A Denotational Semantics for a Process-Based Simulation Language

CHRIS TOFTS and GRAHAM BIRTWISTLE
University of Leeds

In this article, we present semantic translations for the actions of μDemos, a process-based, discrete event simulation language. Our formal translation schema permits the automatic construction of a process algebraic representation of the underlying simulation model which can then be checked for freedom from deadlock and livelock, as well as system-specific safety and liveness properties. As simulation methodologies are increasingly being used to design and implement complex systems of interacting objects, the ability to perform such verifications is of increasing methodological importance. We also present a normal form for the syntactic construction of μDemos programs that allows for the direct comparison of such programs (two programs with the same normal form must execute in identical fashion), reduces model proof obligations by minimizing the number of language constructs, and permits an implementer to concentrate on the basic features of the language (since any program implementation that efficiently evaluates normal forms will be an efficient evaluator for the complete language).

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*simulation*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Discrete event simulation, process algebra, semantics

1. INTRODUCTION

Simulation is widely used for studying the behavior and performance of complex systems. One of the underlying problems of the simulation methodology is that of ensuring the correctness of the representation of the system under study since, in general, simulation systems do not admit formal proofs of their properties. In this article we develop a translation schema for μDemos [Birtwistle and Tofts 1994], a sugared variant of Demos [Birtwistle 1979, 1981] and a typical process-based simulation language, into CCS [Milner 1980, 1990], a mechanized process algebra.

C. Tofts is supported by an EPSRC Advanced Fellowship.
Authors' address: School of Computer Studies, University of Leeds, Woodhouse Lane, Leeds, LS2 9JT, UK; email: ⟨{chris,graham}@scs.leeds.ac.uk⟩.

(Since μDemos is the only simulation system currently with an operational semantics [Birtwistle and Tofts 1993, 1994], it seems to be the most appropriate simulation language with which to commence denotational studies.) Once we have CCS processes representing the systems behavior, we can then prove properties of those systems by describing them in Hennessy–Milner logic [Stirling 1987, 1992] and using the Edinburgh Concurrency Workbench [Moller 1990] to check their validity. For a general introduction to this methodology for simulation problems see Birtwistle et al. [1993]. A similar approach to describing the behavior of a factory can be found in van Vlijmen and van Waveren [1992].

In companion papers [Birtwistle and Tofts 1993, 1994], we presented the operational semantics of Demos. Operational definitions are quite detailed and may be used to guide an implementation (as in Birtwistle and Tofts [1993]) or argue in detail the precise way a particular program should unfold. In this article we develop a denotational definition of Demos by giving a construct-by-construct translation into CCS. Denotational definitions are much more abstract than operational definitions with detailed timings replaced by “now, or some arbitrary time later” and all queues (including the event list) running under random selection. Denotational definitions are thus useful for reasoning about properties that will hold under all possible runs of a model whatever the timing delays and whatever the queueing discipline imposed.

We have already demonstrated the containment relationship between the operational and denotational accounts of Demos in Tofts and Birtwistle [1997c]. Richer process algebras than CCS exist which can be used to account for the timing behavior [Moller and Tofts 1990] or the probabilistic behavior [Tofts 1990, 1994] of systems. However, timed descriptions are highly complex and in many cases system failures result not from timing problems, but rather from underlying interaction errors (see, e.g., Clarke et al. [1995], especially Section 7).

This article complements Birtwistle and Tofts [1994] and presents a denotational semantics for some of the compound instructions of μDemos. Short introductions to μDemos and CCS are given as appendices. For more detail see Birtwistle et al. [1993]. The reader should also be aware of Milner’s [1990] excellent book as the source guide to CCS.

In Section 2 we present a brief overview of the basic translations of the constructs of μDemos into CCS. In Section 3 we introduce the wait-until construction, which allows μDemos processes to perform compound acquisitions of resources, and demonstrate how such a command can be represented within our process algebra. The translation of a conditional branching construction is also presented in Section 4. We then use these translations to give a normal form for μDemos processes.

By providing a formal representation of a system described in μDemos we permit the production of proofs of properties of those systems. Given that these properties are often used to make cost or safety critical decisions, we need to be sure that systemic predictions are genuine and not the result of some programming artifact. CCS is well suited to this task, as it provides a succinct formal notation within which it captures the behavior of the underlying system. Furthermore, formal description of systemic properties provided in Hennessy–Milner logic can be automatically checked against its implementation as described by CCS.

2. TRANSLATING μDEMOS TO CCS

In this section we present the translation of an idealized object-oriented simulation language (μDemos [Birtwistle and Tofts 1994]) into CCS (the translation of wait-until is deferred to Section 3.1). μDemos may be considered as a clutter-free, syntactically checked, designated intermediate language. We demonstrate how a μDemos model can be represented as the composition of CCS processes. To achieve the translation, we must find process encodings for all of the parts that make up a μDemos program. Since process algebras in general and CCS in particular do not have such auxiliary data structures as monitors, queues, or variables, we have to provide explicit process constructions for all our system components. (A similar construction for shared variable While programs is presented in Milner [1990].)

We start with an informal presentation working from the bottom up, giving the style of, and intuition behind, the translation process. We show how resources are modeled as CCS agents, and then how process tasks are translated. Since a process body may be viewed as a sequence of tasks strung together, and a μDemos program a sequence of object definitions, the remainder of the translation process is straightforward. The product of the translation is a CCS description of the model whose properties, such as absence of deadlock and livelock, safety, liveness, and the like can be mechanically checked on the CWB. The validity of this methodology has been proven elsewhere [Tofts and Birtwistle 1997c]. In Section 2.4 we comment on three powerful theorems proved in that paper.

2.1 Translation of Resources

In translating a resource into a process, we use the actions *get*$\mathbf{R}$1, *get*$\mathbf{R}$2, . . . , *get*$\mathbf{Rn}$ to represent the acquisition of amounts 1 to $\mathbf{n}$, respectively, of the resource $\mathbf{R}$. The actions *put*$\mathbf{R}$1, *put*$\mathbf{R}$2, . . . , *put*$\mathbf{Rn}$ represent returning the respective amounts of resource to the resource pool.

A straightforward and elegant way of achieving the foregoing is to predefine a general template in CCS for a resource of size $\mathbf{n}$, for example,

$$\begin{array}{rcl} R(n,\, n) & \overset{def}{=} & \sum_{i=1}^{n} geti.R(n-i,\, n) \\ & \vdots & \\ R(j,\, n) & \overset{def}{=} & \sum_{i=1}^{j} geti.R(j-i,\, n) \\ & + & \sum_{i=1}^{i=n-j} puti.R(j+i,\, n) \\ & \vdots & \\ R(0,\, n) & \overset{def}{=} & \sum_{i=1}^{n} puti.R(i,\, n). \end{array}$$

To create a specific resource process with its appropriate access channels we "instantiate" the template and with its actions renamed:

$$Res(i, \mathbf{n}) \overset{def}{=} \begin{array}{l} R(i, \mathbf{n})[put\mathbf{R}j/putj, get\mathbf{R}j/getj] \\ where\ 1 \le j \le \mathbf{n}, \end{array}$$

which may be read as: $Res(i, \mathbf{n})$ is a resource of size $\mathbf{n}(R(i, \mathbf{n}))$ with all $putj$ actions renamed $put\mathbf{R}j$ and $getj$ actions renamed $get\mathbf{R}j$.

From the preceding we can observe that all resources of a particular size are identical up to renaming of the access channels.

In Demos a distinction is made between those objects that are claimed and released by the same process, called resources, and those into which items are placed by one process and removed by another, called bins. Unfortunately[1] at this point we have to assume that our bins are finite in order to successfully encode them. The only distinguishing feature between a finite bin and a resource is that we do not assume that the same process gives and takes items from the bin. In terms of its operation it is indistinguishable from a resource of the appropriate size.

We choose actions $put\mathbf{B}i$ and $get\mathbf{B}i$ to represent giving or taking i items from a bin **B**. Notice that these are both input actions as the bin is assumed to be responding to instructions from μDemos processes. With the observation that the bin is the same as a resource we can achieve a bin with a maximum capacity[2] of max_{bin} as the following process,

$$Bin(i, n) \overset{def}{=} \begin{array}{l} R(i, n)[put\mathbf{B}j/putj, get\mathbf{B}j/getj] \\ where\ 1 \le j \le max_{bin}. \end{array}$$

2.2 Translation of Processes

The entities that use resources are μDemos processes. It is natural to translate each of these processes as a separate CCS agent. Whenever a μDemos process performs an action **ACQUIRE(R, i)** we translate this into $\overline{get\mathbf{R}i}$, which is the complementary output action to the input actions of the resource. In this style of translation, we need do no checking to see whether this amount of resource is available as the action can only synchronize with its dual when the complementary input can be performed, which implies that the appropriate amount (or more) of the resource is indeed available. We make the sole assumption that processes do not attempt to hand back more of a resource than they took. In the vast majority of cases, resources are acquired and released in constant chunks, and this assumption can be checked syntactically.

[1]In practice this is the case in Demos as well. It is possibly a language design flaw that buffers and bins were distinguished originally—one of the problems of designing languages without a semantic basis from the outset.

[2]We assume that there is some fixed maximum capacity for all bins (max_{bin}) before embarking on the translation.

We use **HOLD(T)** actions as markers to permit tests upon system behavior to be written as Hennessy–Milner formulae. This translation is achieved directly by the use of $s\mathbf{T}.f\mathbf{T}$ (start time and finish time) actions for our representative processes. At this point for all of the activities of a μDemos process we have analogues in terms of actions for a CCS process. In the following summary, we write $\mathcal{A}[\![Obj]\!]$ to mean the translation of a μDemos action into a CCS action.

$$\mathcal{A}[\![\mathbf{ACQUIRE(R,\ i)}]\!] \overset{def}{=} \overline{get\mathbf{R}}i$$

$$\mathcal{A}[\![\mathbf{RELEASE(R,\ i)}]\!] \overset{def}{=} \overline{put\mathbf{R}}i$$

$$\mathcal{A}[\![\mathbf{TAKE(B,\ i)}]\!] \overset{def}{=} \overline{get\mathbf{B}}i$$

$$\mathcal{A}[\![\mathbf{GIVE(B,\ i)}]\!] \overset{def}{=} \overline{put\mathbf{B}}i$$

$$\mathcal{A}[\![\mathbf{HOLD(T)}]\!] \overset{def}{=} s\mathbf{T}.f\mathbf{T}$$

$$\mathcal{A}[\![\mathbf{COOPT(Q)}]\!] \overset{def}{=} \overline{coopt\mathbf{Q}}$$

$$\mathcal{A}[\![\mathbf{FREE(Q)}]\!] \overset{def}{=} \overline{free\mathbf{Q}}$$

$$\mathcal{A}[\![\mathbf{WAIT(Q)}]\!] \overset{def}{=} coopt\mathbf{Q}.free\mathbf{Q}.$$

We assume that the body of a μDemos process is a list of actions, using the infix operator $::$ to denote list concatenation. To translate a list of actions we simply recurse down the list translating each action in turn and prefixing to the remaining actions. Formally our translation of action sequences is as follows,

$$\mathcal{A}[\![action :: acts]\!] \overset{def}{=} \mathcal{A}[\![action]\!].\mathcal{A}[\![acts]\!].$$

The first translation we make is that for πDemos processes; we distinguish the processes that execute once from those that execute repeatedly, assuming that if a repeat is used it starts the process definition (process definitions can always be arranged in this way).

$$\mathcal{A}[\![newP(Name,\ \mathbf{repeat}(acts),\ \mathbf{time})]\!] \overset{def}{=} \mu X.\mathcal{A}[\![acts]\!].X$$
$$\mathcal{A}[\![newP(Name,\ acts,\ \mathbf{time})]\!] \overset{def}{=} \mathcal{A}[\![acts]\!].\mathbf{0}.$$

All the fix-point operator does is to produce arbitrarily many copies of the body of the process definition in sequence. It is merely a different way of writing $X \overset{def}{=} \mathcal{A}[\![acts]\!].X$ but avoids extra names. Notice that this implies two different interpretations of the empty list of actions; in one case it should be instantiated by a process variable, in the other by the **0** process. When we translate the wait-until construct it is necessary to make this separation clear, but for simplicity, we avoid it in this initial translation.

The close relationship between μDemos and CCS is demonstrated by the fact that sequential composition of μDemos actions is directly encoded as an appropriate sequence of action prefixes in CCS.

2.3 Translation of Programs

We are now in a position to translate entire μDemos programs into CCS. A μDemos program is simply a list of definitions, each specifying either a μDemos process or a resource. To achieve our translation we simply take each of these entities in turn and convert it into its representative CCS process. We then compose that collection of processes together in parallel and finally restrict that parallel composition with respect to the communication channels performing the exchanges of information between the parts of the μDemos program.

The first translation we make is that for μDemos processes:

$$\begin{aligned}&\mathcal{A}[\![NEWP(Name, Actions, \mathbf{time})]\!]\\ &\overset{def}{=} \mathcal{A}[\![Actions]\!].\mathbf{0}.\end{aligned}$$

To translate resources and bins we use the following translations,

$$\begin{aligned}&\mathcal{A}[\![NEWR(Name, max)]\!]\\ &\overset{def}{=} R(max, max)[getNamei/geti, putNamei/puti]\\ &where\ 1 \le i \le max\\ &\mathcal{A}[\![NEWB(Name, init)]\!]\\ &\overset{def}{=} R(init, max)[giveNamei/puti, takeNamei/geti]\\ &where\ 1 \le i \le max.\end{aligned}$$

The definitions were supplied in a list, so we need to take each of the entities specified and replace it by the appropriate CCS representation, finally composing those representations together in parallel. Thus we arrive at the following interpretation of a collection of μDemos definitions.

$$\begin{aligned}\mathcal{A}[\![Defn :: Defns]\!] &\overset{def}{=} \mathcal{A}[\![Defn]\!] | \mathcal{A}[\![Defns]\!]\\ \mathcal{A}[\![[]]\!] &\overset{def}{=} 0.\end{aligned}$$

Remember that in CCS, $P|\mathbf{0} = P$. So far we have translated all of the components of a μDemos program but at this point we have not forced them to exchange information; this is achieved by restriction on all the communication labels that have been defined as part of the representative CCS processes. We call the function that gives us the appropriate restriction set L, and define it recursively by cases over the syntax of μDemos.

For our resources we must ensure that each of the possible giving or taking actions is restricted in the eventual process. Hence we define:

$$L(NEWR(Name, max)) \overset{def}{=} \{getNamei, putNamei | 1 \leq i \leq max\}$$

$$L(NEWB(Name, max)) \overset{def}{=} \{giveNamei, takeNamei | 1 \leq i \leq max\}$$

$$L(NEWP(Name, Body, Time)) \overset{def}{=} \{cooptName, freeName\}.$$

Formally we define the set of restriction actions generated by a list of μDemos definitions as follows.

$$\begin{array}{lcl} L(Defn :: Defns) & \overset{def}{=} & L(Defn) \cup L(Defns) \\ L(\llbracket\rrbracket) & \overset{def}{=} & \emptyset. \end{array}$$

So now the final translation of a μDemos program into a CCS process is defined as follows.

$$SIM \overset{def}{=} (\mathscr{A}\llbracket Defns\rrbracket)\backslash L(Defns).$$

This semantics describes μDemos as a collection of interacting parallel components, matching the original intent of μDemos. It should be noted that both the resources and processes of μDemos are equal in this representation; they are all processes. The only difference between them is whether they generate input or output actions on channels. Resources are passive in that they only perform **input** actions. Processes can be considered active as they produce **output** actions (except when being coopted, in which case they are then viewed as subservient resources by their coopting process). The ability to change view of what is an entity and what is a resource (or bin) is often useful when producing system models. Notice that since CCS models both resources and processes as active agents, the techniques we espouse hold good for either style.

2.4 Application

In Birtwistle and Tofts [1993, 1994], Tofts and Birtwistle [1997a, 1997b], and this article we have given an operational semantics and a denotational semantics for μDemos. The two styles of semantic definition serve quite different ends. Operational definitions are quite detailed and may be used to guide an implementation (as in Birtwistle and Tofts [1993]) or argue in detail the precise way a particular program should unfold. Denotational definitions are much more abstract with detailed timings replaced by "now, or some arbitrary time later" and all queues (including the event list) running under random selection. Denotational definitions are thus useful for reasoning about properties that will hold under all possible runs of a model whatever the timing delays and whatever the queueing discipline imposed.

For these semantic definitions to be of full value, we must prove a containment relation between them. In Tofts and Birtwistle [1997c] we established that μDemos processes within CCS were conservative over all possible timing and queueing disciplines.

THEOREM 2.1. *SIM simulates the operational semantics of μDemos.*

Among the important auxiliary theorems proved in Tofts and Birtwistle [1997c] are:

THEOREM 2.2. *The CCS semantics of a μDemos system is independent of the timing model.*

Hence, we can make arbitrary changes to the timings in the μDemos program without affecting provable properties of the CCS description. In other words, the results we obtain for the denotational description will be true with arbitrary distributions in place of the fixed times in the holds, and for any model of the insertion of event notices at equal times into the event list.

THEOREM 2.3. *The CCS semantics of a μDemos system is independent of the queuing model.*

For all queueing models, the CCS denotation is the same wherever a process is inserted in the event list or in the blocked queue associated with a resource.

From the proof in Tofts and Birtwistle [1997c] it is clear that the position that a process is inserted in the event list after performing a hold or delaying action is irrelevant since all entities in that list have equal precedence as a result of the commutativity and associativity of the parallel operator of CCS, and similarly for processes blocked in the waiting queue of any resource. Thus all queueing disciplines are simulated by the CCS translation. Note that within this proof we do take account of the fact that the operational semantics of μDemos uses a first-come-first-served queueing discipline.

These results establish that the CCS description of a μDemos program has captured the structure of the computation, and also successfully abstracted the timing and queueing structures of the execution system. That is, we can execute the model with FIFO, LIFO, or random queueing and the provable properties of the CCS description will still hold under all such strategies. Thus properties established for the CCS description hold under all possible simulation executions. Hence, we can establish systemic properties such as deadlock and livelock freedom with the knowledge that they are independent of the execution model and thus will hold for all timing and queueing structures.

3. A NORMAL FORM

One of the uses of a denotational semantics is to discover normal forms for programming languages. Often these allow us to simplify either the language definition or its implementation. By demonstrating that only a restricted collection of syntactic forms are necessary for the description of any particular problem, the implementer can concentrate on rendering these language components in as efficient a form as possible. Equally, a normal form is highly desirable when attempting to prove properties of a system since these properties need be proved only for the normal forms, and not for the complete language.

In this section we start by giving a very general definition for the wait-until construct. This necessitates our extending our previous definition of a resource in that we must be able to test whether a specified amount of it is free. This in turn requires the introduction of conditionals. We give an informal presentation in Sections 3.1 and 3.2, after which the full technical details are presented in Section 3.3. Section 3.4 presents two theorems that enable us to define a normal form for μDemos programs in terms of wait-until.

Normal forms are of practical importance because they enable us to verify model properties in a standard, consistent, and simple manner.

3.1 Translation of Wait-Until

In Birtwistle et al. [1993] a simple example of the representation of a wait-until instruction is presented. We present a different example of the translation then generalize the constructions necessary to our subset of the Demos language.

Since we cannot form compound actions in CCS whose occurrence is atomic, we have to ensure that while the actions of the wait-until are being performed no other process can interfere with them. Furthermore we cannot simply issue requests to acquire necessary resources and return those claimed earlier if the request is denied. As a definition language CCS only permits us to describe actions that a process may perform, that is, its capabilities. We cannot in any sense observe that an action cannot be performed; it simply may be the case that we have to wait for it for a very long time. Therefore, we must adopt a regime where each object resource, bin, or process in the Demos system will inform a possible acquirer of both its availability or nonavailability with an explicit action (these were available in the original language through the function **Avail** for this very same purpose).

We solve the problem of atomic access by the introduction of a binary semaphore and the availability problem by the introduction of explicit, positive and negative, availability actions. As a small example consider a Demos process that needs to claim 1 unit of resource R (maximum availability 1) and the simultaneous cooperation of process Q in order to perform its task. First we can describe the resource R:

$$\begin{aligned} Sem &\overset{def}{=} gS.pS.Sem \\ R &\overset{def}{=} getR.R' + gavR.R \\ R' &\overset{def}{=} putR.R + nav.R'. \end{aligned}$$

Notice that the resource R has four access actions; two are the usual ones associated with getting or putting the resource, the other two ($gavR$ and $navR$) record whether the resource is available to be taken. We now provide a skeleton for the Demos process Q assuming that it is finishing a hold (called q), before becoming available to be coopted.

$$\begin{aligned} Q &\overset{def}{=} fq.Q1 + navQ.Q \\ Q1 &\overset{def}{=} cooptQ.Q2 + cavQ.Q1 \\ Q2 &\overset{def}{=} freeQ.Q' + nav.Q2 \\ Q' &\overset{def}{=} \ldots. \end{aligned}$$

The state Q' represents Q's continuing behavior after it has been coopted. Notice that again we have introduced two new actions $navQ$ and $cavQ$ that record whether the process Q can be coopted at this point in its execution.

We are now in a position to describe a process that waits until both the resource R and the process Q are available to be acquired and coopted, respectively. P starts by obtaining the token from the semaphore by handshaking on gS, possession of which permits it to claim potential resources.

$$\begin{aligned} P &\overset{def}{=} \overline{gS}.P1 \\ P1 &\overset{def}{=} \overline{gavR}.P2 + \overline{navR}.\overline{pS}.P \\ P2 &\overset{def}{=} \overline{cavQ}.P3 + \overline{navQ}.\overline{pS}.P \\ P3 &\overset{def}{=} \overline{getR}.\overline{cooptQ}.\overline{pS}.P'. \end{aligned}$$

In the foregoing process P must first obtain the acquisition token, then test that all the resources it needs are available. If they are not, it returns the acquisition token and will try again later. If its requirements can be met, the appropriate resources will be claimed and the token returned. Obviously in the preceding we have assumed that P is a process that cannot itself be coopted, as it can respond neither positively nor negatively to a $cavP$ signal. Encoding in general we would have to write:

$$\begin{aligned}
P &\overset{def}{=} \overline{gS}.P1 + navP.P \\
P1 &\overset{def}{=} \overline{gavR}.P2 + \overline{navR}.\overline{pS}.P + navP.P1 \\
P2 &\overset{def}{=} \overline{cavQ}.P3 + \overline{navQ}.\overline{pS}.P + navP.P2 \\
P3 &\overset{def}{=} \overline{getR}.\overline{cooptQ}.\overline{pS}.P' + navP.P3.
\end{aligned}$$

The foregoing process is already becoming somewhat tedious to write by hand. Our fragment of a complete Demos system would look as follows.

$$\begin{aligned}
SYS \overset{def}{=} {} & (P|Q|R|\ldots|Sem)\backslash \\
& \{gS, pS, navQ, cavQ, navP, cavP, getR, putR, navR, gavR\}.
\end{aligned}$$

As we have seen, we need to add information to process, bins, and resources so that other processes can detect whether a request to them will be successful. This is achieved by amending the definitions of bins and resources from Birtwistle and Tofts [1994] in the following way. In the general case, a resource needs to indicate precisely how much of it is available to be claimed. Obviously if more is available than necessary, it must indicate that lower amounts of resource are also all free. Similarly, we must indicate for all possible amounts whether more is not free than the current minimum nongrantable request.

$$\begin{aligned}
Res(m, m) &\overset{def}{=} \textstyle\sum_{i=1}^{i=m} geti.Res(m-i, m) \\
&\quad + \textstyle\sum_{i=1}^{i=m} gavi.Res(m, m) \\
Res(j, m) &\overset{def}{=} \textstyle\sum_{i=1}^{i=j} geti.Res(j-i, m) \\
&\quad + \textstyle\sum_{i=1}^{i=j} gavi.Res(j, m) \\
&\quad + \textstyle\sum_{i=1}^{i=m-j} puti.Res(j+i, m) \\
&\quad + \textstyle\sum_{i=j+1}^{i=m} navi.Res(j, m) \\
Res(0, m) &\overset{def}{=} \textstyle\sum_{i=1}^{i=m} puti.Res(j+i, m) \\
&\quad + \textstyle\sum_{i=1}^{i=m} navi.Res(0, m).
\end{aligned}$$

Similarly we can define our extended bins as follows.

$$\begin{aligned} Bin(i, max) &\overset{def}{=} Res(i, max)[givei/puti, takei/geti] \\ &\phantom{\overset{def}{=}} where\ 1 \leq \mathrm{i} \leq max. \end{aligned}$$

In the "hand" translation of processes we gave previously they needed to produce actions to record whether the process was currently available to be coopted. In order that our translated μDemos process have this (necessary) capability we must know the name of a process throughout its translation. So instead of a semantic function $\mathcal{A}[\![body]\!]$, we have a semantic function $\mathcal{A}[\![body, name]\!]$, where *name* is the (uniquely named) μDemos process under translation, and body is the remaining set of commands to be translated. The range of this function is

$$\mathcal{A}[\![]\!] : AcList \times Name \ \rightarrow \ CCS.$$

We also have semantic functions:

$$\begin{aligned} &\mathcal{D}[\![]\!] : DemosDefns \ \rightarrow \ CCS, \text{ and} \\ &\mathcal{W}[\![]\!] : \mathrm{AcList}\times\mathrm{AcList}\times\mathrm{AcList}\times Name \ \rightarrow \ CCS. \end{aligned}$$

Finally we take advantage of the presence of the process name to change our definition of the **repeat** operation to be compatible with that of Birtwistle and Tofts [1993]. We assume that the translation of the body of any μDemos process (called N) occurs within the scope of a μN operator (alternatively, $N \overset{def}{=}$) and thus whenever we see a **repeat** action we need only replace it with the appropriate process variable name.

3.2 Translation of Conditionals

We regard a conditional branch as having the form $COND([Cacq_i, P_i])$, where $Cacq_i$ is either a simple acquisition action or a wait-until, and P_i are the Demos actions to be executed if the branch is possible and taken. Notice, that we do not use explicit $CondQ$s, as they can be absorbed into the direct request for the resources conditioned upon, and then continue execution. We do not need to permit the returning of the resource to be guarded with a conditional as it is always possible to return resources that are owned.

Ideally we should like to translate the conditional statement as a nondeterministic sum of the acquire actions within the body of the statement. Unfortunately, as a result of encoding the wait-until construct, we no longer have direct access to the process but must go through a check to see that the process is available before attempting to acquire it.

As an example of the encoding we present the following problem. Consider processes P and Q competing for resources A, B, and C. Process P can proceed in one of two ways depending on whether it can acquire A and B, or B and C. Similarly, process Q can proceed in two different ways depending on whether it acquires A and C or B. Obviously we would like the compound acquisitions to be performed within a wait-until for efficiency. This system can be modeled by the following CCS processes.

First the fixed objects in the environment are:

$$\begin{aligned}
A1 &\overset{def}{=} getA1.BA1 + gavA1.A1\\
BA1 &\overset{def}{=} putA1.A1 + navA1.BA1\\
B1 &\overset{def}{=} getB1.BB1 + gavB1.B1\\
BB1 &\overset{def}{=} putB1.B1 + navB1.BB1\\
C1 &\overset{def}{=} getC1.BC1 + gavC1.C1\\
BC1 &\overset{def}{=} putC1.C1 + navC1.BC1\\
Sem &\overset{def}{=} gS.pS.Sem.
\end{aligned}$$

The first of the processes, P, is coded:

$$\begin{aligned}
P &\overset{def}{=} \overline{gS}.PC1\\
PC1 &\overset{def}{=} \overline{gavA}1.PC1R2 + \overline{navA}1.PC2\\
PC1R2 &\overset{def}{=} \overline{gavB}1.PC2C1 + \overline{navB}1.PC2.
\end{aligned}$$

In the foregoing process, we claim possession of the acquisition token, and then see if the first choice A and B is possible; if it is not, we see if the second is; even if the first branch is possible we still need to check the second branch, noting the availability of both.

This process knows that the first branch cannot be taken and checks to see if the second (claiming B and C) can be taken:

$$\begin{aligned}
PC2 &\overset{def}{=} \overline{gavB}1.PC21 + \overline{navB}1.\overline{pS}.P\\
PC21 &\overset{def}{=} \overline{gavC}1.PC2OK + \overline{navC}1.\overline{pS}.P\\
PC2OK &\overset{def}{=} \overline{getB}1.\overline{getC}1.\overline{p1}.pbr2.\overline{putB}1.\overline{putC}1.P.
\end{aligned}$$

If either of the resources is not available, then the process will retry the complete conditional statement, after allowing other processes access to the acquisition token by returning it. If it finds that it can claim the second branch (given that the first is not possible at this time), the resources will be claimed and the process proceed on the appropriate action sequence.

In the following process, the first branch could be taken, so it needs to check the availability of the second.

$$\begin{aligned}
PC2C1 &\overset{def}{=} \overline{gavB}1.PC21C1 + \overline{navB}1.PC1OK\\
PC21C1 &\overset{def}{=} \overline{gavR}.PC2C1OK + \overline{navC}1.PC1OK\\
PC2C1OK &\overset{def}{=} \overline{getB}1.\overline{getC}1.\overline{p1}.pbr2.\overline{putB}1.\overline{putC}1.P\\
&\quad + \overline{getA}1.\overline{getB}1.\overline{p1}.pbr1.\overline{putA}1.\overline{putB}1.P\\
PC1OK &\overset{def}{=} \overline{getA}1.\overline{getB}1.\overline{p1}.pbr1.\overline{putA}1.\overline{putB}1.P.
\end{aligned}$$

If the second is not available, claim the resources A and B and continue appropriately; otherwise make a nondeterministic choice between the two as both are possible.

The process Q is identical to P up to the nature of the resources claimed by the particular conditional statement:

$$\begin{aligned}
Q &\overset{def}{=} \overline{g1}.QC1\\
QC1 &\overset{def}{=} \overline{gavA}1.QC1R2 + \overline{navA}1.QC2\\
QC1R2 &\overset{def}{=} \overline{gavC}1.QC2C1 + \overline{navC}1.QC2\\
QC2 &\overset{def}{=} \overline{gavB}1.QC2OK + \overline{navB}1.\overline{p1}.Q\\
QC2OK &\overset{def}{=} \overline{getB}1.\overline{p1}.qbr2.\overline{putB}1.Q\\
QC2C1 &\overset{def}{=} \overline{gavB}1.QC2C1OK + \overline{navB}1.QC1OK\\
QC2C1OK &\overset{def}{=} \overline{getB}1.\overline{p1}.qbr2.\overline{putB}1.Q\\
&\quad + \overline{getA}1.\overline{getC}1.\overline{p1}.qbr1.\overline{putA}1.\overline{putC}1.Q\\
QC1OK &\overset{def}{=} \overline{getA}1.\overline{getC}1.\overline{p1}.qbr1.\overline{putA}1.\overline{putC}1.Q.
\end{aligned}$$

Finally constructing the appropriate restriction set and composing the system together, we obtain:

$$\begin{aligned}
L &\overset{def}{=} \{getA1, putA1, gavA1, navA1,\\
&\quad getB1, putB1, gavB1, navB1,\\
&\quad getC1, putC1, gavC1, navC1, p1, g1\\
SYS &\overset{def}{=} (P|Q|\ldots|Sem|A1|B1|C1)L.
\end{aligned}$$

Although we would not make any claims for the simplicity of the foregoing, it does encode an appropriate form of conditional statement for the Demos language. The general description of the preceding is going to be quite complicated because it has to check whether each branch is possible, store the result, and then choose among the available branches.

3.3 Technical Detail

To simplify the construction we define the following maps on actions.

Definition 3.1. A *map* $\mathcal{A}$ *: DemosAcq* $\rightarrow$ *Act* for acquisition actions:

$$\mathcal{A}(\mathbf{ACQUIRE}(\mathbf{R},\ \mathbf{n})) \overset{def}{=} \overline{get\mathbf{Rn}}$$
$$\mathcal{A}(\mathbf{TAKE}(\mathbf{B},\ \mathbf{n})) \overset{def}{=} \overline{take\mathbf{Bn}}$$
$$\mathcal{A}(\mathbf{COOPT}(\mathbf{P})) \overset{def}{=} \overline{coopt\mathbf{P}}.$$

A *map* $\mathcal{R}$: *DemosRet* $\rightarrow$ *Act* for release actions:

$$\mathcal{R}(\mathbf{RELEASE}(\mathbf{R},\ \mathbf{n})) \overset{def}{=} \overline{put\mathbf{Rn}}$$
$$\mathcal{R}(\mathbf{GIVE}(\mathbf{B},\ \mathbf{n})) \overset{def}{=} \overline{give\mathbf{Bn}}$$
$$\mathcal{R}(\mathbf{FREE}(\mathbf{P})) \overset{def}{=} \overline{free\mathbf{P}}.$$

A *map* $\mathcal{F}$: *DemosAcq* $\rightarrow$ *Act* to allow a process to determine the availability of a resource or a process:

$$\mathcal{F}(\mathbf{ACQUIRE}(\mathbf{R},\ \mathbf{n})) \overset{def}{=} \overline{gav\mathbf{Rn}}$$
$$\mathcal{F}(\mathbf{TAKE}(\mathbf{B},\ \mathbf{n})) \overset{def}{=} \overline{gav\mathbf{Bn}}$$
$$\mathcal{F}(\mathbf{COOPT}(\mathbf{P})) \overset{def}{=} \overline{cav\mathbf{P}}.$$

A *map* $\mathcal{N}$: *DemosAcq* $\rightarrow$ *Act* to allow a process to determine the nonavailability of a resource or a process:

$$\mathcal{N}(\mathbf{ACQUIRE}(\mathbf{R},\ \mathbf{n})) \overset{def}{=} \overline{nav\mathbf{Rn}}$$
$$\mathcal{N}(\mathbf{TAKE}(\mathbf{B},\ \mathbf{n})) \overset{def}{=} \overline{nav\mathbf{Bn}}$$
$$\mathcal{N}(\mathbf{COOPT}(\mathbf{P})) \overset{def}{=} \overline{nav\mathbf{P}}.$$

We are now in a position to translate our μDemos processes into CCS agents. To simplify our translation we make the following assumptions.

—action a ranges over all μDemos acquisition actions **ACQUIRE**(**R**, **n**), **TAKE**(**B**, **n**), **COOPT**(**P**)

—action r ranges over all μDemos return actions **RELEASE**(**R**, **n**), **GIVE**(**B**, **n**), **FREE**(**P**).

First we translate processes that have finished all their behaviors or are about to start again from the beginning:

$$\mathcal{A}[\![[\,], N]\!] \overset{def}{=} \mathbf{0}$$
$$\mathcal{A}[\![repeat, N]\!] \overset{def}{=} N;$$

a process with no further actions is the **0** agent and the action *repeat* starts execution again from the beginning of the process definition.

We can now translate acquisition-like actions in the following fashion,

$$\mathcal{A}[\![a :: body, N]\!] \overset{def}{=} \overline{gS}.(\mathcal{F}(a).\mathcal{A}(a).\overline{pS}.\mathcal{A}[\![body, N]\!]$$
$$+ \mathcal{N}(a).\overline{pS}.\mathcal{A}[\![a :: body, N]\!]) + navN.\mathcal{A}[\![a :: body, N]\!].$$

In the foregoing process, the acquisition semaphore is claimed, the resource is tested for availability, and if so claimed and the semaphore released; otherwise the semaphore is released and the process retries. Note that this process is not available to be coopted.

Release actions are translated as follows,

$$\mathcal{A}[\![r :: body, N]\!] \overset{def}{=} \mathcal{R}(r).\mathcal{A}[\![body, N]\!]$$
$$+ navN.\mathcal{A}[\![r :: body, N]\!].$$

This leaves the **WAIT**(**P**) action of the basic actions of a Demos process:

$$\begin{aligned}
&\mathcal{A}[\![\mathbf{WAIT}(\mathbf{P}) :: body, N]\!] \\
&\overset{def}{=} cooptN.Wait(\mathcal{A}[\![body, N]\!]) \\
&+ cavN.\mathcal{A}[\![\mathbf{WAIT}(\mathbf{P}) :: body, N]\!] \\
&where\ Wait(\mathcal{A}[\![body, N]\!]) \\
&\overset{def}{=} freeN.\mathcal{A}[\![body, N]\!] \\
&+ navN.W(\mathcal{A}[\![body, N]\!]).
\end{aligned}$$

Finally we translate hold actions as before:

$$\mathcal{A}[\![\mathbf{HOLD}(\mathrm{t}) :: body, N]\!] \overset{def}{=} st.ft.\mathcal{A}[\![body, N]\!].$$

In this sublanguage of Demos we have one compound construction, namely, the wait-until. This takes a list of acquisition actions and waits until they can all be simultaneously satisfied. We do not need to include release-like actions as they can always be performed, and we would never need to wait upon them. Since the wait-until itself consists of a list of actions, we make use of the previously cited auxiliary denotation function

$$\mathcal{W}[\![\,]\!] : AcList \times AcList \times AcList \times Name \rightarrow CCS:$$

$$\mathscr{A}\llbracket \mathbf{WU}(Acts) :: body, N\rrbracket$$
$$\overset{def}{=} \overline{gS}.\mathscr{W}\llbracket Acts, Acts, Acts', N\rrbracket$$

and define the auxiliary denotation function as follows,

$$\begin{aligned}
&\mathscr{W}\llbracket a :: acl, a1, body, N\rrbracket\\
&\overset{def}{=} \mathscr{F}(a).\mathscr{W}\llbracket acl, a1, body, N\rrbracket\\
&+ \mathscr{N}(a).\overline{pS}.\mathscr{A}\llbracket \mathbf{WU}(a1) :: body, N\rrbracket\\
&\mathscr{W}\llbracket [\,], a :: acl, body, N\rrbracket\\
&\overset{def}{=} \mathscr{A}(a).\mathscr{W}\llbracket [\,], acl, body, N\rrbracket\\
&\mathscr{W}\llbracket [\,], [\,], body, N\rrbracket\\
&\overset{def}{=} \overline{pS}.\mathscr{A}\llbracket body, N\rrbracket.
\end{aligned}$$

We also need to add *Cond* to the basic action semantics for μDemos, and are thus required to define the following two auxiliary semantic functions.

$$\begin{aligned}
&\mathscr{C}\llbracket\rrbracket : (CacqList \times (Cacq \times body)list) \times (body \times Name) \rightarrow CCS, \text{ and}\\
&\mathscr{G}\llbracket\rrbracket : AcqList \rightarrow CCS.\\
&\mathscr{A}\llbracket Cond(L) :: body, N\rrbracket\\
&\overset{def}{=} \overline{gS}.\mathscr{C}\llbracket (L, [\,]), (Cond(L) :: body, N)\rrbracket\\
&+ navN.\mathscr{A}\llbracket Cond(L) :: body, N\rrbracket.
\end{aligned}$$

Notice the preceding has to inform other processes that it is not available to be coopted at the moment. The definition of the auxiliary semantic functions for the *Cond* construction is:

$$\begin{aligned}
&\mathscr{C}\llbracket ((a :: L, P) :: t, AvList), Init\rrbracket\\
&\overset{def}{=} \mathscr{F}(a).\mathscr{C}\llbracket ((t, (a, P) :: AvList)), Init\rrbracket\\
&+ \mathscr{N}(a).\mathscr{C}\llbracket ((t, AvList)), Init\rrbracket\\
&\mathscr{C}\llbracket [\,], [\,], Init\rrbracket\\
&\overset{def}{=} \overline{pS}.\mathscr{A}\llbracket Init\rrbracket\\
&\mathscr{C}\llbracket [\,], [(a, P)], (Cond(L) :: body, N)\rrbracket\\
&\overset{def}{=} \mathscr{A}(a).\overline{pS}.\mathscr{A}\llbracket P@body, N\rrbracket\\
&\mathscr{C}\llbracket [\,], (a, P) :: t, (Cond(L) :: body, N)\rrbracket\\
&\overset{def}{=} \mathscr{A}(a).\overline{pS}.\mathscr{A}\llbracket P@body, N\rrbracket\\
&\mathscr{C}\llbracket ((\mathbf{WU}(aL) :: L, P) :: t, AvList), Init\rrbracket
\end{aligned}$$

$$\overset{def}{=} \mathcal{WA}[\![aL, (t, (\mathbf{WU}(aL), P) :: AvList)).$$
$$(t, AvList), Init]\!]$$
$$\mathcal{C}[\![\mathbb{I}, [(\mathbf{WU}(L), P)], (Cond(L) :: body, N)]\!]$$
$$\overset{def}{=} \mathcal{G}[\![L]\!].\overline{pS}.\mathcal{A}[\![P@body, N]\!]$$
$$\mathcal{C}[\![\mathbb{I}, (\mathbf{WU}(L), P) :: t, (Cond(L) :: body, N)]\!]$$
$$\overset{def}{=} \mathcal{G}[\![L]\!].\overline{pS}.\mathcal{A}[\![P@body, N]\!] + \mathcal{C}[\![\mathbb{I}, t, (Cond(L) :: body, N)]\!].$$

The semantic function that checks to see if a wait-until is possible is:

$$\mathcal{WA}[\![\mathbb{I}, P_{av}, P_{nav}, Init]\!]$$
$$\overset{def}{=} \mathcal{C}[\![P_{av}, Init]\!]$$
$$\mathcal{WA}[\![a :: t, P_{av}, P_{nav}, Init]\!]$$
$$\overset{def}{=} \mathcal{F}(a).\mathcal{WA}[\![t, P_{Av}, P_{nav}, Init]\!]$$
$$+ \mathcal{N}(a).\mathcal{C}[\![P_nav, Init]\!].$$

Finally the semantic function to perform the appropriate series of acquisitions is:

$$\mathcal{G}[\![[a]]\!] \overset{def}{=} \mathcal{A}(a)$$
$$\mathcal{G}[\![a :: t]\!] \overset{def}{=} \mathcal{A}(a).\mathcal{G}[\![t]\!].$$

We can use the same semantics for definitions, just extending the range of the function $\mathcal{A}[\![]\!]$ as given previously.

We can now present the translation of the definitional part of a Demos program. First define two relabeling maps

$$Rl1 \overset{def}{=} [getRi/geti, putRi/puti,$$
$$gavRi/gavi, navRi/navi$$
$$where\ 1 \leq i \leq max$$
$$Rl2 \overset{def}{=} [takeBi/geti, giveBi/puti,$$
$$gavBi/gavi, navBi/navi$$
$$where\ 1 \leq i \leq max$$

and our translation of the object definitions:

$$\mathcal{D}[\![NEWP(P, body, t) :: defns]\!]$$
$$\overset{def}{=} \mu P.\mathcal{A}[\![body, P]\!] | \mathcal{D}[\![defns]\!]$$
$$\mathcal{D}[\![NEWR(R, max) :: defns]\!]$$

$$\overset{def}{=} Res(max, max)Rl1|\mathscr{D}[\![defns]\!]$$
$$\mathscr{D}[\![NEWB(B, max) :: defns]\!]$$
$$\overset{def}{=} Res(0, max)Rl2|\mathscr{D}[\![defns]\!].$$

To complete our translation we define the restriction sort of a collection of μDemos definition:

$$L(defn :: defns)$$
$$\overset{def}{=} L(defn) \cup L(defns)$$
$$L(\mathbf{NEWP}(\mathbf{P}, \mathbf{ACTIONS}, \mathbf{y}))$$
$$\overset{def}{=} \{cooptP, freeP, cavP, navP\}$$
$$L(\mathbf{NEWR}(\mathbf{n}, \mathbf{amount}))$$
$$\overset{def}{=} \{get\mathbf{n}i, put\mathbf{n}i, gav\mathbf{n}i, nav\mathbf{n}i | 1 \leq i \leq amount\}$$
$$L(\mathbf{NEWB}(\mathbf{n}, \mathbf{amount}))$$
$$\overset{def}{=} \{take\mathbf{n}i, give\mathbf{n}i, gav\mathbf{n}i, nav\mathbf{n}i | 1 \leq i \leq amount\}.$$

The translation of a μDemos program is:

$$\begin{aligned} Sem &\overset{def}{=} gS.pS.Sem \\ SIM_{CCS}(P) &\overset{def}{=} (\mathscr{D}[\![Defns]\!]|Sem) \\ &\quad \backslash L(Defns) \cup \{gS, pS\}. \end{aligned}$$

3.4 Application to the Normal Form

We can now prove the following two facts about μDemos.

THEOREM 3.2. *Let a be an acquire type action, then*

$$\mathscr{A}[\![a :: body]\!] \sim \mathscr{A}[\![\mathbf{WU}([a]) :: body]\!].$$

The proof follows immediately from the translation. As an example consider the μDemos fragments:

```
⋮                     ⋮
Acquire(Sem, 1);      WU[Acquire(Sem, 1)];
P                     P
```

The CCS translations of these fragments are, respectively,

$$
\begin{aligned}
&\mathcal{A}[\![Acquire(Sem,\ 1);\ P]\!]\\
&\overset{def}{=}\ \overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]\\
&+\overline{navSem}1.\overline{pS}.\mathcal{A}[\![Acquire(Sem,\ 1);\ P]\!]\\
&=\mu X.\overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]\\
&+\overline{navSem}1.\overline{pS}.X\mathcal{A}[\![WU[Acquire(Sem,\ 1)];\ P]\!]\\
&\overset{def}{=}\ \overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]\\
&+\overline{navSem}1.\overline{pS}.\mathcal{A}[\![WU[Acquire(Sem,\ 1)];\ P]\!]\\
&=\mu X.\overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]\\
&+\overline{navSem}1.\overline{pS}.X.
\end{aligned}
$$

$\mathcal{A}[\![Acquire(Sem,\ 1);\ P]\!]$ and $\mathcal{A}[\![WU[Acquire(Sem,\ 1)];\ P]\!]$ denote the same function since they have identical definitions when expressed as fix-point equations.

THEOREM 3.3. *If* **WU**(L) *is a Demos action, then*

$$\mathcal{A}[\![\mathbf{WU}(L)\ ::\ body]\!] \sim \mathcal{A}[\![\mathbf{COND}(\mathbf{WU}(L))\ ::\ body]\!].$$

Again, the proof follows immediately from the translation.

Considering a third version of our code fragment:

```
⋮
COND(WU[Acquire(Sem, 1)], P);
⋮
```

we obtain the CCS translation:

$$
\begin{aligned}
&\mathcal{A}[\![COND[WU[Acquire(Sem,\ 1)],\ P]]\!]\\
&\overset{def}{=}\ \overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]\\
&+\overline{navSem}1.\overline{pS}.\\
&\mathcal{A}[\![COND[WU[Acquire(Sem,\ 1)],\ P]]\!]\\
&=\mu X.\overline{gS}.(\overline{gavSem}1.\overline{getSem}1.\overline{pS}.\mathcal{A}[\![P]\!]+\overline{navSem}1.\overline{pS}.X
\end{aligned}
$$

which is identical to the earlier translation.

The fact that these two theorems obtain is reassuring, since we would intuitively expect them to hold from a naive understanding of the behavior of μDemos programs. From these two facts we can define a normal form for μDemos processes.

Definition 3.4. A Demos process is in *normal form* if it is either:

(1) **repeat**(*body*);

(2) r :: *body*;

(3) **COND**(**WU**(L_i) :: *body*$_i$) :: *body*;

where *body*, $body_i$ are in normal form, r is any return type action, and L_i is a list of acquire type actions.
This syntactically small language has the same expressive power as that of μDemos.

4. CONCLUSIONS AND FURTHER WORK

Increasingly the results of simulated systems are used to make decisions of both cost and safety criticality. Hence, the ability to formally derive properties of simulation systems is of great importance. Our translations from Demos (-like) syntax to Edinburgh Concurrency Workbench [Moller 1990] syntax have been automated; the ability to study simulation systems with these formal systems allows the early detection of design errors.

We have been able to represent a large class of syntactic constructions used in the presentation of simulation problems within the CCS frame. Exploiting these descriptions we can produce normal forms for the underlying languages which begin to address the notion of what constitute the basic constructions of a simulation problem, and allow system designers and implementers to concentrate on the fundamental interactions present in such systems.

In later work, we hope to be able to address issues of representation. Simulation systems admit many possible representations within languages such as Demos, making comparisons between different implementations of the same problem system difficult. In particular there is a duality between those components that are represented as "full-blooded" processes, and those that are represented as auxiliary resources or bins; often these depend on the focus of the implementer's interest. Removing the (artificial) distinction between these entities should allow us to reconcile descriptions of such systems written from any point of view.

Our description of μDemos within CCS was in part motivated by the availability of a tool to formally demonstrate properties of the described systems. More detailed semantic accounts of the timing or probabilistic properties of μDemos can be obtained by using appropriate process calculi; Moller and Tofts [1990] and Tofts [1990, 1994] are examples. When suitable tools exist for exploiting process algebraic descriptions of systems within these calculi it may be useful to extend our descriptions to include these properties. However, the majority of problems within simulation systems are caused by interaction errors (leading to deadlock and livelock, for example) and CCS is more than expressive enough to address these important problems.

The production of a normal form for the μDemos language permits an implementer to concentrate on the basic features of the language. Since any program can be translated to normal form, any implementation that efficiently evaluates normal forms will be an efficient evaluator for the complete language. In the production of proof methods the normal form allows for direct comparison of programs. That is, two programs with the same normal form should execute in identical fashion.

APPENDIX A. μDEMOS

μDemos programs are abstractions consisting of resource declarations, process declarations, and a simulation run length.

Res, bin, and queue declarations take the form:

—NEWR(Rid, max) which defines a new resource *Rid* of size *max* that may be *acquired* and later *released* in chunks. Resources are used to handle finite resources.

—NEWB(Bid, max) defines a new bin *Bid* of maximal capacity *max* that may be *give*n and *take*n in integer chunks. Bins are typically used as buffers to smooth the flow of materials between processing agents.

—NEWQ(Qid) defines a new queue *Qid* in which processes may *wait*, be *coopt*ed by another process, and later *free*d. The queue synchronization is used to handle cooperation among processes.

Process declarations typically take one of the forms:

—NEWP(Pid$_1$, repeat(actions), t$_1$) which defines a process Pid_1 that exhibits its cyclic behavior starting at time t_1;

—NEWP(Pid$_2$, actions, t$_2$) which defines a process Pid_2 that exhibits linear behavior starting at time t_2;

The process bodies are (possibly cyclic) lists of get and put actions on resources. As an example, here is a very short, but complete μDemos program for the classic producer consumer problem, which here runs for 100 time units:

```
[  NEWB("B", 4),
   NEWP("P", repeat[hold(5), give("B", 1)], 0),
   NEWC("C", repeat[take("B", 1), hold(4)], 0),
   hold(100)
].
```

Since delay times are abstracted in CCS, we have contented ourselves here with constant times.

APPENDIX B. CCS

To define CCS we presuppose a set A of atomic action symbols and the dual of the input $a \in A$ being $\bar{a}$. These complements a and $\bar{a}$ are the basis of communication as input and output actions, respectively. The collection of CCS expressions ranged over by E is defined by the following BNF expressions, where a ranges over the set $Act \stackrel{def}{=} \mathrm{A} \cup \tau$, *Names* a set of names; $A \subseteq Names$; S: $Act \rightarrow Act$ such that $\overline{S(a)} = S(\bar{a})$ and $S(\tau) = \tau$; X a set of variable names. Pr is the set of closed process expressions E.

$$E \ :: = a.E|X|0|E + E|E|E|E[S]|E\backslash A|X \stackrel{def}{=} E).$$

The intuitive interpretation of these terms can be given as follows.

$$\frac{}{a.E \xrightarrow{a} E}$$

$$\frac{E \xrightarrow{a} E'}{E + F \xrightarrow{a} E'} \qquad \frac{F \xrightarrow{a} F'}{E + F \xrightarrow{a} F'}$$

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{\bar{a}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

$$\frac{E \xrightarrow{a} E'}{E|F \xrightarrow{a} E'|F} \qquad \frac{F \xrightarrow{a} F'}{E|F \xrightarrow{a} E|F'}$$

$$\frac{E \xrightarrow{a} E'}{E[S] \xrightarrow{S(a)} E'[S]}$$

$$\frac{E \xrightarrow{a} E' \quad a, \bar{a} \notin A}{E\backslash A \xrightarrow{a} E'\backslash A}$$

$$\frac{E \xrightarrow{a} E' \quad X \overset{def}{=} E}{X \xrightarrow{a} E'}$$

Fig. 1. Operational semantics of CCS.

—X represents the process bound to the variable X in some assumed environment.

—$a.P$ represents the process that can perform the (atomic) action a and evolve into the process P upon so doing.

—0 represents the completely dead process. It cannot perform any computation. In terms of a concrete machine, it may be thought of as one that is shut down or with its "plug pulled."

—$P + Q$ represents the notion of choice between the two processes P and Q. The process behaves as the process P or the process Q, with the choice being made only at the time of the first action.

—$P|Q$ represents the parallel composition of the two processes P and Q. Each of the processes may perform any actions independently, or they may synchronize on complementary actions, resulting in a τ action.

—$P\backslash a$ represents the process P with the action $a \in \mathcal{L}$ (as well as its complement action $\bar{a}$) restricted away, that is, not allowed to occur.

—$P[S]$ represents the process P with its actions renamed by the relabeling function f.

—$P\backslash a$ represents the process P with the action $a \in \mathcal{L}$ (as well as its complement action $\bar{a}$) restricted away, that is, not allowed to occur.

—$X \overset{def}{=}$ E represents the (least fixed point) solution to the equation.

REFERENCES

BIRTWISTLE, G. 1979. *Demos—Discrete event modeling on Simula.* Macmillan, New York, NY.

BIRTWISTLE, G. 1981. Demos implementation guide and reference manual. Tech. Rep., 81/70/22, University of Calgary.

BIRTWISTLE, G. AND TOFTS, C. 1993. An operational semantics of process-orientated simulation languages: Part I πDemos. *Trans. Soc. Comput. Simul. 10*, 4, 299–333.

BIRTWISTLE, G. AND TOFTS, C. 1994. An operational semantics of process-orientated simulation languages: Part II μDemos. *Trans. Soc. Comput. Simul. 11*, 4, 303–336.

BIRTWISTLE, G., POOLEY, R., AND TOFTS, C. 1993. Characterizing the structure of simulation models in CCS. *Trans. Soc. Comput. Simul. 10*, 3, 205–236.

CLARKE, E., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D., MCMILLAN, K., AND NESS, L. 1995. Verification of the Futurebus+ cache coherence protocol. *Formal Meth. Syst. Des. 6*, 2 (March), 217–232.

MILNER, R. 1980. *A Calculus of Communicating Systems.* LNCS 92, Springer Verlag, New York, NY.

MILNER, R. 1990. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, NJ.

MOLLER, F. 1990. The Edinburgh Concurrency Workbench. Tech. Rep., Department of Computer Science, University of Edinburgh.

MOLLER, F. AND TOFTS, C. 1990. A temporal calculus of communicating systems. In *CONCUR '90*, LNCS 458, Springer-Verlag, New York, NY, 401–405.

STIRLING, C. 1987. Modal logics for communicating systems. *J. Theor. Comput. Sci. 49*, 311–347.

STIRLING, C. 1992. Modal and temporal logics for processes. Tech. Rep. ECS-LFCS-92-221, Department of Computer Science, University of Edinburgh.

TOFTS, C. 1990. A calculus of relative frequency. In *CONCUR '90*, LNCS 458, Springer-Verlag, New York, NY, 467–480.

TOFTS, C. 1994. Processes with probability, priority and time. *Form. Asp. Comput. Sci. 6*, 5, 536–564.

TOFTS, C. AND BIRTWISTLE, G. 1997a. A denotational semantics for Demos: Part I. Tech. Rep. 97, 12, School of Computer Studies, University of Leeds.

TOFTS, C. AND BIRTWISTLE, G. 1997b. A denotational semantics for Demos: Part II. Tech. Rep. 97, 13, School of Computer Studies, University of Leeds.

TOFTS, C. AND BIRTWISTLE, G. 1997c. A relationship between an operational and a denotational account of Demos. *J. Simul. Pract. Theor. 5*, 1, 1–33.

VAN VLIJMEN, S. F. M. AND VAN WAVEREN, A. 1992. An algebraic specification of a model factory. Tech. Rep., University of Amsterdam Programming Research Group.

Received January 1996; revised March 1997; accepted December 1997