Integrating Behavioural and Simulation Modelling

Rob Pooley
Department of Computer Science
University of Edinburgh

Abstract

Discrete event simulation has grown up as a practical technique for estimating the *quantitative* behaviour of systems, where direct measurement is undesirable or impractical. It is also used to understand the detailed *functional* behaviour of such systems. Its theory is largely that of experimental science, centering on statistical approaches to validating the measures generated by such models, rather than on the verification of their detailed behaviour. On the other hand, much work has been done on understanding and proving functional properties of systems, using techniques of formal specification and concurrency modelling. This paper presents an approach to understanding the correctness of the behaviour of discrete event simulation models, using a technique from the concurrency world, Milner's Calculus of Communicating Systems (CCS) and to deriving behavioural properties of such models without resorting to simulation.

It is shown that a common framework based on the process view of models can be constructed,As a basis for this framework, a hierarchical graphical modelling language (Extended Activity Diagrams) is developed. This language is shown to map onto both the major constructs of the DEMOS discrete event simulation language and their equivalent CCS models. A graphically driven tool based on such a framework is presented, which generates models to use both simulation to answer performance questions (what is the throughput under a certain load) and functional techniques to answer behavioural questions (will the system behave as expected under certain assumptions). An example of the application of this approach to a typical model is presented.

1 Introduction

In designing complex systems, simulation is often used to establish both quantitative (performance) and qualitative (behavioural) properties. Its use is, however, expensive and often yields only approximate results. For qualitative properties, Petri nets, process algebras and formal specification techniques are increasingly used. For quantitative properties analytical or numerical modelling, using queues or stochastic extensions to Petri nets, are often preferred, but simulation remains the only way to handle large models with complex interactions, because of the restricted classes of models suitable for exact solutions and the state space explosion when generating underlying Markov chains for numerical analysis.

Discrete event simulation tools are traditionally categorised as being based on one of a small number of views of a model. A number of modelling tools are based on or can support the process view of simulation as defined by Franta [8]. Several of these, as well as others based on other views, have diagram conventions for users to define their models and some support model construction via graphical interfaces based on such diagrams. Unfortunately, whereas Petri nets generated from graphical tools can be analysed for both functional and performance behaviour, the use of diagrams for simulation is usually specific to one simulation tool and offers no help in understanding the behaviour of models without actually simulating them. Since discrete event simulation is in effect a (pseudo-)random walk through the state space of the model, it is not possible to guarantee to visit all states without pre-analysis by other means.

This paper assesses the benefits of a formal understanding of process based discrete event simulation models. These are expressible in terms of diagrams suitable for direct graphical input on PCs or workstations. At the same time they are amenable to *a priori* functional analysis and have a well developed semantics. The vehicle for this is a mapping from a graphical language of models (known as Extended Activity Diagrams) both to a discrete event simulation language - an extended form of Birtwistle's DEMOS) [5] - and to a process algebra - Milner's Calculus of Communicating Systems (CCS)[13].

The rest of this paper is structured as follows. An overview of DEMOS is given in section 2, along with the symbols of the graphical notation of Extended Activity Diagrams. Section 3 contains a short description of the Calculus of Communicating Systems (CCS), its temporal extension TCCS [22, 14] and an associated process logic, the modal μ -calculus [21]. Section 4 presents a definition of the mechanisms of the DEMOS language in terms of CCS. Some problems with such definitions are identified and remedies discussed. Section 5 presents the tool which supports the ideas in this paper. Section 6 contains a case study which demonstrates the benefits and problems in combining pre-analysis of functional properties with simulation of dynamic, timed behaviour. Not all questions are found to be easily addressed, even with the use of the modal μ -calculus, but some clear benefits are claimed. Section 7 draws together the strands of the earlier sections and assesses the outcome. Open issues and areas for further research and development are identified.

2 DEMOS and Extended Activity Diagrams

Graphical model construction is not new, but it has never been formalised. Nor has a theory of behavioural equivalence for discrete event simulation models been fully developed. Previous work on *simulation graphs*[27] concluded that esatblishing behavoural equivalence of event based models was not even feasible. The approach of this paper is based on models represented by the process view, which proves to be more amenable to such an approach.

2.1 Process based discrete event simulation

The process based view takes as its starting point the idea that the world consists of active and passive components. The term was in common use for several years before the appearance of Franta's book [8], but he gives the first complete description of the approach, using SIMULA as the implementation language. Active components (processes) are described by their life histories, which often form cycles. They interact with the world, in competition or co-operation, through resources, which are passive. This division into two classes is acknowledged to be arbitrary and Franta gives examples where the same object may be seen as active or passive, according to the perspective of the modeller.

The main benefit claimed for the process based approach is that it expresses the model in terms of the structures observable in the real world and so makes modelling more intuitive and interpretation of results easier. It also can have significant implementation benefits.

2.2 SIMULA and DEMOS

Many languages and packages claim to be process oriented or to be capable of representing process oriented models. Rather like the term "object oriented", process oriented has become a victim of its own success in appealing to ease of understanding. There are, in fact, several suitable languages for this purpose, but this paper will refer mainly to the DEMOS package, which is an extension of SIMULA.

SIMULA [4, 16] is a general purpose programming language, defined as a superset of Algol 60. It was designed to support the efficient implementation of event and process based discrete event simulation. The notion of a process is supported by a combination of *inheritance* and *quasi-parallel sequencing* (co-routines or light weight processes) within the *class* concept. This provides an efficient implementation of conditional waiting, since objects suspended as co-routines can wait in heterogeneous lists and can resume themselves when events in the execution of the model allow them to proceed.

SIMULA supports layers of packages, each refining and extending earlier ones. In this way, the DEMOS package known is provided. This has a time ordered event list and a class `ENTITY`, which is the building block for active components in models, adding modelling related abstractions to the co-routine semantics of classes. In addition to these basic features DEMOS has automatic statistical collection and reporting and optional output of event traces. In this way, it allows a wide range of models to be solved to establish their dynamic behaviour, both in terms of quantitative performance (response time, queue lengths etc.) and event based behaviour traces. Although SIMULA does not support directly the concept of a general *wait-until*, Vaucher showed how this could be efficiently implemented in SIMULA by using the Algol name mode for procedure parameters [24, 25]. DEMOS offers such a mechanism, using a conditional queue class, `CONDQ`. A number of more specialised building blocks for the passive elements of a model are also provided, all of which report key statistics automatically. These include `RES`, for resources, `BIN`, for unbounded buffers, and `WAITQ`, for master/slave interactions.

2.3 Extended Activity Diagrams

The key to the tool frontend described in this paper is the definition for the first time of a complete, formally understood graphical notation for hierarchical process based models. For a full definition of this graphical language see [19]. This definition of *Extended Activity Diagrams* has allowed reasoning about the behavioural properties of simulation models, independently of their execution for performance evaluation.

Graphical description of a process class requires both a way of showing the flow of control through such a process and a way of representing interactions and synchronisations engaged in by instances of it. Construction of a model or sub-model defines the linkages between instances of processes, by mapping their required interactions onto instances of those objects which support such interactions. Many synchronisations among processes can be mapped onto queues, which is the only mechanism in queueing network based formalisms such as PAWS [11]. However, the use of higher level abstractions, such as resources in GPSS [20], adds to the ease of description and widens the range of mechanisms which can conveniently be represented. Activity diagrams were used informally by Birtwistle [5] to provide a convenient flow of control description, based on flow charts, and to allow easy description of a wide range of useful synchronisation mechanisms, based on activity cycle diagrams. This made them a good starting point for building a complete diagramming convention for process interaction, as described in [19].

A simple example of an atomic process description is shown in Figure 1. The model is the reader/writer model from [5] chapter 4. It includes Birtwistle's standard symbols of a rectangular box for a delay, annotated with a description of the associated activity, and a circle for a resource, annotated with a description of the resource and the initial amount available. New symbols are needed to complete even this simple example. Thus, Hughes [10] added the lower semi-circle, annotated with the process name, which marks the start of the process life cycle, and an inverted form of the start symbol, with no annotation, to mark the termination of the process. In the Simmer Process Interaction Tool [17] synchronisation nodes were also added, to show where resources are acquired and released. This last extension is a significant change from Birtwistle's convention of attaching synchronisations to hold boxes and allows the exact order of all such synchronisations to be specified.

Various forms of arrowed line might be used to represent the type of a link, but this is fully determined by the types of the nodes which it joins. Thus the lines joining *delay* to *delay*, *delay* to *start* or *delay* to *termination* nodes represent control flow in

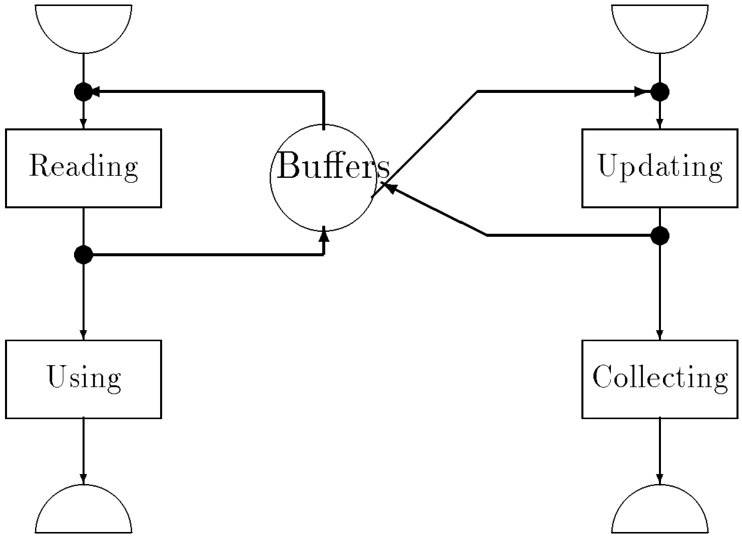

Figure 1: Mutual exclusion resource contention - the reader/writer model

the process, in the same manner as in conventional flow charts. On the other hand, the lines joining *resources* to *synchronisation* nodes represent *acquisition* or *release* of amounts of those resources. *Acquisition* and *release* constitute, respectively, a potential blocking of the flow of control in the process due to contention with other processes and a potential freeing of another process currently blocked by this process. The amount to be acquired or released is shown as an annotation to the link, while the direction of the arrow on the line determines which action is intended. All external interactions are shown by *synchronisation* nodes. In this sort of process type description the objects to which *synchronisation* nodes are linked are there purely to show the type of synchronisation by which any instance of this type will be linked to other process instances. This example uses the process types `Reader` and `Writer`, but their definitions are independent of each other, being linked solely through the intervening `Buffers` resource. In such simple cases the model can be completely described by suitable annotation of the process type description, with amounts of resources and inter-arrival times added in this case.

The presence of *loop-start/end* nodes removes the need for cycles in these graphs. A *decision* or *loop-start* node is associated with the next succeeding *branch/loop-end* node.

Communication through a passive object, such as a *resource* or a *condition queue*, is shown by an outgoing arrow to the passive object from a *synchronisation* node in the output process and an incoming arrow from the passive object to a *synchronisation* node in the input process.

3 CCS and the Modal $\mu-$calculus

The Calculus of Communicating Systems forms the core of the formal semantics for process based simulation developed below. It was created to model the behaviour of systems which can be described in terms of communicating agents. Consider first the basic calculus [13]. This contains the following primitives for defining agents, which will be used in later chapters:

sequential composition	$a.B$	after action a the agent becomes a B
parallel composition	$A \mid B$	agents A and B proceed in parallel
choice	$A + B$	the agent behaves as either A or B, but not both, depending on which acts first
restriction	$A\backslash M$	the set of labels M is hidden from outside agents
relabelling	$A[a_1/a_0, ..]$	in this agent label a_1 is renamed a_0
the null agent	0	this agent cannot act (deadlock)
the divergent agent	$\perp$	this agent can cycle indefinitely and unobservably

Here identifiers starting with lower case letters denote the labels of complementary action pairs, where the use of a label with an overbar, $\overline{c}$, or without one, c, distinguishes two halves (output and input) of an action, both of which must be possible before it can proceed. Identifiers which begin with an upper case letter define agents. Agents are constructed from the forms given above.

Symbolic names for agents are defined using the binding symbol, $\stackrel{def}{=}$. In the Concurrency Workbench this infix operator is replaced by the prefix operator `bi`. Thus the equations $A \stackrel{def}{=} b.C$ and `bi A b.C` are equivalent in the two formats.

CCS uses a notion of observation equivalence, which depends on the assumption that two agents are equivalent if any differences in their behaviours cannot be distinguished by an observer. Where two agents containing the two sides of a complementary action are combined in parallel, the resulting agent may hide the action and regard it as internal. CCS calls such internal actions τs. Under many circumstances such internal actions have no effect on the observable behaviour of agents and so may be ignored. This is not always so, however, notably when a τ is the prefixing action of one half of a choice.

By using the notion of bisimulation as its basis of equivalence, CCS is able to detect equivalence for a wider class of models than the use of isomorphism would permit. It is also inherently compositional, allowing bisimulation results proved for components to be preserved by its combinators and so reducing the effort of proving properties of larger models constructed in this way.

3.1 Temporal CCS

Temporal CCS [22, 14] is an extension to CCS, which allows both explicit delays and wait-for synchronisation (asynchronous waiting), in a manner superficially similar to DEMOS. It adds the primitives:

fixed time delay	(t)	where the agent requires t time units to elapse before it can perform its next action
wait for synchronisation	δ	where the agent may idle indefinitely until its next action is possible

non-temporal deadlock $\underline{0}$ where the agent may idle indefinitely and never engages in further actions

Deadlock now extends to cover situations where time cannot pass, since all parallel components must be ready to advance time for it to move on. Put another way, if there are components composed in parallel where some have as their current action an unsatisfied complementary action, and other agents have a time delay, the system is in temporal deadlock. Non-temporal deadlock allows indefinite idling, i.e. all processes are able to wait indefinitely for actions which cannot happen and so they cannot evolve.

The wait for delay is sometimes written by underlining the next action and sometimes by writing a δ preceding it. The latter form is used throughout this paper. In the Concurrency workbench it is written as a $ symbol preceding the next action.

3.2 Process logics

If process algebras represent a useful way of describing models, with a formally defined semantics, it is natural to use a corresponding process logic to frame properties and queries concerning these models. Although the Concurrency Workbench, for instance, allows simple properties, such as the presence of deadlock, to be queried directly, it needs a suitable logic to express more specific properties and questions. Formally such logics are known as modal logics and express assertions about changing state. Such logics are not confined to reasoning about CCS. They apply generally to labelled transition systems.

There is an appealingly simple modal logic, known as Hennesy-Milner logic [9], for expressing assertions about the immediate possibilities for a model. There is also an extended modal logic, with fixed point operators allowing the expression of recursive definitions, known as the modal $\mu-$calculus. Within the CWB, the modal $\mu-$calculus [21] is used for this purpose.

3.3 Hennesy-Milner logic

The description here follows the outline of Milner's presentation in [58].

Consider the simple system

$$S1 \stackrel{def}{=} a.S2$$
$$S2 \stackrel{def}{=} a.S3$$
$$S3 \stackrel{def}{=} b.S3$$

Using Hennesy-Milner logic it is possible to assert properties of a system's states, using the following operators:

satisfaction	$\models$	the agent on the left hand side of the operator satisfies the formula on its right hand side.
possibility	$\Diamond$	e.g. it is possible to make an a move both from S1 and from S2.

These are expressed respectively as: $S1 \models \Diamond\ a\ True$ and $S2 \models \Diamond\ a\ True$. The state $True$ implies unconditional satisfaction. It is shorthand for the empty conjunction, $\bigwedge_{i \notin \emptyset} \mathcal{F}_i$.

non-satisfaction $\not\models$ e.g. $S3$ cannot make an a move,
i.e. $S3 \not\models \Diamond a\ True$ which means $S3 \models \neg\Diamond a\ True$

It is possible to distinguish between $S1$ and $S2$ if from $S1$ if it is possible to make one a move followed by another, but not to do this from $S2$. This is expressed as:
$S1 \models \Diamond a \Diamond a\ True$ and $S2 \not\models \Diamond a \Diamond a\ True$

necessity $\Box a$ the dual operator to $\Diamond a$.
If $S1 \models \Box a\ \mathcal{F}$ then by performing the move a, $S1$ must always reach a state where $\mathcal{F}$ holds.

$\Diamond a\mathcal{F}$ requires at least one of its currently possible a moves to reach the state $\mathcal{F}$; $\Box a\mathcal{F}$ requires all of its currently possible a moves to reachthe state $\mathcal{F}$.

There are also weak forms of the possibility and necessity operators, which disregard any τs.

3.4 The modal $\mu-$calculus

Hennesy-Milner logic is good for asking questions one or two moves ahead, but cannot cope with recursive definitions. By adding just one construct - fixed point operators - to Hennesy-Milner logic, the result is the *modal $\mu-$calculus*. This is in effect a powerful temporal logic, allowing one to express notions of eventuality and invariance of states and actions. Although the modal $\mu-$calculus is much more general than a process logic, the discussion here is restricted to its use with CCS.

More complete, fairly readable accounts of the modal m-calculus can be found in Stirling [21] and Aldwinckle, Nagarajan and Birtwistle [1].

A fixed point equation might have the form:

$$Y \stackrel{def}{=} a\Diamond bY$$

meaning that each state in Y has the property of being able to perform an a action followed by a b action and then reaching a state in the original set, Y. Once we have allowed such recursive definitions we can examine the properties of fixed point equations and find sets of states which satisfy them, within agents. Not all such equations have solutions, nor are their solutions guaranteed to be unique. However, a restriction that there must be an even number of negations prefixing a recursively defined variable in an equation guarantees that there must be at least one solution. Formally, this property defines that the equation is monotonic.

It is worth noting that a property with respect to a model defines the set of states where that property holds, i.e. the property and the set of states are different ways of expressing the same thing.

There are two very important fixed point operators, defining the maximum and minimum fixed points of a recursive equation. The maximum fixed point is related to the fact that the union of any two solutions to a fixed point equation is a subset of a further solution. This superset is the closure under deduction of the union of the two initial sets. The maximum fixed point of an equation is the closure under deduction of the union of all fixed points of that equation, i.e. it contains every state which can form part of a solution. The minimum fixed point is related to the fact that the intersection of any pair of solutions contains a solution. Thus the minimum fixed point of an equation is the smallest solution to that equation and is a subset of the intersection of all fixed points. It contains only those states guaranteed to be in every solution. It is often the empty set.

Whilst it is not always obvious how to interpret fixed point modal formulae, the general idea is that a maximum fixed point expresses some property which always holds (an invariant), while a minimum fixed point expresses a property which will eventually hold. When verifying systems maximum fixed points are useful for expressing safety properties and minimum fixed points for expressing liveness properties.

Some examples yield to intuition. For example, following [1]:

$$Y \stackrel{def}{=} < x > T \vee [-]Y$$

has a minimum fixed point which can be read as saying that it is possible to perform an x action or all actions lead to a situation where it is eventually possible to do so. The maximum fixed point of the same equation denotes the set of all states.

3.5 Concurrency workbench

The Concurrency Workbench (CWB) [7, 15] is a tool that automates the checking of assertions about CCS models in order to establish properties of the systems they describe. It supports the basic calculus, the temporal extension to this and a synchronous variant of the basic calculus. The CWB allows testing of expressions in the modal $\mu-$calculus.

In the method developed in this paper, the CWB is used for the behavioural analysis of CCS models generated automatically from Extended Activity Diagrams, while DEMOS is used to solve them for their performance measures.

4 DEMOS in CCS

Representations of processes map directly onto Entity declarations in DEMOS and agent definitions in CCS. By using parallel composition of agents in CCS, it is possible to instantiate co-operating and competing processes within a model in the same way as use of new statements in DEMOS. Interactions must be modelled in CCS by complementary actions, shared by the active or passive objects involved in the interaction, while in DEMOS they are calls to procedures (methods) which are attributes of those objects. In CCS internal actions are either disregarded (in un-timed models) or represented by delays matching DEMOS hold statements (in timed versions). Simple DEMOS sequences of actions are matched by the normal CCS prefixing of an agent with an action or a time delay. Termination, shown in DEMOS by the end of an `Entity`'s body, is indicated in

CCS by the non-temporal deadlock agent, $\underline{0}$, which performs no further actions but does not stop time passing. Figure 2 shows a simple example.

```
Entity class Reader;
begin
   Buffers.Acquire(1);
   Hold(3);
   Buffers.Release(1);
end;
```

$$Reader \stackrel{def}{=} \overline{buffersAcq_1}(3)\overline{buffersRel_1}.0$$

Figure 2: A DEMOS sequential Entity and a corresponding TCCS agent

Loops are represented by recursive agent definitions. Figure 3 shows a simple example of this.

```
Entity class Reader;
begin
   while True do
   begin
      Buffers.Acquire(1);
      Hold(3);
      Buffers.Release(1);
   end;
end;
```

$$Reader \stackrel{def}{=} \overline{buffersAcq_1}(3)\overline{buffersRel_1}.Reader$$

Figure 3: A DEMOS repeating Entity and a corresponding TCCS agent

One obvious correspondence that holds in all the following mechanisms is that synchronisations which can block are formed by a communication, preceded by the indefinite wait (δ) in TCCS. Figure 4 shows this in terms of elements of the example used in the case study in Section 6 of this paper.

Note that in the temporal calculus it is necessary to decide whether an action is allowed to block indefinitely or to have the effect of killing the process if it cannot be satisfied immediately. All acquire actions by processes can lead to a process being blocked, awaiting freeing of a resource and so such actions are prefixed with the indefinite waiting action δ. On the other hand, releases should only be permitted in cases where there has already been a matching acquire, leaving the matching resource always ready to accept it. Therefore releases are not prefixed with δ.

```
entity class Reader_C;
begin
   while True do  begin
      Buffers.Acquire(1);    ! grab the buffer;
      Hold(ReadTime);
      Buffers.Release(2);    ! let the buffer go;
      Hold(ThinkTime);
   end;
end-of-Reader;
entity class Writer_C;
begin
   while True do  begin
      Buffers.Acquire(3);   ! grab all the buffers;
      Hold(UpdateTime);
      Buffers.Release(3);  ! let the buffer go;
      Hold(GatherTime);
   end;
end-of-Writer;
ref(Res) Buffers;
Reader1 :- new Reader_c("Reader"); Reader2 :- new Reader_c("Reader");
Writer1 :- new Reader_c("Reader");
Buffers :- new Res("Buffers", 3);
```

$$
\begin{aligned}
Reader &\stackrel{def}{=} \delta.\overline{bAcq_1}.(T_{Read})\delta.\overline{bRel_1}(T_{Think})Reader \\
Writer &\stackrel{def}{=} \delta.\overline{bAcq_3}.(T_{Update})\delta.\overline{bRel_3}(T_{Think})Writer \\
Buffers_3 &\stackrel{def}{=} \delta.((bAcq_1.Buffers_2) + (bAcq_3.Buffers_0)) \\
Buffers_2 &\stackrel{def}{=} \delta.((bAcq_1.Buffers_1) + (bRel_1.Buffers_3)) \\
Buffers_1 &\stackrel{def}{=} \delta.(bRel_1.Buffers_2) \\
Buffers_0 &\stackrel{def}{=} \delta.(bRel_3.Buffers_3) \\
Model &\stackrel{def}{=} (Buffers_3|Reader|Reader|Writer)\backslash\{bAcq_1, bAcq_3, bRel_1, bRel_3\}
\end{aligned}
$$

Figure 4: Demos Res object used by Entities and corresponding TCCS

Resources must be able to wait indefinitely in all states and so all their actions are prefixed with δ. Thus Figure 5 defines a general model of a resource in TCCS. In the basic calculus, where all actions are instantaneous, no δs are needed.

$$
\begin{aligned}
Res0 &\stackrel{def}{=} \sum_{i=1}^{Limit} \delta.resRelease_i.Res_i \\
Resn &\stackrel{def}{=} \sum_{i=1}^{Limit-n} \delta.resRelease_i.Res_{n+i} + \sum_{i=1}^{n} \delta.resAcquire_i.Res_{n-i} \\
Res_{Limit} &\stackrel{def}{=} \sum_{i=1}^{Limit} \delta.resAcquire_i.Res_{Limit-i}
\end{aligned}
$$

Figure 5: General definition of a DEMOS Res in TCCS

5 A Tool for Model Generation

5.1 Related tools

Several tools have appeared which combine simulation and exact quantitative solvers using a common input format [26, 2, 3]. A series of tools, beginning with the SIMMER Process Interaction Tool [17], have shown the potential for generating DEMOS models from graphical input. The translation of a subset of unmodified DEMOS syntax into CWB code for either CCS or SCCS was implemented by Tofts[23], permitting the conversion of DEMOS programs process algebra and the use of the Concurrency Workbench to prove properties of the systems. GreatSPN [6] and DSPNExpress [12], graphically based stochastic Petri net tools, allow both simulation and structural analysis of their underlying place transition net models.

In this paper, a new tool called Demographer allows both modified DEMOS discrete event simulation models and CCS process algebra models to be generated from a common graphical description. The former can be solved by the DEMOS discrete event solver, while the latter can be analysed by the Concurrency Workbench.

5.2 Demographer

Demographer is a simple graphical editor for creating both modified DEMOS discrete event simulation models and Calculus of Communicating Systems (CCS) [13] models directly from extended activity diagrams as described above. The current version runs under MS/DOS and is written entirely in SIMULA. An earlier version, using a less complete definition of extended activity diagrams exists for X Windows under UNIX [18].

Compilation and execution of modified DEMOS models is currently done separately, but it is intended that they should be integrated into the graphical front end.

CCS is generated in the syntax of the Concurrency Workbench for most constructs of Extended Activity Diagrams. Both the basic calculus and its temporal extension can be generated. The Concurrency Workbench remains a separate tool, but it is trivial to load the output of Demographer into it. By integrating the two types of modelling in a pair of compatible tools, the benefits of both approaches are more easily obtained. At the same time the process of modelling is simplified and consistency between the two solvable forms of the model is ensured.

5.3 The basic tool

Demographer allows the user to draw enhanced activity diagrams, by selecting symbols from a menu and placing them on a canvas, which is divided into a grid of squares. Each symbol occupies one square in the grid. Symbols are connected by drawing linking symbols in the squares between them. The types of the symbols joined and the direction of the links determine their meaning, in line with the formal grammar for extended activity diagrams developed in [19].

Many symbols require additional information to be supplied to complete the description of the model. For instance, the Hold symbol requires a description of the duration of the delay it represents. Additionally many symbols can usefully be annotated by a short comment or description. This is possible by selecting a symbol and invoking an *open form* operation. This will cause an *input form* menu appropriate to that symbol to be displayed. The user may then enter the required information by typing into this form.

When a model's description is believed to be correct and complete the user may request that a DEMOS program be generated from it. This is done by activating the Generate button. The user will then be asked for the name of a file into which the DEMOS source is to be written.

6 A Simple Case Study

The reader/writer model used in the earlier examples is typical of a resource used to enforce mutual exclusion. Under appropriate timings this model can produce starvation. Figure 4 showed the mapping into TCCS for that model.

Consider the Reader process. This is a simple cyclical process, defined in CCS by a right recursion. It requires only one buffer to proceed. The Writer process is structurally similar, but needs to acquire all the buffers before it can update them. This simple mutual exclusion example is interesting since it may induce starvation of the *Writer* by the *Reader* processes if the timings of the *Reader*s are unfavourable. The resource is modelled as usual and is simplified as before. Finally the model is a parallel composition of all processes

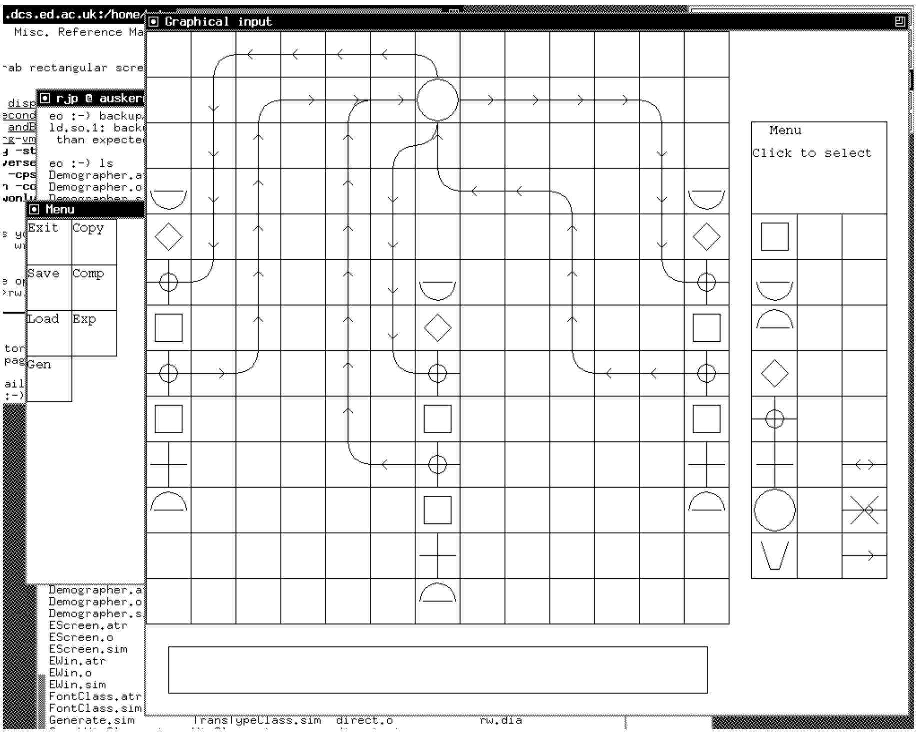

Figure 6: Demographer user interface

Since there are only two *Reader* processes and only in them can a *buffAcq1* take place, and the only way to reach a *Buffs0* state is following a *buffAcq3*, the only possible action of a *Buffs0* agent is a *buffRel3*. Thus the graph of *Model* has two sub-graphs, which are only joined by the start state.

The problem of starvation may be summarised as the situation where, although it is theoretically possible to reach an agent (or sub-graph of the transition graph) within a model, under certain timing and priority or resource conditions, created when the other has proceeded, this cannot happen. Unlike the more general notion of unfairness, without timing information the best that can be said is that the possibility does or does not exist, i.e. that there is a choice from which two or more disjoint sub-agents start and at least one of them contains a cycle which can prevent return to the choice.

In the model being considered this is clearly the start agent, *Model*. The two sub-agents *Reader* and *Writer* both cycle back to this choice, but *Reader* may remain within an internal cycle of activity. This is not strictly the same as livelock, since progress may be made by the overall system, even though part of it is starved. Working without timings the reachability graph of Figure 7 is produced.

It is the secondary cycle between the two reader processes that prevents the writer from engaging in any activity. If timings are added which force the model into bad behaviour, the temporal version of CCS can be used to show this, as in Figure 8. The timings in the Writer agent are unimportant, as it will never be allowed to start as long as both Readers do not release their buffers simultaneously. The *Reader* agent is extended

```
                        (Reader | Reader | Writer | Buffs3)
                          ^     |                 |       ^
                   buffRel1  buffAcq1        buffAcq3  buffRel3
                          |     V                 V       |
(Thinker | Reader | Writer | Buffs2)     (Reader |Reader | Updater | Buffs0)
       ^                   |
   buffRel1           buffAcq1
       |                   V
(Thinker1 | Thinker1 | Writer | Buffs1)
```

Figure 7: Reader/Writer reachability graph without timings

into a series of sub-agents corresponding to time advancing. The overall model uses time prefixes to schedule the various *Reader*s and *Writer*s out of time with each other. The transition graph is now as shown in Figure 9.

$$
\begin{aligned}
Reader0 &\stackrel{def}{=} \delta.buffAcq1.Thinker0 \\
\\
Thinker0 &\stackrel{def}{=} (3)Thinker1 \\
Thinker1 &\stackrel{def}{=} buffRel1.Reader1 \\
Reader1 &\stackrel{def}{=} (1)Reader0 \\
\\
Model &\stackrel{def}{=} (Reader0|(2)Reader0|(1)Writer|Buffers3)\mathrm{L}(Model)
\end{aligned}
$$

Figure 8: Reader/Writer TCCS with timings forcing starvation

The last state is identical, when re-ordered, to an earlier state and so the model will cycle indefinitely without Writer ever acting. Expressing starvation The property that starvation may be possible can be given in English as follows.

Given a choice state, generated by applying the expansion theorem to the parallel composition of two agents, there is, from that state of the model, a path which may revisit that choice, but need not do so. If timing information or priorities are added, it is possible to show cases where such a system will definitely behave badly.It would be comparatively simple to phrase a question in the modal μ-calculus of the form, "Is it possible for the model to reach a state (or perform an action) in the Writer cycle once it has reached (performed) one in the Reader inner cycle" One such question is written in the Workbench syntax as:

```
bi X (Thinker1| Thinker1| Writer| Buffers_1)\{buffAcq_1,buffAcq_3,buffRel_1,buffRel_3}
cp X min(X.<buffAcq_3>T | <->X)
```

```
(Reader0 | (2)Reader0 | (1)Writer | Buffers3)
                | buffAcq1
                V
(Thinker0 | (2)Reader0 | (1)Writer | Buffers2)
                | (2)
                V
((1)Thinker1 | Reader0 | Writer | Buffers2)
                | buffAcq1
                V
((1)Thinker1 | Thinker0 | Writer | Buffers1)
                | (1)
                V
(Thinker1 | (2)Thinker1 | Writer | Buffers1)
                | buffRel1
                V
(Reader1 | (2)Thinker1 | Writer | Buffers2)
                | (1)
                V
(Reader0 | (1)Thinker1 | Writer | Buffers2)
                | buffAcq1
                V
(Thinker0 | (1)Thinker1 | Writer | Buffers1)
```

Figure 9: The Reader/Writer transition graph showing starvation

the command `cp` asks the CWB to heck the proposition that the agent X satisies the modal μ formula that follows. Thus, once the structure of the model is understood, an answer to the possibility of starvation can be expected. The CWB answers false to the command, indicating that once in the state specified, the action $buffAcq_3$ cannot ever be performed. It is perhaps not unreasonable to expect a modeller to be happy to do this kind of reasoning.

To check this, a series of runs of the simulation model were made. The output is shown in table 1.

R1	RT1	TT1	C1	R2	RT2	TT2	C2	W1	UT1	GT1	C3
0.0	3.0	3.0	29	2.0	1.0	1.0	57	1.0	3.0	3.0	28
0.0	3.0	3.0	33	2.0	1.0	3.0	34	1.0	2.0	2.0	33
0.0	3.0	3.0	33	2.0	1.0	3.0	34	1.0	2.0	3.0	33
0.0	3.0	3.0	29	2.0	1.0	3.0	29	1.0	3.0	2.0	28
0.0	3.0	3.0	29	2.0	1.0	3.0	29	1.0	3.0	3.0	28
0.0	3.0	1.0	50	2.0	3.0	1.0	49	1.0	2.0	2.0	0
0.0	3.0	1.0	50	2.0	3.0	1.0	49	1.0	2.0	3.0	0
0.0	3.0	1.0	50	2.0	3.0	1.0	49	1.0	3.0	2.0	0
0.0	3.0	1.0	50	2.0	3.0	1.0	49	1.0	3.0	3.0	0
0.0	3.0	1.0	44	2.0	3.0	3.0	23	1.0	2.0	2.0	21
0.0	3.0	1.0	44	2.0	3.0	3.0	23	1.0	2.0	3.0	21
0.0	3.0	1.0	39	2.0	3.0	3.0	21	1.0	3.0	2.0	19
0.0	3.0	1.0	39	2.0	3.0	3.0	21	1.0	3.0	3.0	19
0.0	3.0	3.0	23	2.0	3.0	1.0	43	1.0	2.0	2.0	22
0.0	3.0	3.0	23	2.0	3.0	1.0	43	1.0	2.0	3.0	22
0.0	3.0	3.0	21	2.0	3.0	1.0	39	1.0	3.0	2.0	19

$$R_n \rightarrow \ Start\ time\ of\ Reader_n$$
$$RT_n \rightarrow \ Read\ time\ of\ Reader_n$$
$$TT_n \rightarrow \ Think\ time\ of\ Reader_n$$
$$C_{1,2} \rightarrow \ Count\ of\ accesses\ by\ Reader_{1,2}$$
$$W_1 \rightarrow \ Start\ time\ of\ Writer$$
$$UT_1 \rightarrow \ Update\ time\ of\ Writer$$
$$GT_1 \rightarrow \ Gather\ time\ of\ Writer$$
$$C_3 \rightarrow \ Count\ of\ updates\ by\ Writer$$

Table 1: Output from simulation of reader/writer

7 Conclusions

The problem of establishing the behaviour of a system is very real for many simulation modellers. Answers to such questions can sometimes be found by expressing the system in a process algebra like CCS and using a process logic like the modal $\mu-$calculus to pose queries.

This paper has used a simple example to demonstrate that expressing process based simulations in both a discrete event simulation language (DEMOS) and a process algebra (CCS) can be achieved from a common graphical representation. Further, the resulting models can be used together to find both behavioural and performance properties.

This approach is still at an early stage of development. In particular, the power of the modal $\mu-$calculus is bought at the expense of an awkward notation and some difficulties in asking general questions easily. Now that the usefulness of combining these techniques has been shown, work remains in integrating the querying with the model description interface, to ease the learning curve for modellers.

Another direction for extension of this work is in applying it to direct derivation of models from widely used formalisms for system specification. For instance the CCITT approved protocol specification language LOTOS is based on process algebra features lik ehtose in CCS. With a well developed semantics for the simulation language, it may prove easier to find such mappings and to exploit them for both quantitative and qualitative results.

References

[1] J. Aldwinckle, R. Nagarajan and G. Birtwistle *An Introduction to Modal Logic and its Applications on the Concurrency Workbench*, University of Calgary Technical Report, June 1992

[2] H. Beilner and F.J. Stewing "Concepts and Techniques of the Performance Modelling Tool HIT", in Proceedings of the European Simulation Multiconference, Vienna, 1987, SCS Europe

[3] H. Beilner, J. Mäter and C. Wysocki "The Hierarchical Evaluation Tool HIT", in Tools Supplement of the 7th International Conference on Techniques and Tools for Computer Performance Evaluation, Vienna, 1994 pp 3-6

[4] G.M. Birtwistle, O.J.Dahl, B.Myhrhaug and K.Nygaard *Simula Begin*, Chartwell-Bratt, 1972

[5] G.M. Birtwistle *Discrete Event Modelling on SIMULA*, MacMillan, 1979

[6] G. Chiola "A Graphical Petri Net Tool for Performance Analysis", in D. Potier Ed. Proceedings of the International Workshop on Modelling Techniques and Performance Evaluation, March 1987, pp 297-307, AFCET, Paris

[7] R. Cleaveland, J. Parrow and B. Steffen "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems", ACM TOPLAS, Vol 15 No 1, 1993

[8] W. Franta *The Process View of Simulation*, North-Holland, 1978

[9] M.C. Hennessy and A.J.R.G. Milner "Algebraic Laews for Non-determinism and Concurrency", Journal of ACM, Vol 32 No 1, 1985, pp137-161

[10] P.H. Hughes *DEMOS Activity Diagrams*, Notat nr 1, FAG 45080 Simulering, Høst 1984, Norges Tekniske Høgskole, Norway

[11] Information Systems Research Associates *PAWS Users Guide*, 1986

[12] C. Lindemann "DSPNExpress: a Software Package for the Efficient Solution of Deterministic and Stochastic Petri Nets", in R. Pooley and J. Hillston Eds. *Computer Performance Evaluation - Modelling Techniques and Tools*, 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Edinburgh, September 1992, Edits 10, Edinburgh University Press,

[13] R. Milner *Communication and Concurrency*, Prentice-Hall, 1989

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

[15] F. Moller *The Edinburgh Concurrency Workbench (Version 6.1)*, Edinburgh University, Department of Computer Science, LFCS Technical Note, TN34, October 1992

[16] R.J. Pooley *An Introduction to programming in SIMULA*, Blackwells, 1987.

[17] R.J. Pooley and M.W. Brown "Automated modelling with the General Attributed (Directed) Graph Editing Tool - GA(D)GET", Proceedings of the European Simulation Multiconference, Nice, June 1988, pp 410-415

[18] R.J. Pooley "Demographer: A Graphical Tool for Combined Simulation and Functional Modelling", in R.Pooley and R. Zobel Eds, UKSS '93: Proceedings of the First Conference of the UK Simulation Society, September 1993, pp 91-95

[19] R.J. Pooley *Formalising the Description of Process Based Simulation Models*, PhD Thesis, Edinburgh University, 1994

[20] Schriber T.J. *Simulation Using GPSS*, Wiley, New York, 1974

[21] C. Stirling *Modal and Temporal Logics for Processes*, Technical Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1992

[22] C. Tofts *Timing Concurrent Processes*, Report ECS-LFCS-89-104, Edinburgh University, Department of Computer Science, 1989

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

[24] J. Vaucher "Simulation Data Structures using Simula 67", in Proceedings of the Winter Simulation Conference, 1971, pp 255-260

[25] J. Vaucher "A Generalised Wait-Until Algorithm for General Purpose Simulation Languages", Proceedings of the Winter Simulation Conference, 1973, pp 177-183

[26] M. Veran and D. Potier "QNAP 2: a Portable Environment for Queueing System Modelling" in D. Potier Ed. Proceedings of Modelling Techniques and Tools for Computer Performance Evaluation, North Holland, 1985, pp 25-63

[27] E. Yücesan and L. Schruben, "Structural and Behavioural Equivalence of Simulation Models", ACM TOMACS, Vol. 2 No 1, January 1992, pp82-103