OPERATIONAL SEMANTICS OF PROCESS-ORIENTED SIMULATION LANGUAGES PART 1 : πDemos*

G. Birtwistle[1] and C. Tofts[2],
1 Department of Computer Science, University of Calgary,
2 Department of Computer Science, University of Swansea

12 October 1993

Abstract

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.

*Transactions of The Society for Computer Simulation, 10(4), December 1994, 299–333

Contents

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 [15, 16] structured operational semantics has gradually emerged as a prominent method of specifying programming languages and reasoning about programs written in them. In [15], Plotkin showed how to describe everyday programming constructs (expressions, statements, procedure/function calls, objects) and in [16], he dealt with guarded commands and CSP flavoured parallelism. Milner [11] used operational semantics to describe the non-deterministic interleaving semantics of CCS, and with others, [12, 13], to describe SML, a modern (mainly) functional programming language. The text by Hennessy [9] 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 [4] 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 [2, 3], 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 [5]. Alternative approaches based upon process logics are being explored in [17].

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.

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)

2.1 Notation

We have adopted certain notations from modern functional programming languages ([1, 7, 10, 14]) 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

$$\text{exec}(\text{current}::\mathcal{EL},\ \mathcal{R}[\text{id}/\text{RD}(\text{true},[])])$$

in simple steps as

$$\begin{array}{lcll} \textbf{let } \mathcal{R}' & = & \mathcal{R}[\text{id}/\text{RD}(\text{true},[])] & \textbf{in} \\ \textbf{let } \mathcal{EL}' & = & \text{current}::\mathcal{EL} & \textbf{in} \\ \quad \text{exec}(\mathcal{EL}',\ \mathcal{R}') & & & \end{array}$$

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.

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).

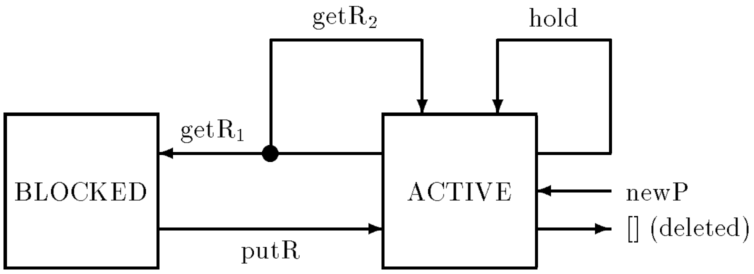

Figure 1: The ways processes change state

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].

```
EL = [ (V2, PD([putR(W)], [W], 6)), (MAIN, PD([close], [], 6))]
R  = [(W,F,[])]
```

[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.

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.

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 $(\mathcal{EL},\ \mathcal{R},\ \Sigma)$ 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

$$(\mathcal{EL},\ \mathcal{R},\ \Sigma) \Longrightarrow (\mathcal{EL}',\ \mathcal{R}',\ \Sigma')$$

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`.

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`.

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

$$\mathcal{EL}= [\ (\mathrm{id}_1,\ \mathrm{PD}(\mathrm{b}_1,\ \mathrm{a}_1,\ \mathrm{t}_1)),\ (\mathrm{id}_2,\ \mathrm{PD}(\mathrm{b}_2,\ \mathrm{a}_2,\ \mathrm{t}_2)),\ ...,\ (\mathrm{id}_n,\ \mathrm{PD}(\mathrm{b}_n,\ \mathrm{a}_n,\ \mathrm{t}_n))\ ]$$

then $\mathrm{t}_1 \leq \mathrm{t}_2 \leq ... \leq \mathrm{t}_n$. 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`.

$$\text{evTime (id, PD(Body,Attrs,evt))} \quad = \quad \text{evt}$$

pName en (event notice `en`) returns the identifier of `en`.

$$\text{pName (id, PD(Body,Attrs,evt))} \quad = \quad \text{id}$$

ENTER en $\mathcal{EL}$: enters the event notice `en` into the event list $\mathcal{EL}$ ranked acording to its event time

$$\begin{array}{lcl} \text{ENTER en []} & = & [\text{ en }] \\ \text{ENTER en1 (en2::EL)} & = & \textbf{if}\ \text{evTime en1} < \text{evTime en2} \\ & & \quad \textbf{then}\ \text{en1::en2::EL} \\ & & \quad \textbf{else}\ \text{en2::(ENTER en1 EL)} \end{array}$$

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.

$$\begin{array}{lcl} \text{DELETE id []} & = & \textit{error} \\ \text{DELETE id (en::}\mathcal{EL}\text{)} & = & \textbf{if } \text{id=pName en } \textbf{then } \mathcal{EL} \textbf{ else } \text{en::(DELETE id } \mathcal{EL}\text{)} \end{array}$$

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:

$$\frac{\begin{array}{c} constraint_1 \\ constraint_2 \\ \ldots \\ constraint_n \end{array}}{(\mathcal{EL}, \mathcal{R}, \Sigma) \quad \Longrightarrow \quad (\mathcal{EL}', \mathcal{R}', \Sigma')}$$

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:

$$\frac{\begin{array}{c}\text{current} = (\text{C, PD((newR(id)::Body, Attrs, time))} \\ \text{current'} = (\text{C, PD(Body, Attrs, time))} \\ \texttt{id} \notin \Sigma\end{array}}{(\text{current}::\mathcal{EL}, \mathcal{R}, \Sigma) \quad \Longrightarrow \quad (\texttt{current'}::\mathcal{EL},\ \mathcal{R}\texttt{[id/RD(true,[])]},\ \Sigma\ \texttt{++ id})}$$

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

$$\begin{array}{lll}
\multicolumn{3}{l}{(\text{current} = (\text{C, PD(newR(id)::Body, Attrs, time))}::\mathcal{EL}, \mathcal{R}, \Sigma)} \\
\Longrightarrow & \textbf{if } \text{id} \in \Sigma \textbf{ then } \textit{error} \textbf{ else} & \\
\quad | & \textbf{let } \mathcal{R}\text{'} = \mathcal{R}[\text{id/RD(true, [])}]) & \textbf{in} \\
 & \textbf{let } \mathcal{EL}\text{'} = \text{current}::\mathcal{EL} & \textbf{in} \\
 & \textbf{let } \Sigma\text{'} = \Sigma \text{ ++ id} & \textbf{in} \\
 & \quad (\mathcal{EL}\text{'}, \mathcal{R}\text{'}, \Sigma\text{'}) &
\end{array}$$

(the **let**s merely break the description into simple steps).

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 ([], $\mathcal{R}$, Σ) $\Longrightarrow$ *error*

2 exec ((C, PD([],Attrs,time))::$\mathcal{EL}$, $\mathcal{R}$, Σ) $\Longrightarrow$
if Attrs=[] then exec($\mathcal{EL}$, $\mathcal{R}$, Σ) else *error*

3 exec ((C, PD(b::Body,Attrs,time))::$\mathcal{EL}$, $\mathcal{R}$, Σ)
$\Longrightarrow$ **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`.

4.4.1 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:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{decP(classId, classDef)}} \\
\Longrightarrow & \textbf{if } \text{classId} \in \Sigma \quad \textbf{then } \textit{error} & \textbf{else} \\
 & \textbf{let } \mathcal{EL}\text{'} = \text{current::}\mathcal{EL} & \textbf{in} \\
 & \textbf{let } \Sigma\text{'} = \Sigma[\text{classId/classDef}] & \textbf{in} \\
 & \quad \text{exec}(\mathcal{EL}\text{'}, \mathcal{R}, \Sigma\text{'}) &
\end{array}
$$

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')

4.4.2 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:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{newP(id, classId, dt)}} \\
\Longrightarrow & \textbf{if } \text{id} \in \Sigma & \textbf{then } \textit{error} & \textbf{else} \\
& \textbf{if } \text{classId} \notin \Sigma & \textbf{then } \textit{error} & \textbf{else} \\
& \textbf{if } \text{not(classId } \textbf{is} \text{ class definition)} & \textbf{then } \textit{error} & \textbf{else} \\
& \textbf{if } \text{dt} < 0 & \textbf{then } \textit{error} & \textbf{else} \\
& \textbf{let } \text{classDef} = \text{LOOKUP classId } \Sigma & & \textbf{in} \\
& \textbf{let } \text{en} = \text{(id, PD(classDef, [], time+dt))} & & \textbf{in} \\
& \textbf{let } \mathcal{EL}\text{'= current::(ENTER en } \mathcal{EL}) & & \textbf{in} \\
& \textbf{let } \Sigma\text{'} = \Sigma \text{ ++ id} & & \textbf{in} \\
& \quad \text{exec}(\mathcal{EL}\text{'}, \mathcal{R}, \Sigma\text{'}) & &
\end{array}
$$

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)`.

4.4.3 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($\mathcal{EL}$', $\mathcal{R}$, Σ)

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)

4.4.4 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:

$$
\begin{array}{ll}
\text{newR(id)} & \\
\Longrightarrow & \textbf{if } \text{id} \in \Sigma \;\; \textbf{then } \mathit{error} \textbf{ else} \\
& \textbf{let } \mathcal{EL}\text{'} = \text{current::}\mathcal{EL} \qquad\qquad\qquad \textbf{in} \\
& \textbf{let } \mathcal{R}\text{'} = \mathcal{R}[\text{id/RD(true, [])}] \quad \textbf{in} \\
& \textbf{let } \Sigma\text{'} = \Sigma \text{ ++ id} \qquad\qquad\qquad\quad \textbf{in} \\
& \quad \text{exec}(\mathcal{EL}\text{'}, \mathcal{R}\text{'}, \Sigma\text{'})
\end{array}
$$

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')

4.4.5 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.

$$
\begin{array}{l}
\text{getR(id)} \\
\quad \Longrightarrow \quad \textbf{if } \text{id} \in \text{Attrs } \textbf{then } \textit{error} \textbf{ else} \\
\qquad\qquad \textbf{case } \text{LOOKUP id } \mathcal{R} \textbf{ of} \\
\qquad\qquad\quad \text{RD(true, [])} \\
\qquad\qquad\quad \Rightarrow \quad \textbf{let } \text{Attrs' = Attrs ++ id} \hfill \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let } \mathcal{EL}\text{' = (C, PD(Body,Attrs',time))::}\mathcal{EL} \quad \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let } \mathcal{R}\text{' = }\mathcal{R}\text{[id/RD(false, [])]} \hfill \textbf{in} \\
\qquad\qquad\qquad\qquad \text{exec(}\mathcal{EL}\text{', }\mathcal{R}\text{', }\Sigma\text{)} \\
\\
\qquad\qquad | \quad \text{RD(false, Q)} \\
\qquad\qquad\quad \Rightarrow \quad \textbf{let } \text{Q' = Q@[current]} \hfill \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let } \mathcal{R}\text{' = }\mathcal{R}\text{[id/RD(false, Q')]} \hfill \textbf{in} \\
\qquad\qquad\qquad\qquad \text{exec(}\mathcal{EL}\text{, }\mathcal{R}\text{', }\Sigma\text{)} \\
\\
\qquad\qquad | \quad \text{anythingelse} \Rightarrow \textit{error}
\end{array}
$$

Interpretation.

1. *id* $\in$ *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* $\Rightarrow$ *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.

4.4.6 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.

$$
\begin{array}{l}
\text{putR(id)} \\
\Longrightarrow \quad \begin{array}[t]{ll}
\textbf{if } \text{id} \notin \text{Attrs } \textbf{then } \textit{error} \textbf{ else} & \\
\textbf{let } \text{Attrs'} = \text{Attrs} - \text{id} & \textbf{in} \\
\textbf{let } \mathcal{EL}\text{'} = \text{(C, PD(Body, Attrs', time))::}\mathcal{EL} & \textbf{in} \\
\textbf{case } \text{LOOKUP id } \mathcal{R} \textbf{ of} & \\
\quad \text{RD(false, [])} & \\
\quad \Rightarrow \quad \textbf{let } \mathcal{R}\text{'} = \mathcal{R}\text{[id/RD(true, [])])} & \textbf{in} \\
\quad \qquad\qquad \text{exec(}\mathcal{EL}\text{', }\mathcal{R}\text{', }\Sigma\text{)} & \\
 & \\
| \quad \text{RD(false, (p1, PD(B1,A1,t1))::Q1)} & \\
\quad \Rightarrow \quad \textbf{let } \mathcal{R}\text{'} = \mathcal{R}\text{[RD(id/(false, Q)]} & \textbf{in} \\
\quad \qquad \textbf{let } \text{A1'} = \text{A1 ++ id} & \textbf{in} \\
\quad \qquad \textbf{let } \text{en} = \text{(p1,PD(B1,A1',time))} & \textbf{in} \\
\quad \qquad \textbf{let } \mathcal{EL}\text{"} = \text{ENTER en } \mathcal{EL}\text{'} & \textbf{in} \\
\quad \qquad\qquad \text{exec(}\mathcal{EL}\text{", }\mathcal{R}\text{', }\Sigma\text{)} & \\
 & \\
| \quad \text{anythingelse} \Rightarrow \textit{error} &
\end{array}
\end{array}
$$

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* $\Rightarrow$ *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`)

4.4.7 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.

5 Applications

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 [14], 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 [6]). 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

[2] The traces shown in this paper came from this toy implementation

```
fun LOOKUP id []          = error
  | LOOKUP id ((r,rd)::R) = if id=r then rd else LOOKUP id R
```

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($\mathcal{EL}$', $\mathcal{R}$', Σ)

| 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($\mathcal{EL}$", $\mathcal{R}$', Σ)

| 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.

5.2 Proofs

Work is underway with Tom Melham of Glasgow University formalising and proving facts about πDemos in the HOL proof assistant [8]. 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.

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 [5]); 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.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, Hemel Hempstead, UK, 1988.

[2] G. Birtwistle. *DEMOS — a system for discrete event modelling on Simula.* Macmillen, London, 1979.

[3] G. Birtwistle. The Demos Implementation Guide and Reference Manual. Technical Report, 260 pages, Computer Science Department, University of Calgary, 1983.

[4] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula begin.* Studentlitteratur, Lund, Sweden, 1973.

[5] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 2: Demos. Computer Science Technical Report, University of Calgary, 1993.

[6] A. Cave Brown. *C.* Macmillan, New York, 1987.

[7] W. Burge. *Recursive Programming Techniques.* Addison-Wesley, New York, 1975.

[8] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, 1993.

[9] M. Hennessy. *The Semantics of Programming Languages.* John Wiley, Chichester, England, 1990.

[10] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, London, 1986.

[11] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[12] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Mass., 1991.

[13] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[14] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, Britain, 1991.

[15] G. D. Plotkin. A Structural Approach to Operational Semantics. Research Report DAIMI FN–19, Computer Science Department, Aarhus University, Denmark, 1981.

[16] G. D. Plotkin. An Operational Semantics for CSP. Research Report CSR–114–82, Computer Science Department, Edinburgh University, Scotland, 1982.

[17] C. Tofts. Process Semantics for Simulation. Technical Report, Department of Mathematics and Computer Science, University of Swansa, Swansea, Wales, 1993.

7 APPENDIX

```
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))], [], []);
```