Operational Semantics of DEMOS 2000

Chris Tofts, Graham Birtwistle[1]
Information Infostructure Laboratory
HP Laboratories Bristol
HPL-2001-263
October 26th , 2001*

E-mail: chris_tofts@hp.com, graham@scs.leeds.ac.uk

semantics, discrete event simulation, functional programming, correctness

We take a process oriented simulation language, DEMOS 2000, and define its operational semantics. This achieves two goals: firstly, we can unambiguously define the behaviour of any DEMOS program; secondly, we can use the semantics to check the behaviour of any implementation of the language. An operational semantics has been given for the simple synchronization fragment of μDEMOS before (Birtwistle & Tofts, 1993, 1994), but that fragment lacked many features of a complete discrete event simulation language. DEMOS 2000 is a slight redesign of DEMOS that frees it from the underlying Simula. In this semantics we give an account of the complete language which includes features such as: parametric classes; value, resource, entity handling synchronisations; multi-way competing synchronisations; priority; conditional synchronisations; random number sampling; data dependent entities; data dependent work.

* Internal Accession Date Only Approved for External Publication
[1] School of Computer Studies, Leeds University, Woodhouse Lane, Leeds, UK

Operational Semantics of DEMOS 2000.

Chris Tofts	Graham Birtwistle
HP Laboratories Bristol	School of Computer Studies
Filton Road	Leeds University
Stoke Gifford	Woodhouse Lane
Bristol	Leeds
chris_tofts@hp.com	graham@scs.leeds.ac.uk

October 10, 2001

Abstract

We take a process oriented simulation laguage, DEMOS 2000, and define its operational semantics. This achieves two goals: firstly, we can unambiguously define the behaviour of any DEMOS program; secondly, we can use the semantics to check the behaviour of any implementation of the language. An operational semantics has been given for the simple synchronisation fragment of μDEMOS before [3, 4], but that fragment lacked many features of a complete discrete event simulation language. DEMOS 2000 is a slight redesign of DEMOS that frees it from the underlying Simula. In this semantics we give an account of the *complete* language which includes features such as: parametric classes; value, resource, entity handling synchronisations; multi-way competing synchronisations; priority; conditional synchronisations; random number sampling; data dependent entities; data dependent work.

1 Introduction

One of the major problems of any programming activity is being certain that the program is behaving correctly. Programming languages are largely defined by example and intent. That is examples of code and its results are presented then the results of executing similar code in other situations must be inferred by the programmer. If the results of the program have no value, an unlikely outcome given the effort to construct any non-trivial progam, then this would be of little consequence. However programs are often used to control potentially life threatening systems, and to support business threatening decisions.

When the program is of a conventional form, a simple variant of the lambda calculus (Imperative, Functional or Declarative) then one may assume that the program inherits its meaning from that underlying implicit formalism. Alternatively when the language is not in this class we have two problems, what mathematical entities does it represent, and how should we execute it correctly. However it is still the case that for a 'real' programming language, even one based on a very clear mathematical underpinning, the effort of providing a semantics is a major undertaking [10].

One of the major activies of business (design) support is the use of simulation, often undertaken by computer using either extensions to general purpose programming languages or via a speciliased language. These languages have representations for interaction and other complex constructions that are presented by intent. Given the need for efficient programming languages for systems modelling to support decision making we attack the second of these problems by precisely defining

the excution model of a simulation oriented programming language. In companion papers the mathematical framework(s) within which this language should be understood are detailed[6, 5].

DEMOS 2000 is a modest extension of the DEMOS system [2] a simulation environment built within the Simula [1] compiler. The aims of the extensions are:

- to remove the dependency on an underlying programming environment, and hence upon its semantics;
- removing all programmer organized syncrhonisation points, no explicit queue maintenance by the programmer;
- use a process algebra view to guide expressivity.

One of the most pleasant outcomes is that this requirements do not impose major changes on either the syntax or the semantics of DEMOS. For further details of the DEMOS system under construction see w3.hpl.hp.com/demos/.

In this form we can regard DEMOS(2000) as a guarded command language (cf unity, Dijkstra like languages). All active commands test the current system state, if the condition they represent can be meet then they are executed otherwise they are blocked until such time as the condition can be met. The major difference between process oriented simulation languages and guarded command languages is that the conditions have side effects, the assignment of resource to the active entity. Hence change of state is mediated not only by assignment to variables, but by assignment and claim of resource, and by entities becoming resources themselves.

One of the major changes from our semantic presentation of DEMOS is that we have exploited the observed normal form, that all guards are simple versions of a compound guard. Hence, the inclusion of the various atomic forms of the commands is syntactic sugar[1]. Consequently the difficulty within the semantics, that is commands that cause more activity than trivial state update: is all located within two areas the satisfaction of case based compound requests; the unblocking effects of a resource return or variable assignment.

Again we permit ourselves to summarize the natural deduction style presentation of the structural operational semantics by using SML like functional presentation. This semantics is a generalisation of that presented in [3, 4]. We present the syntax in its unambiguous functional form. The external input syntax is a more conventional form and as two examples: x:=e; is written Ass(x,e) and try req1 then bd1 etry req2 then bd2; is written Cond([(req1,bd1),(req2,bd2)]). However, since the mapping from the infix to the functional form of the language is trvial we omit these details.

2 Notational Preliminaries

2.1 Lists

There are many ordered structures we wish to maintain within the DEMOS system, such as events, blocked items, waiting entities and ordered queues. All of these structures can be maintained using lists.

The empty list is denoted by `[ ]`. When we wish to display a nonempty list in full, we enumerate it. The process body below with three actions:

```
[ getR(fs "w", 1), hold 3, putR(fs "w", 1) ]
```

[1] strictly its absence of a syntactic wrapper

is actually short for

```
getR(fs "w", 1)::hold 3::putR(fs "w", 1)::[]
```

where *::* is the infix operator (usually called *cons*) used to glue atoms onto lists at their head. Most of the time we wish to focus upon the first action in a process body, since that will be its next action to be carried out, For this we use the technique of *pattern matching*: if we write

```
b::Body = [ getR(fs "w", 1), hold 3, putR(fs "w", 1) ]
```

then in the ensuing text, `b` is matched to the head of the list `getR("w", 1)` and `Body` is matched to its tail `[ hold(3), putR("w", 1) ]`.

We define two higher order functions that can greatly simplify list manipulation:

map f [] = []—
map f h::t = f(h)::(map f t)

imap f [] L = L
imap f h::t L = imap f t f(h,L)

the first applies a function over all of the elements of a list, the second applies a function to a list of elements accumulating the results of repeated application in a result list.

For more detail, see almost any text on a modern functional programming language.

2.2 Storage

We use storage to hold information about the states of resources, synchronizations class definitions and the values of variables and constants. We leave the precise implementation details unstated, but assume that we have the following operations on storage.

Given a store $\mathcal{S}$ then the following operations are defined upon it.

Membership. *(id, rt)* $\in \mathcal{S}$ returns *true* if an entry for *id* of type *rt* lies in $\mathcal{S}$, *false* if not. In this context, the usual question we wish to resolve is whether an identifier is fresh (not found) or not. *notUnique id* $\mathcal{S}$ returns false only if *id* is fresh with respect to the environment $\mathcal{S}$. Some of the values stored are in the form of lists to maintain order. We augment standard store operations to manipulate the list

Lookup an entry. *LOOKUP (id, rt)* $\mathcal{S}$ returns *rd* when *(id, rt, rd)* $\in \mathcal{S}$. The call is an error if *(id,rt)* $\notin \mathcal{S}$.

Remove an entry. When *(id, rt, rd)* $\in \mathcal{S}$, $\mathcal{S}$ — *(id,rt)* removes the entry for *(id,rt)* and returns the reduced store $\mathcal{S}$'. The call is an error if *(id,rt)* $\notin \mathcal{S}$.

Delete an entry. When *(id, rt, rd)* $\in \mathcal{S}$, *DELETE id* $\mathcal{S}$ removes the entry for *id* and returns the entry and the reduced set $\mathcal{S}$' as a pair: *(rd,* $\mathcal{S}$*')*. The call is an error if *(id,rt)* $\notin \mathcal{S}$.

Update an item. If *(id,rt)* $\notin \mathcal{S}$, *UPDATE (id, rt, rd)* $\mathcal{S}$ adds an entry *(id, rt, rd)* to $\mathcal{S}$. If *(id,rt)* $\in \mathcal{S}$, then it overwrites the previous entry for *(id,rt)*. If the update simply adds an item, we will usually write $\mathcal{S}$ *++ (id,rt)*.

Add to a list If *(id,rt)* $\notin \mathcal{S}$, *ADDITEMS (id, rt, vL)* $\mathcal{S}$ adds an entry *(id, rt, vL)* to $\mathcal{S}$. If *(id,rt,oldvL)* $\in \mathcal{S}$, then it replaces the previous entry for *(id,rt)* with *(id,rt,oldvL@vL)*. Note that we preserve the order in which the store is built up.

Get from a list If *(id,rt)* $\notin \mathcal{S}$, then raise an error, else *GETITEMS (id, rt, amt)* $\mathcal{S}$ returns a pair consisting of a list if length *amt* from the list *vL* from the stored item *(id,rt,vL)* and the stored value becomes *(id,rt,reduce(n,vL))*. So we return (vL, $\mathcal{S}$'). Where *reduce(n,L)* is the standard list function that removes the first *n* items from a list.

store union Given two stores $\mathcal{S} \oplus \mathcal{S}$' forms a new store from the stores $\mathcal{S}$ and $\mathcal{S}$', if there is a clash of elements i.e (id,rt,v) $\in \mathcal{S}$ and (id,rt,v') $\in \mathcal{S}$', then the first element takes precedence. This permits us to define local scope, by permitting the local values to take precendence over globals.

make available Given a store convert availability of res, bin and sync record types to variables. `mk_av` $\mathcal{S}$ given *(id,ty,v)* in $\mathcal{S}$ then *(AV_id,var,avail(v))* is constructed in a new store.

store overwrite Given two stores $\mathcal{S} \rightarrow \mathcal{S}$' for all triples it (id,ty,v) $\in \mathcal{S}$ perform *UPDATE (id,ty,v)* $\mathcal{S}$*'* returning the store formed by all of the operations.

Note we do not need an explicit size function for stored lists as *length(LOOKUP (id,rt)* $\mathcal{S}$*)* with the standard list length function will perform this task.

2.3 The event and blocked lists

The event list is an ordered list of the form *PD(id, pri, Body,Attrs,evt)* ranked by increasing time *evt* and *pri* if the times are the same. We postpone the full explanation of process descriptors to Section 3.1.

Given that

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

then $\mathrm{t}_1 \leq \mathrm{t}_2 \leq ... \leq \mathrm{t}_n$ and further if $\mathrm{t}_k = \mathrm{t}_{k+1}$ then $\mathrm{p}_k \leq \mathrm{p}_{k+1}$. We posit two basic event list routines and two auxiliaries: here are their explanations together with concrete FCFS list implementations. The first process in the event list is referred to as *current*. The simulation clock time is the event time of current.

evTime en (event notice *en*) returns the event time of *en*.

evTime PD(id,pr,Body,Attrs,evt) = evt

pPri en (event notice *en*) returns the priority of *en*.

pPri PD(id,pr,Body,Attrs,evt) = pr

future en time makes the event notice active at *time* and returns the new event notice:

future PD(id,pr,Body,Attrs,evt) time = if time<0
then PD(id,pr,Body,Attrs,evt)
else PD(id,pr,Body,Attrs,evt + time)

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

pName (PD(id,pr,Body,Attrs,evt)) = id

ENTER en $\mathcal{EL}$**:** enters the event notice *en* into the event list $\mathcal{EL}$ ranked in according to its event time and priority, using the priority to resolve the order for entities with the same time. It

is entered **after** the last notice with an earlier time, and after all notices with the same time and equal or higher priority.

```
ENTER en []           =  [ en ]
ENTER en1 (en2::EL)   =  if evTime en1 < evTime en2
                           then en1::en2::EL
                           else
                             if evTime en1 = evTime en2
                             then if pPri en1 ≥ pPri en2
                               then en1::en2::EL
                               else en2::(ENTER en1 EL)
                             else en2::(ENTER en1 EL)
```

BENTER projF end $\mathcal{EL}$: enters the event notice data *end* into the blocked list $\mathcal{EL}$ ranked in according to its priority. The projection function allows us to order event notices by priority when the occur within complex data structures, such as the triples used to maintain synchronisations. This is the dual of enter when it is used to maintain blocked lists. We use the same data structure as this permits us to observe the amount of time an entity stays blocked for. Again at equality those objects already in the list take precedence. We pass in a projection function so this function can be used on more complex data structures using event notices, for standard usage the function is simply `pPri`.

```
BENTER projF end []           =  [ end ]
BENTER projF end1 (end2::EL)  =  if pPri(projF end1) ≥ pPri(projF end2)
                                   then end1::end2::EL
                                   else end2::(BENTER projF end1 EL)
```

ADDPrITEM(id, rt, vL) projF $\mathcal{S}$ We maintain lists of prioritised objects within our stores. In particular there a situations where we need to add items to a list which contain process descriptors respecting priorites. In this instance we must provide a projection function $Data \rightarrow evNote$, so we can locate the event notice within the data structure. *ADDPrITEM (id, rt, vL)* $\mathcal{S}$ If *(id,rt,oldvL)* $\in \mathcal{S}$, then it replaces the previous entry for *(id,rt)* with *(id,rt, imap (BENTER proj3) oldvL vL).*

2.4 Attributes

Every time an object releases a share in a resource or a synchronisation, we check that it did indeed own it at the time. It follows that, for each process object, we must keep track of resource seizures as they are made. The simplest way of doing this is to maintain a store of current attributes local to each process object. A major source of error in concurrent systems is the 'mislaying' of semaphores. By ensuring that only resources that are actually held can be freed, we can observe these errors before they propogate through the system. Further for convenience a notion of local variable greatly simplifies entity definitions, and these bindings will be stored as local attributes.

Attributes may be shares in resources, or synced processes. They are maintained in a store with three types, resource, syncs and variables. The syncs are stored as a list of event notices. We exploit the general store operations to maintain this type of store.

We define a couple of auxiliary functions to simplify the manipulation of attributes:

```
getAttrs PD(id,pr,Body,Attrs,evt)          =  Attrs
setAttrs PD(id,pr,Body,Attrs,evt) Attrs'   =  PD(id,pr,Body,Attrs',evt)
```

These functions allow us to get the store within a process descriptor and to set it to a new store.

2.5 Expression evaluation

Define the semantic function $\mathcal{EV}\ \ \mathcal{S}$ e, which given an expression e and a store $\mathcal{S}$ returns its value, straightforward recursion over the syntax. Letting N range over names, V range over values, Uop over unary operators, Bop over binary numerical values, and dis($\tilde{(p)}$) over distributions taking a parameter vector $\tilde{(p)}$ we obtain the following:

$\mathcal{EV}\ \ \mathcal{S}$ V = V
$\mathcal{EV}\ \ \mathcal{S}$ N = let val nm=evalN $\mathcal{S}$ N in
if (nm,var) $\in \mathcal{S}$ then LOOKUP (nm,var) $\mathcal{S}$
else if (nm,cons) $\in \mathcal{S}$ then $\mathcal{EV}\ \ \mathcal{S}$ (LOOKUP (nm,cons) $\mathcal{S}$)
else error unknown variable
$\mathcal{EV}\ \ \mathcal{S}$ Uop(ee) = Uop($\mathcal{EV}\ \ \mathcal{S}$ ee)
$\mathcal{EV}\ \ \mathcal{S}$ Bop(e1,e2) = Bop($\mathcal{EV}\ \ \mathcal{S}$ e1,$\mathcal{EV}\ \ \mathcal{S}$ e2)
$\mathcal{EV}\ \ \mathcal{S}$ DisPtr(nm) = let val (dn($\tilde{p}$),sd,sn)=(LOOKUP (nm, dist) $\mathcal{S}$)
in UPDATE(seed,var,sd) $\mathcal{S}$
in ret=sample(dn(map $\mathcal{EV}\ \mathcal{S}\tilde{(p)}$))
in UPDATE(nm,dist,(de,LOOKUP (seed,var) $\mathcal{S}$,sn+1)) $\mathcal{S}$
in ret

The above is standard except for the fact that constants are stored as expressions, and consequently re-evaluated. This permits us to bind distributions to constants and consequently re-sample them whenever the constant is re-evaluated. To allow us to sample distributions with well seperated seeds, each distribution expression is (essentially) conveted to a triple of the expression, its previous random drawing, and the number of times it has been sampled. We then set the seed to the previous draw *for this distribution* and then sample the distribution, as a side effect this updates seed as the mod-congruence generator defines. Subsequently we store the next draw values and the fact this distribution has been used one more time. See Section 11 for details of the sample functions used.

2.6 Expression conversion

Define the semantic function $\mathcal{EC}\ \ \mathcal{S}$ e, which given an expression e and a store $\mathcal{S}$ returns the expression with all of its fixed terms evaluated. This is used to minimize expressions within distributions as they are stored.

$\mathcal{EC}\ \ \mathcal{S}$ V = V
$\mathcal{EC}\ \ \mathcal{S}$ N = let val nm=evalN $\mathcal{S}$ N in
if (nm,var) $\in \mathcal{S}$ then LOOKUP (nm,var) $\mathcal{S}$
else if (nm,cons) $\in \mathcal{S}$ then $\mathcal{EV}\ \ \mathcal{S}$ (LOOKUP (nm,cons) $\mathcal{S}$)
else error unknown variable
$\mathcal{EC}\ \ \mathcal{S}$ Uop(ee) = Uop($\mathcal{EC}\ \ \mathcal{S}$ ee)
$\mathcal{EC}\ \ \mathcal{S}$ Bop(e1,e2) = if isVal(e1) andalso isVal(e2)
then Bop($\mathcal{EC}\ \ \mathcal{S}$ e1, $\mathcal{EC}\ \ \mathcal{S}$ e2)
else Bop(e1,e2)
$\mathcal{EC}\ \ \mathcal{S}$ Dis($\tilde{p}$) = Dis(map $\mathcal{EC}\ \ \mathcal{S}\ \tilde{p}$)

This function evaluates all of the subexpressions of a distribution expression and grounds them. The predicate isVal checks to see if a term does not containd a reference to a distribution sample and if so the expression can be evaluated.

2.7 Boolean evaluation

Define the semantic function $\mathcal{BV}\ \ \mathcal{S}$ b, which given a boolean expression b and a store $\mathcal{S}$ returns its value, straightforward recursion over the syntax. Allowing Bp to range over binary boolean operators, Up ranging over unary binary operators, Rop over relational operators then we can define the function as follows:

$\mathcal{BV}\ \ \mathcal{S}$ true = true
$\mathcal{BV}\ \ \mathcal{S}$ false = false
$\mathcal{BV}\ \ \mathcal{S}$ Up(be) = Up($\mathcal{BV}\ \ \mathcal{S}$ be)
$\mathcal{BV}\ \ \mathcal{S}$ Bp(b1,b2) = Bop($\mathcal{BV}\ \ \mathcal{S}$ b1, $\mathcal{BV}\ \ \mathcal{S}$ b2)
$\mathcal{BV}\ \ \mathcal{S}$ Rop(e1,e2) = Rop($\mathcal{EV}\ \ \mathcal{S}$ e1, $\mathcal{EV}\ \ \mathcal{S}$ e2)

2.8 Parametrised Names

We permit all of the names used for identifiers to have dependency on the system state. Essentially we have name arrays. The auxiliary function **evalN** takes an identifier and a store pair and returns a fixed name based on the state. Here the first store takes precedence as it represents local definitions. Hence in a context where the variable `i=3` then the identifier `a[i]` is the name `a[3]`.

evalP $\mathcal{S}$ e = "[" string($\mathcal{EV}$ e $\mathcal{S}$) "]"
evalN $\mathcal{S}$(fs s) = s
evalN $\mathcal{S}$(ps (s,eL)) = s (map (evalP $\mathcal{S}$) el)

2.9 Let

We use the *let* notation *let* $x = e$ *in* E to clarify the structure of complicated expressions, preferring, for example, to spell out

exec(cp'::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$ ++ (id, res,2))

in simple steps as

let $\mathcal{S}$' = $\mathcal{S}$ ++ (id, res, 2) **in**
let $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**
exec($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

The advantages are that state changes are reflected textually ($\mathcal{EL}$ and $\mathcal{R}$ have changed; $\mathcal{BL}$has not), and the **let**s are meaningful in both the functional and imperative programming styles.

3 Semantics of DEMOS

We can define the state of a DEMOS program as the product of the states of all of its constituents, both active and blocked, and the state of the store containing all of the data viz: class definitions, resource, bin synchronization states, variable and constant values.

So the program state is represented by a 3-tuple ($\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$) where:

$\mathcal{EL}$ is an event list which contains all the active (scheduled) processes, ranked according to the time of their next scheduled event and their priority . Processes are entered into $\mathcal{EL}$ on creation by *Entity*, rotated down $\mathcal{EL}$ by *hold*, and deleted from $\mathcal{EL}$ when their actions are exhausted. Within the process description the process that have been acquired through syncs

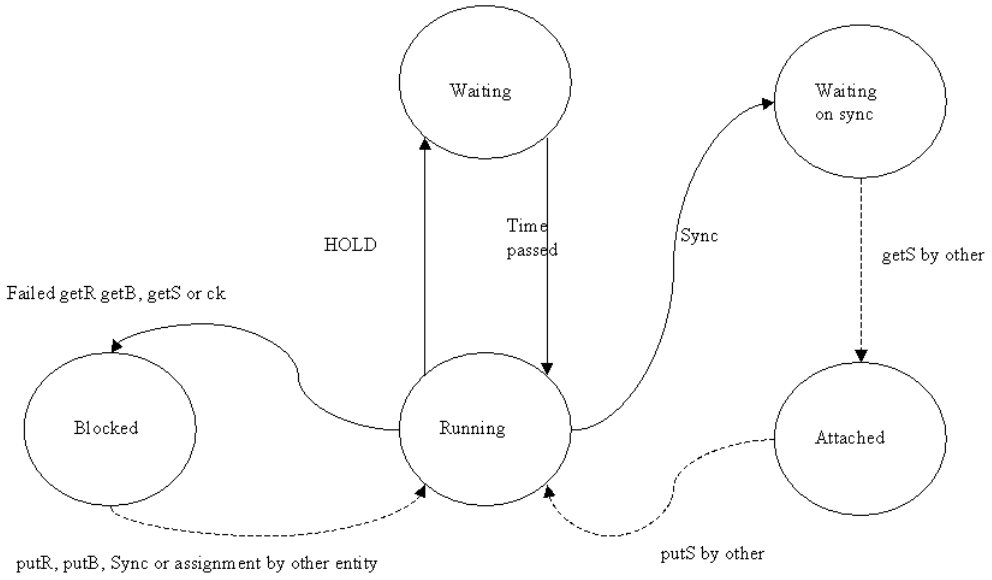

Figure 1: The states of a DEMOS 2000 Entity.

are held, so whilst they are passive they are held either within the store waiting on a getS or within the process whilst being used, until reactivated with a putS. See figure 1

$\mathcal{BL}$ is an event list which contains all the waiting processes, ranked according to their priority . Processes are entered into $\mathcal{BL}$ when any request for resource they make cannot currently be met, or a condition test fails. They are removed from the blocked list and returned to the event list when a resource freeing event, putR, putB, putBS, sync or a variable assignment occurs which causes their blocking condition to become satisfiable.

$\mathcal{S}$ is a store in which the free amounts of resources, bins, and syncs are maintained, the values of variable and constants, as well as the class definitions.

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 contains 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 semantics

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 permit ourselves to add some global information into the store at this point.

The cases are detailed each to its own subsection as indicated below.

```
exec (PD(c,pr,b::Body,Attrs,evt)::ℰℒ, ℬℒ, 𝒮0)
  =   let cp' = PD(c,pr,Body, Attrs, evt)
      𝒮= (UPDATE (DEMOS_TIME,var,evt) mk_av(𝒮0)) → 𝒮0 in
      (   case b of
              Class(id, Pl, Body)                          section 4
          |   Res (id, e)
          |   Bin (id, e)
          |   Entity (id, classId, Pl, dt, ResL, SynL)
          |   Var (id, e)
          |   LVar (id, e)
          |   Sync(id)
          |   Cons (id, e)

          |   hold dt                                      section 7
          |   priority e
          |   do (ec, acL)
          |   ido (n, acL)

          |   putB (id, m)                                 section 5
          |   putR (id, m)
          |   putS (id, m)
          |   putSv(id, eL)
          |   putBS(id, eL)
          |   ass(id, e)

          |   getB (id, m)                                 section 6
          |   getR (id, m)
          |   getS (id, m)
          |   getSv (id, bL, be)
          |   getBv (id, bL, be)
          |   Req(acqL)
          |   Cond(caL)
          |   while (acqL, aL)

          |   close                                        section 8
          |   seed
          |   trace(s, eL)

          |   anything else =error
          )
```

4 Declarations

4.1 Class

Put the definition of a class into the store.
Syntax: Class(n,pl,bd) - (parameters):

1. n is the name of the class possibly an array;

2. pl is a formal parameter list, a simple string, for later substitution either by value or name;

3. bd is the body of the class.

Semantics:

Class(n,pl,bd)

= (evalN (Attrs⊕$\mathcal{S}$) id,class) ∈ $\mathcal{S}$ ⇒ *error*

| **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**

let $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id,class,(pl,bd)) **in**

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

Interpretation:

1. (evalN id (Attrs⊕$\mathcal{S}$),class) ∈ $\mathcal{S}$: error if the class identifier is not fresh

2. *Normal case*

   (a) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction.

   (b) **let** $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, class, (pl,bd)) enter the new class declaration into the environment with the parameter list for later instantiation

   (c) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

4.2 Entity

Puts an entity whose actions are defined by a class into the event list, the entity is enabled at some point in the future. Further, the class definition has name parameters which are instantiated throughout its body at this point.

Syntax: Entity(eid,cn,pl,offset,reL,syL) - (parameters):

1. eid an extra identifier for use of the programmer, to distinguish entities from the same class if necessary;

2. cn the class to instantiate;

3. pl the parameter instantiations for the class;

4. offset the time after the current clock to start executing the entity;

5. reL a resource list [(rn,amt)] of resources claimed from the current entity to be passed to this new one;

6. syL a sync list [sync] of syncs claimed from the current entity to be passed to this new one, note [syncA,syncA] will hand over two instances of syncA;

Semantics:

Entity(eid,cn,pl,offset,reL,syL)
= (evalN (Attrs⊕$\mathcal{S}$) cn,class) $\notin \mathcal{S}$ $\Rightarrow$ *error*
| **let** rcn = evalN (Attrs⊕$\mathcal{S}$) cn **in**
let rid = evalN (Attrs⊕$\mathcal{S}$) eid **in**
let rL = map evalR (Attrs⊕$\mathcal{S}$) reL **in**
let sL = map evalN (Attrs⊕$\mathcal{S}$) syL **in**
let pil = map (evalN $\mathcal{S}$) pl **in**
let nbd = getBody(rcn,pil,state) **in**
let (cp",scL) = removeSL(sl,cp',[]) **in**
let act = evt + ($\mathcal{EV}$ Attrs⊕$\mathcal{S}$offset) **in**
let $\mathcal{EL}$' = (ENTER PD(rid-rcn,0,nbd,(rL⊕scL), act) removeRL(rL,cp")::$\mathcal{EL}$) **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)

Interpretation:

1. (evalN cn (Attrs⊕$\mathcal{S}$),class) $\notin \mathcal{S}$: error the class identifier does not exist;
2. *Normal case*
   (a) **let** rcn = evalN (Attrs⊕$\mathcal{S}$) cn: evaluate the class name to be used;
   (b) **let** rid = evalN (Attrs⊕$\mathcal{S}$) eid: evaluate the entities given name, so the programmer cvan recognise the entity;
   (c) **let** rL = map evalR (Attrs⊕$\mathcal{S}$) reL: evaluate the resource names and amounts that are being handed to the entity;
   (d) **let** sL = map evalN (Attrs⊕$\mathcal{S}$) syL: evaluate the names of the syncs that are being passed to the entity;
   (e) **let** pil = map (evalN $\mathcal{S}$) pl : evaluate the name parameters to instantiate the class parameters into;
   (f) **let** getBody(rcn,pil,state): get the entity body from the class store and perform the name substitution throughout.
   (g) **let** (cp",scL) = removeSL(sl,cp',[]); remove the sync notices from the current entity and update its store, also return a store of the passed on sync notices;
   (h) **let** act = evt + ($\mathcal{EV}$ Attrs⊕$\mathcal{S}$offset): compute the start time for this entity;
   (i) **let** $\mathcal{EL}$' = (ENTER PD(rid-rcn,0,nbd,(rL⊕scL), act) removeRL(rL,cp")::$\mathcal{EL}$move past this instruction and add the new event notice at the appropriate point in the event list, with the store formed from the passed over resources and syncs.
   (j) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)

4.3 Res

Define a resource, similar to a multi value semaphore.
Syntax: Res(n,amt) - (parameters):

1. n the name of the resource, possibly an array;
2. amt the amount of this resource, notice this maximum cannot be changed later;

Semantics:

Res(id,e)

= (evalN (Attrs⊕$\mathcal{S}$) id, res) ∈ $\mathcal{S}$ ⇒ *error*

| **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**

let amt=$\mathcal{EC}$ $\mathcal{S}$ e **in**

let $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id,res,amt) **in**

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

Interpretation:

1. (evalN (Attrs⊕$\mathcal{S}$) id, res) ∈ $\mathcal{S}$: error if the resource identifier is not fresh
2. *Normal case*
   (a) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction.
   (b) **let** amt=$\mathcal{EC}$ $\mathcal{S}$ e how much resource is specified
   (c) **let** $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, res, amt) enter the new resource declaration into the store with the currently available amount of that resource;
   (d) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

4.4 Bin

Define a new bin, similar to a queue or a place in a Petri net
Syntax: Bin(n,amt) - (parameters):

1. n the name of the resource, possibly an array;
2. amt the initial amount in this bin.

Semantics:

Bin(id,e)

= (evalN (Attrs⊕$\mathcal{S}$) id, bin) ∈ $\mathcal{S}$ ⇒ *error*

| **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**

let amt=$\mathcal{EC}$ $\mathcal{S}$ e **in**

let $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, bin,amt) **in**

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

Interpretation:

1. (evalN (Attrs⊕$\mathcal{S}$) id, bin) ∈ $\mathcal{S}$: error if the resource identifier is not fresh
2. *Normal case*
   (a) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction.
   (b) **let** amt=$\mathcal{EC}$ $\mathcal{S}$ e amount of the initial bin contents as specified
   (c) **let** $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, res, amt) enter the new resource declaration into the store with the currently available amount of that resource;
   (d) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

4.5 Sync

Place the current process on the sync list indexed by the sync name.
Syntax: Sync(n) - (parameter):

1. n the name of the sync.

Semantics:
Sync(id)
let $\mathcal{S}$' = ADDPrITEM(evalN (Attrs$\oplus\mathcal{S}$) id, sync, [([],[],PD(cn,pr,Body,Attrs,evt))] val3) $\mathcal{S}$ **in**
promote(Sync(evalN (Attrs$\oplus\mathcal{S}$)) id, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$')

Interpretation:

1. **let let** $\mathcal{S}$' = ADDPrITEM(evalN (Attrs$\oplus\mathcal{S}$) id, sync, [([],[],PD(cn,pr,Body,Attrs,evt)) val3]) $\mathcal{S}$add this entity to those queuing on the sync name id, if there is none then the name is created, *val3* projects out the third value in a tuple. This is a trivial version of a value sync. The entity is inserted into those waiting on the sync in priority order followed by arrival order.

2. perform a promotion on the new state ($\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$') as some other process may now be unblocked. We need to take note of the new resource which has become available to correctly unblock processes with potentially multiple requirements.

4.6 SyncV

Place the current process on the sync list indexed by the sync name, with output expressions and input variable names.
Syntax: SyncV(n, expL, vL) - (parameters):

1. n the name of the sync, possibly an array;

2. expL an expression list of output values, eg [i+j,3];

3. vL a local variable list for return values.

Semantics:
SyncV(id,exL,vL)
let evl = map ($\mathcal{EV}$ Attrs$\oplus\mathcal{S}$) exL **in**
let enl = map (evalN Attrs$\oplus\mathcal{S}$) vL **in**
let $\mathcal{S}$' = ADDPrITEM(evalN (Attrs$\oplus\mathcal{S}$) id, sync, [(evl,enl,PD(cn,pr,Body,Attrs,evt))] val3) $\mathcal{S}$ **in**
promote(Sync(evalN (Attrs$\oplus\mathcal{S}$)) id, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$')

Interpretation:

1. **let** evl = map ($\mathcal{EV}$ Attrs$\oplus\mathcal{S}$) eL: compute tha values bound to the value sync;

2. **let** enl = map (evalN Attrs$\oplus\mathcal{S}$) vL: compute the names of the variables to be bound on return from the sync;

3. **let** $\mathcal{S}$' = ADDPrITEM(evalN (Attrs$\oplus\mathcal{S}$) id, sync, [(evl,enl,PD(cn,pr,Body,Attrs,evt)) val3]) $\mathcal{S}$add this entity to those queuing on the sync name id, if there is none then the name is created, *val3* projects out the third value in a tuple. As part of the store we keep lists of the values bound in the sync and the parameter names in which returned values are to be stored. The entity is inserted into those waitin on the sync in priority order followed by arrival order.

4. perform a promotion on the new state ($\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$') as some other process may now be unblocked. We need to take note of the new resource which has become available to correctly unblock processes with potentially multiple requirements.

4.7 Cons

Enter the definition of the constant into the store.
Syntax: Cons(n,e) - (parameters):

1. n the name of the constant, possibly an array;
2. e an expression;

Semantics:

Cons(id,e)
= (evalN (Attrs⊕$\mathcal{S}$) id, cons) ∈ $\mathcal{S}$ ⇒ *error*
| **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**
let $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, cons, DisC $\mathcal{S}$($\mathcal{EC}$ $\mathcal{S}$e)) **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

Interpretation:

1. (evalN (Attrs⊕$\mathcal{S}$) id, cons) ∈ $\mathcal{S}$: error if the constant identifier is not fresh
2. *Normal case*
   (a) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction.
   (b) **let** $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, cons, DisC $\mathcal{S}$($\mathcal{EC}$ $\mathcal{S}$e)) enter the new constant declaration into the store as a symbolic constant. This permits us to re-evaluate the constant when used. Hence bindings of the form `Cons n=nege(10)`. Will have the expected effect of resampling the distribution each time they are used. Constants in Demos2000 are fixed bindings over a system run, not necessarily fixed values. The function DisC converts the distributions into a triple form of (de,cr,0) with de the distribution expression, cr the current seed for this distribution, and the 0 representing the number of times it has been sampled. We use this to establish well seperated seeds for each of our distributions.
   (c) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

4.8 Var

Enter the definition of the variable into the store.
Syntax: Var(n,e) - (parameters):

1. n the name of the variable, possibly an array;
2. e an expression;

Semantics:

Var(id,e)
= (evalN (Attrs⊕$\mathcal{S}$) id, vars) ∈ $\mathcal{S}$ ⇒ *error*
| **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**
let $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, vars, $\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e) **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

Interpretation:

1. (evalN (Attrs⊕$\mathcal{S}$) id, vars) ∈ $\mathcal{S}$: error if the variable identifier is not fresh;
2. *Normal case*
   (a) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction.
   (b) **let** $\mathcal{S}$' = $\mathcal{S}$++(evalN (Attrs⊕$\mathcal{S}$) id, vars, $\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e) enter the new variable declaration into the store with its current value
   (c) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

4.9 LVar

Enter the definition of the local variable into the entity attributes.
Syntax: Lvar(n,e) - (parameters):

1. n the name of the local variable, possibly an array;
2. e an expression;

Semantics:

LVar(id,e)
= (evalN (Attrs⊕$\mathcal{S}$) id, vars) ∈ Attrs ⇒ *error*
| **let** Attrs' = Attrs ++ (evalN (Attrs⊕$\mathcal{S}$) id, vars, $\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e **in**
let $\mathcal{EL}$' = PD(c,pr,Body,Attrs',evt)::$\mathcal{EL}$ **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)

Interpretation:

1. (evalN (Attrs⊕$\mathcal{S}$) id, vars) ∈ Attrs: error if the variable identifier is not fresh;
2. *Normal case*
   (a) **let** Attrs'=Attrs++(evalN (Attrs⊕$\mathcal{S}$) id, vars, $\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e: add this variable definition to the local store with its new value;
   (b) **let** $\mathcal{EL}$' = PD(c,pr,Body,Attrs',evt)::$\mathcal{EL}$: move past this instruction.
   (c) return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)

5 Assignments

5.1 Variables

Change the value of the variable. Syntax: Ass(x,e) - (parameters):

1. x the name of the constant, possibly an array;
2. e an expression;

Semantics:

Ass(x,e)
= (evalN (Attrs⊕$\mathcal{S}$) id, vars) ∉ $\mathcal{S}$ ⇒ *error*
| **let** Name=evalN (Attrs⊕$\mathcal{S}$) id **in**
| **let** (Attrs',$\mathcal{S}$')=Do_assign(Name,$\mathcal{EV}$ Attrs⊕$\mathcal{S}$ e, Attrs,$\mathcal{S}$ **in**
let $\mathcal{EL}$' = PD(c,pr,Body,Attrs',evt)::$\mathcal{EL}$ **in**
promote(Ass(Name,$\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e), evt, $\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')

Interpretation:

1. (evalN (Attrs$\oplus\mathcal{S}$) id, vars) $\notin \mathcal{S}$: error if the variable identifier does not exist;

2. *Normal case*

   (a) **let** Name=evalN (Attrs$\oplus\mathcal{S}$) id: evaluate and store the name for future use.

   (b) **let** (Attrs',$\mathcal{S}$')=Do_assign(Name,$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e,Attrs,$\mathcal{S}$) enter the new value for the variable into the correct store, obviously only one of these will be changed;

   (c) **let** $\mathcal{EL}$' = PD(c,pr,Body,Attrs',evt)::$\mathcal{EL}$: the local store may have changed so reconstruct the event list;

   (d) use the semantic function promote(Ass(Name,$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e), evt, $\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$') to enable any entities that may have become unblocked, see section 9.3 for details. Use the evaluated name and expression to save time, and avoid local name clashes.

5.2 putR

Return an amount of resource.
Syntax: putR(n,e) - (parameters):

1. n the name of the resource, possibly an array;

2. e is the amount to return;

Semantics:

```
putR(id,e)
  =  (evalN id (Attrs⊕S), res) ∉ S⇒ error
  |  let n = evalN id (Attrs⊕S)                          in
     let v=EV (Attrs⊕S) e                                in
     if pr=-15000 then
     let S' = UPDATE(n, res, v + LOOKUP(n,res) S) S      in
         promote(putR(n,v), evt, EL, BL, S')
     else
     let EL' = removeR(n, v, cp')::EL                    in
     in (ENTER PD(cn,-15000,[putR(n,v)],(),evt) EL',BL,S)
```

Interpretation:

1. (evalN (Attrs$\oplus\mathcal{S}$) id, res) $\notin \mathcal{S}$: error if the resource identifier does not exist;

2. *Normal case*

   (a) **let** n = evalN (Attrs$\oplus\mathcal{S}$) id: work out the resource name;

   (b) **let** v=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e: and the amount to return;

   (c) if pr=-15000: then this is a delayed return so

   (d) **let** $\mathcal{S}$' = UPDATE(n, res, v + LOOKUP(n,res) $\mathcal{S}$) $\mathcal{S}$add the resources back if the previous function did not fail;

   (e) use the semantic function promote(putR(n,v), evt, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$') to enable any entities that may have become unblocked, see section 9.3 for details. Notice this must be a 1 action entity so it immediately dies.

   (f) otherwise

(g) **let** $\mathcal{EL}$' = removeR(n, v, cp')::$\mathcal{EL}$: move past this instruction, and remove the resources deallocated from the entities own store. If they are not held then this will generate the error, see section 9.5;

(h) (ENTER PD(cn,-15000,[putR(n,v)],(),evt) $\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$): and enqueue the return at a priority lower than that which can be assigned to any other entity, to ensure that this is the last activity (other than other returns) undertaken at this clock instant.

5.3 putB

Add an amount to a bin. Syntax: putB(n,e) - (parameters):

1. n the name of the bin, possibly an array;
2. e is the amount to add;

Semantics:

putB(id,e)
= (evalN (Attrs⊕$\mathcal{S}$) id, bins) $\notin \mathcal{S} \Rightarrow$ *error*
| **let** n = evalN (Attrs,$\mathcal{S}$) id **in**
let v=$\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e **in**
if pr=-15000 then
let $\mathcal{S}$' = UPDATE(n, bins, v + LOOKUP(n,bins) $\mathcal{S}$) $\mathcal{S}$ **in**
promote(putB(n,v), evt, $\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')
else
let $\mathcal{EL}$' = cp'::$\mathcal{EL}$ **in**
in (ENTER PD(cn,-15000,[putB(n,v)],(),evt) $\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$)

Interpretation:

1. (evalN id (Attrs⊕$\mathcal{S}$), bins) $\notin \mathcal{S}$: error if the bin identifier does not exist;
2. *Normal case*

(a) **let** n = evalN (Attrs⊕$\mathcal{S}$)id: work out the bin name;

(b) **let** v=$\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e: and the amount to return;

(c) if pr=-15000 then: so we actually perform the return

(d) **let** $\mathcal{S}$' = UPDATE(n, bins, v + LOOKUP(n, bins) $\mathcal{S}$) $\mathcal{S}$add the items to the bin;

(e) use the semantic function promote(putB(n,v), evt, $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$') to enable any entities that may have become unblocked, see section 9.3 for details. Use evaluated forms to save compute and then the current entity must die.

(f) else

(g) **let** $\mathcal{EL}$' = cp'::$\mathcal{EL}$: move past this instruction;

(h) (ENTER PD(cn,-15000,[putB(n,v)],(),evt) $\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$): put in an entity to do the return as the last activity at this instant.

5.4 putS

Return a number of syncs.
Syntax: putS(id,e) - (parameters):

1. id the name of the sync, possibly an array;
2. e is the number to return;

Semantics:
putS(id,e)
= **let** n = evalN (Attrs⊕$\mathcal{S}$) id **in**
let v=$\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e **in**
let (fp,cp")=removeS(n,v,cp') **in**
let $\mathcal{EL}$' = cp"::(imap ENTER fp $\mathcal{EL}$) **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation:

1. **let** n = evalN (Attrs⊕$\mathcal{S}$) id: work out the sync name;
2. **let** v=$\mathcal{EV}$ (Attrs⊕$\mathcal{S}$) e: and the amount to return;
3. **let** (fp,cp")=removeS(n,v,cp'): compute the new process descriptor for this process cp" given by removing v of the entities synced on n from it, if held, returning a list of those freed processes, see section 9.6; otherwise raise error;
4. **let** $\mathcal{EL}$' = cp"::(map ENTER fp $\mathcal{EL}$) use the new process descriptor for this entity and leave it a the live entity, add the freed entities in the appropriate order into the remainder of the event list;
5. **let** $\mathcal{S}$' = UPDATE(n, res, v + LOOKUP(n,res) $\mathcal{S}$) $\mathcal{S}$add the resources back if the previous function did not fail;
6. return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$), note unlike other assignments putS cannot cause any other process to become unblocked. The unblocking causation on getS is the use of a sync to create a potential slave entity.

5.5 putSv

Return an entity synchronized passing in return values.
Syntax: putSv(id,exL) - (parameters):

1. id the name of the sync, possibly an array;
2. exL is a return value list, e.g [i+k,2,5], which is bound to local variables in the freed entity;

Semantics:
putSv(id,eL)
= **let** n = evalN (Attrs⊕$\mathcal{S}$) id **in**
let evL= map ($\mathcal{EV}$ (Attrs⊕$\mathcal{S}$)) exL **in**
let (prp,cp")=removeSv(n,v,cp',$\mathcal{S}$) **in**
let $\mathcal{EL}$' = cp"::(ENTER prp $\mathcal{EL}$) **in**
($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation:

1. **let** n = evalN (Attrs⊕$\mathcal{S}$) id: work out the sync name;
2. **let** evL= map ($\mathcal{EV}$ (Attrs⊕$\mathcal{S}$)) exL : and the values to return;
3. **let** (fp,cp")=removeSv(n,evL,cp',state): compute the new process descriptor for this process cp" given by removing an entity synced on n from it, if held; otherwise raise error, return the freed entity, see section 9.6. Note the internal bindings of the variables of the entity may (should) have been changed by the returning value list;
4. **let** $\mathcal{EL}$' = cp"::(ENTER prp $\mathcal{EL}$) use the new process descriptor for this entity and leave it a the live entity, add the freed entities in the appropriate order into the remainder of the event list;
5. **let** $\mathcal{S}$' = UPDATE(n, res, v + LOOKUP(n,res) $\mathcal{S}$) $\mathcal{S}$add the resources back if the previous function did not fail;
6. return the new state ($\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$), note unlike other assignments putS cannot cause any other process to become unblocked. The unblocking causation on getS is the use of a sync to create a potential slave entity.

5.6 putBS

Similar to syncV but can pass not only values but also resources and syncs. This permits entities to move bound objects to other entities. Obviously to prevent errors these bindings must be tracked. For the resource and sync hanover see entity creation.
Syntax: putBS(id,exL,reL,sL) - (parameters):

1. id the name of the bin, possibly an array;
2. exL an expression list evaluated to give values;
3. reL a resource list, see entity creation;
4. sL a sync list, also see entity creation;

Semantics:

putBS(id,exL,reL,sL)
= (evalN (Attrs⊕$\mathcal{S}$) id, bins) $\notin \mathcal{S} \Rightarrow$ *error*
| **let** n = evalN (Attrs⊕$\mathcal{S}$) id **in**
let vL = map evalN (Attrs⊕$\mathcal{S}$) exL **in**
let rL = map evalR (Attrs⊕$\mathcal{S}$) reL **in**
let sL = map evalN (Attrs⊕$\mathcal{S}$) 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⊕$\mathcal{S}$) id, bins) $\notin \mathcal{S}$: error if the bin identifier does not exist;
2. *Normal case*

(a) **let** n = evalN (Attrs⊕$\mathcal{S}$) id : work out the resource name;

(b) **let** vL = map evalN (Attrs⊕$\mathcal{S}$) exL: evaluate the expressions passed with the bin item;

(c) **let** rL = map evalR (Attrs⊕$\mathcal{S}$) reL: evaluate the amounts of each resource passed along with the bin item;

(d) **let** sL = map evalN (Attrs⊕$\mathcal{S}$) sL: and work out the ground sync names;

(e) if pr=-15000 then: we are actually doing the put;

i. **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;

ii. 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, see section 9.3 for details. 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.

(f) else

(g) **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;

(h) (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.

6 Claims

6.1 getR

Attempt to claim an amount of a resource, if that is not available then wait it until it is. Waiting entities will be satisfied on a priority/FCFS basis:
Syntax: getR(id,v) - (parameters):

1. id the name of the resource, possibly an array;

2. v is the amount to claim;

Semantics:
getR(id,v)
= exec(PD(c,pr,[Cond[([b],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: we take advantage on the normalization observed in [5], and wrap the request up into a conditional, and recall the state evaluation function. We must make sure that the name and value are evaluated now, to fix the current context. If this is blocked then values may change, leading to an incorrect getR. Note b is the current atomic action.

6.2 getB

Attempt to claim an amount of a bin, if that is not available then wait it until it is. Waiting entities will be satisfied on a priority/FCFS basis:
Syntax: getB(id,v) - (parameters):

1. id the name of the resource, possibly an array;

2. v is the amount to claim;

Semantics:
getB(id,v)
= exec(PD(c,pr,[Cond[([b],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$')
Interpretation: see getR above.

6.3 getS

Attempt to claim an amount of a sync, if that is not available then wait it until it is. Waiting processes will be satisfied on a priority/FCFS basis:
Syntax: getS(id,v) - (parameters):

1. id the name of the sync, possibly an array;
2. v is the number to claim;

Semantics:
getS(id,v)
= exec(PD(c,pr,[Cond[([b],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: see getR above.

6.4 getSv

Attempt to claim a sync with values which when bound locally permits a condition (filter) to return true. If no satisfying sync is available then wait it until it is. Waiting processes will be satisfied on a priority/FCFS basis:
Syntax: getSv(id,vL,cc) - (parameters):

1. id the name of the sync, possibly an array;
2. vL is a local variable list to bind its output values to;
3. cc is a boolean expression which when evaluated (possibly) using the local variables above must return true. Obviously getSv(id,vL,true) will take the first sync item and bind the variables.

Semantics:
getSv(id,vL,cc)
= exec(PD(c,pr,[Cond[([b],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: see getR above.

6.5 getBv

Attempt to get a parameterised item from a bin such that when the local variables in a binding list are bound in order to its parameters a condition evaluates true. If there are many items in the bin move down it until the first one that satisfies is found. In that case the local variables are now bound to the values observed, otherwise wait until such an item becomes available. Waiting entities will be satisfied on a priority/FCFS basis. As a side effect any resources or syncs bound to this bin item will be added to this entity. To disambiguate the values are used to indicate what, if any, extra resources came with the bin item.
Syntax: getBv(id,bL,be) - (parameters):

1. id the name of the sync, possibly an array;

2. bL is a local variable list to bind its output values to;

3. be is a boolean expression which when evaluated (possibly) using the local variables above must return true. Obviously getBv(id,bL,true) will take the first sync item and bind the variables.

Semantics:
getBv(id, bl, be)
= exec(PD(c,pr,[Cond[([b],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: see getR above.

6.6 Req

Only proceed when all of its claims can be meet **simultaneously**. This is the representation of the infamous *waituntil*. Notice we can use single acquisitions, and this permits boolean expressions as an acquisition type.
Syntax: Req(acqL)- (parameter):

1. acqL is a list of acqusisitions - getR, getB, getS, getSv, getBv, ck.

Semantics:
Req(acqL)
= exec(PD(c,pr,[Cond[([acqL],Body)]],Attrs,evt)::$\mathcal{EL}$', $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: see getR above.

6.7 Cond

This is similar to a case statement. The condL consists of pairs of claim lists and action lists. We check each of the claim lists in order until one can be satisfied, we then proceed by executing the body of that claim in the state resulting from performing all of the claims. If none of the claim lists can be satisfied then we wait. This is essentially non-determinism with a synchronous process algebra [8, 12, 9] which is known to be an extremely expressive form [12].
Syntax: Cond(condL)- (parameter):

1. condL is a list of pairs (acqL,bd) - where acqL is an acquisition list (see req) and bd a command list. Note that the pair ([],bd) will **always satisfy** and can be used to form non-blocking tests.

Semantics:
Cond(condL)
= if cBL(Cond(condL),$\mathcal{BL}$,(Attrs,$\mathcal{S}$))
then ($\mathcal{EL}$, BENTER(PD(c,pr,Cond(condL)::Body,Attrs,evt),$\mathcal{BL}$), $\mathcal{S}$)
else **let** (fire, (Attrs', $\mathcal{S}$'), fiB)=doCond(evt,c,acqL,(Attrs,$\mathcal{S}$)) **in**
if fire
then let $\mathcal{EL}$'=PD(c,pr,fiB@Body,Attrs',evt)::$\mathcal{EL}$ **in**
($\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$')
else let $\mathcal{BL}$'=BENTER(PD(c,pr,Cond(L)::Body,Attrs,evt),Bl) **in**
($\mathcal{EL}$,$\mathcal{BL}$',$\mathcal{S}$)
Interpretation:

1. if cBL(Cond(acqL),$\mathcal{BL}$,$\mathcal{S}$): we can cause starvation by permitting entities in the event list to have implicit priority over those that are blocked, so if we depend on a claim that something in the blocked list is waiting on then we **must not attempt a claim**

(a) true: then ($\mathcal{EL}$, BENTER(PD(c,pr,Cond(L)::Body,Attrs,evt),$\mathcal{BL}$), $\mathcal{S}$) add this process to the blocked list and return the new state;

(b) false: else **let** (fire, (Attrs',$\mathcal{S}$'), fiB)=doCond(evt,c,condL,(Attrs, $\mathcal{S}$)): attempt to find a list of acquisitions within the Cond list that can fire, return that in fire; if fire is true then $\mathcal{S}$' is the store after the claims have been evaluated, Attrs' is the change to the local attributes (added resources or syncs) and fiB is the action list that should now be executed. See section 9.2 for a detailed explanation;

(c) if fire: check to see if a condition can be satisfied;

i. true: then let $\mathcal{EL}$'=(PD(c,pr,fiB@Body,Attrs',evt)::$\mathcal{EL}$ and return the state ($\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$'): we reconstruct the current process descriptor adding the claimed items to its attribute list (Attrs), putting the action body to be executed in front of the remainder of the entity, and continuing using the updated store;

ii. false: else let $\mathcal{BL}$'=BENTER(PD(c,pr,Cond(L)::Body,Attrs,evt),$\mathcal{BL}$) and return ($\mathcal{EL}$,$\mathcal{BL}$',$\mathcal{S}$): none of the Conds acquisition choices can be satisfied, so block this process in its current state adding it to the blocked list, and leave the store and the remainder of the event list unchanged.

Note we can construct non-blocking tests by using the following Cond([([test],[body]),([],[])]). Which returns true trivially if test fails and then continues.

6.8 While

As long as the claim list can be satisfied **immediately** then execute a body.
Syntax: While(acqL,wB)- (parameters):

1. acqL is a list of acqusisitions - see req;
2. wB the command list to execute.

Semantics:

While(acqL,wB)
= if cBL(Cond([(acqL,wB)]),$\mathcal{BL}$,(Attrs,$\mathcal{S}$))
then (cp'::$\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$)
else let (fire, (Attrs', $\mathcal{S}$'), fiB)=doCond(evt,c,[(acqL,wB)],(Attrs, $\mathcal{S}$)) **in**
if fire
then let $\mathcal{EL}$'=PD(c,pr,fiB@While(acqL,wB)::Body,Attrs',evt)::$\mathcal{EL}$ **in**
($\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$')
else (cp'::$\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$)

Interpretation:

1. if cBL(Cond(acqL),$\mathcal{BL}$,(Attrs,$\mathcal{S}$)): as Cond we must not give implicit priority to 'live' entities:

(a) true: (cp'::$\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$) carry on executing we cannot satisfy the while condition

(b) false: else **let** (fire, (Attrs', $\mathcal{S}$'), fiB)=doCond(evt,c,[(acqL,wB)],(Attrs, $\mathcal{S}$)): as for Cond See section 9.2 for a detailed explanation;

(c) if fire: check to see if a condition can be satisfied;

i. true: then let $\mathcal{EL}$'=(PD(c,pr,fiB@While(acqL,wB)::Body,Attrs',evt)::$\mathcal{EL}$ and return the state ($\mathcal{EL}$',$\mathcal{BL}$,$\mathcal{S}$'): see Cond; we reconstruct the process descriptor and continue with the new store state $\mathcal{S}$';

ii. false: else (cp'::$\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$): ignore the While if it cannot execute its condition.

Note endless repetition is achieved by While([],body).

7 Time and Priority

7.1 HOLD

This entity will wait until the system clock advances by a time. This is considered to be 'active' waiting as opposed to passive waiting when the entity is blocked on some claim.
Syntax: HOLD(e)- (parameter):

1. e the time to wait into the future, an offset not absolute.

Semantics:
HOLD(e)
= (ENTER (FUTURE cp',($\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e) $\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: move the current process descriptor into the future by the current evaluation of e. Will return an error in e is negative.

7.2 PRIORITY

Set the priority level of the current entity, used in resolving equality in event list, blocked list and synchronisation lists.
Syntax: PRIORITY(e)- (parameter):

1. e the level to set priority to -10000 to intMax.

Semantics:
PRIORITY(e)
= (PD(c,lim($\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e), Body, Attrs, evt)::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: set the process priority to e evaluated in the current state, up to a limit of -10000 imposed by the function lim. We need to create processes with lower priority to delay returns to the completion of all activity in the current cycle.

7.3 DO

Repeat an action list a fixed number of times, a clearer form of for next loop, no ambiguity over variable change during loop execution.
Syntax: DO(e, acL)- (parameters):

1. e the number of times to execute the body, evaluated at the start;

2. acl the actions to execute.

Semantics:
DO(e,acL)
= then exec(PD(c,pr,IDO($\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e,acL)@Body, Attrs, evt)::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
else exec(cp'::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: evaluate the number of loops required and use the internal evaluated form.

7.4 IDO

Internal form for evaluated version of above.
Syntax: IDO(n, acL)- (parameters):

1. n a number of times to execute the body;
2. acl the actions to execute.

Semantics:
IDO(n,acL)
= if n>0
then exec(PD(c,pr,acL@IDO(n-1,acL)@Body, Attrs, evt)::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
else exec(cp'::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
Interpretation:

1. if n>0: number of loops left to execute;
   (a) true:exec(PD(c,pr,acL@Do(n-1,acL)@Body, Attrs, evt)::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$) carry on executing the body of the loop acL, then recheck the loop, then perform the rest of the entities actions;
   (b) false: execute the next command.

8 Maintenance

8.1 TRACE

Generate information about the state of the system. This has purely side effect in the form of output, which is annotated with the current time, the supplied string and any parameters supplied.
Syntax: TRACE(s, eL)- (parameters):

1. s a string with escapes. Currently has escape sequences:
   (a) %v print out an expression from an expression list in order;
   (b) %n print out current process name;
   (c) %s print out current process syncs held;
   (d) %r print out current process resources held;
2. el the expressions to evaluate for the %v substitutions in order.

Semantics:
TRACE(s,eL)
= (cp'::$\mathcal{EL}$, $\mathcal{BL}$, $\mathcal{S}$)
Interpretation: do **not** change the state.

8.2 SEED

Set the main random number seed. Must be at the start, once random samplers are defined they have the own well seperated seeds generated from this one.
Syntax: SEED(e)- (parameter):

1. e an expression which evaluates to a seed value.

Semantics:

SEED(e)

= **let** sv=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e **in**

| **let** $\mathcal{S}$')= UPDATE(seed,var,sv) $\mathcal{S}$ **in**

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

Interpretation: set the random number seed to this value. Each time a randome number is needed from the system the following number generator is used

$$\begin{aligned} a &= 16807.0 \\ m &= 2147483647.0 \\ t &:= (a * seed) \\ seed &:= t - m * real(floor(t/m)) \end{aligned}$$

With *seed* having the new random value. By using this mod-congruence psuedo random sequence generator and the standard log sample generators for the various supported theoretic distributions it is possible to completely replicate the behaviour of the reference DEMOS-2000 implementation in any other. Initially *seed* defaults to 120. In the section on sampling it is this function that is called to generate U. Slightly more formally

U $\mathcal{S}$ = cr:=LOOKUP (seed,var) $\mathcal{S}$
t:=(a*cr)
nr:=t -m * real(floor(t/m))
UPDATE (seed,var,nr) $\mathcal{S}$
nr

So a call to the random function U (we onit the $\mathcal{S}$ for brevity) has the side effect of advancing the variable seed to the next number in the psuedo-random sequence. This is used to keep the sequences seperate for each distribution used within demos.

8.3 CLOSE

This simply stops execution elegantly. Formally it is an indication to the wrapper of the exec to stop. It has no effect on state and no parameters.

9 Auxiliary Semantic functions

To simplify the semantics many of the state maintenance functions were abstracted as they appear within more than one semantic rule.

9.1 Do_assign

This function takes a name an value a local store and a global store if the name is a var in the local store then it is updated else the global store is. Note an error will be returned if the name is found in neither.

Do_assign(n,v,Attrs,$\mathcal{S}$) = if (n,vars) $\in$ Attrs then (UPDATE(n,vars,v) Atrrs,$\mathcal{S}$)
else if (n,vars) $\in \mathcal{S}$ then (Attrs, UPDATE(n,vars,v) $\mathcal{S}$)
else **error** Unknown variable

9.2 doCond

We need to check a list of claim lists and action bodies in order to see if the claim can be met in the current state. If it can then we need to know about resources and syncs that have been allocated

to update the state of the calling process, otherwise we reach the end of the list and none of the current claims can be met.

doCond(evt,pn,[], $\mathcal{SP}$) = (false, $\mathcal{SP}$, [])
doCond(evt,pn,(fc,bd)::t,$\mathcal{SP}$) = if checkL(fc,$\mathcal{SP}$)
then let $\mathcal{SP}$'=getResL(evt, pn, fc, $\mathcal{SP}$)
in (true, $\mathcal{SP}$', bd)
else doCond(evt,pn,t,$\mathcal{SP}$)

Explanation:

1. doCond(evt,pn,[],$\mathcal{S}$)=(false, $\mathcal{SP}$, []): reached the end of the list, no claim could be met, so state is unchanged and no resources or syncs were allocated;
2. doCond(evt,pn,(fc,bd)::t,$\mathcal{S}$): the current claim is bound to fc, and if it can be meet then we should continue execution with bd;
3. if checkL(fc,$\mathcal{S}$): can the current claim be met in the current state;
   (a) true: let $\mathcal{SP}$' =getResL(evt, pn, fc, $\mathcal{SP}$): work out the change to the state of making the claims, return the new state, return (true, $\mathcal{SP}$', bd) denoting that firing was successful, the stores changed to $\mathcal{SP}$', and the action list to continue execution bd;
   (b) false: try the next claim execution pair, doCond(evt,pn,t,$\mathcal{SP}$) the store pair is unchanged.

checkL([],$\mathcal{S}$) = true
checkL(h::t,$\mathcal{S}$) = if checkReq(h,$\mathcal{S}$) then checkL(t,$\mathcal{S}$)
else false

Explanation: check that each request can be met, if any fail then the whole fails:

Predicate to test if an atomic claim can be met by a particular store state:

checkReq(getR(n,e),(Attrs, $\mathcal{S}$))
= LOOKUP(evalN (Attrs$\oplus\mathcal{S}$) n, res) $\mathcal{S}$>=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
checkReq(getB(n,e),(Attrs, $\mathcal{S}$))
= length(LOOKUP(evalN (Attrs$\oplus\mathcal{S}$) n,bins) $\mathcal{S}$)>= $\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
checkReq(getBv(n,vL,cc),(Attrs, $\mathcal{S}$))
= checkBVReq(LOOKUP(evalN(n,(Attrs,$\mathcal{S}$)),bins) $\mathcal{S}$), map (evalN (Attrs$\oplus\mathcal{S}$)) vL, cc,(Attrs,$\mathcal{S}$))
checkReq(getSv(n,vL,cc),(Attrs, $\mathcal{S}$))
= checkSVReq(LOOKUP(evalN(n,(Attrs,$\mathcal{S}$)),bins) $\mathcal{S}$), map (evalN (Attrs$\oplus\mathcal{S}$)) vL,cc,(Attrs,$\mathcal{S}$))
checkReq(getS(n,e),(Attrs, $\mathcal{S}$))
= length(LOOKUP(evalN (Attrs$\oplus\mathcal{S}$) n,syncs) $\mathcal{S}$)>=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
checkReq(be,(Attrs, $\mathcal{S}$))
= $\mathcal{BV}$ (Attrs$\oplus\mathcal{S}$) be
checkReq(anything else,$\mathcal{S}$)
= error Non acquisition (getR, getB, getS, getBv ck(..) in test

Explanation: for each of the claims use the store to compute if it can be met, in the case of the parameterised bins, get the current items waiting in the bin, compute the binding names for the parameters, and use a further predicate to examine whether any of the current entries can match the request.

Further predicate in the case of parameterised bins, basically see if we can find an item on which the check condition cc will evaluate true:

checkBVReq([],nvL,cc,(Attrs,$\mathcal{S}$)) = false
checkBVReq((bvl,rL,sL)::t,nvL,cc,(Atts,$\mathcal{S}$)) = if $\mathcal{BV}$ bindL((Attrs,$\mathcal{S}$),nvL,bvL) cc then true
else checkBVReq(t,nvL,cc,(Attrs,$\mathcal{S}$));

Explanation:

1. checkBVReq([],nvL,cc,(Attrs,$\mathcal{S}$))=false: none of the bin items currently satisfy the predicate;
2. checkBVReq((bvl,sL,vL)::t,nvL,cc,(Attrs,$\mathcal{S}$))=if $\mathcal{BV}$ bindL(Attrs$\oplus\mathcal{S}$,nvL,bvL) cc: extend the store with the bindings implied by the bin claim, check the predicate:
3. then true: Ok found a satisfying bin item;
4. else checkBVReq(t,nvL,cc,(Attrs,$\mathcal{S}$)): try the next item;

Extend a store with the bindings implied by the use of a parameterised bin:

bindL($\mathcal{S}$,[],[])	=	$\mathcal{S}$
bindL($\mathcal{S}$,vn::vt,bv::bt)=bindL($\mathcal{S}$++(vn,vars,bv),vt,bt)		
bindL($\mathcal{S}$,[],bv::bt)	=	error
bindL($\mathcal{S}$,vn::vt,[])	=	error

Explanation: construct a new store using the variable and value lists to construct a sequence of bindings.

Further predicate in the case of parameterised syncs, basically see if we can find an item on which the check condition cc will evaluate true:

checkSVReq([],nvL,cc,(Attrs,$\mathcal{S}$))	=	false
checkSVReq((bvl,_,_)::t,nvL,cc,(Atts,$\mathcal{S}$))	=	if $\mathcal{BV}$ bindL((Attrs,$\mathcal{S}$),nvL,bvL) cc then true else checkSVReq(t,nvL,cc,(Attrs,$\mathcal{S}$));

Basically identical to checkBVReq except we need to project data out of a triple.

Given we have a condition that can be meet we need to update a store to reflect the changes, and also keep note of the resources and syncs that have been allocated in making this claim, so they can be returned correctly at some later point. This semantic function essentially mirrors the check just performing the state changes:

getResL(evt,pn,[],$\mathcal{SP}$)	=	$\mathcal{SP}$'
getResL(evt,pn,h::t,$\mathcal{SP}$)	=	getResL(evt,pn,t,getReq(evt,pn,h,$\mathcal{SP}$))

Explanation: do each of the atomic claims in turn, which cause the store to change, and resources or syncs to be allocated, accumulate all of the store changes and allocations and return them.

Performing an atomic resource allocation:

getReq(evt,pn,getR(n,e),(Attrs,$\mathcal{S}$))
= let evn=evalN (Attrs$\oplus\mathcal{S}$) n
let v=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
in (UPDATE(evn,res, LOOKUP(evn,res) Attrs +v) Attrs,
UPDATE(evn,res, LOOKUP(evn,res) $\mathcal{S}$-v) $\mathcal{S}$)
getReq(evt,pn,getB(n,e),(Attrs,$\mathcal{S}$))
= let evn=evalN (Attrs$\oplus\mathcal{S}$) n
let v=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
let (bL,$\mathcal{S}$')=GETITEMS(env,bins,v) $\mathcal{S}$
in (Attrs,$\mathcal{S}$')
getReq(evt,pn,getBv(n,vL,cc),(Attrs,$\mathcal{S}$))
= val evn=evalN (Attrs$\oplus\mathcal{S}$) n
val (aL, bvL, rL, sL, nBi)
=getBVReq(LOOKUP(evn,bins) $\mathcal{S}$, map evalN(Attrs$\oplus\mathcal{S}$) vL, cc, $\mathcal{S}$, [])
(Attrs', $\mathcal{S}$')=upDateBinV(evn, nBi, aL, rL, sL, (Attrs, $\mathcal{S}$))
in (Attrs',$\mathcal{S}$")
getReq(evt,pn,getS(n,e),(Attrs,$\mathcal{S}$))
= let evn=evalN (Attrs$\oplus\mathcal{S}$) n
let v=$\mathcal{EV}$ (Attrs$\oplus\mathcal{S}$) e
let val (syns,$\mathcal{S}$')=GETITEMS(evn,syncs,v) $\mathcal{S}$
let val Attrs'=ADDITEMS(evn, syncs, v) Attrs
in (Attrs',$\mathcal{S}$')
getReq(evt,pn,getSv(n,vL,cc),(Attrs,$\mathcal{S}$))
= val evn=evalN (Attrs$\oplus\mathcal{S}$) n
val (syn,aL,nS)
=getSVReq(LOOKUP(evn,syncs) $\mathcal{S}$,map evalN(Attrs$\oplus\mathcal{S}$) vL,cc,$\mathcal{S}$,[])
$\mathcal{S}$' = UPDATE(evn,syncs,nS) $\mathcal{S}$)
(Attrs', $\mathcal{S}$")=upDateSinV(evn,syn,nS,aL,(Attrs, $\mathcal{S}$'))
in (Attrs',$\mathcal{S}$")
getReq(evt,pn,be,$\mathcal{SP}$)
= $\mathcal{SP}$
getReq(evt,pn,others,$\mathcal{SP}$)
= error Non acquisition (getR, getB, getS, getBv ck(..) in get

Explanation: except for the parameterised gets, we update the store in the obvious fashion. In the case of the value bins, we use an auxiliary function to return the binding values and the bin without the element which was captured. Similarly for the parametrised syncs.

Auxiliary function to get the bin bound values:
getBVReq(bi::t,nvL,cc,bvl,Bi)
= if $\mathcal{BV}$ bindL((Attrs,$\mathcal{S}$),nvL,bi) cc then (mkal(nvL,bi),Bi@t)
else getBVReq(t,nvL,cc,bvl,Bi@[bi]));

Explanation: Search down the list of parameter bindings associated with the bin. There **must** be a satisfying match somewhere, when it is found, return the variables, the bind values and the bin list with this value set removed, but its order preserved.

Auxiliary function to get the sync bound values:
getSVReq((h,bL,pd)::t,nvL,cc,bvl,S)
= if $\mathcal{BV}$ bindL((Attrs,$\mathcal{S}$),nvL,h) cc then ((h,bL,pd),mkal(nvL,h),S@t)
else getSVReq(t,nvL,cc,bvl,S@[(h,bL,pd)]));

Explanation: Search down the list of parameter bindings associated with the sync. There **must** be

a satisfying match somewhere, when it is found, return the descriptor, variables, the bind values and the sync list with this value set removed, but its order preserved.

Auxiliary function to set the bin bound values:

upDateBinV(bn, nBi,nAL, rL, sL,(Attrs, $\mathcal{S}$))
= let val (Attrs',$\mathcal{S}$')=putVarAL(nAL,(Attrs $\mathcal{S}$))
in ((map ADDITEMS sL (map ADDITEMS rL Attrs')),(UPDATE (bn, nBi, bins) $\mathcal{S}$)) end;

Explanation: Update the local and global state to reflect the changes imposed by the removing of a value bin, and the bindings that the get requires. In particular add the passed syncs and resources to the current local attributes.

Auxiliary function to set bound values:

putVarAL([],$\mathcal{S}$) = $\mathcal{S}$
putVarAL((n,v)::t,$\mathcal{S}$) = putVarAL(t,do_assign(n,v,$\mathcal{S}$))

Explanation: return the state obtained by performing a sequence of bindings. Auxiliary function to set the bin bound values:

upDateSinV(sn,syn,nS,nAL,(Attrs, $\mathcal{S}$))
= let val (Attrs',$\mathcal{S}$')=putVarAL(nAL,(Attrs $\mathcal{S}$))
in ((UPDATE (sn,nS,syncs) $\mathcal{S}$), UPDATE (sn,syn,SYNCS) Attrs') end;

Explanation: Update the local and global state to reflect the changes imposed by the removing of a value bin, and the bindings that the get requires.

9.3 Promote

This is the auxiliary function that detects when a state change permits us to unblock an entity that is waiting in the blocked list. We need to be careful about how we continue down the list in the case that a claim still cannot be met. We can easily induce starvation if we do not take care. The basic point is that a blocked request for multiple resources should block all those for less that were blocked at a time after it started waiting. In the context of branches with differing requests we stop if they overlap on any resource.

promote(b,evt,$\mathcal{EL}$,$\mathcal{BL}$,$\mathcal{S}$) = val (PL, WL, $\mathcal{S}$')=first_dep(b,evt,$\mathcal{BL}$,[],$\mathcal{S}$,[])
in (map ENTER PL $\mathcal{EL}$,WL,$\mathcal{S}$')

Explanation: the sub function first dep, returns a list of entities that have been reactivated PL, the resultant blocked list WL, and the new store $\mathcal{S}$ that has resulted from allowing the claims of the activated processes.

This function does the work, it scans the blocked list for the first waiting entity that could be reawakened by the assignment, checks to see if it can be awoken, and then checks if further exploration of the blocked list is necessary.

first_dep(b,evt,[],WL,$\mathcal{S}$,PL) = (PL,WL,$\mathcal{S}$)
first_dep(b,evt,PD(pn,pr,[],Attrs,et)::Bl,WL,$\mathcal{S}$,PL)
= first_dep(b,evt,Bl,WL@[PD(pn,pr,[],Attrs,et)],$\mathcal{SP}$,PL)
first_dep(b,evt,PD(pn,pr,Cond(l)::Body,Attrs,et)::Bl,WL,$\mathcal{S}$,PL)
= if depCond(b,Cond(l),(Attrs,$\mathcal{S}$))
then let (fire, (Attrs',$\mathcal{S}$'), fBody)=doCond(evt,pn,l,(Attrs,$\mathcal{S}$))
in if fire
then first_dep(b, evt, Bl, WL, $\mathcal{S}$',
PL@[PD(pn,pr,fBody@Body,Attrs',evt)])
else if check(b)
then first_dep(b, evt, Bl,
WL@[PD(pn,pr,Cond(l)::Body,Attrs,et)], $\mathcal{S}$,PL)
else (PL,WL@[PD(pn,pr,Cond(l)::Body,Attrs,et)]@Bl,$\mathcal{S}$)
else first_dep(b,evt,Bl,
WL@[PD(pn,pr,Cond(l)::Body,Attrs,et)],$\mathcal{S}$,PL)
first_dep(b,evt,PD(pn,pr,other::Body,aL,et)::Bl,WL,$\mathcal{S}$,PL)
= **error** Blocked object has non leading cond action

Explanation:

1. first_dep(b,evt,[],WL,$\mathcal{S}$,PL)=(PL,WL,$\mathcal{S}$): reached the end of the blocked list so return the current promotion list, the remainder of the blocked list and the current store;

2. first_dep(b,evt,PD(pn,pr,[],Attrs,et)::Bl,WL,$\mathcal{S}$,PL): somehow an empty process descriptor has got here, so ignore it by returning first_dep(b,evt,Bl,WL@[PD(pn,pr,[],Attrs,et)],$\mathcal{S}$,PL), which reinserts it at its current position in the blocked list;

3. first_dep(b,evt,PD(pn,pr,Cond(l)::Body,Attrs,et)::Bl,WL,$\mathcal{S}$,PL): we have a descriptor to check with a condition at its head;

4. if depCond(b,Cond(l),(Attrs,$\mathcal{S}$)) does this blocked item depend on the assignment that has just taken place:

   (a) true: let (fire, (Attrs',$\mathcal{S}$'), fBody)=doCond(evt,pn,l,(Attrs,$\mathcal{S}$)) in if fire: as before we check if this conditional can fire and what are the consequences if it can

      i. true: first_dep(b, evt, Bl, WL, $\mathcal{S}$', PL@[PD(pn,pr,fBody@Body,Attrs,evt)]) continue checking the blocked list to see if others can be fired, place this at the end of the enabled processes so we preserve order.

      ii. false: else if check(b) we need to block if this was a return of resource, a Sync, or a normal bin, since a process dependent on that resource could not fire, so we must wait for enough to accumulate, other wise it was a variable, or a failure of matching on a bin value so we continue;

         A. true: first_dep(b, evt, Bl, WL@[PD(pn,pr,Cond(l)::Body,Attrs,et)], $\mathcal{S}$,PL) continue to attempt unblocking, note this process retains its position in the blocked list;

         B. false: (PL,WL@[PD(pn,pr,Cond(l)::Body,Attrs,et)]@Bl,$\mathcal{S}$) stop checking now and return the effects so far;

   (b) false: first_dep(b,evt,Bl, WL@[PD(pn,pr,Cond(l)::Body,aL,et)],$\mathcal{S}$,PL) leave the current item where it is in the block list and try the next;

5. error: normal form demands that blocked items start with a cond, so this cannot be here.

A sequence of predicates that inform as to whether a conditional is dependent of the assignment that has just taken place.

depCond(b,Cond([]), $\mathcal{SP}$) = false
depCond(b,Cond((reqr,rBd)::t), $\mathcal{SP}$) = map deps b reqr $\mathcal{SP}$orelse depCond(b,Cond(t),$\mathcal{SP}$)
depCond(b,other,s) = error Object has non leading cond action

Explanation: unwrap the claim body pairs so we can check if any of the claims is dependent on this return.

deps putR(n,v) getR(m,w) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps putB(n,v) getB(m,w) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps putBS(n,v) getB(m,w) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps putBS(n,v) getBv(m,pL,cc) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps Sync(n) getS(m,v) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps SyncV(n,exL,vL) getSv(m,vL,cc) $\mathcal{SP}$ = (n=evalN $\mathcal{SP}$ m)
deps ass(v,ee) ck(e) $\mathcal{SP}$ = exp_dep(e,v,$\mathcal{SP}$)
deps ass(v,ee) getBv(m,pL,cc) $\mathcal{SP}$ = exp_dep(cc,v$\mathcal{SP}$))
deps ass(v,ee) getSv(m,pL,cc) $\mathcal{SP}$ = exp_dep(cc,v$\mathcal{SP}$))
deps a1 a2 $\mathcal{S}$ = false

Explanation: check out the names, if they are the same, then true, otherwise false. The same recursion is applied to boolean expressions and expressions, but omitted for brevity as it is standard. Note, the variable name in the assignment has been resolved to a fixed name before this point is reached.

9.4 cBL

This is essentially the same as the dependency check immediately above except that instead of being an assignment we are matching against a claim list. A predicate that examines mutual dependence between a claim list and the blocked objects.

cBL(gcl,[],s) = false
cBL(gcl,PD(_,pr,b::_,ls,_)::t,(st,ilst)) = cdepFC(gcl,b,(st,ilst),(st,ls)) orelse cBL(gcl,t,(st,ilst))
cBL(gcl,h::t,s) = error empty action list inside cond - unreachable

Explanation: check out the leading conditionals of each of the blocked entities in turn, note the local state may be needed to resolve names.

cdepFC(Cond([]),_, $\mathcal{SP}$,$\mathcal{SP}$1) = false
cdepFC(Cond(h::t),Cond(Cl),$\mathcal{SP}$,$\mathcal{SP}$1) = checkCondL(filterConds(h::t),filterConds(Cl),$\mathcal{SP}$,$\mathcal{SP}$1)
cdepFC(_,Cl,$\mathcal{SP}$,$\mathcal{SP}$1) = error("Object has non leading cond action")

Explanation: if none of the current requirements intersects with this blocked process then we can continue. Check each in turn until none is left. Normal form demands that this must be a conditional action. Also a boolean test can never cause starvation by being evaluated so these are filtered out.

checkCondL([],_,$\mathcal{SP}$,$\mathcal{SP}$1) = true
checkCondL(([],_)::t,_,$\mathcal{SP}$,$\mathcal{SP}$1) = false
checkCondL((rql,_)::t,cl,$\mathcal{SP}$,$\mathcal{SP}$1) = if checkConds(rql,cl,$\mathcal{SP}$,$\mathcal{SP}$1)
then checkCondL(t,cl,$\mathcal{SP}$,$\mathcal{SP}$1)
else false

Explanation: for each of the claims in the list see if it collides with the blocked list, all of them must collide with at least one of the competitor for this to block. Basically if their is one free branch then that may fire, we cannot be starving the competitor if it does not compete with that branch.

checkConds(rql,[],$\mathcal{SP}$,$\mathcal{SP}$1) = false
checkConds(rql,(rql1,_)::t,$\mathcal{SP}$,$\mathcal{SP}$1) = if csbL(rql,rql1,[],$\mathcal{SP}$,$\mathcal{SP}$1)
then true else checkConds(rql,t,$\mathcal{SP}$,$\mathcal{SP}$1)

Explanation: check that there is at least one branch in the potential competitor for which our requirements are a strict subset. We can only starve something if we use less than they do!

csbL([],[],[],$\mathcal{SP}$,$\mathcal{SP}$1) = false
csbL([],[],h::t,$\mathcal{SP}$,$\mathcal{SP}$1) = true
csbL(h::t,[],_,$\mathcal{SP}$,$\mathcal{SP}$1) = false
csbL([],h::t,_,$\mathcal{SP}$,$\mathcal{SP}$1) = true
csbL(h::t,h1::t1,L1,$\mathcal{SP}$,$\mathcal{SP}$1) = if c_deps(h,h1,$\mathcal{SP}$,$\mathcal{SP}$1)
then csbL(t,L1@t1,[],$\mathcal{SP}$,$\mathcal{SP}$1)
else csbL(h::t,t1,h1::L1,$\mathcal{SP}$,$\mathcal{SP}$1)

If we manage to discharge all of the first requirements as competing against something in the second, and there are some requirements in the second that are not 'cancelled out'. Then we have a proper subset with all of the requirements competing.

c_deps getR(n,v),getR(m,v1) $\mathcal{SP}$ $\mathcal{SP}$1 = (evalN $\mathcal{SP}$ n=evalN $\mathcal{SP}$1 m)
c_deps getB(n,v),getB(m,v1) $\mathcal{SP}$ $\mathcal{SP}$1 = (evalN $\mathcal{SP}$ n=evalN $\mathcal{SP}$1 m)
c_deps getBv(n,bl,cc),getB(m,v1)$\mathcal{SP}$ $\mathcal{SP}$1 = (evalN $\mathcal{SP}$ n=evalN $\mathcal{SP}$1 m)
c_deps getB(n,v1) getBv(m,bl,cc)$\mathcal{SP}$ $\mathcal{SP}$1 = (evalN $\mathcal{SP}$ n=evalN $\mathcal{SP}$1 m)
c_deps getS(n,v) getS(m,v1) $\mathcal{SP}$ $\mathcal{SP}$1 = (evalN $\mathcal{SP}$ n=evalN $\mathcal{SP}$1 m)
c_deps v be $\mathcal{SP}$ $\mathcal{SP}$1 = false
c_deps other other1 $\mathcal{SP}$ $\mathcal{SP}$1 = false

Explanation: if names match then we are co-dependent, unless its an assignment. Note, that items from parameter bins and value syncs do not cause dependency. In order to check this we would need to resolve the side condition and check that it is overlapping. So we take the view that evaluation carries on, as this will cause incorrect blockage if two entities are waiting on distinct value patterns.

filterConds([]) = []
filterConds((reql,cl)::t) = (filterCond(reql),cl)::filterConds(t)

filterCond([]) = []
filterCond(ck(e)::t)=filterCond(t)
filterCond(a::t)=a::filterCond(t)

Explanation: if a requirement is a conditional then remove it as it cannot be the cause of starvation.

9.5 removeR

Update the store associated with an entity to remove resources.

removeR(n,v,PD(pn,pr,body,attrs,evt))
= if not(MEMBER(n,res) attrs)
then error Process pn does have any resource n
let v1=LOOKUP(n,res) attrs
in if v>v1 then error Process pn does not own v of resource n
PD(pn,pr,body,UPDATE(n,res,v1-v) attrs,evt)

Explanation: find out whether we know about this resource, or have enough to free the amount we state we can. If either are untrue then raise an error, otherwise update the store and return the amended process descriptor.

need to evaluate resource lists when we hand them over in bins:

evalRL([],(Attrs,$\mathcal{S}$))=[]—
evalRL((n,amt)::t,(Attrs,$\mathcal{S}$))
= (evalN Attrs $\oplus\mathcal{S}$n, floor($\mathcal{EV}$ Attrs$\oplus\mathcal{S}$ amt)::evalRL(t,(Attrs,$\mathcal{S}$)

and update the process notice to take account of handed on resources:

removeRL([],cp)=cp—
removeRL((id,v)::t,cp)
= removeRL(t,removeR(id,v,cp))

9.6 removeS

Update the store associated with an entity to remove syncs

removeS(n,v,PD(pn,pr,body,attrs,evt))
= if not(MEMBER(n,syncs) attrs)
then error Process pn does have any sync n
let v1=length(LOOKUP(n,syncs) attrs)
in if v>v1 then error Process pn does not own v of sync n
let (fp,attrs')=GETITEMS(n,syncs,v) attrs
in (map (makeTime evt) (map val3 fp),PD(pn,pr,body,attrs',evt))

Explanation: find out whether we know about this sync, or have enough to free the amount we state we can. If either are untrue then raise an error, otherwise update the store and return the freed entities and the amended process descriptor. As the sync descriptors are triples we must project out the process descriptor part.

Update the store associated with an entity to remove value syncs

removeSv(n,vals,PD(pn,pr,body,attrs,evt))
= if not(MEMBER(n,syncs) attrs)
let val ([(vL,bL,prd)], attrs')=GETITEMS(1,syncs,v) attrs in
let val prd'=putAttrs(putVarAL(mkal(bL,vals)) getAttrs prd)
in (makeTime prd' evt,PD(pn,pr,body,attrs',evt))

Explanation: get the sync record and update the attributes in the current entity. Update the local variables in the deslaved entity to reflect the binding implied by the return values. Finally, make the current time of the promoted process the current event time, and return a pair consisting of the deslaved entity, and the current entity but with this sync element removed.

We may need to do a list of these when we hand a synced process on within a bin.

removeSL([],cp,sl)=(cp,sL)—
removeSL(nm::t,cp,sl)
= val ((rL,idL,sp),cp')=remBSL(nm,cp)
in removeSL(t,cp',addSync(nm,rL,idL,sp,sl))

removing the body of a sync from a process notice:

remBSL(n,PD(pn,pr,body(rL,sL,LV),evt))
= val (syn,sL')=DremBSL(pn,n,sL)
in (syn,PD(pn,pr,body,(rL,sL',LV),evt))

and finally getting the sync from the sync list:

DremBSL(pn,n,[]))=error "pn doesn't own any n"
DremBSL(pn,n,(n1,pl)::t))
= if n=n1
then if length(pl)¡1 error "pn does not own enough n"
else val rs::ral=pl in (rs,(n,ral)::t)
else val (rs,rl)=DremBSL(pn,n,t) in (rs,(n1,pl)::rl)

9.7 getBody

Basically this takes a class name, a list of parameter instantiations and a store. The idea is to get the formal parameter list from the class definition in the store and then replace the formal parameter name throughout.

```
getBody(cn,pil,S)  =  let (pl,bd)=LOOKUP(cn,class) S
                      in doSuB(bd,pl,pil,cn)
                      end;
```

Explanation: get the parameter list and class body from the store form a substitution list of pairs, of names and replacements and then apply it throughout the body of the entity.

```
doSub(bd,[],[],cn)                  =  bd
doSub(bd,[],h::_,cn)                =  error too few parameters in use of cn
doSub(bd,h::_,[],cn)                =  error too many parameters in use of cn
doSub(bd,pn::pt,(fs "s")::pit,cn)   =  doSub(SubA(bd,pn,s,cn),pt,pit,cn)
doSub(bd,pn::pt,(fixe vl)::pit,cn)  =  doSub(SubV(bd,pn,v,cn),pt,pit,cn)
doSub(bd,pn::pt,(ps _)::pit,cn)     =  error unevaluated paramter in cn (shouldn't get here)
```

Explanation: take a parameter name and a replacement string and substitute the replacement for the parameter throughout the body of a class defintion. We can define the effect of substitution over the syntax of the DEMOS language as follows:

subA(Class(nm,pl,bd),p,s) = Class(sub_eN(nm,p,s),subsL(pl,p,s),subAL(bd,p,s))
subA(Entity(enm,cnm,pl,det,rL,sL),p,s)
= Entity(sub_eN(p,s) enm ,sub_eN(p,s) cnm,
map sub_eN(p,s) pl,sub_e(p,s) det, map sub_R(p,s) rL,
map sub_eN sL)
subA(Res(N,e),p,s) = Res(sub_eN(p,s) N,sub_e(p,s) e)
subA(Bin(N,e),p,s) = Bin(sub_eN(p,s) N,sub_e (p,s) e)
subA(Sync(N),p,s) = Sync(sub_eN(p,s) N)
subA(Var(N,ee),p,s) = Var(sub_eN(p,s) N,sub_e (p,s) ee)
subA(LVar(N,ee),p,s) = LVar(N,sub_eN(p,s) ee)
subA(Cons(N,ee),p,s) = Cons(sub_eN(p,s) N,sub_e(p,s) ee)
subA(HOLD(dt),p,s) = HOLD(sub_e(dt,p,s))
subA(TRACE(s,eL),p,s) = TRACE(s,sub_eL(eL,p,s))
subA(PRIORITY(ee),p,s) = PRIORITY(sub_eN(p,s) ee)
subA(CLOSE,p,s) = CLOSE
subA(DO(e,L),p,s) = DO(sub_eN(p,s) e,subAL(L,p,s))
subA(DO(n,L),p,s) = DO(n,subAL(L,p,s))
subA(While(L1,L2),p,s) = While(subAL(L1,p,s),subAL(L2,p,s))
subA(getR(R,e),p,s) = getR(sub_eN(p,s) R,sub_e(p,s) e)
subA(putR(R,e),p,s) = putR(sub_eN(p,s) R,sub_e(p,s) e)
subA(getB(R,e),p,s) = getB(sub_eN(p,s) R,sub_e(p,s) e)
subA(putB(R,e),p,s) = putB(sub_eN(p,s) R,sub_e(p,s) e)
subA(getS(R,e),p,s) = getS(sub_eN(p,s) R,sub_e(p,s) e)
subA(putS(R,e),p,s) = putS(sub_eN(p,s) R,sub_e(p,s) e)
subA(getBv(R,idL,be),p,s) = getBv(sub_eN(p,s) R,map sub_eN(p,s) idL,sb_b(be,p,s))
subA(getSv(R,idL,be),p,s) = getSv(sub_eN(p,s) R,map sub_eN(p,s) idL,sb_b(be,p,s))
subA(putBS(R,eeL,rL,sL),p,s) = putBS(sub_eN(p,s) R, map sub_e(p,s) eeL,
map sub_R(p,s) rL, map sub_eN(p,s) sL)
subA(putSv(R,eeL),p,s) = putSv(sub_eN(p,s) R,map sub_e(p,s) eeL)
subA(be,p,s) = sb_b(p,s) be
subA(ass(V,e),p,s) = ass(sub_eN(V,p,s),sub_e(p,s) e)
subA(Req(L),p,s) = Req(subAL(L,p,s))
subA(Cond(L),p,s) = Cond(subCAL(L,p,s))
subAL([],p,s) = []
subAL(h::t,p,s) = subA(h,p,s)::subAL(t,p,s)
subCAL([],p,s) = []
subCAL((c,bdy)::t,p,s) = (subAL(c,p,s),subAL(body,p,s))::subCAL(t,p,s)

and defining substitution within names as follows:

sub_eN(p,s1) (fs s) = if p=s then fs s1 else fs s
sub_eN(p,s1) (ps (s,pl)) = if p=s then ps (s1 , map sub_e (p,s) pl)

and in resource pairs sub_R(p,s) (rn,ramt) = (sub_eN(p,s) rn,sub_e(p,s) ramt)

for brevity we allow ourselves to use **op** to range over arbitrary operators in both the boolean and the expression syntax. We use variables starting with e and b to distinguish expressions and boolean terms repsectively. Hence we can define the substitution functions with respect to just those parts of the syntax dependant on variable names , distinguished by a **Vi** in the case of expressions and Boi for booleans denoting an expression type:

sub_e(p,s1) Vi(en)	=	Vi(sub_eN (p,s1) en)
sub_e(p,s1) op(e1,e2)	=	op(sub_e(p,s1) e1,sub_e(p,s1) e2)
sub_b(p,s1) op(e1)	=	op(sub_e(p,s1) e1)
sub_b(p,s1) op(e1,e2)	=	op(sub_e(p,s1) e1,sub_e(p,s1) e2)
sub_b(p,s1) op(b1)	=	op(sub_b(p,s1) b1)
sub_b(p,s1) op(b1,b2)	=	op(sub_b(p,s1) b1,sub_b(p,s1) b2)

Explanation: we recurse down the syntax until we hit an id, which is either a fixed string or a string with parameters, if the string matches a string in the substitution list it is replaced otherwise it is left unchanged.

Explanation: take a parameter name and a replacement value and substitute the replacement for the parameter throughout the body of a class defintion. Notice this is only valid as replacement for a parameter in an expression, not in a name. So it must raise an error if the substitution occurs in an illegal position.

We can define the effect of substitution over the syntax of the DEMOS language as follows:

subV(Class(nm,pl,bd),p,s) = Class(sub_eVN(nm,p,s),subsL(pl,p,s),subVL(bd,p,s))
subV(Entity(enm,cnm,pl,det),p,s)
= Entity(sub_eVN(p,s) enm ,sub_eVN(p,s) cnm, map sub_eVN(p,s) pl,
sub_ve(p,s) det, map sub_vR(p,s) rL,
map sub_eVN(p,s) sL)
subV(Res(N,e),p,s) = Res(sub_eVN(p,s) N,sub_ve(p,s) e)
subV(Bin(N,e),p,s) = Bin(sub_eVN(p,s) N,sub_ve(p,s) e)
subV(Sync(N),p,s) = Sync(sub_eVN(p,s) N)
subV(Var(N,ee),p,s) = Var(sub_eVN(p,s) N,sub_ve(p,s) ee)
subV(LVar(N,ee),p,s) = LVar(N,sub_eVN(p,s) ee)
subV(Cons(N,ee),p,s) = Cons(sub_eVN(p,s) N,sub_ve(p,s) ee)
subV(HOLD(dt),p,s) = HOLD(sub_e(dt,p,s))
subV(TRACE(s,eL),p,s) = TRACE(s,sub_eL(eL,p,s))
subV(PRIORITY(ee),p,s) = PRIORITY(sub_eVN(p,s) ee)
subV(CLOSE,p,s) = CLOSE
subV(DO(e,L),p,s) = DO(sub_eVN(p,s) e,subVL(L,p,s))
subV(DO(n,L),p,s) = DO(n,subVL(L,p,s))
subV(While(L1,L2),p,s) = While(subVL(L1,p,s),subVL(L2,p,s))
subV(getR(R,e),p,s) = getR(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(putR(R,e),p,s) = putR(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(getB(R,e),p,s) = getB(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(putB(R,e),p,s) = putB(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(getS(R,e),p,s) = getS(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(putS(R,e),p,s) = putS(sub_eVN(p,s) R,sub_ve(p,s) e)
subV(getBv(R,idL,be),p,s) = getBv(sub_eVN(p,s) R,map sub_eVN(p,s) idL,sb_vb(be,p,s))
subV(getSv(R,idL,be),p,s) = getSv(sub_eVN(p,s) R,map sub_eVN(p,s) idL,sb_vb(be,p,s))
subV(putBS(R,eeL,rl,sL),p,s) = putBS(sub_eVN(p,s) R, map sub_ve(p,s) eeL,
map sub_vR(p,s) rL, map sub_eVN(p,s) sL)
subV(putSv(R,eeL),p,s) = putSv(sub_eVN(p,s) R,map sub_ve(p,s) eeL)
subV(be,p,s) = sb_b(p,s) be
subV(ass(V,e),p,s) = ass(sub_eVN(V,p,s),sub_ve(p,s) e)
subV(Req(L),p,s) = Req(subVL(L,p,s))
subV(Cond(L),p,s) = Cond(subCVL(L,p,s))
subVL([],p,s) = []
subVL(h::t,p,s) = subV(h,p,s)::subVL(t,p,s)
subCVL([],p,s) = []
subCVL((c,bdy)::t,p,s) = (subVL(c,p,s),subVL(body,p,s))::subCVL(t,p,s)

Check substitution in most names, can only change expressions, not names.

sub_eVN(p,s1) (fs s) = error trying to substitute expression for name s
sub_eVN(p,s1) (ps (s,pl)) = if p=s then error trying to substitute expression for name s
else ps (s , map sub_ve (p,s) pl)
sub_eVN(p,s1) (fixe e) = fixe(sub_ve (p,s) e)

and in resource pairs

sub_vR(p,s) (rn,ramt) = (sub_eVN(p,s) rn,sub_ve(p,s) ramt)

For brevity we allow ourselves to use **op** to range over arbitrary operators in both the boolean and the expression syntax. We use variables starting with e and b to distinguish expressions and boolean terms repsectively. Hence we can define the substitution functions with respect to just those parts of the syntax dependant on variable names , distinguished by a **Vi** in the case of

expressions and Boi for booleans denoting an expression type:

```
sub_ve(p,s1) Vi( en )     =  if sub_veN(p,s1) en then s1 else sub_eVN(p,s1) en
sub_ve(p,s1) op(e1,e2)    =  op(sub_ve(p,s1) e1,sub_ve(p,s1) e2)
sub_vb(p,s1) op(e1)       =  op(sub_ve(p,s1) e1)
sub_vb(p,s1) op(e1,e2)    =  op(sub_ve(p,s1) e1,sub_ve(p,s1) e2)
sub_vb(p,s1) op(b1)       =  op(sub_vb(p,s1) b1)
sub_vb(p,s1) op(b1,b2)    =  op(sub_vb(p,s1) b1,sub_vb(p,s1) b2)
```

Explanation: we recurse down the syntax until we hit an id, which is either a fixed string or a string with parameters. If we match the string in a variable name we can replace it by the given expression, otherwise we continue checking the parameters if the name term is of that form.

10 DisC

This function constructs the seperate seed for each distribution so that they are well seperated:

```
DisC S (geo(e))          =  DisPtr(DisAdd S (geo(e)))
DisC S (nege(e))         =  DisPtr(DisAdd S (nege(e)))
DisC S (Poisson(e))      =  DisPtr(DisAdd S (Poisson(e)))
DisC S (bin(e1,e2))      =  DisPtr(DisAdd S (bin(e1,e2)))
DisC S (Normal(e1,e2))   =  DisPtr(DisAdd S (Normal(e1,e2)))
DisC S (wei(e1,e2))      =  DisPtr(DisAdd S (wei(e1,e2)))
DisC S (Erl(e1,e2))      =  DisPtr(DisAdd S (Erl(e1,e2)))
DisC S (Uni(e1,e2))      =  DisPtr(DisAdd S (Uni(e1,e2)))
DisC S (Puni(e1,e2))     =  DisPtr(DisAdd S (Puni(e1,e2)))
DisC S (Pud(l))          =  DisPtr(DisAdd S (Pud(l)))
DisC S (Cud(l))          =  DisPtr(DisAdd S (Cud(l)))
DisC S (fix(n))          =  fix(n)
DisC S (rfix(n))         =  rfix(n)
DisC S (rnd(e))          =  rnd(DisC S (e))
DisC S (plus(e1,e2))     =  plus(DisC S (e1),DisC S (e2))
DisC S (times(e1,e2))    =  times(DisC S (e1),DisC S (e2))
DisC S (minus(e1,e2))    =  minus(DisC S (e1),DisC S (e2))
DisC S (divide(e1,e2))   =  divide(DisC S (e1),DisC S (e2))
DisC S (Vi(n))           =  Vi(DisNC S (n))
DisC S (DisPtr(e))       =  error ("distribution abstraction name" e)
```

This function puts in a pointer to the distribution into the expression, this allows us to store the distrbution with its appropriate seed and sample count whilst still having standard expressions. Notice we use the issue number of the distribution as its reference name.

```
DisAdd S e  =  dn=LOOKUP (disNum,var) S
               do 10000 U
               UPDATE (dn,dist,(e,LOOKUP (seed,var) S ,0)) S
               UPDATE (disNum,var,dn+1) S
               dn
```

Create a triple in the store of the distribution expression, its assigned random seed and the fact it has not been sampled yet.

```
DisNC S (fs s)        =  fs s
DisNC S (ps (s,pl))   =  ps(s, map DisC S pl)
DisNC S (fixp e)      =  fixp (DisC S e)
```

Convert parameters in the obvious fashion.

11 Probability distributions

The function sample(dis($\tilde{p}$)) simple redirects the evaluation to one of the probability distribution functions detailed below. These are all standard from the literature [7] but for completeness we present the algorithms here. All of them a based on a uniform psuedo random sample $U \in [0, 1]$

Geometric Distribution

The following computes a geometric sample from a U.

Geo(mean) = floor(log(U)/log(1-1/mean))

Negative Exponential Distribution

The following computes a negative exponential sample from a U.

nege(mean) = -mean*log(U)

Erlang Distribution

The following computes an Erlang using an internal function.

Erli(0,mean,k) = 0
Erli(k,mean,t) = nege(mean/t)+Erl(k-1,mean,t)
Erl(k,mean) = Erli(k,mean,k)

Normal Distribution

The following computes a normal from U via an internal function. It is slightly less efficient than it may be, but avoids imperative features.

nbs() = let val (V1,V2)=(2U-1,2U-1)
let val S=V1^2 + V2^2
in if S>1 then nbs() else (V1,S)

and we can generate a normal sample using the above.

normal(mean,sigma) = let val (V,S)=nbs()
let val X=V(-2*log(s)/s)$^{1/2}$
in sigma*X + mu

Poisson Distribution

The following computes a poisson using the negative exponential:

poisi(itr,total,mu) = let val nt=total+nexp(1/mu)
in if nt$\geq$ 1 then itr-1 else poisi(itr+1,nt,mu)
poisson(mu) = poisi(0,0,mu)

Binomial Distribution

The following computes a binomial distribution:

bern(p) = if U$\leq$p then 1 else 0
bin(n,p) = bern(p) + bin(n-1,p)

Weibull Distribution

The following computes a geometric from a U.

Wei(a,b) = b*log(1-U)$^{(1/a)}$

Point user distribution

A distribution over distinct values supplied as a list of (prob,data) pairs:

dis_samp(r,[])	=	error underspecified distribution
dis_samp(r,(prob,data)::t)	=	if r≤0 then data
		if r≤prob then data
		else dis_samp(r-prob,t)
Puni(dis)	=	dis_samp(U,dis)

Continuous user distribution

A distribution over distinct values supplied as a hull of (prob,data) pairs, where prob is the **cummulative** probability up to point data:

cdis_samp(r,pp,pprob,[])	=	error underspecified distribution
cdis_samp(r,pp,pprob(prob,data)::t)	=	if r≤0 then pp
		if r≤prob then pp +(r-pprob)*(data-pp)/(prob-pprob)
		else cdis_samp(r,data,prob,t)
Cuni(dis)	=	cdis_samp(U,0,0,dis)

12 Comments on Notation

A more conventional presentation of an operational semantics is the use of a natural deduction style presentation as in Plotkin's SOS [11]. Semantics would be presented as follows:

$$\frac{< C, \sigma > \rightarrow \sigma' \quad < C', \sigma' > \rightarrow \sigma''}{< C; C', \sigma > \rightarrow \sigma''}$$

with $< C, \sigma >$ being our semantic function. In the presence of conditions this form of presentation requires that we present a rule for each possible outcome hence we would have to render our semantics for While as follows:

$$\frac{cBL(Cond[(acqL, wB)],\text{BL,S})}{\mathcal{C}(PD(cn, pr, While(acqL, wB) :: Body, attrs, evt) :: \mathcal{EL},\text{BL,S}) \rightarrow (PD(cn, pr, Body, attrs, evt) :: \mathcal{EL},\text{BL,S})}$$

$$\frac{not(cBL(Cond[(acqL, wB)],\text{BL,S}) \;\; doCond(evt, c, [(acqL, wB)],\text{S}) \rightarrow (true,\text{S}', asR, asP)}{\begin{array}{l}\mathcal{C}(PD(cn, pr, While(acqL, wB) :: Body, attrs, evt) :: \mathcal{EL},\text{BL,S}) \rightarrow \\ (PD(cn, pr, wB@Body, atAdd(asR, asP, attrs), evt) :: \mathcal{EL},\text{BL,S}')\end{array}}$$

$$\frac{not(cBL(Cond[(acqL, wB)],\text{BL,S}) \;\; doCond(evt, c, [(acqL, wB)],\text{S}) \rightarrow (false, \_, \_, \_)}{\begin{array}{l}\mathcal{C}(PD(cn, pr, While(acqL, wB) :: Body, attrs, evt) :: \mathcal{EL},\text{BL,S}) \rightarrow \\ (PD(cn, pr, Body, atAdd(asR, asP, attrs), evt) :: \mathcal{EL},\text{BL,S})\end{array}}$$

So in this presentation each of the three clauses in the conditional is expressed as a separate rule, as would be the clauses in the initial switch within our presentation. Since this presentation is substantially hard to read than our more functional style we retain it. We leave it as an exercise to the determined logician to map our semantic functions into this style.

13 Conclusions

An operational semantics has two major uses:

1. to precisely specify the behavior of the system, and hence disambiguate its response for any particular user;
2. provides a template against which any implementor of the system can validate their implementation for correctness.

To validate a particular implementation we should proceed as follows:

1. we have a representation of the system state within our implementation, given by a tuple $(s_1, s_2, \ldots, s_n)$
2. we provide and invertible function Φ which maps from the tuples to the operational states of the demos system $\Phi(s_1, s_2, \ldots, s_n) = (\mathcal{EL}, \mathcal{BL}, \mathcal{S})$;
3. for each state change within the implementation $(s_1, s_2, \ldots, s_n) \Rightarrow (s'_1, s'_2, \ldots, s'_n)$ we observe $(\mathcal{EL}, \mathcal{BL}, \mathcal{S}) \Rightarrow (\mathcal{EL}', \mathcal{BL}', \mathcal{S}')$ and demonstrate that $(s'_1, s'_2, \ldots, s'_n) = \Phi^{-1}(\mathcal{EL}', \mathcal{BL}', \mathcal{S}')$.

The above methodology allows us to move from inefficient but clear representations of data, and system state to efficient ones whilst maintaining the correctness of the implementation. Hence the reason that we have not expended great effort in specifying efficient representations of store, blocked processes or live processes within the semantic presentation of DEMOS.

We have also defined how random numbers are to be generated. Hence, despite the apparent presence of randomness, it is possible not merely to have an execution in an alternate implementation that is simple, but is in fact identical. This is of great importance when qualifying simulators whose results can have major economic implications.

References

[1] G. Birtwistle, O-J Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, 2nd Edition, Studentlitteratur, Lund, Sweden, 1979.

[2] G. Birtwistle, DEMOS — discrete event modelling on Simula. Macmillen, 1979.

[3] G. Birtwistle and C. Tofts, An Operational Semantics of Process-Orientated Simulation Languages: Part I πDemos, Transactions of the Society for Computer Simulation, 10(4):299-333.

[4] G. Birtwistle and C. Tofts, An Operational Semantics of Process-Orientated Simulation Languages: Part II μDemos. Transactions of the Society for Computer Simulation, 11(4), 303-336.

[5] G. Birtwistle and C. Tofts, A Denotational Semantics for a Process-Based Simulation Language, ACM ToMaCS, 281 - 305:8(3), 1998.

[6] G. Birtwistle and C. Tofts, Relating operational and denotational descriptions of πDemos, Simulation Practice and Theory, 5:1-33 (1997).

[7] R. Jain, The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling John Wiley & Sons, 1991.

[8] R. Milner, Calculi for Synchrony and Asynchrony, Theoretical Computer Science 25(3), pp 267-310, 1983.

[9] R. Milner, Communication and Concurrency, Prentice Hall, 1990.

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

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

[12] R. de Simone Higher-level synchronising devices in Meije- SCCS. *Transactions in Computer Science* **37**, 245-267 (1985).

14 Summary of Semantic functions

We present the domain and range of each of the semantic functions to help in understanding the semantics:

Storage

Function	Domain	Range
$\in$	(name,type) $\times$store	bool
LOOKUP	(name,type) $\times$store	val:type
DELETE	(name,type) $\times$store	store
UPDATE	(name,type,val) $\times$store	store
ADDITEMS	(name,type, val list) $\times$store	store
GETITEMS	(name,type) $\times$store	(val:type list) $\times$store
$\oplus$	store $\times$store	store
ADDPrITEM	(name,type, val list) $\times$projF $\times$store	store
mk_av	store	store
$\rightarrow$	store * store	store

Event notices and lists

Function	Domain	Range
evTime	evNote	time
pPri	evNote	priority
pName	evNote	name
future	evNote $\times$time	evNote
ENTER	evNote $\times$(evNote list)	evNote list
BENTER	projF evNote $\times$(evNote list)	evNote list
getAttrs	evNote	store
setAttrs	evNote $\times$store	evNote
removeR	name $\times$val $\times$evNote	evNote
removeRL	(name, int) list $\times$evNote	evNote
removeS	name $\times$val $\times$evNote	(evNote list) $\times$evNote
removeSL	name list $\times$evNote	(evNote list) $\times$evNote
removeSv	name $\times$(val list) $\times$evNote	evNote $\times$evNote

Expression Evaluation

Function	Domain	Range
$\mathcal{EV}$	store →expression	val
$\mathcal{EC}$	store →expression	val
$\mathcal{BV}$	store →bool_expression	bool
evalN	store →name	string

Conditional Evaluation

The functions that decide whether an acquisition request can be met:

Function	Domain	Range
doCond	time ×evNote ×(cond list) ×store	bool ×store ×(action list)
cBL	action ×(evNote list) ×store	bool
checkL	action list ×store	bool
checkReq	action ×store	bool
getResL	action list ×store	store
getReq	action ×store	store
cdepFC	action ×(evNote list) ×store ×store	bool
checkCondL	(action list ×evNote list) list ×(action list ×evNote list) list×store ×store	bool
checkConds	action list ×(action list ×evNote list) list ×store ×store	bool
csbl	action list ×action list ×action list ×store ×store	bool
c_deps	action →action ×store ×store	bool
filterConds	action list	action list

Promotion

The functions that decide whether a promotion has taken place:

Function	Domain	Range
promote	action ×time ×(evNote list) ×(evNote list) ×store	(evNote list) ×(evNote list) ×store
first_dep	action ×time ×(evNote list) ×(evNote list) ×store ×(evNote list)	(evNote list) ×(evNote list) ×store
depCond	action ×action ×store	bool
deps	action →action →store	bool

Distribution abstraction

The functions creat the well seperated seeds for each distribution:

Function	Domain	Range
DisC	store → expression	expression
DisNC	store → name	name
DisAdd	expression	DisId

Class Parameter Substitution

The functions that substitute parameters in class declarations:

Function	Domain	Range
getBody	name ×(param list) ×store	(action list)
doSub	(action list) ×(string list) ×(param list) ×name	action list
SubA	(action list) ×string ×string ×name	action list
SubV	(action list) ×string ×expression ×name	action list
sub_e	(string ×string) →expression	expression
sub_b	(string ×string) →bool_expression	bool_expression
sub_eN	(string ×string) →name	name
sub_eVN	(string ×expression) →name	name
sub_R	(string ×expression) →(name, expression)	(name, expression)
sub_veN	(string ×expression) ra name	bool
sub_ve	(string ×expression) →expression	expression
sub_vb	(string ×expression) →bool_expression	bool_expression
sub_vR	(string ×expression) →(name, expression)	(name, expression)

Command Evaluation

The top semantic function:

Function	Domain	Range
$\Longrightarrow$	(evNote list) ×(evNote list) ×store	(evNote list) ×(evNote list) ×store