HOLOS – A Simulation and Multi Mathematical Modelling Tool

Chris Tofts
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2001-276
October 31st , 2001*

E-mail: chris_tofts@hp.com

simulation, process algebra, petri net, abstraction, rewriting, compilation

The widespread adoption of analytic modelling techniques for computer systems analysis is hampered by many obstacles including "steep learning curve" on mathematics; lack of "reality" of models; and a widespread lack of belief in abstractions as preserving useful content. We present the HOLOS system an attempt to address these problems, the first by providing compilers from a common executable language DEMOS2k into various mathematical forms, the later two by providing translations and abstractions within the DEMOS2k language. The system is supported by an open extensible interface and is freely available under open source liscencing.

* Internal Accession Date Only Approved for External Publication
© Copyright Hewlett-Packard Company 2001

HOLOS - A Simulation and Multi Mathematical Modelling Tool.

C. Tofts*
HP Research Laboratories Bristol,
Filton Road, Stoke Gifford,
Bristol, BS34 8QZ,
chris_tofts@hp.com

October 25, 2001

Abstract

The widespread adoption of analytic modelling techniques for computer systems analysis is hampered by many obstacles including 'steep learning curve' on mathematics; lack of 'reality' of models; and a widespread lack of belief in abstractions as preserving useful content. We present the HOLOS system an attempt to address these problems, the first by providing compilers from a common executable language DEMOS2k into various mathematical forms, the later two by providing translations and abstractions within the DEMOS2k language. The system is supported by an open extensible interface and is freely available under open source liscencing.

1 Introduction

It is widley known that computer systems engineers and architects rarely model their systems before construction and deployment, unlike their civil engineering counterparts. There are many reasons why they do not model but major influences include: the perception that models are never detailed enough to reflect the 'true' behaviour of the system; and the difficulty of learning the appropriate techniques, especially in the view of the former prevailing view.

Simulation techniques have had some success as generic modelling approaches. In the large part this is because the representation of the models is often undertaken in a standard programing language of which the engineer is already familiar. However, whilst the use of a general purpose programming language frees the engineer from learning any new syntax, it does introduce a risk of errors in their own constructed simulation framework, and often repititous coding of essentially the same structures.

Using Simula[3] as his base language, during the course of constructing more than 70 modelling examples, Graham Birtwistle discovered that he was constantly writing the same structures and consequently wrote a framework (DEMOS) [2] to support those activities. This simple language has had considerable success as a presentation language for discrete event simulations. As the framework provides all of the basic synchronisation and timing mechanisms it is possible (at least via repeated use) to reduce the possiblity of error in the underlying simulator[1].

*On projects of this scale authorship is difficult, but as I wrote the document I will take the blame. Major contributions to this system have been made by R. Taylor, A. Christodoulou, and Maher Rahmouni.

[1]It is of considerable interest that by and large, the software community is aware of the possiblity of programming error, but the simulation community which often produces models of comparable complexity (and cost of error) expects boundary tests to detect mistakes.

More analytic approaches to modelling are typically not adopted as the level of expertise required usually means that any particular modeller will only be aware of a limited number of techniques. As it is often the case that the precise questions that the model needs to address have a major impact on the choice of modelling approach, the consequences of an inappropriate choice (delayed or no results, wasted time) mean that no abstract model is attempted.

Our view is based on the following observations about the experimental comprehension of models (or simulation). Simulation is considered a powerful modelling approach because:

1. it provides a clear debug by execution approach, reassuring the modeller that the basic system description is correct;
2. data is always obtained, little risk of no results;
3. there is no need to know in advance what questions you need to answer
4. there is little need to abstract, if any;
5. the problem will always be representable, if we assume the language is Turing complete;
6. the dynamic presentation that can be very compelling, especially as an explanatory tool for others.

However a pure simulation approach to understanding a system is limited in many ways:

1. the results are fundamentally experimental in nature, and experiments ae always hard to design and frequently ambiguous;
2. we have to execute the model on each data point, rarely do we have an reasonably small parameter space;
3. it does not make the modeller abstract and concentrate on the 'essence' of their problem;
4. it is very language dependent;
5. it is exposed to programming errors both in the model and in the underlying execution environment.

We are attempting to get all of the benefits of both basic approaches to modelling. To achieve some of our goals particularly the proper definition of our presentation language we need it to be closed. The original DEMOS system was dependent in a large part on the underlying Simula and thus to formalise the complete language would require a formalisation for all of Simula, a very difficult task. So given our earlier work on the semantics of DEMOS both operational [5, 6] and denotational [7, 22, 8, 9] we took this language and extended it until we could express (hopefully elegantly) all of the original examples [2]. Equally we attempted to achieve the same level of expressive power as Milner's Pi calculus [14].

The HOLOS system consists of a process oriented concurrent system presentation language which **may** be executed (simulated) along with a set of compilers into multiple formal modelling techniques. This view is given in figure 1.

The current implementation permits the addition of arbitrary extra compilers as sml[16] functions, see below for further details.

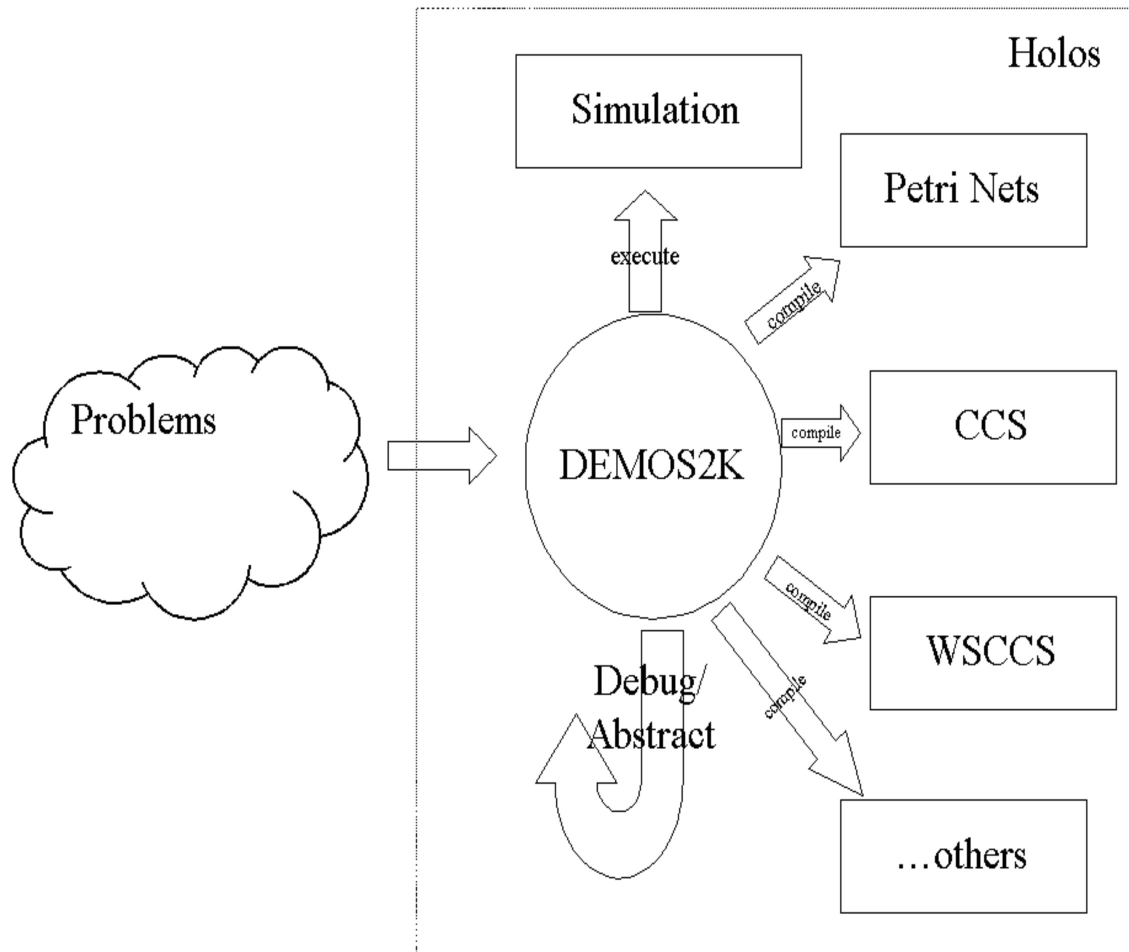

Figure 1: The Holos system.

2 DEMOS2K

DEMOS is a process oriented discrete event simulation language originally defined as an extension to the simula language. The current version has little changed from the original other than in how entities slave themselves to others. DEMOS has essentially 3 components.

1. entities, defined as instances of classes. A class provides the definition of a collection of actions that an instance **entity** will undertake. An entity can be thought of as an inidividual parallel process interacting with others and the environment;

2. resources, are atomic accessible counters an used as locks (limits) similar to semaphores;

3. bins, are atomic counters of numbers of objects in a queue. Bins form desynchronisation points as opposed to resources that are synchronisation points.

These elements all have definitional forms:

1. entity(name,className,offfset) and class name=body

2. res(foo,amt);

3. bin(foo,amt);.

Entities are the active elements of the system, resources can be thought of a semaphores and bins are points of asynchrony. Entities interact with other elements of the system in the following manner:

1. getR(resN,**amt**) and putR(resN,**amt**) repectively claim and free the **amt** of resource resN, an entity cannot return amounts of resource it does not own;
2. getB(binN,**amt**) and putB(binN,**amt**) repectively claim and free the **amt** of bin binN;
3. sync(name) slaves the current entity on name. The current entity waits to be claimed and released on the name and then continues its execution from the next command;
4. getS(syn,**amt**) and putS(syn,**amt**) repectively claim and free the **amt** of sync syn, an entity cannot return amounts of sync it does not own.
5. Entity(name,className, **offset**) spawns a new instance of className called name at time **offset** into the future;
6. Hold(**ti**) advances the simulation clock **ti** into the future.

Collectively getR(resN,**amt**), getB(binN,**amt**) and getS(syn,**amt**) are refered to as acquisitions. The following compound operations are defined on them:

1. **req[acq1,...,acqN]** requires that all of the requests can be granted simutaneously before the entity can proceed.
2. **try [req1] then Bd1 etry [req2] then Bd2 ... etry [reqN] then BdN;** will try each of the requests in turn until one can be satisfied, otherwise it blocks until one can be satisfied. Often used with ... etry [] then hold(t) as a non-blocking test.

Finally we allow loops

1. **repeat body** executes body forever;
2. **do n body** executes body n times;
3. **while [req] body** as long as the requirement can be met execute the body. Note a requirement can be a boolean.

A DEMOS program consists of an entity (conventionally referred to as main) that sets up class and other definitions and then invokes entities to form the running system.

3 Operational Semantics

The execution of a DEMOS program is defined operationally, the first analysis of the interaction fragments are in [5, 6]. The full semantics runs to some 50 pages see [25]. We have to define how the system state will change as a function of all of the commands we may execute. The system state consists of 3 parts.

1. event list, those entities which are ‘active’;
2. blocked list, those entities waiting on acquisitions;

3. a store, holding variable/constant values, class definitions, resource, bin and sync levels;

Since the store can be defined in many different ways we omit its presentation as obvious. Both the event list and the blocked list consist of process descriptors so we give an outline of their definition. Finally in this section we present the outline of the semantics with the full presentation of the semantics of putBS.

3.1 Event notices

The event notice has the following form, **PD(id,pr,Body,Attrs,evt)**, which can be understood as follows:

id a name identifying the entity;

pr the current priority of this entity, can be changed with the *priority* command;

Body the list of actions the entity wishes to engage in, the list providing the order over the actions;

Attrs a store containing three types res, sync and var. The res being the name and amount of currently held resources. The sync a list of synced processes in the order in which they were claimed. The var is the values of local variables.

evt the time at which this notice should next be considered active.

3.2 Operational definitions

As each program command is executed the system will change from one state to another

$$(\mathcal{EL},\ \mathcal{BL},\ \mathcal{S}) \Longrightarrow (\mathcal{EL}',\ \mathcal{BL}',\ \mathcal{S}')$$

Execution is so framed that the next action to be executed is always the first action in the action list of the first object in the event list. Thus given the event list pattern-matching

$$\mathcal{EL} = PD(c,\ pr\ ,b::Body,\ Attrs,\ evt)::\ldots$$

— the next action *must* be b and the system time is said to be at the time of this action, *evt*.

It is now straightforward to give a semantics as a case statement over the structure of DEMOS commands, as sketched below:

1. an *error* arises if the event list becomes empty (the system should be shut down with a call on `close`).

   exec ([], $\mathcal{BL}$, $\mathcal{S}$) =*error*

2. When a process has exhausted its actions, a check is made to see whether it still owns any attributes. An error results if it does. If not, all is well. The object is deleted from the event list. The simulation proceeds from the next action of the new current.

   exec (PD(c,pr,[],Attrs,evt)::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
   = **if** Attrs = empty
   then exec($\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
   else *error*

3. The normal case — we focus on *PD(c, pr, b::Body, Attrs, evt)* the object at the head of the event list, and execute its next action *b*. The names *Body*, *Attrs*, and *evt* are directly accessible in the case clause, as is *cp'*, the "expected next current process object". We add some global information, current time along with bin and resource levels, into the store at this point.

3.2.1 putBS(id,exL,reL,sL)

Add a bin item with parameters given in the expression list exL to the bin *id*, also has resources as specified by reL and syncs specified by sL;

Semantics:

putBS(id,exL,reL,sL)

= (evalN (Attrs$\oplus\mathcal{S}$) id, bins) $\notin \mathcal{S} \Rightarrow$ *error*

| **let** n = evalN (Attrs$\oplus\mathcal{S}$) id **in**
  let vL = map evalN (Attrs$\oplus\mathcal{S}$) exL **in**
  let rL = map evalR (Attrs$\oplus\mathcal{S}$) reL **in**
  let sL = map evalN (Attrs$\oplus\mathcal{S}$) sL **in**
  if pr=-15000 then
  let $\mathcal{S}$' = ADDITEM(n, bins, (vL,rL,syN)) $\mathcal{S}$ **in**
    promote(putBS(n,eL,reL,sL), evt, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$')
  else
  let (cp",syN) = removeSL(sL,cp',[]) **in**
  in (ENTER PD(cn,-15000,[putBS(n,vL,rL,syN)],(),evt) removeRL(rl,cp")::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)

Interpretation:

1. (evalN (Attrs$\oplus\mathcal{S}$) id, bins) $\notin \mathcal{S}$: error if the bin identifier does not exist;
2. *Normal case*
   (a) **let** n = evalN (Attrs$\oplus\mathcal{S}$) id : work out the resource name;
   (b) **let** vL = map evalN (Attrs$\oplus\mathcal{S}$) exL: evaluate the expressions passed with the bin item;
   (c) **let** rL = map evalR (Attrs$\oplus\mathcal{S}$) reL: evaluate the amounts of each resource passed along with the bin item;
   (d) **let** sL = map evalN (Attrs$\oplus\mathcal{S}$) sL: and work out the ground sync names;
   (e) if pr=-15000 then: we are actually doing the put;
   (f) **let** $\mathcal{S}$' = ADDITEM(n, bins, (vL,rL,syN)) $\mathcal{S}$put the parameterised bin item in the store, with its associated values, resources and syncs;
   (g) use the semantic function promote(putBS(n,eL,reL,sL), evt, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$') to enable any entities that may have become unblocked. Since the evaluated form has been stored in the state, we only use the name and hence have no state overwrite problems. Note the current entity has only one action and hence will die.
   (h) else
   (i) **let** (cp",syN) = removeSL(sL,cp',[]): remove the syncs from the current process notice, note may lead to an error if we pass on syncs we do not have;
   (j) (ENTER PD(cn,-15000,[putBS(n,vL,rL,syN)],(),evt) removeRL(rl,cp")::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$): then enqueue the actual return as the last activity at this clock instant.

4 Execution Support

For any model simulation, or exploration by execution provides us with two major benefits. Firstly it allows us to explore our model dynamically and often identify errors in our modelling. Secondly, there are limits to the scale of models that can be addressed analytically and consequently a simulator guarantees that the modelling effort will deliver some results, even if their provenance is limited. The DEMOS simulator is component based:

1. an engine written in sml, to give closest match to the operational semantics;
2. an sml parser;
3. a perl[27]/Tk front end, displayed in Figure 2;
4. a perl/Tk data accumulator;
5. a VRML[1] data visualisation system, displayed in Figure 3.

One of the major drivers in the above choices was that all of the components should be based on open software. We have also concentrated our compilation efforts into those areas where tool support is also open.

Furthermore the front end supports plug in compilers. It can be extended with sml components that can be executed on the currently live DEMOS program. These additions can be made without any coding within the primary interface. Other plugins allow the addition of different data filters written in perl, the ability to add perl components to the main interface, and the ability to activate executables and batch files.

5 Translations

To formally present each of the following translations will require approximately 20 pages each so in the interest of brevity we will take the example DEMOS program below and show its translation into each of the four formal presentations.

Our simple example works as follows: work is generated by a source at an a rate *arrive*, this work accumulates in a bin called em work. A server takes an item of work and then processes it using resource *R1*, this takes an amount of time *task1*. This resource is needed for all processing. It then has a choice for the second stage of processing. If there is a *sec_serv* slave available it is used and returned and this stage of the job takes *quick* amount of time. This path is prefered. Alternatively it can make use of *R2* and the secondary task will take time *slow*. Having completed the task whichever set of resources/syncs are held are returned, and the complete job begins again.

```
Bin(work,0);          //an empty bin
Res(R1,2);            //a number of slots
Res(R2,1);            //backup server

(*a Source of work*)
class src={repeat{hold(arrive);putB(work,1);}}

(*a secondary server-note we can put hysterysis here*)
class sec_serv={repeat{sync(service);hold(hysterysis);}}
```

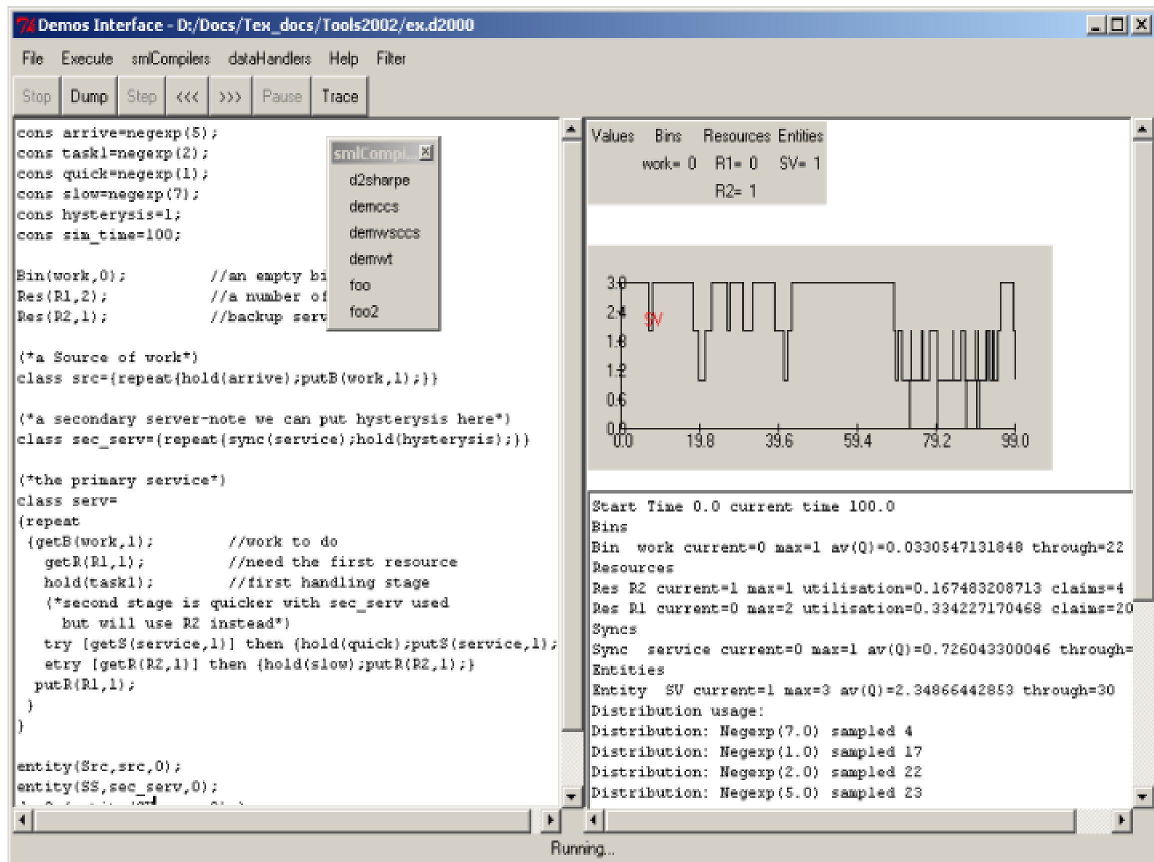

Figure 2: The interface with a DEMOS program on the left, results on the right (including graphs) and the compilers menu active.

```
(*the primary service*)
class serv=
{repeat
 {getB(work,1);         //work to do
   getR(R1,1);          //need the first resource
   hold(task1);         //first handling stage
   (*second stage is quicker with sec_serv used
     but will use R2 instead*)
   try [getS(service,1)] then {hold(quick);putS(service,1);}
   etry [getR(R2,1)] then {hold(slow);putR(R2,1);}
  putR(R1,1);
 }
}

entity(Src,src,0);
entity(SS,sec_serv,0);
```

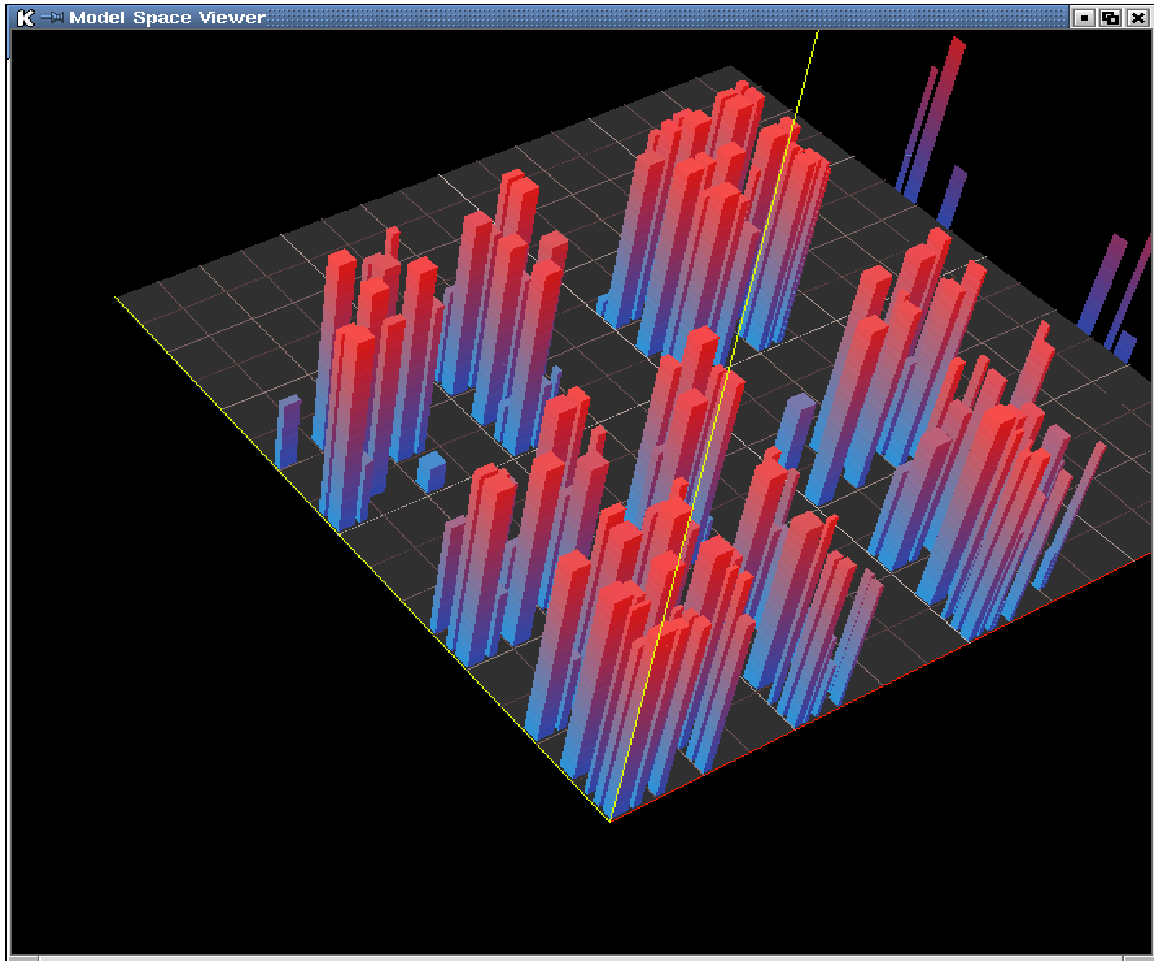

Figure 3: The data visualisation system.

```
do 3 {entity(service,serv,0);}
hold(sim_time);
close;
```

5.1 Petri Nets

The translation of the above DEMOS program into a probabilistic Petri net [17, 18, 19] is presented below. An account of the full translation of DEMOS currently understood can be found in [26]. These models can be automatically analysed with tools such as [20].

5.2 CCS

The original work on the relationship between DEMOS and CCS [12] was in the form of structural observation [5]. An automatable translation is presented in [22, 8, 9] and its relationship with the operational semantics demonstrated in [7]. The CCS representatoin allows us to address questions such as livelock and deadlock, using tools including the Edinburgh Concurrency Workbench [15, 10]

```
*Bin work max=3
```

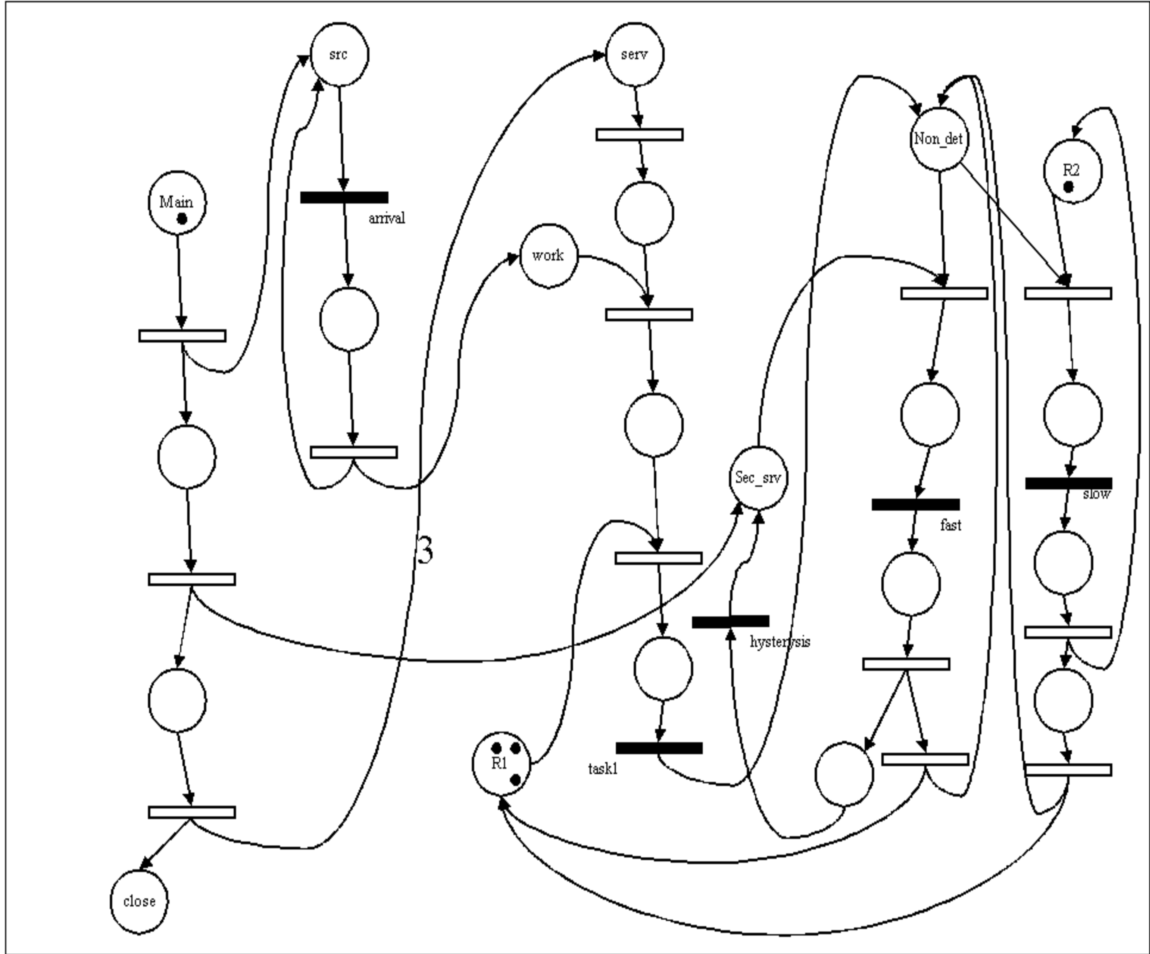

Figure 4: The simple example as a Petri net.

```
agent Bwork_3 getwork.Bwork_2;
agent Bwork_2 getwork.Bwork_1 + putwork.Bwork_3;
agent Bwork_1 getwork.Bwork_0 + putwork.Bwork_2;
agent Bwork_0 putwork.Bwork_1;

*Resource R1
agent R1_2 getR1.R1_1;
agent R1_1 getR1.R1_0 + putR1.R1_2;
agent R1_0 putR1.R1_1;

*Resource R2
agent R2_1 getR2.R2_0;
agent R2_0 putR2.R2_1;

*class src
agent Csrc 'putwork.holdArrival.Csrc;
```

```
*class sec_serv
agent Csec_serv getSservice.putSservice.holdHysterysis.Csec_serv;

*class serv
agent Cserv 'getwork.'getR1.holdtask1.CservCh;
agent CservCh 'getSservice.holdfast.'putSservice.'putR1.Cserv
             + 'getR2.holdslow.'putR2.'putR1.Cserv;

set R {getwork,putwork,getR1,putR1,getR2,putR2,getSservice,putSservice};

agent Main (Bwork_0|R1_2|R2_1|Csrc|sec_serv|Cserv|Cserv|Cserv)/R;
```

5.3 WSCCS(1)

To formally analyse the probabilistic, priority and timing properties of DEMOS programs we need a formal language that can encompass all of these phenomena. A prototypical extension to Milner's SCCS [11] is WSCCS [21, 24]. This translation can be extended to a large fragment of DEMOS [25] and there is detail of automated analysis support in the same document.

```
*Bin work max=3
bs Bwork_3 1.getwork:Bwork_2 + 1.t:Bwork_3
bs Bwork_2 1.getwork:Bwork_1 + putwork.Bwork_3 + 1.t:Bwork_2
bs Bwork_1 1.getwork:Bwork_0 + putwork.Bwork_2 + 1.t:Bwork_1
bs Bwork_0 1.putwork:Bwork_1 + 1.t:Bwork_0

*Resource R1
bs R1_2 1.getR1:R1_1 + 1.t:R1_2
bs R1_1 1.getR1:R1_0 + 1.putR1:R1_2 + 1.t:R1_1
bs R1_0 1.putR1:R1_1 + 1.t:R1_0

*Resource R2
bs R2_1 1.getR2:R2_0 + 1.t:R2_1
bs R2_0 1.putR2:R2_1 + 1.t:R2_0

*class src
bs Csrc 1@1.putwork^-1:Csrc1 + 1.t:Csrc
bs Csrc1 Arrival.t:Csrc + (1-Arrival).t:Csrc1

*class sec_serv
bs Csec_serv 1.getSservice.Csec_serv_1 + 1.t:Csec_serv
bs Csec_serv_1 1.putSservice:Csec_serv_2 + 1.t:Csec_serv_1
bs Csec_serv_2 Hysterysis.t:Csec_serv + (1-Hysterysis).t:Csec_serv_2

*class serv
bs Cserv 1@1.getwork^-1:Cserv_1 + 1.t:Cserv
bs Cserv_1 1@1.getR1^-1:Cserv_2 + 1.t:Cserv_1
bs Cserv_2 task1.t:CservCh + (1-task1).t:Cserv_2
bs CservCh 1@2.getSservice^-1:CservCh_1_1 + 1@1.getR2^-1:CservCh_2_1 + 1.t:CservCh
```

```
bs CservCh_1_1 fast.t:CservCh_1_2 + (1-fast).t:CservCh_1_1
bs CservCh_1_2 1@1.putSservice:Cserv_3 + 1.t:CservCh_1_2
bs CservCh_2_1 slow.t:CservCh_2_2 + (1-slow).t:CservCh_2_1
bs CservCh_2_2 1@1.putR2^-1.Cserv_3 + 1.t:CservCh_2_2 '
bs Cserv_3 1@1.putR1^-1#done:Cserv + 1.t:Cserv_3

basi R done

btr Main Bwork_0|R1_2|R2_1|Csrc|sec_serv|Cserv|Cserv|Cserv\R;
```

5.4 WSCCS(2)

A dual approach to the representation of performance problems within a synchronous process algebra allows us to greatly reduce the size of the state space[23], and consequently the analysis time can also be exploited to analyse DEMOS programs. An example of that form of translation is presented below.

```
*Bin work max=3
bs Bwork_3 1.getwork:Bwork_2 + 1.t:Bwork_3
bs Bwork_2 1.getwork:Bwork_1 + putwork.Bwork_3 + 1.t:Bwork_2
bs Bwork_1 1.getwork:Bwork_0 + putwork.Bwork_2 + 1.t:Bwork_1
bs Bwork_0 1.putwork:Bwork_1 + 1.t:Bwork_0

*Resource R1
bs R1_2 1.getR1:R1_1 + 1.t:R1_2
bs R1_1 1.getR1:R1_0 + 1.putR1:R1_2 + 1.t:R1_1
bs R1_0 1.putR1:R1_1 + 1.t:R1_0

*Resource R2
bs R2_1 1.getR2:R2_0 + 1.getR2#putR2.R2_1 + 1.t:R2_1
bs R2_0 1.putR2:R2_1 + 1.t:R2_0

*class src
bs Csrc 1@1.putwork^-1#dArrival:Csrc + 1.t:Csrc

*class sec_serv
bs Csec_serv 1.getSservice.Csec_serv_1 + 1.getSservice#putSservice:Csec_serv_2 + 1.t:Csec_se
bs Csec_serv_1 1.putSservice:Csec_serv_2 + 1.t:Csec_serv_1
bs Csec_serv_2 1.dHysterysis:Csec_serv

*class serv
bs Cserv 1@1.getwork^-1#getR1^-1#dtask1:CservCh + 1.t:Cserv
bs CservCh 1@2.getSservice^-1#dfast#putSservice^-1#putR1^-1#done:Cserv \
           + 1@1.getR2^-1#dslow#putR2^-1#putR1^-1#done:Cserv + 1.t:CservCh

basi R done, dArrival, dtask1, dfast, dslow

btr Main Bwork_0|R1_2|R2_1|Csrc|sec_serv|Cserv|Cserv|Cserv\R;
```

6 Abstraction

Abstraction is an activity largely undertaken by modellers. In most modelling contexts the problem owner will not be responsible for constructing the model. They engage in a discourse with a modeller who, usually, abstracts the system under consideration and then derives what they believe the information the problem presenter required. One of the consequences of this is that often the amount of abstraction used gives the problem owner little or no belief in the results of the model. As organisations grow and the gap between problem owner and modeller grow increase problem can become acute.

One of the consequences of the hegemony of the C programming language has been the work on optimising compilers onto particular processors. In this context there are both globally useful work; optimisations that work on all underlying architectures, and locally useful work; optimisations that work on particular underlying architectures. In the context of modelling we may regard a particular mathematical analysis as a processor and the act of translating from some, neutral representation, into that as being compilation. In this context if the neutral representation can be comprehended by execution then we have the chance to explain our abstraction. We can proceed by presenting a chain of models aimed at a particular analysis technique. In the worst instance we can demonstrate by experiment that the models are broadly in agreement. In the best case we may be able to prove that particular rewrites cannot change the results of the model.

However, for the above to work we must agree to concentrate on a particular presentation language something that has been singularly lacking within the formal methods community[2]. The major problems in this space will be the temptation to use a language that is particularly suited to one mathematical technique and the obvious consequences of the triviality of achieving Turing completeness with almost any programtic presentation.

7 Conclusions and Further Work

There are clear and immediate benefits to being able to take one representation of a model and derive formal models that permit multiple forms of an anlysis. However the attempt to produce such systems present us with major challenges. Firstly the definition of the primary presentation language must extend beyond that of its syntax. Any choice of initial presentation language will have an intended execution model and unless this is well defined it will be impossible to comment on the relationship of any results derived from a more formal analysis of a compilation of the original text. One pleasing result is that the outcome in our examples above of the DEMOS2k text being much shorter than the other forms is the usual outcome. To the point where the author, despite being the originator of WSCCS uses DEMOS as his primary presentation language.

The above observation limits our choice of original presentation given the difficulty of defining the behaviour of even relatively 'clean' languages [13]. Our choice of presentation language has been guided by two principles; firstly, it has been widely used in the past; secondly, we already knew that it could be formally defined and compiled into analysable mathematics which could be related to the original definition. The current form of DEMOS is largely a consequence of having to remove all dependencies on an underlying Simula system. Otherwise we would need to provide a semantics for that language, an even greater challenge. Equally we wished to remove all explicit scheduling activities from the coder and to achieve (as much as possible) expressive power comparable with Milner's Pi calculus [14].

[2] It has always been amusing that the main concurrency theory conference acronym is CONCUR.

Our interface design is deliberately intended to permit the easy addition of alternative compilers to encourage experimentation within as many forms as possible.

We are currently investigating how we might derive queueing and difference equation models directly from DEMOS representations. One major area of interest is the translation of one DEMOS program into another. The major benefit of process algebraic representations is that we can (occasionally) prove that various alternative forms of model are equivalent. Often merely changing the view of the model can lead to better forms of compilation. In particular the two forms (Petri net/automata) views shown in this document provide an interesting view of DEMOS2k. The Petri view is based essentially translating all interactions as mediated by bins. The automata (Process Algebra) view would be of all interactions being syncs. Both of these views provide normal forms for DEMOS which may admit efficient compilation.

Finally there is abstraction in the traditional sense, of omitting some of the complexity of the problem. One of our major requirements was the ability to construct 'chains' of models as witnesses to abstraction, and indeed this is relatively straightforward within the syntax of DEMOS. Clearly an ideal situation to reach would be one where this abstraction is (largely) automated in response to the particular questions the modeller has. Equally clearly this is an immense technical challenge, but possbily achievable when viewed from the perspective of complier optimisation.

References

[1] A. Ames, D. Nadeau & J. Moreland, VRML 2.0 Sourcebook (2nd Edition) John Wiley & Sons, 1996.

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

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

[4] G. Birtwistle, R. Pooley, and C. Tofts. Characterising the Structure of Simulation Models in CCS. *Transactions of the Society for Computer Simulation*, 10(3):205–236, 1993.

[5] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 1: πDEMOS. *Transactions of the Society for Computer Simulation*, 10(4):299–333, 1993.

[6] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 2: μDEMOS. *Transactions of the Society for Computer Simulation*, 11(4):303–336, 1994.

[7] G. Birtwistle and C. Tofts. Relating Operational and Denotational Descriptions of πDEMOS. *Simulation Practice and Theory*, 5(1):1–33, 1997.

[8] G. Birtwistle and C. Tofts. Getting DEMOS Models Right - Part I: Practice. to appear *Transactions of the Society for Computer Simulation*, 2001.

[9] G. Birtwistle and C. Tofts. Getting DEMOS Models Right - Part II: ... and Theory. to appear *Transactions of the Society for Computer Simulation*, 2001.

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[11] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[13] R. Milner, M, Tofte, D. MacQueen & R. Harper. Definition of Standard ML. The MIT press, 1997.

[14] R. Milner. *Communication and Mobile Systems*. CUP, 1999.

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

[16] L. C. Paulson ML for the Working Programmer, Cambridge University Press, 1996.

[17] W. Reisig. A Primer in Petri Net Design, Springer Compass International, 1992.

[18] W. Reisig. Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets, Springer Verlag, 1998.

[19] W. Reisig & G. Rozenburg (Eds), Lectures on Petri Nets Basic Models : Advances in Petri Nets, Springer Verlag Lecture Notes in Computer Science, 1491, 1998.

[20] R. Sahner , K. S. Trivedi & A. Puliafito. Performance and Reliability Analysis of Computer Systems : An Example-Based Approach Using the Sharpe Software Package, Kluwer Academic Publishers, 1995

[21] C. Tofts. Processes with Probability, Priority and Time. *Formal Aspects of Computer Science*, 6(5):536–564, 1993.

[22] C. Tofts and G. Birtwistle. A denotational semantics for a process-based simulation language. *ACM Transactions on Modelling and Simulation*, 8(3):281–305, 1998.

[23] C. Tofts. Symbolic Approaches to Probability Distributions in Process Algebra. To appear, *Formal Aspects of Computer Science*, 2001.

[24] C. Tofts. Performance Modelling Using Probabilistic Process Algebra. In T. Hoare, M. Broy, R. Steinbruggen, editors, *Engineering Theories of Software Construction* Vol. 180, pp223-257, 2001.

[25] C. Tofts. The Operational Semantics of DEMOS2k. Report HPL-2001-263 Hewlett Packard Research Laboratories Bristol, 2001.

[26] C. Tofts. Translating DEMOS into Petri Nets.. Report HPL-2001-274 Hewlett Packard Research Laboratories Bristol, 2001.

[27] L. Wall, T. Christiansen & J. Orwant, Programming Perl (3rd Edition), O'Reilly & Associates, 2000.