the agent *0* which can do nothing. From there CCS permits several ways of building more interesting agents.

Prefixing: The agent $a.\overline{b}.0$ can do an a action, then a $\overline{b}$ action, and after that, nothing more. Recursive definitions are allowed. We can define a clock as: $Clock = \overline{tick}.\overline{tock}.Clock$ This clock ticks and then tocks forever.

Here are abstract descriptions of the vans, lorries, and production lines that appear in our model.

$$\begin{array}{lcl} Van & = & enter.unload.leave.nextV.Van \\ Lorry & = & enter.load.leave.nextL.Lorry \\ PLine & = & first.second.PLine \end{array}$$

Vans carry out the actions $enter$, $unload$, $leave$, and $nextV$ repeatedly. A lorry carries out the actions $enter$, $load$, $leave$, and $nextL$ repeatedly. A production line carries out the actions $first$, $second$ repeatedly. Each action may take an arbitrary time but for each individual object they can only occur in the stated order.

Resources We model the weighbridge by:

$$\begin{array}{lcl} W & = & \overline{gW}.W' \\ W' & = & \overline{pW}.W \end{array}$$

A weighbridge exhibits cyclic behaviour forever: think of it as a token that may first be “got”, but then must be “put” before it can be “got” again. The weighbridge may only be “got” when in state W whereupon it moves to state W' in which it may only be “put”.

We now generalise definition to that of a resource of size n that may be got/put unit chunks.

$$\begin{array}{lclcl} R_n & = & \overline{gR}.R_{n-1} & & \\ R_{n-1} & = & \overline{gR}.R_{n-2} & + & \overline{pR}.R_n \\ & \ldots\ldots & & & \\ R_k & = & \overline{gR}.R_{k-1} & + & \overline{pR}.R_{k+1} \\ & \ldots\ldots & & & \\ R_1 & = & \overline{gR}.R_0 & + & \overline{pR}.R2 \\ R_0 & = & & & \overline{pR}.R1 \end{array}$$

In state R_k, k units of the resource are available and $n-k$ have been taken. Initially the resource in state R_n and may only be acquired. In the $n-2$ intermediate states, a unit chunk may be returned (through a $\overline{pR}$ action) or acquired (through a $\overline{gR}$ action). When empty (in state R_0) the resource may only be incremented.

Competition: We enforce mutual exclusion in a system comprising several vans and/or lorries by enclosing the $enter$ and $leave$ actions of these object classes in **handshake** calls to get and put the weighbridge as below:

$$\begin{array}{lcl}
W & = & \overline{gW}.\overline{pW}.W \\
Van & = & gW.enter.pW.unload. \\
 & & \quad gW.leave.pW.nextV.Van \\
Lorry & = & gW.enter.pW.load. \\
 & & \quad gW.leave.pW.nextL.Lorry \\
PLine & = & first.second.PLine \\
SYS & = & (W \mid Van \mid ... \mid Lorry \mid ... \mid PLine..\} \\
 & & \backslash\{gW, pW\}
\end{array}$$

The rules of the handshake game are simple: we use names and co-names (same name without/with the overline) for handshakes and list the all handshakes when defining the complete system. A handshake action can only fire when its “partner” is available. Thus when a van or a lorry has executed as far as a gW action, it can only progress when the weighbridge is in state W and can carry out a $\overline{gW}$ action — neither handshake can progress alone. The handshake operation will move the object past its gW into its *enter* or *leave* action *and* flip the weighbridge into state W' where it shows $\overline{pW}$. A handshake is strictly between two objects, so the even if two or more vans and lorries are showing their individual gW actions, only one may progress at a time.

Co-operation: Parallel agents may also rendezvous to carry out a task jointly, as with lorries and production lines over fillings. A production line may not start a second filling until it can load the first onto a waiting lorry. The handshaking style is retained, but this time it is directly between co-operating objects and does not require an intermediate resource.

$$\begin{array}{lcl}
Lorry & = & gW.enter.pW.gL.pL. \\
 & & \quad gW.leave.pW.nextL.Lorry \\
PLine & = & first.\overline{gL}.second.\overline{pL}.PLine \\
SYS & = & (W \mid Van \mid ... \mid Lorry \mid ... \mid PLine..\} \\
 & & \backslash\{gW, pW, gL, pL\}
\end{array}$$

G.3 The model structure

Since there is interplay amongst all three entities, we present the process interactions in two gentle iterations. The vans and lorries interact only at the plant entrance where they compete for the weighbridge on the way in and on the way out. We ignore any time spent by van and lorry movements within the plant. This interaction has already been modelled with the weighbridge described as a token of size 1.

Van/production line interplay is also straightforward. Van require empty hoppers to unload (each van fills three hoppers) full hoppers are passed to the production lines. The production line empties a full hopper per batch and returns it back to the van unloading area. We assume instantaneous hopper movements between vans and production lines.

Resources may used in a different way to model production line scenarios where one object produces items for its successor(s) to fettle further. In this style, the resource counts how much the producer is ahead of the consumer and is used to block the consumer if there is nothing for it to fettle, and block the producer should it get “too far ahead”. In our model, vans acquire empty hoppers, fill them with aluminium plates and pass them on as full hoppers to a production line. A production line, drains full hoppers and returns them as empty for further use by another van. We use two resources EH and FH which follow the general template for a resource but with the signals $\overline{gR}$ and $\overline{pR}$ renamed to $\overline{gE}$ and $\overline{pE}$ and $\overline{gF}$ and $\overline{pF}$.

$$\begin{array}{lcl} EH & = & R_{8,5}[gE/gR, pE/pR] \\ FH & = & R_{8,3}[gF/gR, pF/pR] \end{array}$$

There are 8 hoppers in the system. Initially 5 are empty and three are full. The notation $R_{8,5}$ indicates that EH is a resource of limit 8, and is initially in state 5 et sim.

CCS code for the relevant segments of Van and PLine are:

$$\begin{array}{lcl} Van & = & gW.enter.pW. \\ & & \qquad gE.pF.gE.pF.gE.pF. \\ & & \qquad\quad gW.leave.pW.nextV.Van \\ PLine & = & gF.first.pE.gF.second.pE.PLine \end{array}$$

in which each van will thrice get an empty hopper and pass on a full one, and a production line will twice get a full hopper and empty it during each work cycle.

The production line/lorry interplay is the hardest to understand. Once started each batch takes 25 minutes to complete, but it takes 10 minutes before the first crate emerges. There is no storage space at the end of a production line — crates are put straight onto lorries. Thus production is halted unless a lorry is there. Once a lorry has been “claimed” it is filled with two hoppers worth of crates.

$$\begin{array}{lcl} Lorry & = & gW.enter.pW.gL.pL.gW.leave.pW.Lorry \\ PLine & = & gF.first.\overline{gL}.pE.gF.second.pE.\overline{pL}.Pline \end{array}$$

G.4 Refined model

Having detailed their co-operations, we are now in position to refine the entity outlines one by one.

class van

The vans arrive at the plant periodically. Once across the weighbridge (which takes two minutes in or out), each van goes to the rear of the factory to an unloading area where its load is removed with the assistance of a crane. The load of aluminium

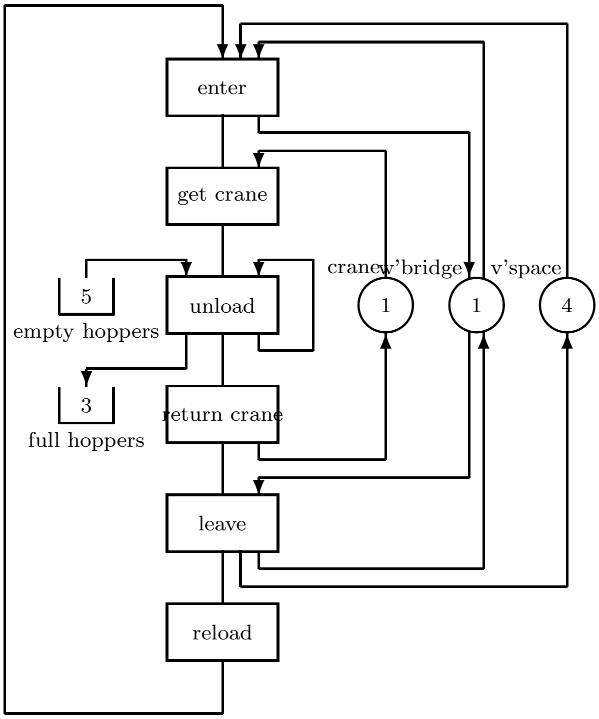


Figure G.2: Van activity diagram

sheets fills three empty hoppers one by one. Once full, a hoppers is passed onto the production lines. Each van then leaves, again passing over the weighbridge. To prevent congestion, at most four vans are allowed in the factory grounds at a time.

A pool of seven vans serves the factory. A resource VS (vanspaces) of limit 4 is used to limit the number in the factory grounds at any one time. Unloading into the 3 empty hoppers requires a crane (a resource of size 1) and may start even if only one or two empty hoppers are available. The crane is only released by its owning van when three hoppers have been filled.

$$\begin{aligned} VS &= R_{1,1}[gV/gR, pV/pR] \\ Crane &= R_{1,1}[gC/gR, pC/pR] \\ Van &= gV.gW.enter.pW. \\ &\quad\quad gC.gE.pF.gE.pF.gE.pF.pC. \\ &\quad\quad\quad gW.leave.pW.pV.nextV.Van \end{aligned}$$

class lorry

A full hopper fits onto a production line (of which there are five). The aluminium sheets are removed from the hopper and processed one by one. As the sheets pass

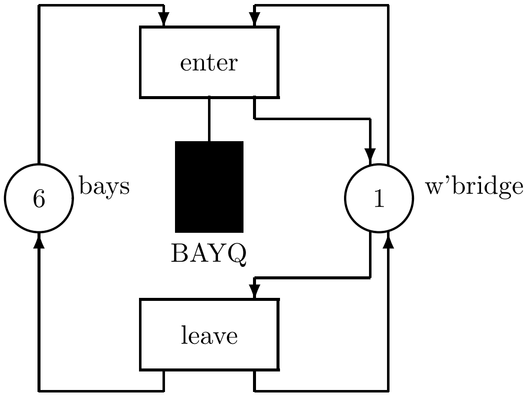


Figure G.3: Lorry activity diagram

down the line, they are formed into cans, filled, and capped. It takes two hoppers to fill one container. If all goes smoothly, the processing time per hopper is 25 minutes.

The containers are loaded onto articulated trucks. The trucks wait outside the factory until a loading bay is free. They take three minutes to cross the weighbridge (in and out) and then manoeuvre into a loading bay. When the lorry is loaded, it departs via the weighbridge.

Lorries enter the factory grounds when they have a bay (there are 6 bays in the model) and the weighbridge. Once in, they accept two containers and then leave.

$$\begin{array}{lcl} Bays & = & R_{6,6}[gB/gR, pB/pR] \\ Lorry & = & gB.gW.enter.pW.gL.pL.ten. \\ & & \quad gW.leave.pW.pB.Lorry \end{array}$$

class production

When a hopper is put on the line, the plant starts producing cans. The first can is ready ten minutes later. If there is no waiting container, production is halted but can continue without penalty when one arrives. After a further fifteen minutes, the hopper has to be replaced with another possible production line halt. Twenty five minutes later the second hopper will have been emptied, but the last can will not arrive at the end of the production line until another five minutes have elapsed.

A second container load can be started on the production line immediately after the first if required, there being no need to wait for the final can of the first load to be ready before the first can of the second can be started.

$$PLine \quad = \quad gF.first.\overline{gL}.pE.gF.second.pE.\overline{pL}.Pline$$

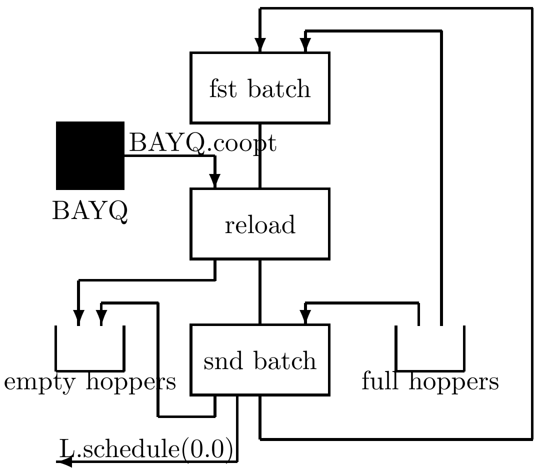


Figure G.4: Production line activity diagram

The CCS description of the whole system may now be pieced together by composing the various resources and processes and hiding off the handshake lines.

$$\begin{array}{lll}
Resources & = & EH \mid FH \mid Crane \mid VS \mid W \mid Bays \\
HS & = & \{gE, pE, gF, pF, gC, pC, gL, pL, \\
 & & \quad gV, pV, gW, pW, gB, pB\} \\
 & & \\
Vans & = & Van \mid ... \mid Van \\
Lorries & = & Lorry \mid ... \mid Lorry \\
Plines & = & Pline \mid ... \mid Pline \\
Processes & = & Vans \mid Lorries \mid Plines \\
 & & \\
Model & = & (Resources \mid Processes) \setminus HS
\end{array}$$

G.5 Checking the model

Once we have a model description we can check its characteristic properties using the Edinburgh Concurrency Workbench (CWB). Checks take the form

$$\text{Model} \models \text{property}$$

which examine the Model from its initial configuration and check to see if the logical property holds. Some of the simpler checks are

Deadlock:

$$\text{Model} \models \text{ALL} < - > \text{T}$$

which traverses every state reachable from the initial state (ALL) and asks "can you make a move" (expressed by $< - >$ T).

Livelock:

$$\text{Model} \models \text{PATH} < \tau > \text{T}$$

which traverses every state reachable from the initial state and tries to find a cyclic path (PATH) containing only handshake moves (expressed by $< \tau >$ T).

The above properties are posed the same way for all models. Safety, liveness, progress, and fairness properties are require the model to be decorated with "trace variables" at appropriate points. Very often these can be manufactured mechanically by inserting for every Demos

$$hold(D.sample)$$

$startD.endD$ in the CCS description, together with say $startVan$, $startLorry$, $startP$ prefixing each process description. (Actually here $nextV$, $nextL$ would serve that purpose.)

Typical tailored checks might now be

Progress checks to make sure that each process will eventually carry out a next action.

$$\begin{array}{lll} Model & \models & ALL[startEnter] \\ & & \quad (EV < endEnter > T) \end{array}$$

read as "in every state following an action startEnter (ALL [startEnter]) however the system unfolds (EV)entually one must reach a state permitting an *endEnter* action.

Safety which tests to see that bad things cannot happen. We may want to check that it is never possible to get two weighbridge users at the same time:

$$\begin{array}{lll} Model & \models & ALL[startEnter] \\ & & \quad (MUST\ endEnter\ startEnter) \end{array}$$

read as "in every state following an action startEnter (ALL [startEnter]) however the system unfolds, one must do an *endEnter* action before a *startEnter* action.

Liveness tests to see that good things may happen (e.g. each request may be accepted).

For any system *SYS*, and a particular action a, liveness of action a may be expressed as:

$$Model \quad \models \quad \text{ALL POSS } \langle a \rangle \text{ T}$$

read as "from every state (ALL) there exists a path (POSS) to some state where action a is enabled ($\langle a \rangle$ T)".

Fairness means that a system can not "spin" forever without enabling some particular input or output action. For any system *SYS*, and for a particular action a, this may be expressed as:

$$\text{Model} \quad \models \quad \text{ALL EV } \langle a \rangle \text{ T}$$

which reads as "from every state (ALL) for each path (EV)entually there is a state in which a is enabled ($\langle a \rangle$ T)". E.g.

$$Model \quad \models \quad ALL(EV < nextV > T)$$

read as "in every state (ALL) however the system unfolds (EV)entually one must reach a state permitting a van to finish is workcycle again.

G.6 Conclusions

Via an example, we have presented an approach to developing discrete event simulation models which entails first sketching its decomposition into objects, and then describing the synchronisation points in their code (mutual exclusions, producer/consumer interactions, rendezvous etc etc). The CCS description may now be entered into the CWB and straightforwardly checked to ensure that deadlock and livelock can never arise, and that the system has particular mutual exclusion, fairness and progress properties. Since the CWB explores all possible ways in which a model can unfold, these findings are valid whatever distributions are use, whatever queueing discipline is selected and for all possible runs. Once the model structure is deemed safe and sound, it is a trivial matter to render it into your favourite simulation notation (see appendix for mine) and to run that model knowing full well that deadlock etc cannot arise.

Model in Demos

The driving program includes distribution, resource and entity declarations and initialisations. Initially there are five empty hoppers and three full hoppers. The first lorry arrives at time zero; each lorry schedules the next. A seven-strong van fleet is established, weakly spaced in time, and five production lines are ready for action. The model is run for 8 hours with a cold start and an abrupt end.

```
external class Demos = "/usr/local/simulabin/demos.atr";

Demos
  begin
    ref(rdist) nextlorry, fill, nexttrip;
    ref(res) weighbridge, crane, bays, vanspaces;
    ref(bin) fullhoppers, emptyhoppers;
    ref(waitq) BAYQ;

    entity class van;
    begin
      integer k;

      vanspaces.acquire(1);
      weighbridge.acquire(1);
      hold(2.0);
      weighbridge.release(1);

      crane.acquire(1);
      for k := 1 step 1 until 3 do
      begin
        emptyhoppers.take(1);
        hold(fill.sample);
        fullhoppers.give(1);
      end;
      crane.release(1);

      weighbridge.acquire(1);
      hold(2.0);
      weighbridge.release(1);
      vanspaces.release(1);

      hold(98.0 + nexttrip.sample);
      repeat;
    end***van***;

    entity class lorry;
    begin
      new lorry("lorry").schedule(nextlorry.sample);

      bays.acquire(1);
      weighbridge.acquire(1);
      hold(3.0);
      weighbridge.release(1);

      BAYQ.wait;

      weighbridge.acquire(1);
      hold(3.0);
      weighbridge.release(1);
      bays.release(1);
    end***lorry***;

    entity class production;
    begin
      ref(lorry) L;

      fullhoppers.take(1);
      hold(10.0);

      L :- BAYQ.coopt;
      hold(15.0);
      emptyhoppers.give(1);

      fullhoppers.take(1);
      hold(25.0);
      emptyhoppers.give(1);
      L.schedule(10.0);
      repeat;
    end***production line***;

    integer k;

    nextlorry    :- new negexp("next lorry", 0.1);
    fill         :- new normal("fill hopper", 5.0, 1.0);
    nexttrip     :- new negexp("van return", 0.1);
    weighbridge  :- new res("weighbridge", 1);
    vanspaces    :- new res("van spaces", 4);
    crane        :- new res("crane", 1);
    bays         :- new res("bays", 6);
    fullhoppers  :- new bin("full hoppers",  3);
    emptyhoppers :- new bin("empty hoppers", 5);

    BAYQ         :- new waitq("await container");

    new lorry("l").schedule(0.0);

    for k := 1 step 1 until 7 do
      new van("v").schedule((k-1)*14.0);

    for k := 1 step 1 until 5 do
      new production("P-line").schedule(0.0);

    hold(480.0);
  end;
```

OUTPUT

```
                       clock time =    480.000
**********************************************************************
*                                                                    *
*                            r e p o r t                             *
*                                                                    *
**********************************************************************



                      d i s t r i b u t i o n s
                      *************************

title       /   (re)set/   obs/type     /        a/        b/     seed
next lorry        0.000     41 negexp        0.100            33427485
fill hopper       0.000     74 normal        5.000     1.000  22276755
van return        0.000     24 negexp        0.100            46847980

                          r e s o u r c e s
                          *****************

title       /   (re)set/   obs/ lim/ min/ now/  % usage/ av. wait/qmax
weighbridge       0.000    121    1    0    1    65.417      1.168    4
van spaces        0.000     24    4    0    3    40.525      0.000    1
crane             0.000     24    1    0    0    81.393      8.943    2
bays              0.000     33    6    0    0    88.570     12.900    4

                               b i n s
                               *******

title       /   (re)set/   obs/init/ max/ now/ av. free/ av. wait/qmax
full hoppers      0.000     73    3    3    0     0.516      5.477    5
empty hopper      0.000     71    5    8    2     2.571      0.219    1

                        w a i t   q u e u e s
                        *********************

title       /   (re)set/   obs/ qmax/ qnow/ q average/zeros/  av. wait
await contai      0.000     38     4     0      0.315    27      3.977
await contai*     0.000     38     3     1      0.578    12      6.743
```

Appendix H

Operational semantics for mini-demos

In this chapter (joint work with Chris Tofts [43,44]) we give an operational semantics for the synchronisation mechanisms of πDemos, a small process-oriented discrete event simulation language based upon Simula and Demos. The operational semantics gives a clear, concise and precise meaning to πDemos programs and have been extended to full Demos. The paper includes applications of the semantics as an implementation blueprint and in verifying the consistency of event list operations.

H.1 Background

Since they describe dynamically changing scenarios, discrete event simulations are difficult to program and to reason about. They are usually "debugged" (whatever that might mean) by extensive validation runs. We present the basis for an alternative approach — mathematical reasoning about simulation models. Since the pioneering work of Plotkin [76, 77] structured operational semantics has gradually emerged as a prominent method of specifying programming languages and reasoning about programs written in them. In [76], Plotkin showed how to describe everyday programming constructs (expressions, statements, procedure/function calls, objects) and in [77], he dealt with guarded commands and CSP flavoured parallelism. Milner [59] used operational semantics to describe the non-deterministic interleaving semantics of CCS, and with others, [63, 64], to describe SML, a modern (mainly) functional programming language. The text by Hennessy [37] serves as a good introduction to the basic techniques of operational semantics.

We extend this range to deal with the basic synchronisations of object-oriented discrete-event simulation. Giving the operational semantics of a complete simulation language, such as Simula [6] would require many pages, most of which would contain nothing new. What is needed is an operational semantics for that part of the problem of interest, and since the bugs that give the most trouble are those caused by scheduling and synchronisation problems, we concentrate upon giving simple and consistent formulation of the routines for event list scheduling and inter-process communication. Since it is unlikely that the intended audience (simulars) is experienced in reading and applying semantic definitions, we have chosen to present the development in stages. In this paper we use a stripped down version of Demos [4, 5], called

πDemos, to put across our mental model and explain the essence of the technique. The operational semantics of the synchronisations of full Demos language is given in a companion paper [10]. Alternative approaches based upon process logics are being explored in [101].

The paper is organised as follows: In section 2 we give the structure of πDemos programs and sketch its built-in facilities. In section 3 we present a simple πDemos model and explain how it is executed. The presentation gives insight into the structure of the semantic definition. In section 4 we give operational definitions for the scheduling and synchronisation facilities of πDemos. In section 5 we give some applications of semantic techniques: how to derive implementations from semantic definitions and how to verify the correctness of event list operations.

H.2 πDemos programs

In this section we introduce the facilities of πDemos informally using weighbridge access as a running example. Delivery vans arriving at a factory must pass over a weighbridge on entry. The weighbridge accepts one van at a time and each weighing operation takes 3 time units. Vans arrive at clock times 0 and 2 and the model is to be run for 6 time units. The complete πDemos program reads:

line no. πDemos code

```
1      MAIN  =
2            [  decP(van, [getR(W), hold(3), putR(W)]),
3               newR(W),
4               newP(V1, van, 0),
5               newP(V2, van, 2),
6               hold(6)
7            ]
```

πDemos programs have a particularly simple structure:

- the whole program (lines 1–7) is defined as a process called `MAIN` whose body is a list of commands, separated by commas, and enclosed in square brackets.
- a *static* section (lines 2–3) giving (a) templates for the process definitions (line 2 declares the class of `vans`) and (b) establishing the resources (line 3 creates a resource `W` representing the weighbridge)
- a *dynamic* section (lines 4–6) wherein (a) the individual processes are created and scheduled (line 4 generates a van named V1 and inserts it into the event list at time 0 and line 5 generates a van named V2 and inserts it into the event list at time 2) and the length of the model run is established ((line 6 sets the simulation run length at 6)

H.2.1 Notation

We have adopted certain notations from modern functional programming languages ([3, 16, 75, 74]) to express lists and sub-expressions.

Lists. The empty list is denoted by `[]`. When we wish to display a nonempty list in full, we enumerate it. The process body below with three actions:

```
[ getR(W), hold(3), putR(W) ]
```

is actually short for

```
getR(W)::hold(3)::putR(W)::[]
```

where `::` is the infix operator (usually called *cons*) used to glue atoms onto lists at their head. Most of the time we wish to focus upon the first action in a process body, since that will be its next action to be carried out, For this we use the technique of *pattern matching*: if we write

```
b::Body = [ getR(W), hold(3), putR(W) ]
```

then in the ensuing text, `b` is matched to the head of the list `getR(W)` and `Body` is matched to its tail `[ hold(3), putR(W) ]`.

Updating. We use the notation $\mathcal{S}$`[id/x]` to mean:

- case id $\in \mathcal{S}$: update the value of `id` $\in \mathcal{S}$ by `x`
- case id $\notin \mathcal{S}$: add the name `id` to $\mathcal{S}$ and initialise it to `x`

let x = e in E. We use the `let` notation `let x = e in E` to clarify the structure of complicated expressions, preferring, for example, to spell out

$$\text{exec(current::}\mathcal{EL}\text{, }\mathcal{R}\text{[id/RD(true,[])])}$$

in simple steps as

$$\begin{array}{lcll} \textbf{let } \mathcal{R}' & = & \mathcal{R}\text{[id/RD(true,[])]} & \textbf{in} \\ \textbf{let } \mathcal{EL}' & = & \text{current::}\mathcal{EL} & \textbf{in} \\ \quad \text{exec(}\mathcal{EL}'\text{, }\mathcal{R}'\text{)} & & & \end{array}$$

H.2.2 Processes

In the process-oriented approach to discrete event modelling, programs are collections of interacting processes which compete for resources with other processes before a task can be undertaken. Processes are given distinctive names and their bodies are described as a list of the tasks that they carry out.

Process classes are defined by the `decP` command. `decP(classId, classDef)` saves away the class body definition under the lookup name of `classId`.

New processes are generated by `newP` commands. `newP(id, classId, dt)` enters a new *event notice* for the process `id` into the event list delayed by `dt`. The second argument gives the lookup name for the class actions.

Each process in the model is represented by its own event notice, of the form `(id, PD(Body, Attrs, evTime))` where

1. `id` is a unique identifier, e.g. `MAIN`, `V1`, `V2`
2. `Body` is the sequence of actions in the process's body, e.g. `[ getR(W), hold(3), putR(W) ]`.
3. `Attrs` is a list of properties local to this specific object. In the sequel we will maintain as attributes, the names of resources acquired but not yet released. This enables checks to be made which ensure that resources already owned cannot be acquired again, resources must be acquired before they are released and that all acquired resources are eventually released.
4. `evTime` is a non-negative number fixing when the process is scheduled to carry out its next action. Every time we pass a time delay as argument to a function, we insert a check to ensure that it is indeed not negative. In some more modern programming languages, it would be possible to give event times a non-negative type obviating the need for such explicit checks.

As an example of this notation in action.

```
EL = [ (MAIN, PD([newP(V2,van,2),hold(6),hold(0),close],[],0)),
       (V1,   PD([getR(W),hold(3),putR(W)],             [],0))
     ]
```

displays the event list with two active processs:

1. `MAIN` scheduled to carry out the statement `newP(V2,van,2)` at time `0`
2. `V1` scheduled to carry out `getR(W)` also at time `0`

Both MAIN and V1 have empty attribute lists. The event notice at the head of the event list is called *current*. Above, this is

```
(MAIN, PD([newP(V1,van,2),hold(6),hold(0),close], [], 0))
```

We take the simulation clock time to be the event time of current. The πDemos executor is so framed that the next action to be executed is always the first action in the action list of the current event; in this case, newP(V2, van, 2). When this action has been carried out, the event list takes the form

```
EL = [ (MAIN, PD([hold(6),hold(0),close],  [],0)),
       (V1,   PD([getR(W),hold(3),putR(W)],[],0)),
       (V2,   PD([getR(W),hold(3),putR(W)],[],2))
     ]
```

in which there are three event notices. Notice that V2 has been scheduled at time 2 and the list of actions of MAIN has been decremented (in object oriented parlance, its local sequence control has moved past the last action). *time* remains unchanged by this action.

H.2.3 Resources

Mutual exclusion is implemented by getR/putR operations on a resource. For pedagogic simplicity, resources are always of size 1 in this presentation. The weighbridge resource is introduced by newR(W).

The resource W is aquired via a call getR(W). Requests are always considered on the first-come, first-served basis. A request is granted immediately if the resource is free. Otherwise the requester is blocked and held in a (hidden) queue local to the resource. There it remains until it is first in the queue and the resource is free again.

A call on putR(W) not only frees the resource but also unblocks the first waiting process, if any. An unblocked process leaves the resource queue, claims ownership of the resource, and enters the event list at the same clock time as its unblocker, but after it.

The most appropriate place to locate a blocked process is in a list local to the resource itself. It follows that the state of a resource is captured by a (name, descriptor) pair (id, RD(avail, Q)) where

1. id is a unique identifier, e.g. W

2. avail is a true/false (free/busy) flag

3. `Q` is a list of processes wanting to acquire the resource. Processes are queued first-come, first served.

Then

- `(W, RD(true, []))` represents a free weighbridge,
- `(W, RD(false, []))` represents a busy weighbridge with no blocked processes, and
- `(W, RD(false, (V2, PD(Body, Attrs, evTime’))::Q))` represents a busy weighbridge with `V2` at the head of the list of blocked processs. We represent a blocked process by its event notice. Its `evTime` field is not required but, if left, contains the time at which it was blocked which can be useful debug information.

Summary of πDemos commands.

command	::=	decP(classId, classDef)
	|	newP(id, classId, dt)
	|	hold(dt)
	|	newR(id)
	|	getR(id)
	|	putR(id)
	|	close

where:

decP(classId, classDef) defines a fresh class of process under the name `classId`. `classId` must be a fresh identifier.

newP(id, classId, dt) creates a new object, named `id`, and enters it into the event list at `time+dt`. The class body actions are looked up under the name `classId`. `id` must be a fresh identifier. `classId` must be already declared.

hold(dt) re-enters the current object into the event list at `time + dt`.

newR(id) establishes a new `resource`. `id` must be a fresh identifier.

getR(id) seeks to acquire the resource `id` on behalf of `current`. If the request cannot be met, `current` (the requesting process) is blocked. A blocked process has to wait until the resource is freed by a subsequent call on `putR(id)` by another process.

putR(id) frees the resource `id` and awakens the first process blocked on `id` (if any) who can now go into the event list after `current` but at the same simulation clock time.

close shuts down a simulation run.

At any given time, a process may be in one of two states: *scheduled* in the event list or *blocked* awaiting a resource to become available. As an illustration, figure 1 shows how the operations described above move processes between these states. The arrow for `getR` forks because a request may cause `current` to be blocked (subscript 1) or to be granted at once (subscript 2).

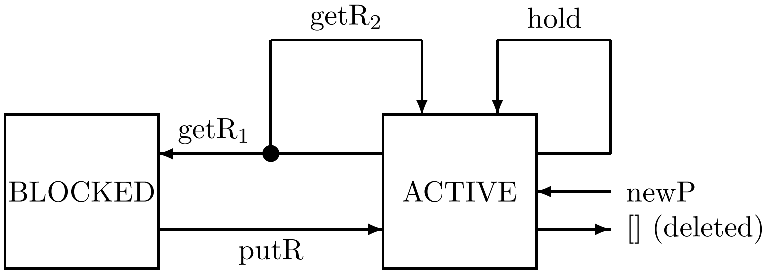


Figure H.1: The ways processes change state

H.3 Execution of a simple πDemos program

In order to give insight into our presentation of the semantics of πDemos, we now explain how the weighbridge program unfolds. NB. We purposely omit lookup information to simplify the presentation.

1. The system initialises itself by entering `MAIN` into the event list, with an empty attribute list, at clock time zero. An empty resource pool is also established.

   ```
   EL = [(MAIN, PD([ decP(van, [getR(W),hold(3),putR(W)]),
                     newR(W),
                     newP(V1,van,0),
                     newP(V2,van,2),
                     hold(6),
                     hold(0),
                     close
                   ], [], 0))]
   R  = []
   ```

   The main program has the five actions supplied explicitly by us

   ```
   [ decP(van, ...), newR(W), newP(V1,van,0), newP(V2,van,2), hold(6) ]
   ```

   which define the class of vans, initialise the weighbridge resource, create and schedule the two van processes into the event list, and then run the model for 6 time units. Two actions, `hold(0)` and `close` are automatically added to every user definition of `MAIN`. `hold(0)` permits all other actions scheduled for the system shut down time to complete, and `close` first calls a final report and shuts down the run gracefully.

2. The system is so framed that the next action to be executed is the first action of the current object, here `decP(van, [getR(W),hold(3),getP(W)])` which saves the class actions under the name `van`.

3. The next action is to create and initialise the weighbridge.

   ```
   EL = [(MAIN,PD([newP(V1,van,0),newP(V2,van,2),hold(6),hold(0),close],[],0))]
   R  = [(W,T,[])]
   ```

4. Then `newP(V1,van,0)` creates a new van process `V1` and schedules it for time 0.

```
EL = [ (MAIN, PD([newP(V2,van,2),hold(6),hold(0),close],[],0)),
       (V1,   PD([getR(W),hold(3),putR(W)],             [],0))
     ]
R  = [(W,T,[])]
```

5. The event list now has two members each scheduled for time 0. The next two steps create a second van at time 2

```
EL = [ (MAIN,PD([hold(6),hold(0),close],  [],0)),
       (V1,  PD([getR(W),hold(3),putR(W)],[],0)),
       (V2,  PD([getR(W),hold(3),putR(W)],[],2))
     ]
R  = [(W,T,[])]
```

6. and then reschedule `MAIN` for time 6.

```
EL = [ (V1,   PD([getR(W),hold(3),putR(W)],[],0)),
       (V2,   PD([getR(W),hold(3),putR(W)],[],2)),
       (MAIN, PD([hold(0),close],          [],6))
     ]
R  = [(W,T,[])]
```

Notice that `MAIN` has now done the first part its job (established all the dynamic entities in the system). Now it waits to shut down at the appropriate time. We now have a new `current` but the simulation clock time remains at 0.

7. `V1` now acquires the weighbridge and remembers that fact in its attribute list.

```
EL = [ (V1,   PD([hold(3),putR(W)],        [W],0)),
       (V2,   PD([getR(W),hold(3),putR(W)],[], 2)),
       (MAIN, PD([hold(0),close],          [], 6))
     ]
R  = [(W,F,[])]
```

8. `V1` now carries out `hold(3)` which moves it down the event list.

```
EL = [ (V2,   PD([getR(W),hold(3),putR(W)],[], 2)),
       (V1,   PD([putR(W)],                [W],3)),
       (MAIN, PD([hold(0),close],          [], 6))
     ]
R  = [(W,F,[])]
```

9. Again we have a new current, this time `V2`, and the simulation clock moves up to 2. `V2` is blocked. It is removed from the event list and waits on the resource `W`.

```
EL = [ (V1,   PD([putR(W)],       [W], 3)),
       (MAIN, PD([hold(0),close], [],  6))
     ]
R  = [(W,F,[(V2, PD([hold(3),putR(W)],[],2))])]
```

10. This moves the simulation clock up to 3 and makes `V1` the new current. It owns one share of `W` and can thus release it. This unblocks `V2` which returns to the event list behind `V1` and owning `W`.

```
EL = [ (V1,   PD([],                [],  3)),
       (V2,   PD([hold(3),putR(W)],[W], 3)),
       (MAIN, PD([hold(0),close],  [],  6))
     ]
R  = [(W,F,[])]
```

11. `V1` has now exhausted its actions and is deleted (having checked that its attribute list is empty)

```
EL = [ (V2,   PD([hold(3),putR(W)],[W], 3)),
       (MAIN, PD([hold(0),close],  [],  6))
     ]
R  = [(W,F,[])]
```

12. `V2` now carries out the weighing task

```
EL = [ (MAIN, PD([hold(0),close], [],  6)),
       (V2,   PD([putR(W)],       [W], 6))
     ]
R  = [(W,F,[])]
```

13. which brings back `MAIN` to be `current`. Now you see the need for the `hold(0)` — it allows the program to complete the final release action[1].

[1] Stopping the simulation at exactly the right time is made easier in process based languages if one treats the main program block as just another process. If need be, it can then be blocked on a bin (or passivated) and woken up at the right instant.

```
EL = [ (V2, PD([putR(W)], [W], 6)), (MAIN, PD([close], [], 6))]
R  = [(W,F,[])]
```

14. `V2` now releases its share in `W`

    ```
    EL = [ (V2, PD([], [], 6)), (MAIN, PD([close], [], 6))]
    R  = [(W,T,[])]
    ```

15. and is deleted

    ```
    EL = [ (MAIN, PD([close],[],6))]
    R  = [(W,T,[])]
    ```

    leaving `MAIN` to execute the final `close` action which issues a final report and then shuts down.

H.4 Semantics of πDemos synchronisations

As a simulation run unfolds, we have to keep track of the current states of the processes and resources it contains. Thus we may define the state of a program as the product of the states of its constituents (resources and processes) together with the set of valid class, process and resource names. We represent the state of a program at any time by a triple ($\mathcal{EL}$, $\mathcal{R}$, σ) where:

$\mathcal{EL}$ is an event list which contains all the active (scheduled) processes, ranked according to the time of their next scheduled event.

$\mathcal{R}$ is the set of resources. It is convenient to keep blocked processes local to the resource upon which they are waiting, so $\mathcal{R}$ implicitly contains all the blocked processes as well.

σ is an environment of defined names. In this account, σ contains class, individual object and resource definitions.

NB σ is used in our presentation to save and lookup definitions. If is possible to combine $\mathcal{R}$ and σ into a single set. We have chosen to represent them separately to ephasize that all uses of σ (checks that identifiers are fresh, and that definitions exist and are of the appropriate type) could be carried out by a πDemos compiler.

As each program command is executed the system will change from one state to another

$$(\mathcal{EL},\ \mathcal{R},\ \sigma\ ) \Longrightarrow (\mathcal{EL}\text{'},\ \mathcal{R}\text{'},\ \sigma\ \text{'})$$

Execution is so framed that the next action to be executed is always the first action in the action list of the first object in the event list. Thus given the event list pattern-matching

$$\mathcal{EL} = \texttt{(C, PD(b::Body, Attrs, time))::...}$$

— the next action *must* be `b` and the system takes the time of this action to be `time`.

H.4.1 Accessing sets

We use sets to hold resources, names, and attributes. The basic operations over a set are: the test for set membership, looking up an entry, adding an entry, and deleting an entry. We do not impose an implementation, but adopt the following conventions:

Membership. `id` $\in \mathcal{S}$ returns `true` if an entry for `id` lies in $\mathcal{S}$, `false` if not.

Lookup an entry. `LOOKUP id` $\mathcal{S}$ returns `rd` when `(id, rd)` $\in \mathcal{S}$. The call is an error if `id` $\notin \mathcal{S}$.

Remove an entry. $\mathcal{S}$ `--id` returns $\mathcal{S}$ when `(id, rd)` $\in \mathcal{S}$. The call is an error if `id` $\notin \mathcal{S}$.

Update an item. If `id` $\notin \mathcal{S}$, we add an entry `(id, rd)` to $\mathcal{S}$ by $\mathcal{S}$`[id/rd]` If `id` $\in \mathcal{S}$, then $\mathcal{S}$`[id/rd]` overwrites the previous entry for `id`. If the update simply adds an identifier, we will usually write $\mathcal{S}$ `++ id`.

H.4.2 Accessing the event list

The event list is an ordered list of pairs of the form `(id, PD(Body,Attrs,evt))` ranked by increasing time `evt`. Given that

$$\mathcal{EL}\text{= [(id}_1\text{, PD(b}_1\text{, a}_1\text{, t}_1\text{)), (id}_2\text{, PD(b}_2\text{, a}_2\text{, t}_2\text{)), ..., (id}_n\text{, PD(b}_n\text{, a}_n\text{, t}_n\text{))]}$$

then $t_1 \leq t_2 \leq ... \leq t_n$. We posit two basic event list routines and two auxiliaries: here are their explanation together with concrete FCFS list implementations.

evTime en (event notice `en`) returns the event time of `en`.

evTime (id, PD(Body,Attrs,evt)) = evt

pName en (event notice `en`) returns the identifier of `en`.

pName (id, PD(Body,Attrs,evt)) = id

ENTER en $\mathcal{EL}$: enters the event notice `en` into the event list $\mathcal{EL}$ ranked acording to its event time

```
ENTER en []            =  [ en ]
ENTER en1 (en2::EL)    =  if evTime en1 < evTime en2
                             then en1::en2::EL
                             else en2::(ENTER en1 EL)
```

DELETE id $\mathcal{EL}$ returns a copy of the event list $\mathcal{EL}$ with the event notice for `id` located and deleted. It raises an error if `id` is not scheduled.

$$\begin{array}{lcl} \text{DELETE id []} & = & \textit{error} \\ \text{DELETE id (en::}\mathcal{EL}\text{)} & = & \textbf{if } \text{id=pName en } \textbf{then } \mathcal{EL} \textbf{ else } \text{en::(DELETE id } \mathcal{EL}\text{)} \end{array}$$

H.4.3 Semantic rules

Structural operational semantics takes a language, command by command, and tells us how the system state will change when we carry out that command. In general, a command will fire only if certain constraints are satisfied, and it may fire in different ways depending upon the constraints. The typical rule is written:

$$\frac{\begin{array}{c} constraint_1 \\ constraint_2 \\ \dots \\ constraint_n \end{array}}{(\mathcal{EL},\ \mathcal{R},\ \sigma\ ) \quad \Longrightarrow \quad (\mathcal{EL}\text{'},\ \mathcal{R}\text{'},\ \sigma\ \text{'})}$$

where the constraints are listed above the horizontal line and the firing rule below. It is interpreted as "when all the constraints above the line are satisfied, then fire the rule below it".

As an example, suppose that the next command is `newR(id)`. We pattern match the current state of the system to

$$\texttt{((C, PD(newR(id)::Body, Attrs, time))::}\mathcal{EL}\texttt{, }\mathcal{R}\texttt{, }\sigma\texttt{)}$$

There are two cases to consider:

1. *error* if `id` is already in use, expressed by

$$\frac{\begin{array}{c} \text{current = (C, PD(newR(id)::Body, Attrs, time))} \\ \texttt{id} \in \sigma \end{array}}{\text{(current::}\mathcal{EL},\ \mathcal{R},\ \sigma\ ) \quad \Longrightarrow \quad \textit{error}}$$

2. normal case : the name `id` is added to the name pool, and a new res is added to the set of system resources. The system state changes to

$$\texttt{((C,PD(Body,Attrs,time))::}\mathcal{EL}\texttt{, }\mathcal{R}\texttt{[id/RD(true,[])],}\sigma\texttt{ ++ id)}$$

where `C` remains current, at the same clock time, but has moved on to the next instruction; the set of resources $\mathcal{R}$ has been incremented by `(id, RD(true, []))`, a pair with identifier `id` and a resource descriptor intialised to free and with an empty listy of blocked processes. `id` is added to the set of names σ . This is expressed by:

$$\frac{\begin{array}{c}\text{current} = (\text{C, PD((newR(id)::Body, Attrs, time))}\\ \text{current'} = (\text{C, PD(Body, Attrs, time))}\\ \texttt{id} \notin \sigma\end{array}}{(\text{current::}\mathcal{EL}, \mathcal{R}, \sigma\ ) \quad \Longrightarrow \quad \texttt{(current'::}\mathcal{EL}\texttt{, }\mathcal{R}\texttt{[id/RD(true,[])], } \sigma \ \texttt{ ++ id)}}$$

It is common practice to list the firing rules separately as above. However after consideration of the target readership of this paper (with simulation rather than proof backgrounds), we prefer to coalesce the rules into a single case structure which has the merit of being closer to normal programming practice, as below

$$\begin{array}{lll}
\multicolumn{3}{l}{(\text{current} = (\text{C, PD(newR(id)::Body, Attrs, time))::}\mathcal{EL}, \mathcal{R}, \sigma\ )}\\
\quad\Longrightarrow & \textbf{if } \text{id} \in \sigma \ \ \textbf{then } \textit{error } \textbf{else} & \\
\quad\quad | & \textbf{let } \mathcal{R}\text{'} = \mathcal{R}\text{[id/RD(true, [])])} & \textbf{in}\\
 & \textbf{let } \mathcal{EL}\text{'} = \text{current::}\mathcal{EL} & \textbf{in}\\
 & \textbf{let } \sigma\ \text{'} = \sigma \ \ \text{++ id} & \textbf{in}\\
 & \quad (\mathcal{EL}\text{', } \mathcal{R}\text{', } \sigma\ \text{')} &
\end{array}$$

(the **let**s merely break the description into simple steps).

H.4.4 Event list commands

It is now straightforward to give a semantics as a case statement over the structure of πDemos commands, as sketched below:

```
1   exec ([], $\mathcal{R}$, $\sigma$ ) $\Longrightarrow$ error

2   exec ((C, PD([],Attrs,time))::$\mathcal{EL}$, $\mathcal{R}$, $\sigma$ ) $\Longrightarrow$
        if Attrs=[] then exec($\mathcal{EL}$, $\mathcal{R}$, $\sigma$ ) else error

3   exec ((C, PD(b::Body,Attrs,time))::$\mathcal{EL}$, $\mathcal{R}$, $\sigma$ )
      $\Longrightarrow$   let current = (C, PD(Body, Attrs, time)) in
              (    case b of
                      decP(classId, classDef)     § 4.4.1
                |     newP(id, classId, dt)       § 4.4.2
                |     hold(dt)                    § 4.4.3
                |     newR(id)                    § 4.4.4
                |     getR(id)                    § 4.4.5
                |     putR(id)                    § 4.4.6
                |     close                       § 4.4.7
              )
```

1. an *error* arises if the event list becomes empty (the system should be shut down with a call on `close`).

2. When a process has exhausted its actions, a check is made to see whether it still owns any attributes. It should not and an error results if it does. If not, all is well. The process is deleted from the event list and the simulation proceeds from the next action of the new current.

3. The normal case — we focus on `(C, PD(b::Body,Attrs,time))` the object at the head of the event list, and execute its next action `b`. The names `Body`, `Attrs`, and `time` are directly accessible in the case clause. The cases are detailed each to its own subsection as indicated above. For convenience, we name the expected next `current`.

decP(classId, classDef)

Informally, a new entry `(classId, classDef)` is entered into the set of process declarations. An error arises if `classId` is not fresh.

Semantics:

```
decP(classId, classDef)
  $\Longrightarrow$  if classId $\in \sigma$   then $error$   else
      let $\mathcal{EL}$' = current::$\mathcal{EL}$    in
      let $\sigma$ ' = $\sigma$ [classId/classDef]   in
         exec($\mathcal{EL}$', $\mathcal{R}$, $\sigma$ ')
```

Interpretation:

1. *classId* $\in \sigma$: error if the process identifier is not fresh
2. *Normal case*
   (a) **let** $\mathcal{EL}$'= current::$\mathcal{EL}$: put the (diminished) `current` back as head of the event list at the current clock time.
   (b) **let** σ ' = σ [classId/classDef]: add the entry for the process class `id` to σ .
   (c) continue execution from ($\mathcal{EL}$', $\mathcal{R}$, σ ')

newP(id, classId, dt)

Informally, a new process named `id` is entered into the event list at the simulation clock `time + dt`. An error arises if the delay `dt` is negative or if `id` is not unique. The same process remains as current and the simulation clock time is unchanged.

Semantics:

```
    newP(id, classId, dt)
      $\Longrightarrow$  if id $\in \sigma$                           then error   else
            if classId $\notin \sigma$                      then error   else
            if not(classId is class definition)  then error   else
            if dt $< 0$                            then error   else
            let classDef = LOOKUP classId $\sigma$                   in
            let en = (id, PD(classDef, [], time+dt))       in
            let $\mathcal{EL}$'= current::(ENTER en $\mathcal{EL}$)          in
            let $\sigma$ ' = $\sigma$ ++ id                              in
               exec($\mathcal{EL}$', $\mathcal{R}$, $\sigma$ ')
```

Interpretation:

1. $id \in \sigma$: error if the process identifier is not fresh
2. $classId \notin \sigma$: error is the identifier `class Id` is not already defined
3. *not(classId* **is** *class definition)*: error if `classId` is not a process class definition
4. $dt < 0$: error if the relative time of scheduling is negative
5. *Normal case*

   (a) **let** classDef = LOOKUP classId σ : lookup the definition of class

   (b) **let** en = (id, PD(classDef, [], time+dt)): prepare an event list entry for `id` at the current clock time + dt.

   (c) **let** $\mathcal{EL}$'= current::(ENTER en $\mathcal{EL}$): and enter it into the event list after current

   (d) **let** σ ' = σ ++ id: add the name of the fresh object to σ

   (e) continue execution from ($\mathcal{EL}$', $\mathcal{R}$, σ ')

Notice that we make no attempt to give a semantics for arithmetic values. In a full semantic definition, we should include an extra clause stating that if the argument `dt` evaluates to `error`, then so does `newP(id, classId, dt)`.

hold(dt)

Informally we move `current` down the event list with a delay of `dt`. An error arises if `dt` is negative. Typically a new `current` will result.

Semantics:

```
hold(dt)
  $\Longrightarrow$   **if** dt $<$ 0 **then** *error* **else**
        **let** $\mathcal{EL}$' = ENTER (C, PD(Body,Attrs,time+dt)) $\mathcal{EL}$ **in**
           exec($\mathcal{EL}$', $\mathcal{R}$, $\sigma$ )
```

Interpretation:

1. *dt* $<$ *0*: error if `dt` is negative
2. *Normal case*
   (a) **let** $\mathcal{EL}$' = ENTER (C, PD(Body, Attrs, time+dt)) $\mathcal{EL}$: enters the updated event notice for `current` into the tail ($\mathcal{EL}$) of the event list.
   (b) continue evaluation from the new state ($\mathcal{EL}$', $\mathcal{R}$, σ)

newR(id)

Informally a new resource is added to the resource set. It is saved as a pair `(id, RD(true, []))`. An error occurs if the resource name `id` has been used before.

Semantics:

```
newR(id)
  ⟹  if id ∈ σ   then error else
       let $\mathcal{EL}$' = current::$\mathcal{EL}$            in
       let $\mathcal{R}$' = $\mathcal{R}$[id/RD(true, [])]     in
       let σ ' = σ ++ id                in
           exec($\mathcal{EL}$', $\mathcal{R}$', σ ')
```

Interpretation:

1. *id* $\in \sigma$: an error if the resource identifier is not fresh

2. *Normal case*

   (a) **let** $\mathcal{EL}$' = current::$\mathcal{EL}$: update the event list.

   (b) **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(true, [])] : add the new resource descriptor for id (free and with an empty blocked queue) to the resource pool $\mathcal{R}$.

   (c) **let** σ ' = σ ++ id: add the fresh identifier to the list of resource names

   (d) continue from ($\mathcal{EL}$', $\mathcal{R}$', σ ')

getR(id).

Informally `current` acquires the resource `id` only if there are no blocked processes waiting on `id` and `id` is free at the time of the request. Otherwise `current` is blocked and waits in a queue local to the resource `id`. A successful request is recorded in the attribute list of current. An error arises if the resource is already owned since a second attempt must deadlock the system.

Semantics.

```
getR(id)
  ⟹  **if** id $\in$ Attrs **then** *error* **else**
      **case** LOOKUP id $\mathcal{R}$ **of**
          RD(true, [])
          ⇒  **let** Attrs' = Attrs ++ id                          **in**
              **let** $\mathcal{EL}$' = (C, PD(Body,Attrs',time))::$\mathcal{EL}$   **in**
              **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(false, [])]               **in**
                  exec($\mathcal{EL}$', $\mathcal{R}$', $\sigma$ )

        | RD(false, Q)
          ⇒  **let** Q' = Q@[current]                              **in**
              **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(false, Q')]              **in**
                  exec($\mathcal{EL}$, $\mathcal{R}$', $\sigma$ )

        | anythingelse ⇒ *error*
```

Interpretation.

1. *id ∈ Attrs*: an error if `current` already owns the resource (otherwise, the system deadlocks)
2. *Normal case*
3. **case** LOOKUP id $\mathcal{R}$ **of**: returns the descriptor for `id`.
   (a) RD(true, []): the resource is available and no other process is blocked. Acquire it and continue on as `current`
      i. **let** Attrs' = Attrs ++ id: add id to the attribute list of current.
      ii. **let** $\mathcal{EL}$' = (C, PD(Body,Attrs',time))::$\mathcal{EL}$: update the event list
      iii. **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(false, [])]: update the entry for id to reflect its busy status
      iv. and continue on from exec($\mathcal{EL}$', $\mathcal{R}$', σ)
   (b) RD(false, Q) : the resource is already in use. Current is blocked.

   i. **let** Q' = Q@[current]: add `current` to the tail of the blocked queue associated with resource `id`
   ii. **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(false, Q')]: update the entry for the resource `id`
   iii. continue on with a fresh current, exec($\mathcal{EL}$, $\mathcal{R}$', σ)

(c) *anythingelse* ⇒ *error*: an error if the lookup fails. NB we can prove that the LOOKUP case *RD(true, q::Q)* (a free resource with one or more blocked process) cannot arise.

putR(id)

Informally when `current` releases a resource `id`, the resource count is made free and then its pending queue is examined. The leading blocked process, if any, can now be promoted. Promotion entails seizing the resource and entering the event list at the current clock time, but note that the "putter" will remain as `current`. An error arises if an attempt is made to release a resource that has not been aquired.

Semantics.

```
putR(id)
    $\Longrightarrow$  if id $\notin$ Attrs then $error$ else
        let Attrs' = Attrs – id                                         in
        let $\mathcal{EL}$' = (C, PD(Body, Attrs', time))::$\mathcal{EL}$    in
        case LOOKUP id $\mathcal{R}$  of
            RD(false, [])
            $\Rightarrow$  let $\mathcal{R}$' = $\mathcal{R}$[id/RD(true, [])])                  in
                    exec($\mathcal{EL}$', $\mathcal{R}$', $\sigma$ )

          | RD(false, (p1, PD(B1,A1,t1))::Q1)
            $\Rightarrow$  let $\mathcal{R}$'  = $\mathcal{R}$[RD(id/(false, Q)]                in
                let A1' = A1 ++ id                                      in
                let en = (p1,PD(B1,A1',time))                           in
                let $\mathcal{EL}$" = ENTER en $\mathcal{EL}$'                          in
                    exec($\mathcal{EL}$", $\mathcal{R}$', $\sigma$ )

          | anythingelse $\Rightarrow$ $error$
```

Interpretation.

1. *id* $\notin$ *Attrs*: any attempt to return a resource that is not owned is in *error*

2. *Normal case*: proceed by removing `id` from the attributes of `current` and pre-computing the updated event list

3. **case** LOOKUP id $\mathcal{R}$ **of**: three cases arise

   (a) RD(false, []): there are no blocked processes

      i. **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(true, [])]: simply update `id` to be free, and

      ii. continue on from exec($\mathcal{EL}$', $\mathcal{R}$', σ)

   (b) RD(false, (p, PD(b1,a1,t1))::Q): there are blocked processes, and the leading one is `p`

      i. **let** $\mathcal{R}$' = $\mathcal{R}$[id/RD(false, Q)]: the resource is now busy and `p` is deleted from its blocked queue

  ii. **let** A1' = A1 ++ id: update the attributes of `p1` with the resource name `id`
  iii. **let** en = (p1,PD(B1,A1',time)): create a new event notice for the unblocked process `p1`
  iv. **let** $\mathcal{EL}$" = ENTER en $\mathcal{EL}$': the event notice for `p1` is entered into the (precomputed) event list at the current simulation time.
  v. continue on from exec($\mathcal{EL}$", $\mathcal{R}$', σ)

(c) *anythingelse* ⇒ *error*: an error if the lookup fails. NB we can prove that the LOOKUP case *RD(true, Q)*, an attempt to free an already free resource cannot arise (we have already checked that it is owned by `current`) *close*

A call on `close` shuts down the simulation run.

Semantics.

$$\text{close} \quad \Longrightarrow \quad \textbf{report } \mathcal{R}$$

Interpretation.

In a fully-fledged simulation, a final report on resource usage would be issued at this point.

H.5 Applications

H.5.1 Implementation

An implementation of πDemos was developed as the operational semantics was being formulated. This style of co-development was important as it helped both debug the semantics (especially missing error cases) and streamline its presentation.

As implementation language we used SML [74], a modern functional language with strong datatypes. As one might have expected, it was a straightforward matter to convert from the operational semantics into SML (much easier than it would have been to convert to an imperative language with weak datatypes such as C [15]). It is interesting commentary on the power of modern functional languages that only 184 lines of code were required (with full tracing[2] but no resource utilisation statistics), and that the object-oriented style can be modelled in such a direct fashion.

Two fully-explained representative code expansions are presented below. A full listing of πDemos, our running example and its full execution trace are given in an Appendix.

Implementation of sets in SML

Here are the intuitive definitions and implementations of the basic routines for set membership, updating an item, and look-up together with the actual implementations. A set is represented by a list; each member of a set is held as a pair `(r, rd)`, where `r` is a (unique) identifier and `rd` is its associated descriptor.

Membership. `id` $\in \mathcal{R}$ returns `true` if an entry for `id` lies in $\mathcal{R}$, `false` if not. This definition is implemented by

```
fun isMEM id []             = false
  | isMEM id ((r, rd)::R) = if id=r then true else isMEM id R
```

`id` is not in the empty list `[]`. Otherwise search the nonempty list `((r, rd)::R)` from its head `(r, rd)`: if `id = r` then return true; else search the rest of the list.

Lookup an entry. `LOOKUP id` $\mathcal{R}$ returns `rd` when `(id, rd)` $\in \mathcal{R}$. The call is an error if `id` $\notin \mathcal{R}$. This definition is implemented by

```
fun LOOKUP id []            = error
  | LOOKUP id ((r,rd)::R) = if id=r then rd else LOOKUP id R
```

[2] The traces shown in this paper came from this toy implementation

Lookup on an empty list is an error. Otherwise lookup in the nonempty list `((r, rd)::R)` from its head `(r, rd)`: if `id = r` then return the associated descriptor `rd`; else search the rest of the list.

Remove an entry. `REMOVE id` $\mathcal{R}$ returns $\mathcal{R}$ `--(id, rd)` when `id` $\in \mathcal{R}$. The call is an error if `id` $\notin \mathcal{R}$. This definition is implemented by

```
fun REMOVE id []           = error
  | REMOVE id ((r,rd)::R) = if id=r then R else (r,rd)::(REMOVE id R)
```

Removing an item from an empty list is an error. Otherwise search nonempty list `((r, rd)::R)` from its head `(r, rd)`: if `id = r` then return the rest of the list `R`; otherwise save the current list head and add it onto the result of removing `id` from the tail `R`.

Add/update an item. If `id` $\notin \mathcal{R}$, we add an entry `(id, rd)` to $\mathcal{R}$ by $\mathcal{R}$`[id/rd]` If `id` $\in \mathcal{R}$, then $\mathcal{R}$`[id/rd]` overwrites the previous entry for `id`. This definition is implemented by

```
fun UPDATE [] (r, rd)
     = [(r, rd)] (* add a new entry *)
  | UPDATE ((r',rd')::R) (r, rd)
     = if r'=r then (r, rd)::R else (r',rd')::(UPDATE R (r, rd))
```

If the list is empty, return a list with one item. Otherwise search nonempty list `((r', rd')::R)` from its head `(r', rd')`: if `r'=r` then replace the old entry `(r' rd')` by the update `(r, rd)`; otherwise save the current list head and add it onto the result of updating `(r, rd)` in the tail `R`.

Implementation of putR in SML

The implementation of the `putR` synchronisation is again very close to that of the semantic definition. The major change being the syntactic form of `lets` in SML.

```
putR(id)
  $\Longrightarrow$  **if** id $\notin$ Attrs **then** *error* **else**
      **let** Attrs' = Attrs – id                                        **in**
      **let** $\mathcal{EL}$' = (C, PD(Body, Attrs', time))::$\mathcal{EL}$                **in**
      **case** LOOKUP id $\mathcal{R}$ **of**
          RD(false, [])
          $\Rightarrow$  **let** $\mathcal{R}$' = $\mathcal{R}$[RD(id/(true, []))])                   **in**
                  exec($\mathcal{EL}$', $\mathcal{R}$', $\sigma$ )

        |  RD(false, (p1, PD(B1,A1,t1))::Q1)
          $\Rightarrow$  **let** $\mathcal{R}$'  = $\mathcal{R}$[RD(id/(false, Q)]                   **in**
              **let** A1' = A1 ++ id                                 **in**
              **let** $\mathcal{EL}$" = ENTER (p1,PD(B1,A1',time)) $\mathcal{EL}$'   **in**
                  exec($\mathcal{EL}$", $\mathcal{R}$', $\sigma$ )

        |  anythingelse $\Rightarrow$ *error*
```

is implemented by

```
putR(id)
  => if not(isMEM id Attrs) then error else
     let val Attrs' = REMOVE id Attrs
         val EL'    = (C, PD(Body, Attrs', time))::EL
     in
        ( case LOOKUP id R of
          (RD(false, []))
             => let val R' = UPDATE R (id, RD(true, []))
                in
                    exec (EL', R', Sigma)
                end

        | (RD(false, (p1, PD(B1,A1,t1))::Q1))
             => let val R'   = UPDATE R (id, RD(false, Q1))
                    val A1'  = UPDATE A1 (id, RA)
                    val en   = (p1,PD(B1,A1',time))
                    val EL'' = ENTER en EL'
                in
                    exec (EL'', R', Sigma)
                end

        | anythingelse => error
        )
```

```
    end
```

With set definitions established, and allowing for a sugared `let` construct, the translation is quite mechanical.

H.5.2 Proofs

Work is underway with Tom Melham of Glasgow University formalising and proving facts about πDemos in the HOL proof assistant [34]. The HOL description is a direct encoding of the operational semantics presented here. Initial work has shown that the event list does indeed remain ordered. In further work we expect to prove that terminating πDemos models evolve uniquely and that "well-formed" πDemos models must terminate. Formalising systems and carrying out proofs in proof assistants like HOL is time consuming and requires a reasonable level of expertise. Proof assistants are very demanding and see to it that every detail has to be properly proved (no corners can be cut, which can be tedious), that all sub-proofs are completed, and (in this case) that an appropriate induction schema be used. The effort is justified by the extra confidence that a formal proof bestows and the intrinsic interest of the proof.

H.6 Summary and conclusions

In this paper we have given an operational definition of the synchronisations and event list operations of a small discrete event simulation language, πDemos. The same style and techniques can be applied to give a semantics for other common synchronisation mechanisms, such as producer/consumer, buffers, waitqs, waituntil, broadcasting, and interrupts (see [10]); and to compare and contrast simulation languages.

Giving a language a "good" semantics is important because it serves as a clear (taking care with notation), short (using good abstractions), and unambiguous statement of the intent of each language construct and how a model will evolve. The semantic description can be used by implementors to ensure consistent developments across different hardwares, by simulars to understand how models unfold in "tricky" situations, and in proving facts about models.

This semantics was developed hand-in-hand with an implementation in SML. This co-development was important as is helped debug and simplify the semantic definitions. In general, we would contend that languages designed in this way via semantic principles will be simpler, cleaner, and safer. The work presented here is leading to further research on proofs about simulation models, comparisons with other semantics bases (the more abstract denotational semantics), formal checking of the properties of simulation models, and meta-level abstractions over synchronisations to ensure their consistency.

Acknowledgements

This work has been supported by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada and by a Science and Engineering Research Council (UK) Advanced Research Fellowship tenable at University College, Swansea.

H.7 Listing in SML

```
exception BLOW_UP;

fun error s
  = ( output(std_out, "***error -- " ^ s ^ " ***\n\n");
      raise BLOW_UP
    );

fun pNum x  = (makestring (x:int));
fun showB b = if b then "T" else "F";

fun bkt s         = "(" ^ s ^ ")";
fun sbkt s        = "[" ^ s ^ "]";
fun splice []     = ""
  | splice [a]    = a
  | splice (a::A) = a ^ "," ^ (splice A);
fun rbs S         = bkt(splice S);
fun sbs S         = sbkt(splice S);

type Id   = string
type Time = int

datatype ACTION
  = decP      of Id * ACTION list
  | newP      of Id * Id *  Time
  | hold      of Time
  | close
  | newR      of Id
  | getR      of Id
  | putR      of Id

datatype PROC
  = PD of ACTION list * (Id * ATTRIBUTE) list * Time

and RESOURCE
  = RD of bool * (Id * PROC) list

and ATTRIBUTE
  = RA

and DECL
  = PDEC of Id * Time
  | RDEC
  | CDEC of ACTION list;

fun showACT a
  = case a of
    decP(classId,classDef)  => "decP:" ^ classId ^ " = " ^ (showACTS classDef)
  | newP(id,classId,t)      => "newP" ^ rbs[id,classId,pNum t]
  | hold(t)                 => "hold" ^ bkt(pNum t)
  | close                   => "close"
  | newR(id)                => "newR" ^ (bkt id)
  | getR(id)                => "getR" ^ bkt(id)
  | putR(id)                => "putR" ^ bkt(id)

and showACTS L = sbs(map showACT L)

and pName  (id, PD(Body, Attrs, evt)) = id
and evTime (id, PD(Body, Attrs, evt)) = evt
and sysTime []       = error "empty EL"
  | sysTime (en::EL) = evTime en

and showEN (id, PD(Body, Attrs, evt))
  = rbs[id, "PD" ^ rbs[showACTS Body, showATTRS Attrs, pNum evt]]

and showENS L = sbs(map showEN L)

and showRESOURCE (id, RD(b, Q)) = rbs[id, showB b, showENS Q]
and showRESOURCES L             = sbs(map showRESOURCE L)

and showATTR (id, RA) = id
and showATTRS L       = sbs(map showATTR L)

and showDEC (id, PDEC(cId,dt)) = rbs["PROC", id, cId, pNum dt]
  | showDEC (id, RDEC)         = rbs["RES",  id]
  | showDEC (id, CDEC cDef)    = rbs["CLASS",id, showACTS cDef]
and showDECS L                 = sbs(map showDEC L)

and showEL  L = showENS L
and showRHO L = showDECS L;

fun showState (EL, R, SIGMA)
  = ( output(std_out,("**Clock time = " ^ pNum(sysTime EL) ^ "\n"));
      output(std_out,("EL  = " ^ (showEL       EL) ^ "\n"));
      output(std_out,("R   = " ^ (showRESOURCES R) ^ "\n"));
      output(std_out,("RHO = " ^ (showDECS  SIGMA) ^ "\n\n"))
    );

fun isMEM id []           = false
  | isMEM id ((r, rd)::R) = if id=r then true else isMEM r R;

fun LOOKUP id []          = error ("LOOKUP " ^ id ^ ":: " ^ id ^ " not found")
  | LOOKUP id ((r,rd)::R) = if id=r then rd else LOOKUP id R;

fun REMOVE id []                    = error ("REMOVE " ^ id ^ ":: " ^ id ^ " not found")
  | REMOVE id ((item as (r,rd))::R) = if id=r then R else item::(REMOVE id R);

fun UPDATE [] (r, rd)            = [(r, rd)] (* add a new entry *)
  | UPDATE ((r’,rd’)::R) (r, rd) = if r=r’ then (r, rd)::R else (r’,rd’)::(UPDATE R (r, rd))

fun ENTER en1 [] = [ en1 ]
  | ENTER en1 (en2::EL)
      = if evTime en1 < evTime en2
          then en1::en2::EL
          else en2::(ENTER en1 EL)

fun DELETE id []       = error ("attempt to DELETE non-scheduled process " ^ id)
  | DELETE id (en::EL) = if id = pName en then EL else en::(DELETE id EL);

fun exec state
  = ( showState state;
      case state of

        ([], R, Sigma) => error ("exec:: empty event list")

      | ((C, PD([], Attrs, time))::EL, R, Sigma)
            => if not(Attrs = [])
                 then error ("exec:: process " ^ C ^ " dies owning attrs::" ^ (showATTRS Attrs))
                 else exec(EL, R, Sigma)

      | ((C, PD((b::Body), Attrs, time))::EL, R, Sigma)
```

```
=> ( let val current = (C, PD(Body, Attrs, time))
     in
        case b of
          decP(classId, classDef)
            => if (isMEM classId Sigma) then error ("decP:: class id '" ^ classId ^ "' used before") else
               let val EL'    = current::EL
                   val Sigma' = UPDATE Sigma (classId, CDEC classDef)
               in
                   exec (EL', R, Sigma')
               end

        | newP(id, classId, dt)
            => if (isMEM id Sigma)         then error ("newP:: proc id '" ^ id ^ "' used before") else
               if not(isMEM classId Sigma) then error ("newP:: " ^ classId ^ " not declared")     else
               if dt < 0                   then error ("newP with negative delay " ^ (pNum dt))   else
               let val (CDEC classDef) = LOOKUP classId Sigma
                   val en              = (id, PD(classDef, [], time+dt))
                   val EL'             = current::(ENTER en EL)
                   val Sigma'          = UPDATE Sigma (id, PDEC(classId, dt))
               in
                   exec (EL', R, Sigma')
               end

        | hold(dt)
            => if dt < 0  then error ("hold::  negative argument " ^ (pNum dt)) else
               let val EL' = ENTER (C, PD(Body, Attrs, time+dt)) EL
               in
                   exec (EL', R, Sigma)
               end

        | close => output(std_out, "\n\n*** end of simulation run ***\n\n")

        | newR(id)
            => if (isMEM id Sigma) then error ("newR:: resource id '" ^ id ^ "' used before") else
               let val R'  = UPDATE R (id, RD(true, []))
                   val EL' = current::EL
                   val Sigma' = UPDATE Sigma (id, RDEC)
               in
                   exec (EL', R', Sigma')
               end

        | getR(id)
            =>if (isMEM id Attrs) then error ("deadlock --- re-acquire of owned resource " ^ id) else
                ( case LOOKUP id R of
                  (RD(true, []))
                     => let val Attrs' = UPDATE Attrs (id, RA)
                            val EL'    = (C, PD(Body, Attrs', time))::EL
                            val R'     = UPDATE R (id, RD(false, []))
                        in
                            exec (EL', R', Sigma)
                        end

                | (RD(false, Q))
                     => let val Q'  = Q@[current]
                            val R'  = UPDATE R (id, RD(false, Q'))
                        in
                            exec (EL, R', Sigma)
                        end

                | anythingelse => error "LOOKUP failure in getR"
                )
                | putR(id)
                   => if not(isMEM id Attrs) then error ("putR of not owned resource " ^ id) else
                      let val Attrs' = REMOVE id Attrs
                          val EL'    = (C, PD(Body, Attrs', time))::EL
                      in
                         ( case LOOKUP id R of
                           (RD(false, []))
                              => let val R' = UPDATE R (id, RD(true, []))
                                 in
                                     exec (EL', R', Sigma)
                                 end

                         | (RD(false, (p1, PD(B1,A1,t1))::Q1))
                              => let val R'   = UPDATE R (id, RD(false, Q1))
                                     val A1'  = UPDATE A1 (id, RA)
                                     val EL'' = ENTER (p1,PD(B1,A1',time)) EL'
                                 in
                                     exec (EL'', R', Sigma)
                                 end

                         | anythingelse => error ("LOOKUP error on putR")
                         )
                       end
              end
            )
    );

fun e M
  = exec([("MAIN", PD(M@[hold(0), close], [], 0))], [], []);
```

Appendix I

Getting Demos Models Right

In this chapter (again joint work with Chris Tofts [45, 46]) we present a method for translating the synchronisation behaviour of a process oriented discrete event simulation language into a process algebra. Such translations serve two purposes. The first exploits the formal structure of the target process algebraic representations to enable proofs of such properties of the source system as deadlock freedom, safety, fairness and liveness which can be very difficult to establish by simulation experiment. The second exploits the denotational semantics to better understand the language constructs as abstract entities and to facilitate reasoning about simulation models. Here we give the intuition and the basic translation mechanisms using a variety of the Demos simulation language and the CCS and SCCS process algebras. The translations have been automated as SML programs and produce CWB compatible input allowing the automated checking of formal system properties.

I.1 Introduction

Many complex problems are studied by building simulation models that are intended to replicate selected aspects of their behaviour [14, 19, 29, 31, 49, 50, 67, 79, 84, 85, 89, 109, 110]. Once we have a model, the first step is to validate it, that is show it exhibits certain expected behaviours. The standard practice of running the simulation "enough" times cannot guarantee that all possible model paths will be covered, so harmful behaviours such as deadlock may be missed by this approach. This paper works towards a more formal approach whose motivation is giving guaranteed answers to such questions as: "Can the system ever hang?" (deadlock), "Can the system spin forever" (livelock), "Can there ever be more than one train on this section of track at a time?" (safety), "Is it always possible to return to a certain state?" (liveness), "Once a request is made, will it be granted?" (fairness), etc. We are able to give guaranteed answers however the model may unfold on any run and under quite general circumstances (whatever the particular waiting time distributions and whatever the queueing disciplines).

In two previous papers [9, 10], we have given an operational semantics for the synchronisations of Demos [4], a typical process-based simulation language. These operational semantics are quite detailed and are best suited to serve as an implementation guide or for reasoning about an individual program run. In this paper we present more abstract denotational definitions of Demos. Our abstraction replaces the Demos model, say M, by a coarser generalisation, say M', in which structurally irrelevant actions (such as data collection) are omitted and the strict timings of M become arbitrary. Informally, we can think of M' as being akin to a worst case containment of M: if nothing bad can happen however M' unfolds, then our translation from M to M' guarantees that it cannot happen in M either. Once we have an algebraic characterisation of M', we can then check its characteristic properties using the modal-μ calculus [48] on the public domain Concurrency Workbench[1] (henceforth the *CWB* [21, 66]). Since M and M' are structurally the same, once M' has been corrected, it is straightforward to rectify the actual model M.

The most popular styles of programming simulation models have been the activity based approach of (E)CSL [17, 23, 29, 79, 99], the event based approach of GASP and early Simscript [85, 43], and the process based approach of GPSS, Simscript II.5 and Simula [89, 18, 6]. However it is quite straightforward to translate between these styles [7, 57]. Hence, without loss of generality, we concentrate upon the process based approach as typified by the Demos extension to Simula.

In this paper we give a formal denotational semantics for a variety of Demos by providing translation schemata from Demos down to CCS [59] and to SCCS [58], two of the simplest process algebras both of which are mathematically well-defined. These semantics provide us with a system description in CCS/SCCS that extends, but faithfully contains, their original description in Demos [11]. Both random delays and queueing disciplines are generalised in the translation. Informally we replace all Demos random drawings by “some time later” in CCS/SCCS and track all possible ways in which the actions of the constituent processes may interleave as the model unfolds.

Given such a translation schema, we can exploit the CWB to examine the properties and consequences of the underlying simulation and thus make our models that much more reliable. Translation into a formal system also permits proofs of properties of constructs within Demos (see section 4), as well as of particular Demos programs.

The disadvantage of using a coarser model is that it will permit some behaviours that are unrealisable in practice. However every possible failure in the original M will be captured by the translation M'. A more refined target process algebra should be used when errors unrealisable in M occur in M'. Richer process algebras can be used to account for timing [65] or probabilistic [100, 102] behaviour of systems, but their descriptions are usually much more complicated. In most cases, systems failures result from underlying interaction errors rather than specific problems and can be captured in the simpler process algebras we espouse [8, 9, 10, 11, 103].

[1] The Edinburgh CWB home page is `www.dcs.ed.ac.uk/home/cwb`.

The presentation is in two parts. Part I (sections 1..4) is motivational. In it we present the various notations, hand translations of the core synchronisations, and demonstrate model checking. The translation methods needed for more advanced synchronisations are technically more mature and are held over to part II (sections 5..8). We introduce the core synchronisations of Demos in section 2 through an example. Section 3 introduces CCS using the same example, shows that the model deadlocks and how to eliminate the deadlock. We then introduce our second process algebra, SCCS, and in section 4 show how to check for a variety of properties in the modal-μ calculus.

Part II (sections 5..8) formalises the hand translations exhibited in the part I for both CCS and SCCS. Section 5 gives formal translations for core Demos. Section 6 adds wait until and section 7 conditional commands. A normal form for Demos is presented as an application. Using these schemata, we can translate a complete Demos model to a process algebra and use the translation to formally derive properties of the original system. Finally we summarise the work completed.

I.2 Modelling in Demos

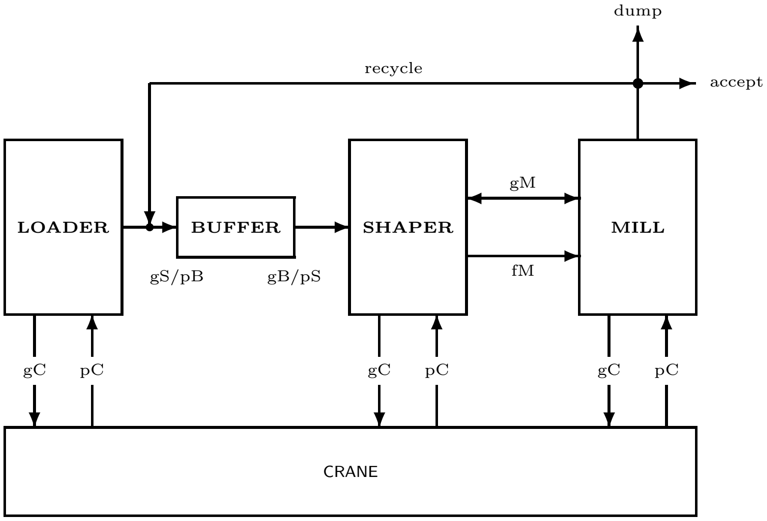


Figure I.1: System interactions

In Demos we model major components by *entities* (synonyms: agents, objects, processes, transactions) and abstract away minor components to *resources*. Demos has two types of resource: the *res* to model competition between entities and the *bin* to model co-operative (producer/consumer) entity interactions. The intention is that a res represents a fixed supply which entities fight to share, and a bin models "tokens" that can be handed on from a producing entity to consuming entity.

We illustrate the core of Demos by a small example (outlined in figure I.1) which is used as a running example throughout the paper. It is a compacted and idealised version of several models given in [4] and is sufficient to illustrate the basic synchronisations of Demos and to point out some the pitfalls of using simulation languages to model real concurrency.

A LOADER delivers scrap metal to a factory. When there is room for a load, it is placed in a 2-slot buffer. If both buffer slots are already filled, the LOADER must wait. When ready and there is (at least) one load, the SHAPER picks one up from the buffer, heats it and forms it into a hot, malleable billet. The billet is then handed over to a MILL for rolling into a flat plate. Because the billet must be at the right temperature for rolling, we cannot permit it to be dumped by the SHAPER and left to wait for the MILL to be ready — the SHAPER and MILL objects must synchronise for the handover. Once the MILL has rolled the billet into a plate, it is inspected. If deemed good it is accepted; if not it is recycled back to the buffer area for another run through the SHAPER if there is space. In later versions of the model, it may be dumped instead.

All movements of the metal — scrap, billet, or plate — require the use of a CRANE. We use the following short names for these movements:

task name	from		to	carried out by
a2b	arrival	→	buffer	LOADER
b2s	buffer	→	shaper	SHAPER
s2m	shaper	→	mill	SHAPER&MILL
m2s	mill	→	stack	MILL
m2b	mill	→	buffer	MILL
m2x	mill	→	dump	MILL

Entities. Entities are major system components whose behaviours warrant modelling in detail. In Demos, these behaviours are described by lists of time consuming tasks. (Syntactically, items in list are separated by commas and enclosed in square brackets.) Behaviours may be repeated forever. Initial (informal) definitions of LOADER, SHAPER and MILL are:

```
LOADER   =   repeat [ startL, a2b ]
SHAPER   =   repeat [ b2s, heat, s2m ]
MILL     =   repeat [ s2m, roll, m2s or m2b ]
```

The SHAPER and MILL must synchronise to perform the movement *s2m.* This we model by a Demos master/slave cooperation. We choose the MILL as slave. It signals its readiness by *wait(m).* As master, the SHAPER *coopt*s the MILL *m.* The master or slave that signals first is blocked until its partner arrives. The master carries out the task *s2m* on behalf of both entities, and then frees up the slave to go its own way by *free(m)*. A refined definition of the system with these changes highlighted is:

```
LOADER   =   repeat [ startL, a2b ]
SHAPER   =   repeat [ b2s, heat, coopt(m), s2m, free(m) ]
MILL     =   repeat [ wait(m), roll, m2s or m2b ]
```

Initially the MILL waits passively for the next billet. When the SHAPER has heated a billet, it coopts the MILL and they carry out the transfer *s2m* together. When the MILL is freed by the SHAPER's *free(m)*, it goes its own way carrying out the remainder of its tasks before waiting again.

Bins. The drop area is best modelled in Demos by a *bin*. In fact by two Demos bins: one to count the number of unbooked slots, and the other to keep track of the number of loads actually in there. *newB(slots, 2)* creates an initial pool of two free slots where loads can be deposited; *newB(buff, 0)* creates a load pool, initially empty. The first argument to newB names the bin, the second argument gives the initial size of the pool.

To load a buffer, a producer must first book a slot and then move the load into the buffer from whence it is available to be picked up. We model the LOADER actions by:

```
LOADER   =   repeat [ startL, gB(slots, 1), a2b, pB(buff, 1) ]
```

The LOADER first checks that there is a free slot *gB(slots, 1)* (and will be blocked if the buffer is currently full), moves the load into the guaranteed slot *a2b*, and then increments the number of loads by *pB(buff, 1)*. The call on *pB* would awaken the SHAPER if it were waiting for a fresh load. The code for SHAPER, the mirrored consumer process, is refined to:

```
SHAPER  =  repeat [ gB(buff, 1), b2s, pB(slots, 1), heat, coopt(m), s2m, free(m) ]
```

The SHAPER makes sure there is a load with *gB(buff, 1)*, moves it out by *b2s*, and then frees up the vacated slot with *pB(slots, 1)*, which would awaken a waiting LOADER, if any.

The correspondingly refined code for the MILL is:

```
MILL   =   repeat [ wait(m), roll, m2s or gB(slots, 1), m2b, pB(buff, 1) ]
```

Res(ources). Whereas bins enable us to model co-operation, we use the *res* to model competition. Resources are created by executing *newR* whose arguments name the resource and fix its size. *newR(crane, 1)* creates a res object named *crane* with one token. The token may be acquired by *gR(crane, 1)*. Such a request is granted immediately if sufficient of the resource is free and no other entity is blocked. Otherwise the requester is itself blocked and held in a queue local to the resource. There it remains until it is first in the queue and there is sufficient of the resource available for it to proceed. When a current user releases its share by *pR(crane, 1)*, the call on *pR* not only increments the resource pool but also unblocks any waiting entity whose request can be granted. An unblocked entity leaves the resource queue and enters the event list behind its unblocker and at the same clock time. If we include competion for the crane, our next model refinement is:

```
newB(buff,   0)
newB(slots,  2)
newR(crane,  1)

LOADER  = repeat [ startL,
                   gB(slots, 1),
                   gR(crane, 1), a2b, pR(crane, 1),
                   pB(buff,  1)
                 ]

SHAPER  = repeat [ gB(buff,  1),
                   gR(crane, 1), b2s, pR(crane, 1),
                   pB(slots, 1),
                   heat,
                   coopt(m),
                     gR(crane, 1), s2m, pR(crane, 1),
                   free(m)
                 ]

MILL    = repeat [ wait(m),
                   roll,
                   if accept then
                     then gR(crane, 1), m2a, pR(crane, 1),
                     else gS(slots, 1),
                          gR(crane, 1), m2b, pR(crane, 1),
                          pB(buff,  1)
                 ]

newP(L, LOADER, 0)
newP(S, SHAPER, 0)
newP(M, MILL,   0)
```

The above model description would be completed by defining appropriate distributions for the task times and their calls (holds), initialising, and setting a run length. But when the model is run, it might grind to a halt (deadlock). Can you see why?

When a simulation run results in deadlock, there is usually no explicit indication that this has happened. One has to infer the fact from reading the trace and the final report and then set about locating the cause and rectifying it. Given a more complex scenario, detecting the precise cause of a deadlock can be extremely difficult.

It may be buried in a welter of complexity, or it may be a rare event which depends crucially upon precise timings, and if the timings are sufficiently delicate, a potential deadlock may well be missed even in a large set of runs. At best this will be irritating and costly to remove, but in safety-critical systems it is also dangerous since most systems do not have stationary fail-safe behaviour. It is important find a methodology that will cope.

I.3 Modelling in process algebras

One can detect the possibility of deadlock directly from a model structure by using the techniques of process algebra. The idea developed here is to map a Demos program into CCS or SCCS (both are object-oriented and preserve the original model structure) and then to test the translation using the modal μ-calculus. In next subsections we introduce CCS and SCCS and show how to test for properties from a static code description rather than running the model. Once we have the mathematical apparatus to discover deadlock, we can also apply it to test models for livelock, fairness, and a variety of safety and progress properties.

I.3.1 Modelling in CCS

CCS is a process algebra developed by Milner over the last 20 years [60]. CCS is a very small language (just half a dozen syntactic rules) with a clean semantics and a rich equational reasoning system for analyzing behaviours and equalities. It permits descriptions of a system by the composition of its constituent parts called *agents*. CCS scales hierarchically and is hence well suited to abstract modelling. Although its scope is limited to modelling interactions rather than functionality, it has been used successfully to describe and reason about protocols and self-timed hardware.

The simplest agent in CCS is **0** which can do nothing. CCS has three ways of building more interesting agents.

Prefixing The agent $a.\overline{b}.\mathbf{0}$ can do an a action, then a $\overline{b}$ action, and after that nothing more. By convention we overline output actions. Recursive definitions are allowed so that the agent $GFclock \ = \ \overline{tick}.\overline{tock}.GFclock$ will omit a $\overline{tick}$ and then a $\overline{tock}$ forever.

We can model a unit resource by $Res \ = \ \overline{gR}.\overline{pR}.Res$. We think of Res as possessing a token and giving permission for it to be seized a $\overline{gR}$ action, and later giving permission for it to be returned by a $\overline{pR}$ action. After the return action, the resource evolves back into a Res and is available for seizing again.

Non-deterministic choice We use + to represent choice of action. Here is a description of a Bin of size 2 which can be incremented or decremented by 1.

$$\begin{array}{lcccccc} Bin_2 & = & & & \overline{gB}.Bin_1 & + & \overline{isB2}.Bin_2 \\ Bin_1 & = & \overline{pB}.Bin_2 & + & \overline{gB}.Bin_0 & + & \overline{isB1}.Bin_1 \\ Bin_0 & = & \overline{pB}.Bin_1 & & & + & \overline{isB0}.Bin_0 \end{array}$$

When the bin is not empty, we may decrement by $\overline{gB}$. When bin spaces are free, we may increment by pB. Further we may quiz the buffer as to its current state by the $\overline{is}$ actions.

Parallel composition The | operator allows the concurrent operation of agents. Parallel agents synchronize by handshaking. A *handshake* can occur when an action is an input to one agent, an output from another agent, and both are enabled. When a signal is hidden (syntactically via \{}), the communication becomes local to those agents.

Here is a small system in which two LOADERs compete for a CRANE.

$$\begin{array}{lcl} CRANE & = & \overline{gC}.\overline{pC}.CRANE \\ LOADER & = & startL.gC.a2b.pC.LOADER \\ SYS & = & (CRANE \mid LOADER \mid LOADER) \setminus \{gC, pC\} \end{array}$$

SYS has three concurrent agents: two LOADERs and a CRANE, with gC and pC as handshakes. $\setminus\{gC, pC\}$ internalises the handshakes and only the LOADERs can signal on them. In contrast $startL$ and $a2b$ are “independent” actions which may be considered as self-determined (they can fire when they want). In the simulation model, explicit timing distributions are used for $startL$ and $a2b$. In CCS these actions last for an arbitrary time and the properties we prove are valid for all interleaving possibilities.

It is worth pointing out that in a CCS handshake on x, it does not matter which agent says x and which says $\overline{x}$. They are interchangeable. We have found it worthwhile in practice to be systematic and put output handshake actions ($\overline{x}$) in the resources and their corresponding input handshake actions in the entities.

Renaming We are permitted to reuse a definition. For example, our Demos model contains two resources of size 1, $PERMIT$ and $CRANE$. We may either define them from first principles as

$$\begin{array}{lcl} CRANE & \overset{def}{=} & \overline{gC}.\overline{pC}.CRANE \\ PERMIT & \overset{def}{=} & \overline{gP}.\overline{pP}.PERMIT \end{array}$$

or *rename* the existing standard template for a unit sized Res:

$$
\begin{aligned}
CRANE &\overset{def}{=} Res[gC/gR, pC/pR] \\
PERMIT &\overset{def}{=} Res[gP/gR, pP/pR]
\end{aligned}
$$

in which the construction a/b means replace action b with action a.

Later on we shall exploit renaming to define *generic* objects and instantiate them by appropriate renaming functions. To ease its initial reading, we do not rename in the following CCS descriptions of our Demos model.

Version 1.

$$
\begin{aligned}
CRANE &= \overline{gC}.\overline{pC}.CRANE \\
BUFF0 &= \overline{isB0}.BUFF0 + \overline{pB}.BUFF1 \\
BUFF1 &= \overline{isB1}.BUFF1 + \overline{pB}.BUFF2 + \overline{gB}.BUFF0 \\
BUFF2 &= \overline{isB2}.BUFF2 + \overline{gB}.BUFF1 \\
SLOTS2 &= \overline{isS2}.SLOTS2 + \overline{gS}.SLOTS1 \\
SLOTS1 &= \overline{isS1}.SLOTS1 + \overline{pS}.SLOTS2 + \overline{gS}.SLOTS0 \\
SLOTS0 &= \overline{isS0}.SLOTS0 + \overline{pS}.SLOTS1 \\
LOADER &= startL.gS.gC.a2b.pC.pB.LOADER \\
SHAPER &= startS.gB.gC.b2s.pC.pS.heat.S2R \\
S2R &= \overline{gM}.gC.s2r.pC.\overline{fM}.SHAPER \\
MILL &= startM.gM.fM.roll.(M2S + M2B) \\
M2S &= ok.gC.m2s.pC.MILL \\
M2B &= no.gS.gC.m2b.pC.pB.MILL \\
RESOURCES &= CRANE \mid BUFF0 \mid SLOTS2 \\
ENTITIES &= LOADER \mid SHAPER \mid MILL \\
SYS &= (RESOURCES \mid ENTITIES) \\
&\quad \setminus \{gM, fM, gB, pB, gC, pC, gS, pS, \\
&\qquad isB0, isB1, isB2, isS0, isS1, isS2\}
\end{aligned}
$$

Table I.1: Model (version 1) in CCS

We enter the model description (table I.1) into the CWB and try the built-in test for finding deadlock *fd*.

```
fd SYS;
--- startS startL tau<gS> tau<gC> a2b tau<pC> tau<pB> tau<gB>
```

```
                      tau<gC> b2s tau<pC> tau<putS> heat
startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
startL startM tau<gM> tau<gC> s2r tau<pC> tau<fM> roll no
startS tau<gB> tau<gC> b2s tau<pC> tau<putS> tau<gS>
                           tau<gC> a2b tau<pC> tau<pB> heat
startL ---> 0
```

The traces show how the system evolves (by the shortest route) into its final deadlocked state. Handshakes are shown as `tau<gS>` — the `tau` saying "handshake" and `<gS>` indicating which one.

By following the trace, we see that the LOADER has started five times; two loads have worked their way through the buffer, one is in the MILL, and one is in the SHAPER. Two others are in the buffer, which thus has no free slots. The 5th load is waiting on gS to gain entry to the buffer but is stuck. The SHAPER has completed one cycle, and handed over one billet to the MILL. The SHAPER is stuck on $\overline{gM}$ — the MILL is not ready. Why? Well it has rolled, rejected the result and is waiting on a buffer space (gS).

Version 2. The only definition to change is that of the MILL (table I.2) in which we use the is? check on the slots to dump a plate if it is faulty and no slot is available (M2X); otherwise (M2B), we have tested positively for a slot by $(isS2 + isS1)$, claim the slot and move the faulty plate into it.

$$\begin{array}{rcl}
MILL & = & startM.gM.fM.roll.(M2S + M2BX) \\
M2S & = & ok.gC.m2s.pC.MILL \\
M2BX & = & no.((isS2 + isS1).M2B + is0.M2X) \\
M2B & = & gS.gC.m2b.pC.pB.MILL \\
M2X & = & gC.m2x.pC.MILL
\end{array}$$

Table I.2: Model (version 2) in CCS

But this system too deadlocks. Since the claim and the seizure (either $isS2.gS$ or $isS1.gS$) are not atomic, one parallel interleaving will permit the MILL to make the latter check and then interleave a gS from the LOADER. The MILL is now blocked on its gS action, but the LOADER can cycle round to deadlock itself on another gS action.

As with many process-oriented simulators, the semantics of Demos keep an entity active as current until it itself decides to move on. Thus interleaving is entirely dictated by the current object. The above potential deadlock would not be picked up by Demos. But CCS examines every possible interleaving and does pick up this one:

```
--- startS startL tau<gS> tau<gC> a2b tau<pC> tau<pB> tau<gB>
                          tau<gC> b2s tau<pC> tau<putS> heat
    startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
    startL tau<gS> tau<gC> a2b tau<pC> tau<pB>
    startL startM tau<gM> tau<gC> s2r tau<pC> tau<fM> roll no
    startS tau<gB> tau<gC> b2s tau<pC> tau<putS> tau<isS1>
           tau<gS> tau<gC> a2b tau<pC> tau<pB> heat
   startL ---> 0
```

with the entities no further on than last time.

Version 3. The usual way to make sure that a sequence of actions is not interruptible, is to wrap them inside calls to a semaphore (the $PERMIT$ res). The fresh solution (we note only changed definitions) is given in table I.3.

$$
\begin{array}{rcl}
PERMIT & = & \overline{gP}.\overline{pP}.PERMIT \\[1ex]
LOADER & = & startL.gP.(A2B + LWAIT) \\
A2B & = & (isS2 + isS1).gS.gC.a2b.pC.pB.pP.LOADER \\
LWAIT & = & isS0.pP.LOADER \\[1ex]
MILL & = & startM.gM.fM.roll.(M2S + M2BX) \\
M2S & = & ok.gC.m2s.pC.MILL \\
M2BX & = & no.gP.((isS2 + isS1).M2B + is0.M2X) \\
M2B & = & gS.gC.m2b.pC.pB.pP.MILL \\
M2X & = & gC.m2x.pC.pP.MILL \\[1ex]
RESOURCES & = & CRANE \mid BUFF0 \mid SLOTS2 \mid PERMIT \\
SYS & = & (RESOURCES \mid ENTITIES) \\
 & & \quad \setminus \{gM, fM, gB, pB, gC, pC, gP, pP, gS, pS, \\
 & & \qquad isB0, isB1, isB2, isS0, isS1, isS2\}
\end{array}
$$

Table I.3: Model (version 3) in CCS

Having acquired permission, the LOADER makes sure there is a slot before calling gS and buffering its load. Only then does it release the semaphore. If there is no slot currently free, it releases the semaphore and tries again later. Similarly, the MILL only goes ahead with gS if there is a free slot; if not it dumps the load before releasing the semaphore. Testing on the CWB confirms that this model is deadlock free. Once that is established, a variety of other property checks can be carried out (as in section 4).

Version 4. Since the original Demos was embedded in Simula, it was easy to extend the language by tailor-made constructs. We follow that lead in table I.4 and modify the definition of $SLOTS$ in such a way as to permit the MILL to request and seize a free slot ($hasS2$ or $hasS1$) in one action (just as gS); on the other hand $hasS0$ returns at once so that dumping can go ahead.

$$\begin{aligned}
SLOTS2 &= \overline{hasS2}.SLOTS1 &&&& + \ \overline{gS}.SLOTS1 \\
SLOTS1 &= \overline{hasS1}.SLOTS0 &+&\ \overline{pS}.SLOTS2 && + \ \overline{gS}.SLOTS0 \\
SLOTS0 &= \overline{hasS0}.SLOTS0 &+&\ \overline{pS}.SLOTS1 \\
LOADER &= startL.gS.gC.a2b.pC.pB.LOADER \\
\\
MILL &= startM.gM.fM.roll.(M2S + M2BX) \\
M2S &= ok.gC.m2s.pC.MILL \\
M2BX &= no.gP.((hasS2 + hasS1).M2B + has0.M2X) \\
M2B &= gC.m2b.pC.pB.MILL \\
M2X &= gC.m2x.pC.MILL \\
\\
RESOURCES &= CRANE \mid BUFF0 \mid SLOTS2 \\
SYS &= (RESOURCES \mid ENTITIES) \\
&\qquad \backslash \ \{gM, fM, gB, pB, gC, pC, gS, pS, \\
&\qquad\quad isB0, isB1, isB2, hasS0, hasS1, hasS2\}
\end{aligned}$$

Table I.4: Model (version 4) in CCS

The CWB reports no deadlock on this model. Whilst it is easy to mimic such a tailored Demos monitor in CCS, this is usually left as a last resort as it would also imply tailored extensions to the translation toolkit.

I.3.2 Modelling in SCCS

As their names imply, SCCS and CCS are similar approaches to the description of concurrent systems. (For an introduction, read [59, chapter 9]). Their major difference lies in the ways they handle parallel composition. In SCCS we write $P \times Q$ for the synchronous parallel composition of agnets P and Q. In synchronous parallel, we insist that if the composition is to perform some action, then each of the components must perform some action (and, of course, the resulting action is the composition of the component behaviours).

Temporarily overloading the prefix operator, if we are given $P \overset{def}{=} a.P$ and $Q \overset{def}{=} b.Q$, then we take $P|Q = a.(P|Q) + b.(P|Q)$ and $P \times Q = (a\#b).(P \times Q)$. Within SCCS we have a structure upon actions. When we have a complementary pair of actions, say a and $\overline{a}$ in SCCS then we define that $a\#\overline{a} = \mathbf{1}$. The **1** action witnesses one synchronous time step being performed with no visible computation, so it can also be used to represent the passage of one unit of time. The action composition operator $\#$ is commutative and associative, hence the order of formation of a parallel composition will not matter.

Our presentation of SCCS mirrors that given for CCS earlier. The simplest agent within SCCS is the process $\mathbf{0}$ which does nothing. However, unlike its CCS counterpart, if we compose $\mathbf{0}$ in synchronous parallel with any other process, that too is deadlocked.

Prefixing The agent $\overline{tick} : \mathbf{0}$ can perform a single $\overline{tick}$ action. To form a persistent process, we write $GFclock \overset{def}{=} \overline{tick} : \overline{tock} : GFclock$, which can perform any number of $\overline{tick}$ then $\overline{tock}$ actions. We could define a crane in SCCS by $CRANE \overset{def}{=} \overline{gC} : \overline{pC} : CRANE$. However, an important use of choice in SCCS is in defining an operator that allows actions to wait for their complements. The actions defined so far demand to be acted upon *at once*. For instance the $CRANE$ agent demands that it be able to perform the $\overline{gC}$ action immediately or else it deadlocks the *whole* system.

To permit such agents to wait, we make use of an auxiliary operator $\$$: $\$P \overset{def}{=} P + \mathbf{1} : \$P$. P can idle by performing any number of $\mathbf{1}$'s whilst awaiting a process that will synchronise with it. So a more appropriate description of the crane might be $CRANE \overset{def}{=} \$\overline{gC} : \$\overline{pC} : CRANE$. This crane can wait until it is wanted, and similarly wait until it is returned.

Non-Deterministic choice We define a bin by:

$$\begin{array}{lcllclcl}
Bin_0 & = & \$(\overline{pB} : Bin_1 & & & + & \overline{isB0} : Bin_0) \\
Bin_1 & = & \$(\overline{pB} : Bin_2 & + & \overline{gB} : Bin_0 & + & \overline{isB1} : Bin_1) \\
Bin_2 & = & & & \$(\overline{gB} : Bin_1 & + & \overline{isB2} : Bin_2)
\end{array}$$

in which the position of the $\$$ prefix is important. If we simply distribute it before the actions, then some choices would no longer be available after one $\mathbf{1}$ had gone by.

Parallel Composition We form the synchronous parallel composition of agents with the × operator. In order to enforce local communication, we have a *permission* operator which lists those actions a process may perform. To perform actions *not* in the permission sets components must find other components willing to synchronise upon that action. Consider the example system with just two LOADERs and a CRANE:

Two forms of synchronisation are exhibited above: one in which both the input and output actions are guarded by $\$$ (the gC action pair); and one where only the output

$$\begin{array}{lcl} CRANE & = & \$\overline{gC}.\$\overline{pC}.CRANE \\ LOADER & = & startL.\$gC.a2b.pC.LOADER \\ SYS & = & (CRANE \times LOADER \times LOADER)\lceil\{\mathbf{1}, startL, a2b\} \end{array}$$

is guarded by a \$ (the pC pair). In the first case these represent two asynchronous actions, in that they can both wait an arbitrary time before firing. In the second case one action can wait, the other must fire as soon as it is encountered. Thus, LOADERs are willing to wait for the CRANE to become available, but will return the CRANE *as soon as* they are finished with it. The CRANE simply waits to be either taken or returned, and cannot affect the timing of the behaviours. In effect, we are assuming that we can always put the CRANE back at once, but we may have to wait to get it.

For a synchronisation on an action pair, say gC we have the following four possible handshaking options:

1. $\$gC$ ... $\$\overline{gC}$: the actions can wait for an arbitrary time before performing the synchronisation, by performing **1** actions;
2. gC ... $\$\overline{gC}$: the output can wait, but the input will be performed straight away;
3. $\$gC$ ... $\overline{gC}$: the input can wait, but the output must be performed straight away;
4. gC ... $\overline{gC}$: must synchronise at once. If both actions are not available at the same time step, then the above synchronisation can deadlock.

Renaming As with CCS.

Version 3 of our Demos model in SCCS We can enter our SCCS (see table I.5) description into the CWB and demonstrate that it is deadlock free.

Our descriptions in CCS and SCCS exhibit the same behaviour, as Demos entities are deterministic. However, we have the ability in SCCS to form compound atomic actions (e.g., $a\#b$ begin a and b performed *simultaneously* in parallel). As we shall see in sections 6 and 7, SCCS models may be much more compact for more complicated scenarios with waits until and conditions. Further using SCCS gives us the capability of addressing simple timing properties of Demos models.

$$
\begin{aligned}
PERMIT &= \$\overline{gP} : \$\overline{pP} : PERMIT \\
CRANE &= \$\overline{gC} : \$\overline{pC} : CRANE \\
\\
BUFF0 &= \$(\overline{isB0} : BUFF0\ \ + \overline{pB} : BUFF1) \\
BUFF1 &= \$(\overline{isB1} : BUFF1\ \ + \overline{pB} : BUFF2\ \ \ + \overline{gB} : BUFF0) \\
BUFF2 &= \$(\overline{isB2} : BUFF2 \qquad\qquad\qquad\qquad\ \ + \overline{gB} : BUFF1) \\
\\
SLOTS2 &= \$(\overline{isS2} : SLOTS2 \qquad\qquad\qquad\qquad\ \ + \overline{gS} : SLOTS1) \\
SLOTS1 &= \$(\overline{isS1} : SLOTS1 + \overline{pS} : SLOTS2\ \ + \overline{gS} : SLOTS0) \\
SLOTS0 &= \$(\overline{isS0} : SLOTS0 + \overline{pS} : SLOTS1) \\
\\
LOADER &= startL : \$gP : (A2B + LWAIT) \\
A2B &= \$(isS2 + isS1) : \$gS : \$gC : a2b : pC : \$pB : pP : LOADER \\
LWAIT &= \$isS0 : \$pP : LOADER) \\
SHAPER &= startS : \$gB : \$gC : b2s : \$pC : \$pS : heat : S2R \\
S2R &= \$\overline{gM} : \$gC : s2r : \$pC : \overline{fM} : SHAPER \\
MILL &= startM : \$gM : \$fM : roll : (M2S + M2BX) \\
M2S &= ok : \$gC : m2s : pC : MILL \\
M2BX &= no : \$gP : (\$(isS2 + isS1) : M2B + \$isS0 : M2X) \\
M2B &= \$gS : \$gC : m2b : pC : \$pB : pP : MILL \\
M2X &= \$gC : m2x : pC : pP : MILL \\
\\
RESOURCES &= CRANE \times BUFF0 \times SLOTS2 \times PERMIT \\
ENTITIES &= LOADER \times SHAPER \times MILL \\
VISIBLE &= \{\mathbf{1}, startL, a2b, startS, b2s, heat, s2r, startM, \\
&\qquad roll, m2b, m2s, m2x\} \\
\\
SYS &= (RESOURCES \times ENTITIES) \lceil VISIBLE
\end{aligned}
$$

Table I.5: Model (version 3) in SCCS

I.4 Checking the translated model

Once we have a model description in CCS, we can check its characteristic properties using the CWB. Claims take the form $SYS \models property$. The CWB examines SYS from its initial state and checks to see if the logical property holds. The first step on the CWB is to minimise the CCS definition down to the equivalent state machine with the least number of states, henceforth SYS'. The theory behind CCS assures us that there is but one such machine and (loosely speaking) that its non-τ properties will be preserved. The minimised machines are often orders of magnitude smaller than the composed definitions (about 10× in our case) so minimisation speeds up property checking. Typical simple "one-move" checks are $SYS' \models \langle startL \rangle\, T$ which asks “Can SYS’ make an initial move of startL?” which would return true, as would *SYS’* $\models \langle - \rangle\, T$ which asks “Can SYS’ make any initial move at all?”. The notation extends in an obvious way so that we may make two move checks such as *SYS’* $\models \langle startL \rangle \langle startS \rangle\, T$ which asks “Having made a startL move, can SYS’ then make a startS move?” (true) and *SYS’* $\models \langle startL \rangle \langle startL \rangle\, T$ which returns false since SYS’ cannot make successive startL moves. Whereas $\langle a \rangle$ tracks one a move from a state (one path), its dual [a] follows all a moves from a state.

It is possible (see [94, 97]) to define macros in modal-μ which traverse all the reachable states of a CCS agent. The most used are:

S $\models$ **BOX p** returns true only if S and every state reachable from S has property p. Our check for deadlock is simply *SYS’* $\models$ *BOX*$\langle - \rangle\, T$ — can each and every reachable state make a move?

S $\models$ **POSS p** returns true only if S or at least one state reachable from S has property p. Thus *SYS’* $\models$ *POSS*$\langle roll \rangle\, T$ asks if it is possible under some system unfolding to execute a roll action.

Note that BOX and POSS are duals; any property formulated with one operator can be reformulated in terms of the other. The dual check for deadlock is *SYS’* $\models$ $\neg$*POSS(*$\neg\langle - \rangle$*T)* or *SYS’* $\models$ $\neg$*POSS([-]F).*

S $\models$ **EVENT p** returns true only if S has property p or else p holds at some state on every path from S. In otherwords, however the system unfolds, you must eventually reach a state with property p. *SYS’* $\models$ *EVENT*$\langle roll \rangle\, T$ asks whether SYS’ always reaches a state capable of a roll action.

S $\models$ **PATH p** returns true there is a loop of actions with property p reachable from S. Thus *SYS* $\models$ *PATH*$\langle \tau \rangle\, T$ returns true if SYS is livelocked (has a cycle of τ moves). Since minimisation mainly removes chains of internal handshakes, we have to be careful when checking for livelock to operate with the unminimised definition, since the livelock check explicitly looks for τ operations.

PATH is actually more general than stated — there may also be a chain of states with property p leading to a deadlocked state. Note that EVENT and PATH are duals.

It is possible to combine the above macros for extra expressiveness. Safety, liveness, progress, and fairness properties require the model to be decorated with trace actions at appropriate points. ADVANCE and HOLD actions of GPSS and Demos models serve this purpose. Corresponding to every Demos $hold(T.sample)$, we write T in the CCS description, together with say $startP$ prefixing each agent definition P. Typical tailored checks might now be:

Progress checks to make sure that each process will eventually carry out a certain action.

$$\begin{array}{lll} SYS & \models & BOX[startL](EVENT\langle a2b\rangle T) \\ SYS & \models & BOX[startL](EVENT\langle startL\rangle T) \end{array}$$

read as "in every state following an action *startL* (BOX[startL]), (EVENT)ually one must reach a state permitting an *a*2*b*/*startL* action however the system unfolds".

Safety tests to see that bad things cannot happen. We may want to check that it is never possible to get two crane movements to the buffer as consecutive moves:

$$SYS \quad \models \quad BOX[a2b][m2b]F$$

read as "however the system unfolds, in every state following an *a*2*b* action (BOX[a2b]) an immediate *m*2*b* action is impossible".

Liveness tests to see that good things may happen (e.g. each movement from the MILL may be accepted).

$$\begin{array}{lll} SYS & \models & \mathrm{BOX(POSS}\langle m2s\rangle \mathrm{T)} \\ SYS & \models & \mathrm{BOX(POSS}\langle m2b\rangle \mathrm{T)} \\ SYS & \models & \mathrm{BOX(POSS}\langle m2x\rangle \mathrm{T)} \end{array}$$

read as "from every state (BOX) there exists a path (POSS) to some state where action *m2s*, or *m2b*, or *m2x* is enabled ($\langle m2s/m2b/m2x\rangle$ T)".

Fairness means that a system can not "spin" forever without enabling some particular input or output action. For any system *S*, and for a particular action *a*, this may be expressed as S $\models$ BOX EVENT$\langle a\rangle$T which reads "from every state (BOX) for each path (EVENT)ually there is a state in which *a* is enabled ($\langle a\rangle$T)". E.g.

$$\begin{array}{lll} \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startL\rangle \mathrm{T)} \\ \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startB\rangle \mathrm{T)} \\ \mathrm{SYS} & \models & \mathrm{BOX(EVENT}\langle startM\rangle \mathrm{T)} \end{array}$$

read as "in every state (BOX) however the system unfolds (EVENT)ually one must reach a state permitting the LOADER/SHAPER/MILL to start its workcycle again.

Although the above tests may be combined into one large statement of the system properties, it is much clearer and more readable to display them one at time, as above. The following books and papers are recommended for background material on property checking. Walker [107] and Milner [59] for material on CCS; Clarke et.al [20], and Manna and Pnueli [55, especially part II, pages 177–387] and [56] for a basic understanding of safety, liveness, and fairness properties and how to express them; Stirling [93, 94, 95, 97, 98] for links between CCS and process logics; Lynch and Wing [53, 52, 108] for some practical applications.

Summary of part I

We have demonstrated the core of Demos, a typical process oriented discrete event simulation language, and given informal translations to two different process calculi, CCS and SCCS. Such translations can be used to reason formally about the properties of original model on CWB and ensure that deadlock and livelock can never arise, and that specific safety (e.g. mutual exclusion), fairness and progress properties hold. Typical templates were exhibited for carrying out these checks.

In part II [13] of this paper we formalise and extend these translations, exhibit key portions of some interesting translations, and give and apply several theorems.