1 Motivation

In the main, discrete-event simulation is used to investigate "hard" models with no known analytic solutions. Although there are several popular programming styles in which to describe discrete event models, it is easy to translate amongst them [6]. Thus without loss of generality, we concentrate upon the process oriented style popularised by Simula in the 1960's. Simula programs are collections of interacting processes in which each entity of interest in the model gets mapped into a specific object in the Simula description. An object has its own data values, its own operators (local procedures), and its own actions. Each object is also equipped with a local sequence control that keeps track of which action it is currently carrying out.

If we strip away all the purely computational aspects (data collection and reporting), we are left with an object description with the same structure but containing only synchronisations. Such skeletal descriptions are closely allied to system descriptions in process algebras for which there exists a considerable body of results and techniques (see CCS [11], BPA [2, 3] for example). Using these techniques we can test a system description expressed in a process algebraic notation to see if it is deadlock-free, livelock-free, is safe, is live, and is fair. These techniques are applied directly to the static code skeleton, and we can establish that certain properties hold without running the model at all. The properties we prove are guaranteed valid for all possible system timings, i.e. they hold no matter how long a process is blocked waiting upon on a resource, no matter how long a task takes, and whatever queueing strategy is adopted. This very conservative approach which has its pros and cons. It means any proofs are sample (indeed distribution) independent, and are valid for all possible runs of the model. On the other hand, we may not be able to show specialised results that depend upon specific timings.

The main point of this paper 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 methodology 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.

We have chosen to work with CCS, one of the simpler process algrbras. CCS descriptions erase functionality and retain only the synchronisations. The bonus is that CCS descriptions are crisp and clear, and their testing can be mechanized efficiently. The main drawback, that CCS covers synchronisations but not functionality, is partially countered by the argument that getting the synchronisations right is by far the hardest part of the task of debugging a complex interacting system. 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 [9, 12, 13, 14, 15, 18, 19]. See also [2, 3].

The following papers are recommended for backgound material. Walker [20] and Milner [11] for material on CCS; Manna [10, part II, pages 177–387] for a basic understanding of safety and liveness properties and how to express them; and Stirling [17] for the link between CCS and process logics.

The paper is structured as follows. Section 2 serves as an introduction to CCS. Section 3 shows how to express simulation synchronisations in CCS. Section 4 introduces a particular process logic and shows how to use it on a non-trivial simulation model. Finally we summarize our findings and suggest how these techniques could become part of a standard methodology for developing simulation models.

2 CCS notation

CCS is a language in one which may describe the various ways in which co-operating sequential processes can interact with each another. Examples of typical processes are: the receive, send, and retransmit processes in the X25 link-level; arbiters and mutual exclusion elements in asynchronous hardware design; boats, trucks, cranes in a discrete-event harbour simulation; etc. We will find that such simulation processes map directly into CCS processes, one for one. To help distinguish simulation model processes from CCS processes we call the former *objects* and the latter *agents*.

CCS agents may be constructed by prefixing ., non-deterministic choice $+$, or by parallel composition \{}.

Sequencing. The simplest agent is the one that does nothing, **0**. We can construct agents with more interesting behaviours by prefixing 0 with any number of actions. The agent $Match_1 \overset{def}{=} strike.0$ may engage in a single *strike* action and then evolves into 0, whereas

$$Match_2 \overset{def}{=} strike.\overline{burn}.0$$

strikes then burns before becoming spent. By convention, output actions are given *co-names* (they are overlined). I.e. the co-name of action α is $\overline{\alpha}$. We take the co-name of $\overline{\alpha}$ to be α, i.e. $\overline{\overline{\alpha}} = \alpha$.

Recursive definitions are allowed. A clock that ticks away forever is described by $Clock \overset{def}{=} \overline{tick}.Clock$.

In general, the agent defined by $A \overset{def}{=} \alpha_1.\alpha_2.....\alpha_n.B$ may also be written in "stages" as

$$A_1 \overset{def}{=} \alpha_1.A_2 \qquad A_2 \overset{def}{=} \alpha_2.A_3 \qquad \ldots \qquad A_n \overset{def}{=} \alpha_n.B$$

where the α_k are actions and B is an agent (e.g. 0 or A or ...). We interpret the agent A as one which first executes the action α_1 and evolves into the agent A_2 which then executes the action α_2 and evolves into the agent A_3 which then ... and then executes the action α_n and evolves into the agent

B. We are guaranteed that the action α_1 preceeds action α_2 that the action α_2 preceeds action α_3, ..., and the action α_{n-1} preceeds the action α_n.

Example: send process We can now start to illuminate the mapping between a fully fledged simulation object and its skeleton in CCS. In general, synchronisations (such as acquiring or releasing a resource) are mapped into CCS handshakes, task durations (ADVANCE in GPSS or HOLD in Simula) are mapped into markers (names), and all other statements (e.g. explicit data collection, assignments) are ignored. Here is the partial description in DEMOS [4, 5], a Simula extension, of a *SEND* process in a long haul network.

```
entity class SEND;
begin
  integer NextFrameToSend;
  receiveFromHost;
  NextFrameToSend := NextFrameToSend + 1;
  sendFrame(NextFrameToSend);
  startTimer;
  ....
  repeat;
end***SEND***;
```

The CCS code for *SEND* is simply:

$$SEND \overset{def}{=} \mathit{receiveFromHost}.\overline{\mathit{sendFrame}}.\overline{\mathit{startTimer}}\ldots SEND$$

In the CCS description of *SEND*, we know that *receiveFromHost* completes before $\overline{\mathit{sendFrame}}$ can start, and that $\overline{\mathit{sendFrame}}$ completes before $\overline{\mathit{startTimer}}$ can start, but we do not know how long each action will take. This means that, if we so wish, we can absorb the time taken to effect computational statements like $NextFrameToSend := NextFrameToSend + 1$ into the next action without penalty. As noted before, proofs we make about CCS models are valid whatever the timings, so that if we replaced $NextFrameToSend := NextFrameToSend + 1$ by computations that were longer or shorter it would not affect our reasoning in the CCS model one jot.

The CCS description erases functionality and retains only the synchronisations. The bonus is that CCS descriptions are crisp and clear, and their testing can be mechanized efficiently. The main drawback, that CCS covers synchronisations but not functionality, is partially countered by the argument that getting the synchronisations right is by far the hardest part of the task of debugging a complex interacting system. Additionally CCS is one of the simplest process algebras: varieties exist which can cope with hard timings and functionality (value passing), but they are harder to reason with and are correspondingly less well mechanised [9, 12, 13, 14, 15, 18, 19].

Choice. $+$ is used to represent non-deterministic choice, as in

$$Match_3 \overset{def}{=} strike.(\overline{burn} + Match_3)$$

which describes a match that after being struck may burn and become spent, or be struck again. The match has an unfair behaviour in that it may never do a $\overline{burn}$ action.

Example : mouse click handler[1]**.** A mouse sends d and u signals to MCH (a mouse click handler) which interprets them and sends individual commands to a single click handler, or a double click handler, or a menu-select handler.

Three types of reaction are catered for by the mouse click handler:

1. a du pair are close together — send sc to the single click handler

$$d.u.\overline{sc}.MCH$$

2. a du pair is swiftly followed by another — $dudu$ this is a double click, send dc to the double click handler

$$d.u.d.u.\overline{dc}.MCH$$

3. a d is NOT swiftly followed by a u — in this case we are menu browsing, and want to stop browsing when the u signal appears. Send a select start ss to the selector handler after the d, and a select stop se to the selector handler after the u.

[1]Taken from IEEE Transaction on Software Engineering, September 1992, pages 810–811. See also, Proc SIGGRAPH, 1985, pages 199–205.

$$d.\overline{ss}.u.\overline{se}.MCH$$

Combining possible tracks with the choice operator + results in the specification:

$$MCH \overset{def}{=} d.(u.(\overline{sc}.MCH + d.u.\overline{dc}.MCH) + \overline{ss}.u.\overline{se}.MCH)$$

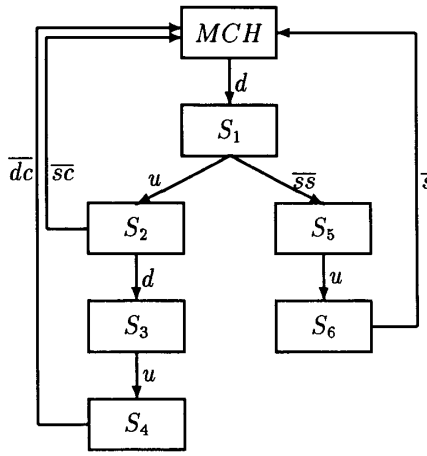

Figure 1: Possible evolutions of MCH

CCS can be seen as a compact algebraic notation in which to express the behaviour of a finite state machine (see figure 1). The CCS definition describes all the possible ways in which the system can evolve.

Parallel composition. We give the intuition behind this construct in simple stages.

Example: interleaving behaviour. Consider an system with two users running freely in parallel. Each user U_k has cyclic behaviour consisting of a non-critical section n_k followed by a critical section c_k. The system is specified by describing its constituent objects and them composing them together (with |):

$$\begin{aligned} U_1 &\stackrel{def}{=} n_1.c_1.U_1 \\ U_2 &\stackrel{def}{=} n_2.c_2.U_2 \\ \\ SYS_1 &\stackrel{def}{=} (U_1 \mid U_2) \end{aligned}$$

The possible behaviour paths of SYS_1 are shown in figure 2.

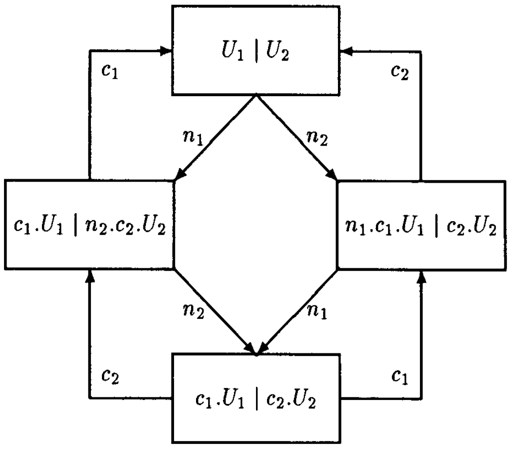

Figure 2: State-by-state expansion of SYS_1

At any given stage, the system may evolve by carrying out *either* the next (or *leading*) action from U_1 *or* the leading action from U_2, ***but not both at the same time.*** For example from the initial state $n_1.c_1.U_1 \mid n_2.c_2.U_2$,

- $U_1 = n_1.c_1.U_1$ may complete an n_1 action and evolve into $c_1.U_1$ leaving U_2 unchanged, or
- $U_2 = n_2.c_2.U_2$ may complete an n_2 action and evolve into $c_2.U_2$ leaving U_1 unchanged.

As can be seen from figure 2, four states are distinguished, one of which $(c_1.U_1 \mid c_2.U_2)$ has both users in their critical sections at the same time. This is presumably not what we want.

Example: mutual exclusion. We can prevent overlapping critical sections by protecting them with a semaphore *Sem* and insisting that users wishing to enter their critical section have to gain control of the semaphore beforehand. We think of the semaphore as controlling a token which each user must get before entering its critical section and put back after completing its critical sections. A user is blocked should the token not be available when it makes a request. Here is a semi-formal CCS description of the system components:

$$\begin{tikzcd}
U_1 & \overset{def}{=} & nc_1. & get. & c_1. & put. \arrow[d, Rightarrow] & U_1 \\
Sem & \overset{def}{=} & & \blacksquare \arrow[u, Rightarrow] \arrow[d, Rightarrow] & & \square. & Sem \\
U_2 & \overset{def}{=} & nc_2. & get. & c_2. & put. \arrow[u, Rightarrow] & U_2
\end{tikzcd}$$

We define the semaphore formally by $Sem \overset{def}{=} \overline{gT}.pT.Sem$. Initially it can carry out a $\overline{gT}$ action (outputting the token), then a pT action (accepting the token back), and then it evolves into a *Sem* again. The user template is updated to $U_k \overset{def}{=} nc_k.gT.c_k.\overline{pT}.U_k$, (k = 1, 2) in which the critical section code of each user is bracketted by calls to get the token and then return it. Notice that the specification of the semaphore *Sem* and the users U_k are closely related — names in one are co-names in the other. The complete CCS description of a system with one semaphore and two users is:

$$\begin{aligned}
U_1 &\overset{def}{=} n_1.gT.c_1.\overline{pT}.U_1 \\
U_2 &\overset{def}{=} n_2.gT.c_2.\overline{pT}.U_2 \\
Sem &\overset{def}{=} \overline{gT}.pT.Sem \\
\\
SYS_2 &\overset{def}{=} (U_1 \mid U_2 \mid Sem) \setminus \{gT, pT\}
\end{aligned}$$

What are new there are the notions of a handshake and hidden actions ($\setminus\{gT, pT\}$). The ways in which the system may evolve are depicted in figure 3.

In our first description, SYS_1, none of the actions were hidden, and any leading action could fire at any time. In this description gT and pT are hidden. A hidden action has a different firing mode: when leading, it cannot fire until another agent offers a leading corresponding co-action. I.e.

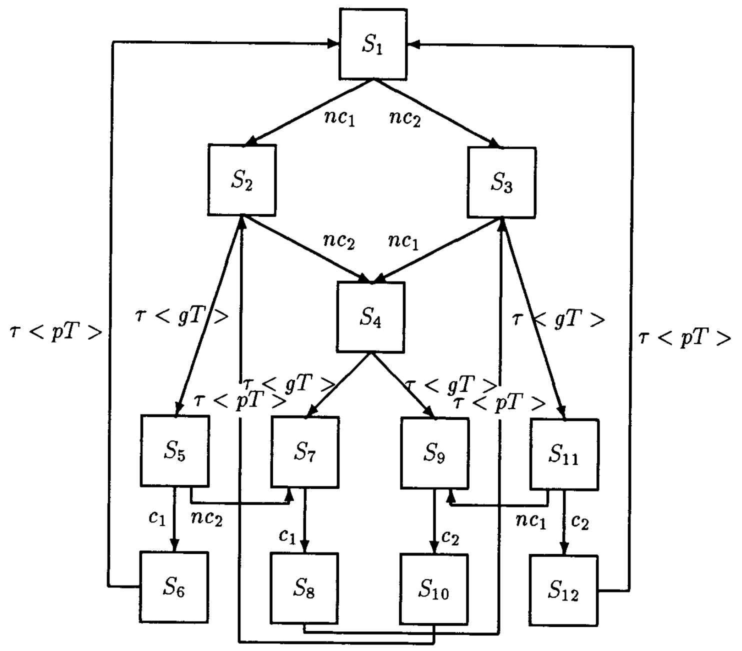

Figure 3: State-by-state expansion of SYS_2

the semaphore, $\overline{gT}.pT.Sem$ cannot fire until (at least) one user offers a gT and then a handshake may occur. When that happens *both* cooperating agents advance by one action (each moves past its guard). Here is one valid sequence of moves by SYS_2:

	(	Sem	|	U$_1$	|	U$_2$	)	\{gT,pT}	state
$\equiv$	(	$\overline{gT}.pT.Sem$	|	$n_1.gT.c_1.\overline{pT}\ldots$	|	$n_2.gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	1
$\xrightarrow{n_1}$	(	$\overline{gT}.pT.Sem$	|	$gT.c_1.\overline{pT}\ldots$	|	$n_2.gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	2
$\xrightarrow{\tau}$	(	$pT.Sem$	|	$c_1.\overline{pT}\ldots$	|	$n_2.gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	5
$\xrightarrow{n_2}$	(	$pT.Sem$	|	$c_1.\overline{pT}\ldots$	|	$gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	7
$\xrightarrow{c_1}$	(	$pT.Sem$	|	$\overline{pT}\ldots$	|	$gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	8
$\xrightarrow{\tau}$	(	Sem	|	$\ldots$	|	$gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	3
$\equiv$	(	$\overline{gT}.pT.Sem$	|	$\ldots$	|	$gT.c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	3
$\xrightarrow{\tau}$	(	$pT.Sem$	|	$\ldots$	|	$c_2.\overline{pT}\ldots$	)	$\backslash\{gT, pT\}$	11

Notice that CCS does NOT broadcast: an agent offering an internal action say $\overline{gT}$ can handshake with only one other agent offering a gT. If several are on offer, a choice will be made.

It is possible to describe this system state by state using only . and + as follows, with handshakes represented by τ's:

$$\begin{array}{lcrcr} S_1 & \overset{def}{=} & n_1.S_2 & + & n_2.S_3 \\ S_2 & \overset{def}{=} & \tau.S_5 & + & n_2.S_4 \\ S_3 & \overset{def}{=} & \tau.S_{11} & + & n_1.S_4 \\ S_4 & \overset{def}{=} & \tau.S_7 & + & \tau.S_9 \\ & \ldots & & & \end{array}$$

Even with two users, the state space is complicated. For n users, it explodes as $(n+1) * 2^n$ so that attempting state-by-state expansions is not the right approach. The system description is much clearer and simpler if we describe it in terms of its constituent objects (individual users and a semaphore) and then compose them using | and hiding. In particular, this style of system specification is easy to extend to several users.

Spelling out the system state by state using only . and + is akin to how we would program it in a language without objects. The simplicity of the specification arrived at by the composition of a set of interacting objects is a vivid demonstration of the power of the object-oriented approach when system components interact strongly.

Summary of how CCS systems may evolve. If E_1, E_2, ..., E_n are agents, then so is

$$\mathrm{E} = (E_1 \mid E_2 \mid ... \mid E_n) \setminus L$$

ﬃwhere L is a set of actions.

$$\mathrm{E} = (E_1 \mid E_2 \mid ... \mid E_n) \setminus L$$

At any given stage of its behaviour, E may evolve in one of two ways:

1. τ and $\alpha, \overline{\alpha} \notin L$: by a single visible interaction with the operating environment. Either $\alpha.E_j$ receives an α from the environment and evolves into E_j, or $\overline{\alpha}.E_k$ has $\overline{\alpha}$ accepted by the environment and evolves into E_k.

2. $\alpha, \overline{\alpha} \in L$: by an invisible, internal handshake (called τ) between two agents with one agent performing an *output* action on a hidden line and the other performing a complementary *input* action on the same hidden line simultaneously. That is if $\alpha \in L$, then $\alpha.E_j \mid \overline{\alpha}.E_k$ may evolve to $E_j \mid E_k$ in one step by performing τ.

τ represents a completed action in that both the complementary input and output actions have been performed.

Thus a composed agent evolves in stepwise fashion and its possible behaviours interleave the individual behaviours of E_1, E_2, ..., E_n in all possible ways except that the actions in L must be performed pairwise.

Renaming. If we require several instances of the same template in a model we are allowed to instantiate the latter with appropriate names. E.g we require two semaphores A and B with operations $\overline{gA}$ and pA, and $\overline{gB}$ and pB respectively, we write

$$\begin{array}{lcl} Sem & \overset{def}{=} & \overline{gR}.pR.Sem \\ A & \overset{def}{=} & Sem[gA/pR, pA/pR] \\ B & \overset{def}{=} & Sem[gB/pR, pB/pR] \end{array}$$

Read gA/gR as "substitute gA for gR", etc. Renaming makes it clear that agents have the same structure, but with varying access behaviours.

3 Expressing simulation synchronisations in CCS

In this section we show how to model in CCS the four most common synchronisation mechanisms, namely mutual exclusion, producer consumer, the rendezvous, and waitsuntil.

Mutual exclusion. We have already seen the simplest sort of mutual exclusion mechanism, the semaphore. GPSS and Demos have more general devices with limits greater than 1 which can be seized and released in chunks instead of just units.

Suppose a file is used to record the current status of elements in a dynamic system. 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. The trick is simply to define a resource of size r where r is the maximum number of readers in the system (or greater) and to let readers acquire and drop the resource in units, and the writers to seize and drop all r shares at once. Here is the Demos code:

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

entity class reader;
begin
  file.acquire(1);
    hold(read.sample);
  file.release(1);
  hold(use.sample);
  repeat;
end***reader**;

entity class writer;
begin
  hold(acq.sample);
  file.acquire(r);
    hold(write.sample);
  file.release(r);
  repeat;
end***writer**;
```

We model this variant of a resource of size n in CCS with $nBUFF$ (maximum size n). An $nBUFF$ can be acquired or released in unit chunks via gb and $\overline{pb}$ nad acquired and released *in toto* by calls gB and $\overline{pB}$.

$$
\begin{array}{lclclcl}
nBUFF_n & \overset{def}{=} & \overline{gb}.nBUFF_{n-1} & + & & + & \overline{gB}.nBUFF_0 \\
nBUFF_{n-1} & \overset{def}{=} & \overline{gb}.nBUFF_{n-2} & + & pb.nBUFF_n & & \\
 & \cdots & & & & & \\
nBUFF_k & \overset{def}{=} & \overline{gb}.nBUFF_{k-1} & + & pb.nBUFF_{k+1} & & \\
 & \cdots & & & & & \\
nBUFF_1 & \overset{def}{=} & \overline{gb}.nBUFF_0 & + & pb.nBUFF_2 & & \\
nBUFF_0 & \overset{def}{=} & & + & pb.nBUFF_1 & + & pB.nBUFF_n
\end{array}
$$

Reader and *Writer* definitions are now easy to write.

$$
\begin{array}{lclllll}
Reader_j & \overset{def}{=} & record_j. & gb. & use_j. & \overline{pb}.Reader_j \\
Writer_k & \overset{def}{=} & acq_k. & gB. & write_k. & \overline{pB}.Writer_k
\end{array}
$$

Here is the definition of a system with m readers and n writers:

$$
\begin{array}{lcl}
SYS_{m,n} & \overset{def}{=} & (\prod_{j=1}^{m} Reader_j \mid \prod_{k=1}^{n} Writer_k \mid mBUFF_m) \\
 & & \backslash\{pb, gb, pB, pB\}
\end{array}
$$

Note that *hold* statements (*ADVANCE* in GPSS) need not be modelled explicitly as we could consider them as being absorbed into the next action (synchronisation). However they are useful as "markers" when we test specifications for their properties (see further in section 4).

Producer/consumer. A simple manifestation of this second common synchronisation occurs when we have two cooperating entities, the first of which produces items for the second one to consume. Typical Demos code for a producer and a consumer are:

```
bucket :- new bin("bucket", 0);

entity class producer;
begin
  hold(prod.sample);
  bucket.give(1);
  repeat;
end***producer**;
```

```
entity class consumer;
begin
  bucket.take(1);
  hold(cons.sample);
  repeat;
end***consumer**;
```

The point is that 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.

The CCS code for a *bin* is trival. We again use a buffer but this time let one agent increment it and the other decrement it. The maximum size of the buffer tells us how many items the *Producer* can be in front of the *Consumer*. $nBUFF_0 \setminus \{pB, pB\}$ hides away the gB and pB operations (renders them invisible to producers and consumers).

$$\begin{array}{lcl}
Producer & \stackrel{def}{=} & prod.\overline{pb}.Producer \\
Consumer & \stackrel{def}{=} & gb.cons.Consumer \\
nBUFF'_0 & \stackrel{def}{=} & nBUFF_0 \setminus \{pB, pB\} \\
SYS & \stackrel{def}{=} & (Producer \mid Consumer \mid nBUFF'_0) \setminus \{pb, gb\}
\end{array}$$

The rendezvous. The rendezvous is used in Demos when a number of objects join together in a common task. Instead of having all of them travelling in unison down the event list, it is easier to arrange for one to be a master which "gathers" up the rest as slaves, carries out the task, and then releases the slaves allowing them to carry on as independent processes. Here is the Demos description of a crane loading a lorry

```
RDV :- new waitq("C + L");

entity class crane;
begin
  hold(other.sample);
  L :- RDV.coopt;
    hold(load.sample);
  L.schedule(0.0);
  repeat;
end***crane***;

entity class lorry;
begin
  hold(trip.sample);
  RDV.wait;

  repeat;
end***lorry***;
```

The CCS model is quite natural: the "master" sends out one signal to collect the slave ($\overline{srdv}$ — start rendezvous) and another ($\overline{erdv}$) to free it.

$$\begin{array}{lclllll}
Crane & \overset{def}{=} & other. & \overline{srdv}. & load. & \overline{erdv}. & Crane \\
Lorry & \overset{def}{=} & trip. & srdv. & & erdv. & Lorry \\
SYS & \overset{def}{=} & \multicolumn{5}{l}{(Crane \mid Lorry) \setminus \{srdv, erdv\}}
\end{array}$$

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 R_1; acquire R_2; ... acquire R_n;
  hold(activity duration);
release R'_1; release R'_2; ... release R'_m;

where the extra resources required (R_1, R_2, ..., R_n), be they modelled as *res*, *bin* or *entity* objects, have been requested and acquired one at a time. However this may not always happen. Suppose 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 there are several entities E_k which use one or more of these resources to carry out a task. Specifically let E_1 require both R_1 and R_2 to start a task. The coding

```
entity class E;
begin
  R1.acquire(1);
  R2.acquire(1);
    hold(task.sample);
    ........
end***E***;
```

is manifestly undesirable since E_1 may seize R_1 and wait a long time before R_2 is available. 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*. Informally, code for E_1 takes the shape:

```
Q :- new condq("Q");

entity class E;
begin
  Q.waituntil(R1.avail >= 1 and R2.avail >= 1);
  R1.acquire(1);
  R2.acquire(1);
   hold(task.sample)
   ........
```

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.

Modelling this behaviour in CCS is not easy. Essentially we require an enhanced resource:

$$\begin{array}{lcl} RES & \stackrel{def}{=} & \overline{gR}.RES' \; + \; \overline{fR}.RES \\ RES' & \stackrel{def}{=} & pR.RES \; + \; \overline{nfR}.RES' \end{array}$$

which can be probed to see if it is free (fR) or not free (nfR). Accesses to the three such resources, A, B, C must be enclosed in a semphore which makes sure that we can test and seize them all when all are free, and escape and try again when they are not.

$$
\begin{array}{lll}
Sem & \stackrel{def}{=} & \overline{gS}.pS.Sem \\
\\
A & \stackrel{def}{=} & RES[gA/gR, pA/pR, fA/fR, nfA/nfR] \\
B & \stackrel{def}{=} & RES[gB/gR, pB/pR, fB/fR, nfB/nfR] \\
C & \stackrel{def}{=} & RES[gC/gR, pC/pR, fC/fR, nfC/nfR] \\
\\
E & \stackrel{def}{=} & n.gS.E_0 \\
E_0 & \stackrel{def}{=} & fA.E_a \quad + \quad nfA.\overline{pS}.E \\
E_a & \stackrel{def}{=} & fB.E_{ab} \quad + \quad nfB.\overline{pS}.E \\
E_{ab} & \stackrel{def}{=} & fC.E_{abc} \quad + \quad nfC.\overline{pS}.E \\
E_{abc} & \stackrel{def}{=} & gA.gB.gC.\overline{pS}.cs.gS.\overline{pA}.\overline{pB}.\overline{pC}.\overline{pS}.. \\
\\
SYS & \stackrel{def}{=} & (Sem \mid A \mid B \mid C \mid E \mid ...) \setminus \{gS, pS, ..., nfC\}
\end{array}
$$

Structure of Simula programs. Simula programs typically contain the main program, say $MAIN$, and a number of objects, say P_1, P_2, ..., P_n. Their structure usually falls into one of two categories:

1. distributed control: $MAIN$ establishes the objects P_k and then control passes from one object to another object. The system terminates when control returns to $MAIN$. In this style, the active object decides for itself where to pass control. $MAIN$'s role is restricted to system initialisation and shut down.

   The action possibilities are captured in CCS by:

$$
\begin{array}{lll}
MAIN & \stackrel{def}{=} & \overline{s_1}.r_1.\overline{s_2}.r_2....\overline{s_n}.r_n.MAIN' \\
MAIN' & \stackrel{def}{=} & \overline{go}.go.report.0 \\
\\
P_k & \stackrel{def}{=} & s_k.init_k.\overline{r_k}.P'_k \\
P'_k & \stackrel{def}{=} & go.action_k.\overline{go}.P'_k \\
\\
SYS & \stackrel{def}{=} & (MAIN \mid \prod_k P_k) \setminus \{s_j, r_j, go\}
\end{array}
$$

   Here, $MAIN$ fires up $P_1, P_2, \ldots, P_n$ in turn. Upon receiving its firing signal s_k, process P_k carries out an initialising sequence $init_k$ and then, by emitting signal r_k, returns control back to $MAIN$. When $MAIN$

then fires a $\overline{go}$ signal, it may be accepted by any of the P_k. When its next action sequence is completed, it to emits a $\overline{go}$ signal which may be accepted by any other P_k (in which case more actions will be carried out) or indeed by $MAIN$ (in which case the simulation will be closed with a final report. this definition does not preclude the model running forever!

2. centralised control: as before, $MAIN$ establishes the objects P_k. Then $MAIN$ decides which single object to fire. When that object has finished its task, it returns control to $MAIN$. In this style, it is always $MAIN$ that decides which object may execute next.

   The action possiblities are captured in CCS by:

$$\begin{array}{lll}
MAIN & \stackrel{def}{=} & \overline{s_1}.r_1.\overline{s_2}.r_2.\ldots.\overline{s_n}.r_n.MAIN' \\
MAIN' & \stackrel{def}{=} & (\sum_{k=1}^{n} \overline{g_k}.d_k.MAIN' + report.0) \\
 & & \\
P_k & \stackrel{def}{=} & s_k.init_k.\overline{r_k}.P'_k \\
P'_k & \stackrel{def}{=} & g_k.action_k.\overline{d_k}.P'_k \\
 & & \\
SYS & \stackrel{def}{=} & (MAIN|\prod_k P_k) \setminus \{s_j, r_j, go\}
\end{array}$$

Here, $MAIN$ initialises the system as above. Thereafter, when a process P_k is fired, it always returns control back to $MAIN$ This time, $MAIN$ decides which process to fire next.

Process-oriented simulation languages such as Simula and Demos use style 2. When a process finishes its current task, the first object in the event list is the next one to go.

4 Process logics

We can never know that a specification is correct, but we gain confidence in its appropriateness if its consequences fulfill our expectations. A good source for the sorts of test that are used in practice (e.g. deadlock, livelock, safety, liveness, fairness) is [10, pages 177-387]. [1] gives a tutorial introduction to process logics.

Associated with CCS are two logics, Hennessy-Milner logic and the modal μ-calculus which enable one to ask such questions of CCS specifications. Consider the simple system

$$\begin{aligned} \mathrm{S1} &\overset{def}{=} \mathrm{a.S2} \\ \mathrm{S2} &\overset{def}{=} \mathrm{a.S3} \\ \mathrm{S3} &\overset{def}{=} \mathrm{b.S3} \end{aligned}$$

Using Hennessy-Milner logic we can ask questions of the system states:

- "it is possible to make an *a* move" from S_1 and from S_2. These are expressed as $S_1 \models$ <a> T and $S_2 \models$ <a> T respectively.
- S_3 does not have this property. $S_1 \not\models$ <a> T.
- we may distinguish between S_1 and S_2 because from S_1 we may make one a move followed by another, but not from S_2. These are expressed as $S_1 \models$ <a> <a> T and $S_2 \not\models$ <a> <a> T respectively.

[] is the dual operator to <>. <a> requires at least one *a* move; [a] requires all *a* moves. Some useful extra notation: – stands for all actions; –k,l,m stands for all actions except k,l,m; $< a, b, c > S$ is short for <a> S $\vee$ <b> S $\vee$ <c> S; and $[a, b, c]S$ is short for [a] S $\wedge$ [b] S $\wedge$ [c] S. Here are some common uses of Hennessy-Milner logic:

E	$\models$	[a] F	E cannot do an a move
E	$\models$	<a> T	E can do an a move
E	$\models$	[–] F	E is deadlocked
E	$\models$	<–> T	E can make a move
E	$\models$	<–> T $\wedge$ [– a] F	E can only do an a

Hennessy-Milner logic is good for asking questions one or two moves ahead, but cannot cope with recursive definitions. By adding just one construct — fix point operators — to Hennessy-Milner logic, we get the modal μ-calculus. Unfortunately, modal μ formulae are hard to read. Fortunately it is a very expressive logic, and we express many well-known temporal operators in it. Amongst these are:

- BOX: $S \models BOX\ a$ is true if a holds in each state reachable from S. E.g. the test for whether S can deadlock is simply $S \models BOX$ <–> T (we ask of each state reachable from S "can you make a move?").
- POSS: $S \models POSS\ b$ is true if S or (at least) one state reachable from S satisfies b.
- EVENT: $S \models EVENT\ c$ is true if c holds for (at least) one state on each and every path from S.
- CAN: $S \models CAN\ d$ is true if d holds along at least one path from S
- LOOP: $S \models LOOP\ e$ is true if there is an unending path of e moves from S. E.g. $POSS(LOOP\ <\tau>T)$ is a test for livelock.
- MUST_DO: $S \models MUST\_DO\ f$ is true if the only move that S can make is an f move.
- NEC: $S \models NEC\ g\ h$ is true if however the system evolves, we cannot do an h until after a g
- CYCLE$_n$: $S \models CYCLE_n\ i_1 ... i_n$ is only true if, however the system evolves from S, $i_1 \prec i_2 ... \prec i_n \prec i_1 ...$ where $\prec$ reads "must come before". This is a useful test to check that agents maintain their integrity and perform actions in the expected sequence no matter what the rest of the system does.

$$\begin{array}{lcl} U_1 & \stackrel{def}{=} & n_1.gT.sc_1.ec_1.\overline{pT}.U_1 \\ U_2 & \stackrel{def}{=} & n_2.gT.sc_2.ec_2.\overline{pT}.U_2 \\ Sem & \stackrel{def}{=} & \overline{gT}.pT.Sem \\ & & \\ SYS & \stackrel{def}{=} & (U_1 \mid U_2 \mid Sem) \setminus \{gT, pT\} \end{array}$$

Using these operators on a slightly modified SYS_2 system

we can ask such questions as:

$$\begin{array}{lll} \text{i} & SYS & \models \; CYCLE_3\; n_1\; sc_1\; ec_1 \\ \text{ii} & SYS & \models \; BOX[sc_1](NEC\; ec_1\; sc_1 \;\&\; NEC\; ec_1\; sc_2) \end{array}$$

i.e. (i) U_1 maintains its expected behaviour cycle regardless of other activity in the system and (ii) after U_1 entered its critical section $BOX[sc_1]$ (read this as "from every state in which we can do an sc_1 action, do it and then") it must exit its critical section ec_1 before re-entering its critical section or before U_2 is permitted access to its critical section via sc_2. Splitting "holds" into *shold* (start hold) and *ehold* (end hold) is sometimes used to aid modal testing.

Plant simulation

We close by giving a larger example of the methodology in action. Consider an industrial plant where machines periodically break down and have to be repaired. We model such generic behaviour by:

$job \stackrel{def}{=}$ *working; broken;* `repeat`

The plant employs a number of repairers to fix broken machines; each repair requiring just one repairer. Each repairer carries his/her own basic tool kit. In addition there are two spanner sets shared amongst the repairers, denoted by $spanner_1$ and $spanner_2$.

We model four types of machine breakdown:

1. requires just a basic tool kit
2. requires a basic tool kit plus $spanner_1$

3. requires a basic tool kit plus $spanner_2$

4. requires a basic tool kit plus $spanner_1$ and $spanner_2$

Each repairer is capable of fixing any type of breakdown. Their generic behaviour is thus:

```
repairer  $\overset{def}{=}$  effect $repair_1$
          or  acquire $spanner_1$; effect $repair_2$; release $spanner_1$
          or  acquire $spanner_2$; effect $repair_3$; release $spanner_2$
          or  acquire $spanner_1$ and $spanner_2$;
              effect $repair_4$;
              release $spanner_1$ and $spanner_2$
              repeat
```

The obvious way of modelling this system in DEMOS is to use a *waitq* and let the machines be slaves and the repairers be masters (or *vice versa*). Here is a sketch of the Demos code:

```
ref(waitq) BDQ;
ref(res) spanner1, spanner2;

entity class job(BDtype); integer BDtype;
begin
  hold(useful[DBtype].sample);
  DBQ.wait;
  repeat;
end***entity***;

entity class repairer;
begin
  ref(job) J;

  J :- BDQ.coopt;
  if J.BDtype  = 1
    then
      hold(repair[1].sample)
    else
  if J.BDtype  = 2
    then
      begin
        s1.acquire(1); hold(repair[2].sample); s1.release(1);
      end
    else
  if J.BDtype  = 3
    then
      begin
        s2.acquire(1); hold(repair[3].sample); s2.release(1);
      end
    else
  if J.BDtype  = 4
    then
      begin
        (s1 and s2).acquire(1);
          hold(repair[3].sample);
        (s1 and s2).release(1);
      end
    else ERROR;
  repeat;
end***repairer***;
```

where $useful_k$ and $repair_k$ are appropriate distributions.

Before simulating for performance, we use CCS to analyse the temporal properties of this little system. The resources are modelled by simple semaphores:

$$\begin{aligned} SPANNER_1 &\overset{def}{=} \overline{gS1}.pS1.SPANNER_1 \\ SPANNER_2 &\overset{def}{=} \overline{gS2}.pS2.SPANNER_2 \end{aligned}$$

The *waitq* synchronisation is standard

$$\begin{aligned} J_1 &\overset{def}{=} \overline{gR1}.pR1.J_1 \\ J_2 &\overset{def}{=} \overline{gR2}.pR2.J_2 \\ J_3 &\overset{def}{=} \overline{gR3}.pR3.J_3 \\ J_4 &\overset{def}{=} \overline{gR4}.pR4.J_4 \end{aligned}$$

The final system component we need to define is the repairer:

$$\begin{array}{lll}
R_1 & \stackrel{def}{=} & gR1.st1.et1.\overline{pR1}.Rman \\
R_2 & \stackrel{def}{=} & gR2.gS1.st2.et2.\overline{pS1}.\overline{pR2}.Rman \\
R_3 & \stackrel{def}{=} & gR3.gS2.st3.et3.\overline{pS2}.\overline{pR3}.Rman \\
R_4 & \stackrel{def}{=} & gR4.\ \begin{array}{ll} (& gS1.gS2.st4.et4.\overline{pS1}.\overline{pS2}.\overline{pR4}.Rman \\ + & gS2.gS1.st4.et4.\overline{pS1}.\overline{pS2}.\overline{pR4}.Rman \\ ) & \end{array} \\
Rman & \stackrel{def}{=} & R_1 + R_2 + R_3 + R_4
\end{array}$$

the first three sub-behaviours (R_1, R_2, R_3) are obvious. In the final branch we model the notion of fixing a job of type 4 as permitting to pick up the extra spanners in either order. We add actions $st_k.et_k$ (start task of class k, end task of class k respectively) to facilitate modal testing.

We can build a mini-system two jobs of type 4 and two repairers who handle jobs of type 4 only by

$$\begin{array}{lll}
S_1 & \stackrel{def}{=} & (SPANNER1 \mid SPANNER2 \mid J4 \mid J4 \mid R4 \mid R4) \\
 & & \backslash\{gR4, pR4, gS1, pS1, gS2, pS2\}
\end{array}$$

Does this system have desirable properties? We first check for deadlock using the CWB (Concurrency Workbench [7, 8, 16]) a mechanized support tool for CCS. The CWB command is *fd* — find deadlock. The CWB uses t for τ.

```
fd S1
--- use4 t<gR4> t<gS1> use4 t<gR4> t<gS2>
   ---> ( pS1.SPANNER1 | pS2.SPANNER2
        | pR4.J4 | pR4.J4
        | gS1.st4.et4.'pS1.'pS2.'pR4.Rman
        | gS2.st4.et4.'pS1.'pS2.'pR4.Rman
        )\{gR4,gS1,gS2,pR4,pS1,pS2}

--- use4 t<gR4> t<gS2> use4 t<gR4> t<gS1>
   ---> ( pS1.SPANNER1 | pS2.SPANNER2
        | pR4.J4 | pR4.J4
        | gS2.st4.et4.'pS1.'pS2.'pR4.Rman
        | gS1.st4.et4.'pS1.'pS2.'pR4.Rman
        )\{gR4,gS1,gS2,pR4,pS1,pS2}
```

The output shows both ways in which deadlock can be achieved, both the steps to reach a certain configuration, and the deadlocked end state. Either way, one repairer has *spanner*$_1$ and is waiting for *spanner*$_2$ whilst the other one has *spanner*$_2$ and is waiting for *spanner*$_1$.

This we can circumvent by requiring resources to be prioritized and picked up in priority order. We try again with a modified version of $R4$ and hence *Rman*,

$$\begin{array}{lcl}
R_1 & \stackrel{def}{=} & gR1.st1.et1.\overline{pR1}.Rman \\
R_2 & \stackrel{def}{=} & gR2.gS1.st2.et2.\overline{pS1}.\overline{pR2}.Rman \\
R_3 & \stackrel{def}{=} & gR3.gS2.st3.et3.\overline{pS2}.\overline{pR3}.Rman \\
R_4 & \stackrel{def}{=} & gR4.gS1.gS2.st4.et4. \\
 & & \quad (\overline{pS1}.\overline{pS2} + \overline{pS2}.\overline{pS1}).\overline{pR4}.Rman
\end{array}$$

$$Rman \stackrel{def}{=} R_1 + R_2 + R_3 + R_4$$

this time modelling a system with two repairers and two jobs of each category.

$$\begin{array}{lcl}
S_2 \stackrel{def}{=} & (& SPANNER1 \mid SPANNER2 \\
 & \mid & J1 \mid J1 \mid J2 \mid J2 \mid J3 \mid J3 \mid J4 \mid J4 \\
 & \mid & Rman \mid Rman \\
 &) & \backslash\{gS_k, pS_k, gR_k, pR_k\}
\end{array}$$

We now use the CWB to assure ourselves certain desirable consequences obtain. Read the CWB command *cp* as "check proposition".

- the test for deadlock is built into the concurrency workbench:

  ```
  fd S2
  No such agents.
  ```

  It is also easy to express in the modal μ-calculus: in every state (BOX), it is possible to make a move; or equivalently, it is not possible to reach a state ($\sim POSS$) from which no move can be made

  ```
  cp S2
  BOX(<->T)
  true
  ```

```
cp S2
(~POSS([-]F))
false
```

- can the system livelock? — is it possible to reach a state (*POSS*) which permits a cycle of internal handshakes (*LOOP* τ)

```
cp S2
POSS(LOOP t)
false
```

- mutual exclusion holds on the spanners? always after entering a critical section$_2$ (literally —in every state that can make an *st*2 move, having made it) we must end critical section$_2$ (do an *et*2 move) before entering another critical section$_2$, or a critical section$_4$, and similarly. The CWB represents $\wedge$ by &.

```
cp S2
BOX[st2](NEC et2 st4 & NEC et2 st4)
true

cp S2
BOX[st3](NEC et3 st3 & NEC et3 st4)
true

cp S2
BOX[st4](NEC et4 st2 & NEC et4 st3 & NEC et4 st4)
true
```

- having entered a critical section, we must complete it, i.e. all paths (*EVENT*) lead to a state where all we can do is end it

```
cp S2
BOX[st2](EVENT (MUST_DO et2))
false
```

This is too strong — the other repairer may cycle on jobs of type 1 quite happily. We content ourselves with the weaker statement

```
cp S2
BOX[st2](EVENT <et2>T)
true
```

the possibility of ending the critical section remains open on all paths, i.e. the option is never taken away.

- the basic behaviour patterns of a repairer always open up after the curent task is completed — it is always possible to get to a state where one can start to fix a job of type 1, etc

```
cp S2
BOX ( POSS <st1>T
    & POSS <st2>T
    & POSS <st3>T
    & POSS <st4>T
    )
true
```

- unfair behaviour: it is possible *never* to repair a job of type 1, etc

```
cp S2
CAN <-st1,et1>T
true
```

5 Summary and conclusions

In this paper we have sought to explain the advantages of using well-known techniques from process algebra in simulation model development. Simply by expressing the model in a process algebra notation, means that we can test a system description for many properties that are very difficult and time consuming to find by simulation runs (for example, we know of hardware projects that have been delayed by over one elapsed year trying to locate and fix a deadlock). Adopting this methodology makes our models much more reliable.

For pedagogic reasons, we illustrated the approach using CCS, one of the simplest process algebras. There are many notions not covered by CCS: but there are many more advanced process algebras which can cope with priorities, hard timings and/or functionality (value passing) [9, 12, 13, 14, 15, 18, 19].

Finally, more work is required on the appropriateness of the match between the simulation model and CCS. For example, DEMOS uses priority queueing for processes blocked on resources. The CCS model mimics all possible variations of behaviour. Ipso facto, it encompasses the DEMOS behaviours. Our model is therefore safe, but too conservative. Thus there will be some questions to which CCS says no which will be valid in the DEMOS regime.

6 Acknowledgements

The authors are happy to acknowledge the strong influences on their work of: Keith Tocher (entities, methodology), Steve Mathewson (central model notations), Dahl, Myrhaug, Nygaard (Simula and objects), Milner, Stirling, and Moller (CCS and process logics); Kendall, Ord-Smith, Rokne (mobility).

The work of the first author has been supported by an Operating Grant bestowed by the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. Aldwinckle, R. Nagarajan, and G. Birtwistle. An introduction to Modal Logic and its Applications on the Concurrency Workbench. Technical Report, Computer Science Department, University of Calgary, 1991.

[2] J. C. M. Baeten. *Applications of Process Algebra.* Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, Cambridge, 1990.

[3] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, Cambridge, 1990.

[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, P.A.Luker, G.Lomow, and B.Unger. Process style packages for discrete event modelling: Experience from the transaction, activity, and event approaches. *Transactions on Simulation*, 2(1):25–56, 1985.

[7] R. Cleaveland, J. Parrow, and B.Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 24–37. Springer Verlag, 1990.

[8] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification fo concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[9] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Mass, 1988.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification.* Springer-Verlag, New York, 1992.

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

[12] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, 1991.

[13] 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.

[14] 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.

[15] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. Technical Report ECS-LFCS-89-104, Department of Computer Science, University of Edinburgh, Edinburgh, 1989.

[16] F. G. Moller. The Edinburgh Concurrency Workbench, Version 6.0. Tech Report, Computer Science Department, University of Edinburgh, 1991.

[17] 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.

[18] 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.

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

[20] 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.