A Guide to Specifying Concurrent Behaviour with *Abacus*[1]

Oscar Nierstrasz[2]

Abstract

We present the syntax, semantics and usage of Abacus, an executable notation for specifying concurrent computations that extends CCS with *label prefixing* and *filtering* operators for encapsulating systems of communicating agents and a *pattern* mechanism for parameterizing behaviour expressions. Abacus is intended to be used as a semantic target and a prototyping tool for the specification of concurrent object-based languages and systems. We illustrate the use of Abacus through a series of standard concurrency examples, concluding with an executable specification of *SAL*, a Simple Actor Language.

1. Introduction

Abacus is a notation for specifying the behaviour of concurrent systems, intended primarily as a prototyping tool to support the design of concurrent object-based languages and systems. One may either directly specify concurrent systems with Abacus or one may use it as a semantic target for the specification of language constructs. Syntactic patterns of a programming language are thus mapped to behavioural patterns expressed in Abacus in a denotational fashion [7]. Since Abacus specifications are executable, one immediately obtains a running prototype of an interpreter for the language being designed. In this way the benefits of exploratory prototyping support and complement those of formal specification techniques.

Abacus is based on Milner's process calculus and provides the standard operators of the basic calculus [15]. Value-passing, on the other hand, is simulated by *patterns*, which are functions that evaluate to agents of the calculus. Patterns also serve as semantic functions taking as arguments syntactic constructs of a source language and returning Abacus specifications.

The rule-based specification of the calculus and of the corresponding implementation makes it very easy to add new operators more convenient for modeling certain kinds of behaviours. These may either be added as patterns or as new primitive operators in the notation. We present *label prefixing* and *filtering*, which are particularly useful for encapsulating systems of cooperating agents.

We shall present the syntax and semantics of Abacus in §2. We then proceed to illustrate the use of Abacus through a series of progressively more ambitious examples from the standard concurrency literature, concluding with an executable specification of *SAL*, a Simple Actor Language [1]. As an appendix we provide the code for a minimal implementation in Prolog.

1. In *Object Management*, ed. D.C. Tsichritzis, CUI, University of Geneva, July 1990, pp. 267-293.

2. Author's address: Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland. E-mail: oscar@cui.unige.ch, oscar@cgeuge51.bitnet. Tel: +41 (22) 787.65.80. Fax: +41 (22) 735.39.05.

2. Abacus Syntax and Semantics

Computations in Abacus are modeled as systems of communicating agents. Two agents may communicate if an *output offer* for a communication event made by one agent is matched by a corresponding *input offer* from another agent. Whenever a communication event takes place the two participating agents replace themselves by their new behaviour.

The current state of any agent or system of agents is captured explicitly by a *behaviour expression*, which determines what input and output offers are made by agents or subsystems and, consequently, which events may take place. The occurrence of an event yields a new behaviour expression for each of the participating agents, and thus for the whole system. We shall first present the abstract syntax of agent declarations and behaviour expressions and then the semantic rules that permit us to interpret them.

2.1 Abstract syntax

Abacus is essentially equivalent to CCS [15], modifying the syntax somewhat to simplify implementation, and adding two new operators that are convenient for encapsulating systems of cooperating agents. In the following, *A* stands for an *agent name*, *B* for a *behaviour expression*, *E* for an *event label*, and *X* for a *prefix*. For the present we shall suppose that agent names are identifiers. When we introduce *patterns*, we shall see that agent names may be parameterized, thus giving us the possibility of defining a set of agent names with a single declaration. Event labels are either identifiers or tuples enclosed in [square brackets]. If *E* is an event label and *X* a prefix, then *X*:*E* is also an event label.

An agent name is bound to a behaviour expression by a declaration of the form: *A* := *B* .

Behaviour expressions have the following syntax:

1.	*A*	*Behaviour of agent A*
2.	nil	*Inactive agent*
3.	*B* & *B*	*Concurrent composition*
4.	*B* + *B*	*Summation* (exclusive choice)
5.	*E* ! *B*	*Output offer*
6.	*E* ? *B*	*Input offer*
7.	*X* : *B*	*Prefixing*[1]
8.	*B* \: *X*	*Filtering*
9.	*B* \ *E*	*Restriction*
10.	*B* / [*E*/*E*, ...]	*Relabelling*

Table 1 The syntax of behaviour expressions

1. The term *prefixing* is used in CCS to refer to input and output offers preceding a behaviour expression; we shall use “prefixing” in this paper to refer to label prefixing.

The operators are listed in order from loosest to tightest binding so, for example: p&q+u\:x will be parsed as: p & (q + (u\:x)) .

2.2 Transition semantics

We define the semantics of behaviour expressions by a set of *transition rules*. For every behaviour expression there may be several *visible transitions* to replacement behaviour expressions, corresponding to *offers* to communicate, and several *invisible transitions* corresponding to internal communications.

We write $p \xrightarrow{e} p'$ to indicate that p offers to input e and replace itself by p', and $p \xrightarrow{\bar{e}} p'$ if p offers to output e and become p'. Furthermore, we adopt the convention that $\bar{\bar{e}} = e$. If $a = \bar{b}$ we say that offers a and b *match.*

If p represents a system of concurrent agents, these agents may communicate with one another. In this case we write $p \xrightarrow{\tau} p'$ to indicate that p may make an invisible transition to p'. Such a transition is "invisible" because it is no longer visible as an offer to agents external to p. As we shall see, it is possible that p may simultaneously support both visible and invisible transitions.

In the semantic rules that follow, the expressions over the bar represent preconditions and those under the bar the conclusions. The symbol e represents a visible transition, τ an invisible transition, and α represents either.

Input, Output and Summation

$$\frac{}{e?p \xrightarrow{e} p} \qquad \frac{}{e!p \xrightarrow{\bar{e}} p} \qquad \frac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p+q \xrightarrow{\alpha} q'}$$

The first rule tells us that a?nil describes an agent that offers to input a and then become nil (i.e., it terminates or becomes inactive). Similarly, the second rule tells us that a!nil will output a and become nil, and that a!b!nil will first output a, then b and then terminate.

The two rules for summation tell us that a?nil+b?nil offers to input *either* a or b, then terminate. Note that summation is both associative and commutative, thus p+q is equivalent to q+p and (p+q)+v is equivalent to p+(q+v). [The equivalent forms in CCS for $e?p$ and $e!p$ are respectively $e.p$ and $\bar{e}.p$. The + operator is identical to that of CCS.]

Concurrent composition

$$\frac{p \xrightarrow{e} p',\ q \xrightarrow{\bar{e}} q'}{p\&q \xrightarrow{\tau} p'\&q'} \qquad \frac{p \xrightarrow{\alpha} p'}{p\&q \xrightarrow{\alpha} p'\&q} \qquad \frac{q \xrightarrow{\alpha} q'}{p\&q \xrightarrow{\alpha} p\&q'}$$

The first composition rule tells us that concurrent agents may communicate if they present matching input and output offers, thus a!b!nil & a?nil+b?nil may silently change state to b!nil & nil by an internal communication a.

The next two rules tell us that concurrent agents may communicate independently with other agents that may be present, thus a!b!nil & a?nil+b?nil may output an a to an external agent and become b!nil & a?nil+b?nil or it may input either an a or a b to become a!b!nil & nil.

We can also conclude that the & operator is both associative and commutative.

[In CCS, one writes p|q instead of p&q. In Abacus, + binds more tightly than &, thus reducing parentheses in most of our examples, whereas in CCS, | binds more tightly than +.]

Restriction and Relabelling

$$\frac{p \xrightarrow{\alpha} p',\ \ \alpha \notin \{e, \bar{e}\}}{p\backslash e \xrightarrow{\alpha} p'\backslash e} \qquad \frac{p \xrightarrow{\alpha} p'}{p/[f] \xrightarrow{f(\alpha)} p'/[f]}$$

Restriction is used to hide input and output offers for a particular event label with the effect that the corresponding communication can *only* occur internally. For example, in the expression (a!b!nil & a?nil+b?nil)\a the a offers are hidden from external agents. Thus either an internal communication may take place, yielding (b!nil & nil)\a, or an external agent may communicate a b yielding (a!b!nil & nil)\a. In the latter case, no further actions are possible, since the output offer for a can neither be matched internally, nor is it visible externally. [In CCS one may restrict a set of labels, as in p\{a,b}. To do so in Abacus one restricts each member of the set, as in p\a\b.]

Relabelling is used to change the label of a visible transition. Relabelling functions are written as a finite sequence of *replacement/transition* mappings, such as [a/b,c/d], which maps b to a and d to c. All transitions not explicitly mentioned are mapped to themselves (τ is never relabelled). For example, in the expression: ((a!b!nil)/[b/a] & a?nil+b?nil)\a the output offer for a has been relabelled as an offer for b. The first agent is therefore equivalent to b!b!nil.

There are three possible transitions:

1. An internal b event yielding: ((b!nil)/[b/a] & nil)\a
2. An external output of b by the first agent yielding: ((b!nil)/[b/a] & a?nil+b?nil)\a
3. An external input of b by the second agent yielding: ((a!b!nil)/[b/a] & nil)\a

In all three cases further transitions are possible.

[Relabelling is identical to the corresponding operator in CCS, except no slash is required before the function, i.e., in CCS, one would write p[a/b,c/d] instead of p/[a/b,c/d].]

Prefixing and Filtering

$$\frac{p \xrightarrow{e} p'}{x{:}p \xrightarrow{x:e} x{:}p'} \qquad \frac{p \xrightarrow{\tau} p'}{x{:}p \xrightarrow{\tau} x{:}p'} \qquad \frac{p \xrightarrow{x:e} p'}{p\backslash{:}x \xrightarrow{e} p'\backslash{:}x} \qquad \frac{p \xrightarrow{y:e} p',\ x \neq y}{p\backslash{:}x \xrightarrow{y:e} p'\backslash{:}x} \qquad \frac{p \xrightarrow{\tau} p'}{p\backslash{:}x \xrightarrow{\tau} p'\backslash{:}x}$$

The first rule tells us that a prefix applied to a behaviour expression (rather than to just a single offer) has the effect that all offers of that agent and its descendents will be prefixed, so x:(a!b!nil) is equivalent to x:a!x:b!nil. The second rule tells us that prefixing has no effect on internal transitions. For finite label sets, one can simulate prefixing by a relabelling function, for example, the preceding expressions are also equivalent to (a!b!nil)/[x:a/a,x:b/b].

Filtering is used to hide all *except* prefixed offers. If the filter argument matches the prefix, it is stripped off, otherwise the prefix remains. For example, (x:a?nil+b?nil)\:x permits only a vis-

ible offer to input a. Since the prefix matches, it is removed. The b offer is not prefixed, so it does not pass through the filter.

Filtering is useful for encapsulating systems of cooperating agents. Whereas restriction can be used to hide a specific list of visible transitions, filtering hides all *except* prefixed transitions. As we shall see, this will provide us with a convenient mechanism for specifying the visibility scope of an offer by using prefixes to represent the name of a scope.

[There is no direct equivalent to filtering in CCS, though one can generally simulate it by a combination of restriction and relabelling.]

Agent declarations

$$\frac{p := q, \quad q \xrightarrow{\alpha} q'}{p \xrightarrow{\alpha} q'}$$

When we bind an agent name to a behaviour expression, that name may in future be used to stand for that expression. It is by this means that we may define non-terminating, recursive behaviours, such as: res := a?res + b?res. This agent repeatedly offers to input either an a or a b.

It should be clear that occurrences of an agent name appearing in its own recursive definition must be *guarded* [11][15], i.e., preceded by an input or output offer. For example,

```
p := p.
```

defines nothing at all, as our inference rules do not allow us to conclude $p \xrightarrow{\alpha} p'$ for any α.

3. Communication, Concurrency and Synchronization

Let us take the recursively-defined agent res introduced above and use it to model a shared resource for a number of concurrent clients:

```
res := a?res + b?res.
```

Clients c1 and c2 each present two output offers and then terminate:

```
c1 := a!a!nil.
c2 := b!b!nil.
```

We may compose the resource with its two clients as follows:

```
example1 := res & c1 & c2.
```

Within example1, res will accept the offers of c1 and c2 interleaved arbitrarily (there are six possible interleavings: aabb, abab, abba, baab, baba and bbaa). In all cases, the final configuration will be: res & nil & nil. Since the nil agents contribute nothing to the behaviour of this system, it is equivalent to res. We shall make use of such basic equivalences to simplify many of the examples.

Suppose that clients require exclusive access to a resource for a period during which they may make multiple requests. One way of accomplishing this is for clients to synchronize by means of a binary semaphore [3][6]:

```
bsem := p!v?bsem + v?bsem.
```

Recall that to acquire a semaphore one performs a P and to release it, a V. Our bsem agent is initially available, offering a p to any interested client. (We could just as well have made p an input offer, but we find it more intuitively appealing to think of clients *requesting* a p but *issuing* a v.) Once a p has been delivered bsem replaces itself by v?bsem, which refuses all further p requests until a v has been received. Note that bsem will *always* accept a v request, but that it simply discards multiple v's. (In the next section we shall see how to model a counting semaphore).

Within the system example2, clients c3 and c4 synchronize via bsem before communicating with the resource:

```
c3 := p?a!a!v!nil.
c4 := p?b!b!v!nil.
example2 := res & bsem & c3 & c4.
```

Now there are only two possible computation paths, namely paavpbbv and pbbvpaav, with c3 and c4 having exclusive access to res.

4. Encapsulation

It is often useful to encapsulate subsystems by restricting the visibility of offers to a certain scope. Abacus provides two complementary sets of operators for encapsulation: restriction and relabelling, which are used to hide only selected offers, and filtering and prefixing, which are used to hide all *but* a selected set of offers. In most of our examples we shall use filtering and prefixing, as they yield very compact specifications, however we shall encounter at least one situation in which restriction is more convenient.

Let us consider the specification of a counting semaphore. The semaphore is initially available, permitting a p event, but it also remembers how many v events have occurred and accepts one p request for every matching v. We can specify such a semaphore as follows:

```
sem := p!v?sem + v?(d?s:sem & avail\:x)\:s.
avail := s:p!x:d!nil + s:v?(d?avail & avail\:x).
```

The agent sem permits one p event, with replacement v?sem, or one v event, with replacement (d?s:sem & avail\:x)\:s. Upon each v event, a new avail agent is created to "remember" the v. The incremented semaphore is encapsulated by a \:s filter, which permits *only* prefixed offers to be exported. As a consequence, the agent d?s:sem must wait for a matching d offer from the agent avail\:x. If a d event occurs, it will be internal to the encapsulated semaphore. The avail\:x agent is itself encapsulated by a \:x filter. This permits us to *link* together the agents internal to the semaphore. At most one avail agent will ever be able to communicate with external agents; all others will be forced to wait for their neighbour to consume a p offer.

Since the entire semaphore is encapsulated, an avail agent can only communicate with external agents by prefixing its offers with s:. Note that the offers s:p and s:v pass through *two* filters. They first pass unaltered through \:x, and then they pass through the filter \:s, which strips off the s: prefix. The offers visible to the outside are simply p! and v?. Let us step through the state changes of sem when composed with the agent v!v!v!p?p?nil. First, a v event yields:

```
v!v!p?p?nil & (d?s:sem & (avail)\:x)\:s
```

Two more v events:

```
v!p?p?nil & (d?s:sem & (d?avail & (avail)\:x)\:x)\:s
p?p?nil & (d?s:sem & (d?avail & (d?avail & (avail)\:x)\:x)\:x)\:s
```

At this point it may help to visualize the agents of the system to see how they are linked together:

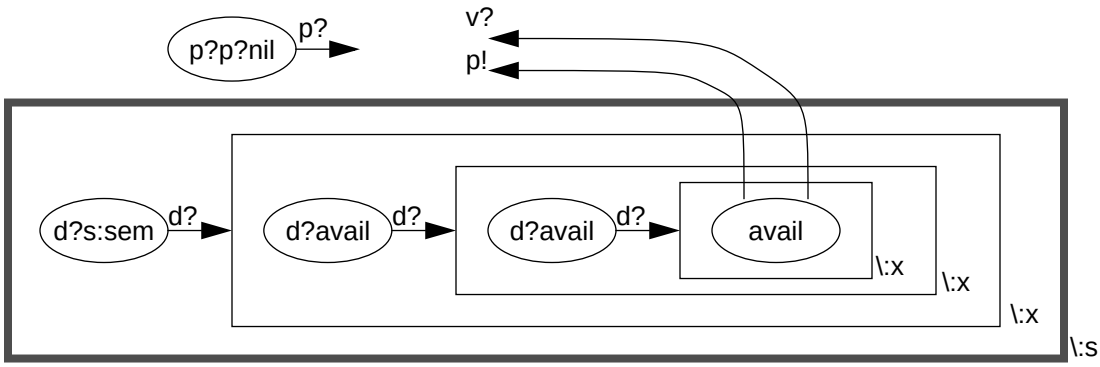

Figure 1 Communication offers in a counting semaphore agent

The arrows in this figure represent offers to communicate. Thus, the innermost avail agent can communicate with the external agent because its p and v offers use the unique prefix s of the enclosing filter. The other agents internal to the semaphore are linked because they are each enclosed by a non-unique \:x filter.

A p event yields:

```
p?nil & (d?s:sem & (d?avail & (d?avail & (x:d!nil)\:x)\:x)\:x)\:s
```

Now an internal d event takes place. The x:d offer is only visible to the nearest adjacent avail agent as the prefix is immediately consumed by the \:x filter:

```
p?nil & (d?s:sem & (d?avail & (avail)\:x)\:x)\:s
```

Another p and another internal d:

```
nil & (d?s:sem & (d?avail & (x:d!nil)\:x)\:x)\:s
nil & (d?s:sem & (avail)\:x)\:s
```

As expected, we finally obtain the same semaphore that we had after a single v.

We can specify an equivalent semaphore using restriction and relabelling. The agent rsem is adapted from an example by Milner:

```
rsem := p!v?rsem + v?( pos/[unlink/done] & unlink?rsem ) \ unlink.
pos := p!done!nil + v?( pos/[unlink/done] & unlink?pos ) \ unlink.
```

In this case we build up a chain of pos agents to count the v events. To ensure that each agent only communicates with its nearest neighbour, it offers to output a done, which is immediately translated to an unlink by the relabelling /[unlink/done]. The unlink is then hidden by the restriction \unlink from surrounding agents. Let us see what happens when we compute v!v!p?p?nil & rsem. First, the two v offers are accepted:

```
v!p?p?nil & (pos/[unlink/done] & unlink?rsem)\unlink
p?p?nil & ((pos/[unlink/done]&unlink?pos)\unlink/[unlink/done] & unlink?rsem)\unlink
```

Since the p and v offers of pos are neither restricted nor relabelled, they are visible to the outside. Next, a p is accepted:

```
p?nil & (((done!nil)/[unlink/done]&unlink?pos)\unlink/[unlink/done] & unlink?rsem)\unlink
```

The done offer is relabelled as an unlink and consumed by the neighbouring unlink?pos. Since unlink offers are restricted, they are not visible to unlink?rsem:

```
p?nil & (pos\unlink/[unlink/done] & unlink?rsem)\unlink
```

Another p is accepted:

```
((done!nil)\unlink/[unlink/done] & unlink?rsem)\unlink
```

And finally the internal unlink yields:

```
(rsem)\unlink
```

Note that rsem\unlink is equivalent to rsem, since in any case it makes no unlink offers. In fact, it is relatively simple to prove that sem and rsem are equivalent specifications, since they have precisely the same *external* behaviour.[1]

As an aside, it is interesting to note that Abacus is computationally complete since it is possible to simulate a Turing machine by using a slight variation of our counting semaphore. We can exploit the fact that each semaphore effectively stores a *stack* of avail or pos agents. It is quite easy to define a binary stack that stores and returns 0 or 1 values by modifying the sem specification. The infinite tape of a Turing machine can then be modelled by two such stacks, one for the tape symbols to the left, and the other for those on the right. We can change the current position by popping a value off one stack and pushing it onto the other. The finite logic of the Turing machine can be also expressed as an Abacus specification. Since we can model a Turing machine using *either* prefixing and filtering or restriction and relabelling, this means that either set of operators is as powerful as the other. They do not have the same expressive capabilities, however, and there appears to be no straightforward way to translate specifications using one set to those using the other.

5. Using Patterns to Model Value-Passing

Patterns are simply functions that evaluate to agents. Syntactically, patterns are parameterized behaviour expressions or agent names. Patterns can be used to express value-passing, but they are also more generally useful for defining classes of agents, agents with dynamically modifiable behaviour, new operators and, as we shall see, semantic functions for defining programming language constructs.

Our approach to modelling value-passing between agents differs somewhat from that of Milner, but has the same general flavour in the sense that patterns ultimately translate to agents of the basic notation. Since the Abacus interpreter is implemented in Prolog, we exploit Prolog's unification mechanism to interpret patterns. (Aside from the ability to define new operators, we generally avoid the use of advanced features of Prolog in the examples; it should therefore be possible for the reader to follow the examples without having any prior exposure to Prolog.)

1. They are *observation equivalent* [15] since we can establish a "bisimulation" relation between the reachable states of either; furthermore, since both sem and rsem are *stable* – that is, they do not initially seek to change state through an internal event – we can conclude that they are *observation congruent*, i.e., equal as specifications.

Variables in Prolog begin with upper case letters. We may define a parameterized agent just as we would an ordinary agent by supplying Prolog variables as parameters to the agent's name and using those variables in the behaviour expression bound to that name. Furthermore, all variables used in any behaviour expression must be introduced *either* as parameters to the agents *or* as parameters to an input offer.

Let us take as a simple example the specification of an agent that simulates the tuple space of Linda [4]. Linda provides a small set of primitives to allow concurrent processes to communicate and synchronize by writing and reading tuples to a so-called tuple space. A process may write a tuple using the non-blocking *out* primitive, and a process may read a tuple either destructively with the *in* primitive, or non-destructively with the *rd* primitive. Both read primitives block if no matching tuple exists. The following agent, linda, supports these three primitives:

```
linda := [out,T]?(linda & tuple(T)).
tuple(T) := [in,T]!nil + [rd,T]!tuple(T).
```

In this example we introduce compound event labels as tuples enclosed in square brackets[1]. In response to an [out,T] request, linda will generate a tuple agent that stores the value T. An agent attempting to input [in,T] will block unless such a tuple exists. An [in,T] event is destructive, causing the tuple to be consumed, whereas a [rd,T] event is non-destructive.

In the specification of tuple(T) the only variable that occurs is a parameter to the agent. We can interpret this as a definition of a *set* of agents tuple(T), for all possible values of T, i.e.,

```
tuple(a) := [in,a]!nil + [rd,a]!tuple(a).
tuple(b) := [in,b]!nil + [rd,b]!tuple(b).
...
```

and so on. In this interpretation, each tuple(T) is a new agent name.

In the specification of linda, on the other hand, the variable T is not an agent parameter but is introduced in an input offer. In this case we must understand the definition of linda as an infinite sum of behaviour expressions [out,T]?(linda & tuple(T)) for all possible values of T, i.e.,

$$\text{linda} := \sum_{T} [\text{out},T]?(\text{linda} \,\&\, \text{tuple}(T)) \ .$$

We can simulate the behaviour of a counting semaphore by generating a tuple(sem) for each V and consuming one for each P. Our synchronizing clients now look like this:

```
c5 := [in,sem]?a!a![out,sem]!nil.
c6 := [in,sem]?b!b![out,sem]!nil.
```

And the system looks like this:

```
linda & tuple(sem) & res & c5 & c6
```

We must start with a single instance of tuple(sem) to permit an initial P. As before, c5 and c6 obtain exclusive access to the resource by acquiring the semaphore.

1. We adopt this convention primarily to improve readability: there is nothing to prevent us, for example, from writing out(T) rather than [out,T] , but we then risk confusing the parameterized *event label* out(T) with a parameterized *agent* called out(T).

Incidentally, the *eval* primitive of Linda can be easily simulated by creating an agent that evaluates an expression before replacing itself by a tuple. The non-blocking variants of *in* and *rd* (*inp* and *rdp*) are more problematic, however, as they require the ability to detect the *absence* of a particular tuple[1].

6. Using Patterns to Specify Agents

Specifying Classes of Agents

There is no inherent reason why pattern parameters must be restricted to the domain of event labels. Consider the following alternative specification of a counting semaphore:

```
psem := p!v?psem + v?inc(psem).
inc(S) := p!S + v?inc(inc(S)).
```

As with the example of the previous section, we can interpret the inc(S) pattern as defining a *set* of agents with names inc(psem), inc(inc(psem)) etc. Provided a pattern is well-defined, the semantics of a pattern is that of the agents it evaluates to.

The agent psem is equal (observation congruent) to both sem and rsem defined earlier, even though it performs no internal events. In this sense Abacus specifications (just as CCS specifications) are *fully abstract*: they specify only external behaviour, not implementations. Any two specifications that exhibit identical external behaviour are to be considered interchangeable.

Specifying Operators

The pattern inc(S) can also be viewed as an *operator* over the domain of behaviour expressions. We may similarly define binary operators over behaviour expressions as patterns. Let us suppose that we have instructed Prolog to recognize ~ as a right-associative infix operator[2]. We could then define ~ as a *linking* operator as follows:

```
P ~ Q := P & Q\:x.
```

(Of course, this means that the prefix x has become “special” since it has its own operator.)

We may now define a version of the counting semaphore using the linking operator:

```
lsem := p!v?lsem + v?(d?s:lsem ~ next)\:s.
next := s:p!x:d!nil + s:v?(d?next ~ next).
```

Since ~ is extremely useful for linking together a series of communicating agents, we actually provide it as a supplementary operator to Abacus rather than as a pattern. This has the convenient side-effect that linking will be visible in all reachable states, rather than being translated to its equivalent form using & and \:x. For example, after three v events, lsem reaches the state:

```
(d?s:lsem~d?next~d?next~next)\:s
```

which is equivalent to its (more verbose) translation:

```
(d?s:lsem & (d?next & (d?next & next\:x)\:x)\:x)\:s
```

1. One way of modelling this would be to introduce a “clearinghouse” agent that keeps track of which tuples currently exist in the tuple space.

2. This is done using the built-in predicate op/3, as in :- op(660,xfy,~).

Linking is *not* associative, since p~q~u = p~(q~u) = p & (q & u\:x)\:x, which is not the same as: (p~q)~u = (p & q\:x) & u\:x.

A Concurrent Queue

We can use the linking operator to specify a queue whose head and tail may be accessed concurrently by a producer and a consumer:

```
queue := [put,X]?(head(X) ~ tail)\:q.
head(X) := q:[get,X]!ok!nil.
tail := q:[put,X]?(x:ok?head(X) ~ tail) + x:ok?q:queue.
```

The queue initially accepts only a [put,X] request. Subsequent states consist of a chain of agents starting with a head(X) agent that attempts to deliver its contents to a consumer, zero or more head agents each waiting to become the true head of the queue, and a tail that accepts further [put,X] requests.

Suppose we have the following producer and consumer:

```
prod := [put,a]![put,b]![put,c]!nil.
cons := [get,X]?[get,Y]?[get,Z]?nil.
```

In the system: (cons & queue & prod) the put and get requests may be arbitrarily interleaved (provided there are no more gets than puts!). If the producer succeeds in outputting all its values before the consumer reads any, the queue will reach the state:

```
(head(a)~x:ok?head(b)~x:ok?head(c)~tail)\:q
```

After the consumer finishes reading the queue, we reach the state:

```
(nil~nil~ok!nil~tail)\:q
```

An internal ok yields (nil~nil~nil~q:queue)\:q, which is equivalent to queue.

Concurrent Bounded Buffers

Linking is also useful for passing responsibilities amongst a collection of cooperating agents. As an example we shall specify a pattern for arbitrary-length concurrent bounded buffers. We shall model the buffer as a chain of agents that may hold the values written to the buffer. There is always at most one agent at the head of the chain responding to [put,X] offers and always at most one at the tail responding to [get,X] offers. Whenever a value is read or written, the responsibility of being the head or the tail passes on to the next agent. When the end of the chain is reached, the responsibility cycles back to the beginning of the chain. When the buffer is empty or full, requests to get or put are respectively blocked.

Initially the buffer is empty and an empty agent acts as the tail. If a value is written, this agent becomes the head and passes the responsibility of being the tail to the agent "behind it" by communicating ok. (There is always a free agent or the end of the buffer following empty.)

```
empty := b:[put,X]?ok!head(X).
free := x:ok?tail.
```

When tail accepts a put request it knows that the buffer is not empty and so becomes a taken(X) agent that waits to become the head until that responsibility is passed to it:

```
tail := b:[put,X]?ok!taken(X) + x:ok?empty.
taken(X) := x:ok?head(X).
```

If the responsibility of being the head passes to tail, then we know that the buffer is empty again, and tail simply becomes empty.

When the buffer is not empty the head attempts to deliver its value to a consumer. If the responsibility of being the tail passes to the head then we know the buffer is full:

```
head(X) := b:[get,X]!ok!free + x:ok?full(X).
full(X) := b:[get,X]!ok!tail.
```

In addition, we need to handle the eventuality that the last agent in the chain tries to pass the responsibility to the "next" agent. We close the loop with the agent end that simply repeats the communication to the agent at the start of the chain. The agent start passes the communication on to the first agent in the chain:

```
end := x:ok?a:x:ok!end.
start := ok?ok!start.
```

Finally, a bounded buffer is an encapsulated system consisting of a start agent and a chain of agents of the form empty~free~free~...~end:

```
buf(Chain) := (start & Chain\:a\:x)\:b.
```

To see the buffer pattern in action, let us consider the system (cons & buf(empty~free~end) & prod) consisting of a consumer, a two-slot buffer and a producer, where prod and cons are:

```
prod := [put,a]![put,b]![put,c]!nil.
cons := [get,X]?[get,Y]?[get,Z]?nil.
```

If we follow one possible computation path, we see the buffer undergo the following transitions:

```
[put,a]   → (start & ((ok!head(a)~free~end)\:a)\:x)\:b
ok        → (start & ((head(a)~tail~end)\:a)\:x)\:b
[put,b]   → (start & ((head(a)~ok!taken(b)~end)\:a)\:x)\:b
ok        → (start & ((head(a)~taken(b)~a:x:ok!end)\:a)\:x)\:b
ok        → (ok!start & ((head(a)~taken(b)~end)\:a)\:x)\:b
ok        → (start & ((full(a)~taken(b)~end)\:a)\:x)\:b
[get,a]   → (start & ((ok!tail~taken(b)~end)\:a)\:x)\:b
ok        → (start & ((tail~head(b)~end)\:a)\:x)\:b
[put,c]   → (start & ((ok!taken(c)~head(b)~end)\:a)\:x)\:b
ok        → (start & ((taken(c)~full(b)~end)\:a)\:x)\:b
[get,b]   → (start & ((taken(c)~ok!tail~end)\:a)\:x)\:b
ok        → (start & ((taken(c)~tail~a:x:ok!end)\:a)\:x)\:b
ok        → (ok!start & ((taken(c)~tail~end)\:a)\:x)\:b
ok        → (start & ((head(c)~tail~end)\:a)\:x)\:b
[get,c]   → (start & ((ok!free~tail~end)\:a)\:x)\:b
ok        → (start & ((free~empty~end)\:a)\:x)\:b
```

Note that in our specification of the bounded buffer, there is no global "locking" of the buffer to synchronize or inhibit concurrent requests to put or get values as is the case with a solution based on monitors [10]. Instead, our solution permits producers and consumers to concurrently access the buffer, as in solutions using critical sections [8] or synchronizing resources [2], except that we achieve synchronization by distributing responsibilities rather than by maintaining global knowledge of the state of the buffer.

A Concurrent Prime Sieve

In certain situations it is convenient to define agents that compute the value of simple expressions. We shall make use of such agents to specify a concurrent prime sieve. gen(J,N) is an agent that outputs the values [test,J] for all values of J up to N. It will be used to generate a list of numbers for the sieve to test:

```
gen(J,N) := [test,J]!gen(K,N)          :- J<N, K is J+1.
gen(N,N) := [test,N]!nil.
```

We express the behaviour of gen(J,N) through the use of a Horn clause that verifies that J is less than N and then computes the value of K for the replacement behaviour. Otherwise, if J=N, then the next value to test is generated and the agent terminates. As with our previous pattern examples, we interpret this as the definition of a set of agents named gen(0,0), gen(0,1), etc.:

```
gen(0,0) := [test,0]!nil .
gen(0,1) := [test,0]!gen(1,1) .
gen(1,1) := [test,1]!nil .
gen(0,2) := [test,0]!gen(1,2) .
gen(1,2) := [test,1]!gen(2,2) .
gen(2,2) := [test,2]!nil .
...
```

We similarly define agents eq(X,Y), which reports whether X is equal to Y, div(N,P), which reports whether N is divisible by P, and square(P), which outputs the value of P^2:

```
eq(X,Y) := true!nil                    :- X=Y.
eq(X,Y) := false!nil                   :- not(X=Y).
div(N,P) := true!nil                   :- 0 is N mod P.
div(N,P) := false!nil                  :- not(0 is N mod P).
square(P) := [val,P2]!nil              :- P2 is P*P.
```

The prime sieve itself consists of a chain of agents, each of which stores a prime number and performs tests on candidate primes, and a prime generator, which adds new primes to the end of the chain. If a candidate fails a test it is discarded. If it passes a test it is forwarded to the next prime in the chain for testing. A candidate that passes all division tests up to its square root is approved as a prime. A sieve to compute primes up to N is defined as the following pattern:

```
primes(N) := gen(3,N) ~ last(2,4) ~ genprime.
```

The agent gen(3,N) generates integers to test starting with 3. The last(P,P2) agent approves as a prime any number less than P2 (the square of the prime P) since that number is not divisible by any prime up to its square root. When it encounters P2 itself, that number is discarded (since it is divisible by P), and the agent replaces itself by sieve(P), which performs division tests on candidates and forwards those that pass the test to the next agent in the chain.

```
last(P,P2) := x:[test,N]?(eq(N,P2) & findprime(N,P,P2)).
findprime(N,P,P2) := true?sieve(P) + false?p:[prime,N]!last(P,P2).

sieve(P) := x:[test,N]?(div(N,P) & dotest(N,P)).
dotest(N,P) := true?sieve(P) + false?[test,N]!sieve(P).
```

Finally, genprime is the prime generator, whose responsibility it is to append new primes to the end of the chain:

```
genprime := p:[prime,P]?((square(P) & [val,P2]?last(P,P2)) ~ genprime).
```

As a demonstration, consider the following execution trace of primes(10) and note how the work performed by the various agents is interleaved to reflect their concurrent execution:

```
[test,3]    → gen(4,10)~(eq(3,4)&findprime(3,2,4))~genprime
false       → gen(4,10)~p:[prime,3]!last(2,4)~genprime
p:[prime,3] → gen(4,10)~last(2,4)~(square(3)&[val,X]?last(3,X))~genprime
[test,4]    → gen(5,10)~(eq(4,4)&findprime(4,2,4))~(square(3)&[val,X]?last(3,X))~genprime
true        → gen(5,10)~sieve(2)~(square(3)&[val,X]?last(3,X))~genprime
[test,5]    → gen(6,10)~(div(5,2)&dotest(5,2))~(square(3)&[val,X]?last(3,X))~genprime
false       → gen(6,10)~[test,5]!sieve(2)~(square(3)&[val,X]?last(3,X))~genprime
[val,9]     → gen(6,10)~[test,5]!sieve(2)~last(3,9)~genprime
[test,5]    → gen(6,10)~sieve(2)~(eq(5,9)&findprime(5,3,9))~genprime
[test,6]    → gen(7,10)~(div(6,2)&dotest(6,2))~(eq(5,9)&findprime(5,3,9))~genprime
true        → gen(7,10)~sieve(2)~(eq(5,9)&findprime(5,3,9))~genprime
[test,7]    → gen(8,10)~(div(7,2)&dotest(7,2))~(eq(5,9)&findprime(5,3,9))~genprime
false       → gen(8,10)~[test,7]!sieve(2)~(eq(5,9)&findprime(5,3,9))~genprime
false       → gen(8,10)~[test,7]!sieve(2)~p:[prime,5]!last(3,9)~genprime
p:[prime,5] → gen(8,10)~[test,7]!sieve(2)~last(3,9)~(square(5)&[val,X]?last(5,X))~genprime
[test,7]    → gen(8,10)~sieve(2)~(eq(7,9)&findprime(7,3,9))
                                    ~(square(5)&[val,X]?last(5,X))~genprime
[test,8]    → gen(9,10)~(div(8,2)&dotest(8,2))~(eq(7,9)&findprime(7,3,9))
                                    ~(square(5)&[val,X]?last(5,X))~genprime
true        → gen(9,10)~sieve(2)~(eq(7,9)&findprime(7,3,9))
                                    ~(square(5)&[val,X]?last(5,X))~genprime
[test,9]    → gen(10,10)~(div(9,2)&dotest(9,2))~(eq(7,9)&findprime(7,3,9))
                                    ~(square(5)&[val,X]?last(5,X))~genprime
false       → gen(10,10)~[test,9]!sieve(2)~(eq(7,9)&findprime(7,3,9))
                                    ~(square(5)&[val,X]?last(5,X))~genprime
false       → gen(10,10)~[test,9]!sieve(2)~p:[prime,7]!last(3,9)
                                    ~(square(5)&[val,X]?last(5,X))~genprime
p:[prime,7] → gen(10,10)~[test,9]!sieve(2)~last(3,9)~(square(5)&[val,X]?last(5,X))
                                    ~(square(7)&[val,Y]?last(7,Y))~genprime
[test,9]    → gen(10,10)~sieve(2)~(eq(9,9)&findprime(9,3,9))~(square(5)&[val,X]?last(5,X))
                                    ~(square(7)&[val,Y]?last(7,Y))~genprime
[test,10]   → ((div(10,2)&dotest(10,2))~(eq(9,9)&findprime(9,3,9))~(square(5)&[val,X]?last(5,X))
                                    ~(square(7)&[val,Y]?last(7,Y))~genprime)\:x
true        → (sieve(2)~(eq(9,9)&findprime(9,3,9))~(square(5)&[val,X]?last(5,X))
                                    ~(square(7)&[val,Y]?last(7,Y))~genprime)\:x
true        → (sieve(2)~sieve(3)~(square(5)&[val,X]?last(5,X))
                                    ~(square(7)&[val,Y]?last(7,Y))~genprime)\:x
[val,25]    → (sieve(2)~sieve(3)~last(5,25)~(square(7)&[val,Y]?last(7,Y))~genprime)\:x
[val,49]    → (sieve(2)~sieve(3)~last(5,25)~last(7,49)~genprime)\:x
```

7. Defining a Programming Language

As we stated initially, our purpose in developing Abacus was to use it as a specification and prototyping tool to support the design of computational models and language constructs for concurrent object-based programming languages. We shall now step through an example of how one might use Abacus to specify a small programming language. Rather than invent a new language we shall take SAL, the Simple Actor Language introduced by Agha [1] to explain the actor model. In this way we not only demonstrate at least some degree of generality in our approach, but we also show how a notation based on synchronous message passing and dynamic agent creation (i.e., Abacus) is at least as powerful as one based on asynchronous message passing (i.e., SAL), thus reinforcing the observation of Liskov et al. [13] that either asynchrony or an extendible process structure are necessary to obtain adequate expressive power for concurrent or distributed computing.

We shall start by giving a short introduction to actors before presenting the syntax and informal semantics of SAL. We use a standard example of a factorial actor to illustrate some of the features of SAL. Then we provide an overview of the Abacus specification of SAL, followed by the specification itself. We close with part of the trace of the running factorial actor to demonstrate the correspondence between SAL's actor model and Abacus agents.

7.1 Actors

Actors are computational entities that communicate by asynchronous message-passing[1][9]. An actor consists of a queue of pending messages and a "behaviour" that accepts and responds to messages. Every actor is associated with a unique identifier which is the "mail address" of its message queue. An actor may know the mail addresses of other actors which are its *acquaintances*. When an actor accepts a message, it can do three things:

1. Create new actors.
2. Send messages to its acquaintances.
3. Specify the replacement behaviour to handle the next message.

An actor automatically becomes acquainted with any new actors it creates. This permits it to send messages to a new actor, or to send its mail address to another actor that will become acquainted with it. (An actor that has no pending messages and with which no other actor is acquainted is effectively dead.) An actor is normally acquainted with itself, and so can always send itself messages. The replacement behaviour may be specified at any time, thus permitting an actor to begin processing the next message concurrently with the processing of the current one.

7.2 SAL

We have modified the syntax of SAL only slightly in order to take advantage of Prolog's ability to support user-defined operators. We give the abstract syntax for SAL below in extended BNF. Non-terminals are in *italics*, optional items are within [tall square brackets], and zero or more repetitions are within {brace brackets}* with a trailing asterisk. Keywords and literals are in **bold**. *beh-name*, *selector*, *target* and *name* are all identifiers. *acquaintance-list* and *parameter-list* are instances of *name-list*.

initial-behaviour ::= **initially** *command*

behaviour-definition ::= **def** *beh-name* [**with** *acquaintance-list*]
accept *selector* [**:** *parameter-list*] **=>** *command*
{ **or** *selector* [**:** *parameter-list*] **=>** *command* }*

name-list ::= **[** *name* { **,** *name* }* **]**

command ::= **skip** | *command* **;** *command* | **{** *command* **}**
| **send** *selector* [**:** *expression-list*] **to** *target*
| **become self**
| **become** *beh-name* [**with** *expression-list*]
| **if** *logical-expression* **then** *command* [**else** *command*]
| **let** *name* **=** *expression* { **and** *name* **=** *expression* }* **in {** *command* **}**

expression ::= *number* | *name* | *expression-list*
| *expression* **+** *expression* | *expression* **-** *expression*
| *expression* ***** *expression* | *expression* **/** *expression*
| **new** *beh-name* [**with** *expression-list*]

expression-list ::= **[** *expression* { **,** *expression* }* **]**

logical-expression ::= *expression* **=** *expression*

A SAL program consists of a set of actor behaviour definitions and an initial behaviour (a command to execute). Each actor has a unique mail address, a queue of pending messages, and a current behaviour responsible for handling the next message. An actor may have a list of acquaintances, which are the mail addresses of other actors it may send messages to. An actor may always send a message to itself by using the pseudo-variable **self** as a target. Messages contain a *selector* and an optional list of values. A behaviour specifies how to handle the next message by indicating for each possible selector what command to execute.

An actor may evaluate arithmetic and logical expressions (to keep SAL simple, we only provide a tiny set of arithmetic and logical operators), send messages to acquaintances, create new actors and specify its replacement behaviour. It is possible to temporarily assign names to the values of expressions using the **let** command. It is important to note that the names bound to acquaintances, message contents and expression results are *not* variables: the **let** command only provides a temporary scope during which a name is bound to some value; after that scope has ended the old value bound to that name (if any) is exposed. As a consequence the only way to model state change is by using the **become** command.

The **become** command indicates which behaviour is to handle the next message in the queue. It may be executed before the handling of the current message has been completed, thus allowing the possibility of internal concurrency. If no replacement is specified, the default is to copy the current behaviour (i.e., to **become self**).

Let us consider Hewitt's standard example of a factorial actor [9] and complete Agha's pseudo-code [1] for a SAL implementation:

```
def recFact     accept fact:[n,client] =>
                    become self ;
                    if (n=0)
                    then        send result:[1] to client
                    else        let c = new factCust with [n,client]
                                    in { send fact:[n-1,c] to self }.

def factCust with [n,c] accept result:[k] => send result:[n*k] to c.
```

The behaviour recFact accepts requests of the form fact:[n,client] to compute the factorial of n and eventually causes the message: result:[*factorial of n*] to be sent back to the client. If the request is for the factorial of 0, the factorial actor responds immediately. Otherwise it dynamically creates a *customer* whose acquaintances are n and client, and it sends itself a request to compute the factorial of n-1 and send the result to the customer:

The customer will eventually receive this result, compute the product of n and the factorial of n-1 and send the value to the client. For a request to compute n factorial, then, recFact will end up creating n customers, thus simulating an execution stack [1].

Since recFact maintains no state information itself (it uses the customer to remember the original client) it immediately specifies its replacement as **self** to begin processing the next message. As a consequence, the factorial actor may service multiple requests concurrently.

Now all we need is a client definition and an **initially** declaration to create the factorial actor, its client and two requests to compute factorials:

```
def factClient accept result:[n] => skip.

initially   let f = new recFact
            and c = new factClient
                in { send fact:[5,c] to f ; send fact:[3,c] to f }.
```

7.3 Mapping Actors to Agents

In order to specify SAL computations in terms of Abacus patterns we must decompose actors into a number of agents that cooperate to give us the required behaviour. We shall give an overview of the approach before going into the details of the specification of each pattern.

We model every SAL program in terms of three kinds of agents: a *command* agent that performs the **initially** command, a *factory* agent that creates new actors, and a number of dynamically created *actor* agents. The initial command agent is responsible for creating the first actor agents and sending the messages that will start the computation. The factory agent is responsible for creating new actor agents and assigning a unique mail address to each. This address is reported to the actor requesting the creation.

Actor agents are encapsulated systems of agents consisting of a *message queue* agent and a *behaviour* agent. A behaviour agent consists of an *environment* agent that keeps track of the actor's acquaintances and other values to remember, a *handler* agent that accepts the next message and proceeds to respond to it, and a *cleanup* agent responsible for starting the replacement behaviour at the appropriate time. The main interactions are shown in Figure 2.

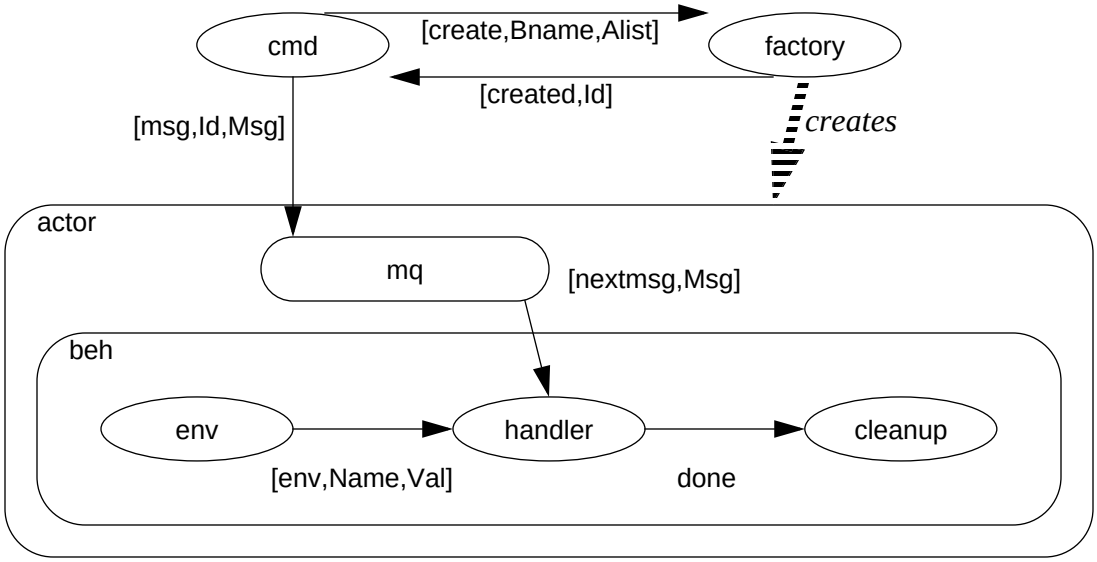

Figure 2 Mapping Actors to Agents

The cleanup agent will create a default replacement behaviour if none has already been specified by the time the handler reports that it has terminated by outputting done. If an early replacement has already been requested, it remembers this fact and does not create a second one.

The handler agent accepts a message from the queue and replaces itself by an environment agent that binds the message contents to local names, and a command agent that performs the appropriate actions. The command agent is responsible for reporting done to the cleanup agent.

Commands may send messages to other actors, specify replacement behaviours, or cause expressions to be evaluated by creating an *expression* agent. An expression agent optionally performs some computation and eventually outputs the value of the expression in the scope of the current environment. A new scope is created for a **let** command, which requires that a sequence of expressions be evaluated, a new environment binding names to the values of those expressions be created, and a command be executed with those names visible. The command following the **let** command executes within the old scope.

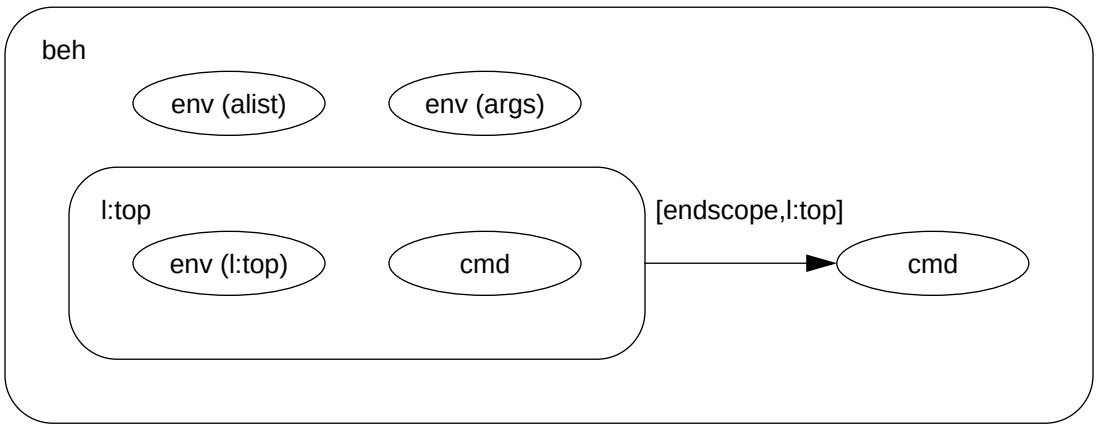

Figure 3 Lexical scoping in SAL

In Figure 3 we see the currently running command agent inside a scope called l:top. When the command terminates, it communicates [endscope,l:top] to the command agent waiting “out-

side." All names bound are accessible to the running command with the exception that names bound locally hide previous bindings of those names. When the scope terminates, the subsequent command sees only the bindings previously in effect. An environment for a given scope terminates when it receives the communication [quit,Scope].

The complete set of communications exchanged and their interpretation is as follows:

```
[create,Bname,Alist]        — create a new actor (behaviour Bname, acquaintances Alist)
[created,Id]                — the actor created has mail address Id
[msg,Id,Msg]                — message Msg is sent to address Id
[nextmsg,Msg]               — the next message for a given actor is accepted
mqnext                      — next message in the queue becomes the head
[env,Name,Val]              — the name Name is currently bound to Val
[endscope,Scope]            — Scope has just ended
[quit,Scope]                — the bindings of Scope are discarded
done                        — the current handler has terminated
[replacement,Bname,Alist]   — a replacement behaviour is created
[replacement,self]          — the replacement is the current behaviour
[val,E,V]                   — expression E has the value V
```

As a specification shortcut, we simulate the syntax of SAL by declaring SAL keywords and operators as prefix and infix operators to be recognized by Prolog[1] using the op/3 predicate (we do not consider this a particularly convenient way to define the syntax of a language, but it is adequate for small examples):

```
:- op(990,fx,[def,initially]) .
:- op(980,xfy,[accept,or]) .
:- op(970,xfy,[=>]) .
:- op(960,xfy,[;]) .
:- op(955,fy,[if]) .
:- op(950,xfy,[else]) .
:- op(945,xfy,[then]) .
:- op(940,fx,[send,become,let]) .
:- op(935,xfy,[in]) .
:- op(930,xfy,[to,and]) .
:- op(600,fy,[new]) .
:- op(600,xfx,[with]) .
```

7.4 An Abacus Specification of SAL

We shall now proceed with the specification of the patterns that interpret the semantics of SAL programs as Abacus agents. The patterns are specified in a denotational fashion, defining the semantics of SAL language constructs in terms of patterns that interpret their parts. The patterns correspond to the agents we introduced in the previous section.

To generate new actors and mail addresses, we make use of the following "actor factory":

```
factory(Num) := [create,Bname,Alist]?
                    (actor([id,Num],Bname,Alist)
                    & [created,[id,Num]]!factory(NextNum))        :- NextNum is Num + 1.
```

An agent requesting the creation of a new actor is expected to wait for the reply containing the mail address of the actor created. Mail addresses are of the form [id,Num], rather than simply Num to distinguish them from numerical values in expressions (we wish to prevent actors from performing computations with mail addresses and mailing to computed addresses). To prevent con-

1. The effect of this is that SAL declarations will be parsed as Prolog expressions. For example, the declaration of factCust has an abstract syntax tree with the linear representation:
def(accept(with(factCust,[n,c]),((result:[k]) => send(to((result:[n*k]),c)))))

fusion between multiple requests, the factory refuses new requests until the mail address is delivered.

An actor simply consists of a message queue and a behaviour. The actor is encapsulated using a \:a filter so that only requests to create new actors and messages between actors will be externally visible. The message queue is similar to the queue agent we defined earlier, except that it only accepts messages sent to the mail address Id.

```
actor(Id,Bname,Alist) := (mq(Id) & beh(Id,Bname,Alist))\:a.

mq(Id) := a:[msg,Id,X]?(head(X) ~ tail(Id))\:q.
head(X) := q:[nextmsg,X]!mqnext!nil.
tail(Id) := a:[msg,Id,X]?(x:mqnext?head(X) ~ tail(Id)) + x:mqnext?q:mq(Id).
```

The initial configuration consists of an agent that realizes the **initially** command and an actor factory. The agent sal is defined only if an **initially** command has been declared. As we shall see, the second and third arguments to the cmd pattern are the current scope and the command continuation.

```
sal := cmd(Cmd,top,nil)\:a & factory(0)                    :- initially Cmd.
```

For every **def** declaration we obtain a beh pattern that realizes the behaviour defined. The body of a behaviour consists of an *environment* that stores the bindings of names to values, a *handler* that accepts and handles the next message, and a *cleanup* agent responsible for creating the replacement behaviour. There are two possible cases, since the **with** clause is optional for actors with no acquaintances:

```
beh(Id,Bname,Alist) := body(Id,Bname,Anames,Alist,Handler)
                                                  :- def Bname with Anames accept Handler.
beh(Id,Bname,[ ]) := body(Id,Bname,[ ],[ ],Handler)    :- def Bname accept Handler.

body(Id,Bname,Anames,Alist,Handler) :=   ( env(alist,[self|Anames],[Id|Alist])
                                         & handler(Handler)
                                         & cleanup(Id,Bname,Alist) ) \: b.
```

Environments are identified by a current *scope*, and manage a set of name to value bindings. The outermost scope is called alist, and contains the mail addresses of the acquaintances and of self. The next is called args and is created when a message is accepted. The scope of the command to execute is called top, and all other scopes created by **let** commands are called l:top, l:l:top, and so on. The environment self-destructs when it receives the message [quit,Scope]. The lookup agent services requests to look up name bindings. It is defined recursively in terms of a list of names and a list of values. (In Prolog, [X|L] is a list with X as the first element and L as the rest of the list.) Note the use of the Self parameter to the lookup pattern that enables it to replace itself by the same environment after servicing each request:

```
env(Scope,Names,Vals) := [quit,Scope]?nil + lookup(Names,Vals,env(Scope,Names,Vals)).
lookup([N1|Names],[V1|Vals],Self) := [env,N1,V1]!Self + lookup(Names,Vals,Self).
```

The handler pattern simply accepts any of a series of messages and starts up an environment containing the message content bindings and a cmd agent that executes the handler command and reports done when the handler has terminated.

```
handler(Msg => Cmd) := handle(Msg => Cmd).
handler(Msg => Cmd or Others) := handle(Msg => Cmd) + handler(Others).
```

```
handle(Sel:Vars => Cmd) := hbody(Sel:Vars => Cmd).
handle(Sel => Cmd) := hbody(Sel:[ ] => Cmd)                    :- atom(Sel).

hbody(Sel:Vars => Cmd) := b:[nextmsg,Sel:Args]?( env(args,Vars,Args)
                                                 & cmd(Cmd,top,done!nil)).
```

The cleanup pattern handles requests to start the replacement behaviour, making sure that at most one such replacement is created. If the done message is received and no replacement has been created, a copy of the current behaviour is created. The args and alist environments are told to self-destruct (this is optional, since in any case they will not be accessible to the replacement behaviour):

```
cleanup(Id,Bname,Alist) := atend(b:beh(Id,Bname,Alist))
      + [replacement,self]?(atend(nil) & b:beh(Id,Bname,Alist))
      + [replacement,self,NewAlist]?(atend(nil) & b:beh(Id,Bname,NewAlist))
      + [replacement,Rname,NewAlist]?(atend(nil) & b:beh(Id,Rname,NewAlist)).

atend(End) := done?[quit,args]![quit,alist]!End.
```

The cmd pattern serves as a semantic function for SAL commands. The first argument is a SAL command, the second the name of the current scope, and the third the behaviour expression of the command *continuation*, i.e., the agent that realizes the rest of the computation. The first three rules are straightforward. The **skip** command does nothing; a semi-colon separates two commands to perform sequentially; and brace brackets simply serve as parentheses:

```
cmd(skip,Scope,Cont) := Cont.
cmd((C1;C2),Scope,Cont) := cmd(C1,Scope,cmd(C2,Scope,Cont)).
cmd({C},Scope,Cont) := cmd((C),Scope,Cont).
```

There are two versions of the **send** command since message contents are optional. The message expression is evaluated, the mail address of the target is retrieved from the environment, and the message is sent.

```
cmd(send Sel:Expr to Target, Scope, Cont) :=
                              expr(Expr) & [val,Expr,Val]?[env,Target,Id]?a:[msg,Id,Sel:Val]!Cont .

cmd(send Sel to Target, Scope, Cont) := [env,Target,Id]?a:[msg,Id,Sel:[ ]]!Cont    :- atom(Sel).
```

The **become** command simply sends a request to the cleanup agent:

```
cmd(become self, Scope, Cont) := [replacement,self]!Cont.
cmd(become Bname, Scope, Cont) := [replacement,Bname,[ ]]!Cont.
cmd(become Bname with Elist, Scope, Cont) :=
                              expr(Elist) & [val,Elist,Alist]?[replacement,Bname,Alist]!Cont.
```

The **if** command evaluates the logical expression and then decides to execute either the **then** part or the **else** part. If the **else** clause is missing, a **skip** command is inserted:

```
cmd(if Bool then C1 else C2, Scope, Cont) := expr(Bool) & [val,Bool,true]?cmd(C1,Scope,Cont)
                                                        + [val,Bool,false]?cmd(C2,Scope,Cont).
cmd(if Bool then C1,Scope,Cont) := cmd(if Bool then C1 else skip, Scope, Cont).
```

The **let** command is specified by means of the bind pattern, which generates agents to evaluate a list of expressions, and then creates a new environment in which the values of the expression list are bound to a list of names. The base case occurs when there is only one expression to evaluate. At this point a new environment called l:Scope is created together with the command

to be executed within this scope. Both are encapsulated using the hide pattern, which uses restriction to ensure that the names defined locally hide any prior bindings of those names in enclosing scopes. Bindings of names not locally defined are, of course, still accessible (lexical scoping applies). The continuation waits until the scope terminates. The continuation must be *outside* the new scope since it must be able to access the old bindings.

```
cmd(let Bindings in {Cmd}, Scope, Cont) := bind(let Bindings in {Cmd},Scope,[ ],[ ],Cont).
bind(let Name = Expr and Bindings in {Cmd},Scope,Names,Vals,Cont) := expr(Expr)
        & [val,Expr,Val]?bind(let Bindings in {Cmd},Scope,[Name|Names],[Val|Vals],Cont).

bind(let Name = Expr in {Cmd},Scope,Names,Vals,Cont) :=
                                        expr(Expr)
                                        & [val,Expr,Val]?
                                            ( hide(Names,
                                                env(l:Scope,[Name|Names],[Val|Vals])
                                                & cmd(Cmd,l:Scope,endscope(l:Scope)))
                                            & [endscope,l:Scope]?Cont).

endscope(l:Scope) := [quit,l:Scope]![endscope,l:Scope]!nil.

hide([N|Names],P) := hide(Names,P)\[env,N,_].
hide([ ],P) := P.
```

The remaining patterns deal with expressions. Expressions always terminate by reporting [val,E,V], where E is the expression to be evaluated and V is its value. Since there are never any local side-effects in the computation of an expression (names cannot be re-bound) subexpressions can be computed concurrently. The expression to be evaluated is repeated in the reply to disambiguate the results of concurrent subexpressions. (If the same numerical expression is computed in two subexpressions, both evaluations will yield the same result.)

The evaluation of numbers, names and lists of expressions is straightforward:

```
val(E,V) := [val,E,V]!nil.

expr([ ]) := val([ ],[ ]).
expr([E|Elist]) := expr(E) & expr(Elist) & [val,E,V]?[val,Elist,Vlist]?val([E|Elist],[V|Vlist]).

expr(N) := val(N,N)                                     :- number(N).
expr(X) := [env,X,Val]?val(X,Val)                       :- atom(X).
```

The **new** expression simply forwards the request to the actor factory and evaluates to the mail address of the newly created actor:

```
expr(new Bname) := create(new Bname,Bname,[ ])          :- atom(Bname).
expr(new Bname with Elist) :=
                expr(Elist) & [val,Elist,Alist]?create(new Bname with Elist,Bname,Alist).

create(E,Bname,Alist) := a:[create,Bname,Alist]!a:[created,Id]?val(E,Id).
```

To evaluate arithmetic and logical expressions we concurrently evaluate the subexpressions and then ask Prolog to compute the result. One simple way of doing this is as follows[1]:

1. This solution is somewhat verbose but easy to follow. In order to factor out the redundancy we may make use of Prolog's "univ" predicate to decompose E1 Op E2 and construct V1 Op V2.

```
expr(E1+E2) := expr(E1) & expr(E2) & [val,E1,V1]?[val,E2,V2]?arith(E1+E2,V1+V2).
expr(E1-E2) := expr(E1) & expr(E2) & [val,E1,V1]?[val,E2,V2]?arith(E1-E2,V1-V2).
expr(E1*E2) := expr(E1) & expr(E2) & [val,E1,V1]?[val,E2,V2]?arith(E1*E2,V1*V2).
expr(E1/E2) := expr(E1) & expr(E2) & [val,E1,V1]?[val,E2,V2]?arith(E1/E2,V1/V2).

arith(E,VE) := val(E,V)                                          :- V is VE.

expr(E1=E2) := expr(E1) & expr(E2) & [val,E1,V1]?[val,E2,V2]?bool(E1=E2,V1=V2).

bool(E,Bool) := val(E,true)                                      :- Bool.
bool(E,Bool) := val(E,false)                                     :- not(Bool).
```

7.5 Executing SAL Programs

This completes our specification of SAL. To execute a SAL program, we need only declare our behaviour definitions and our initial configuration and then execute the agent sal. Let us take as our example the recursive factor actor defined earlier:

```
def recFact      accept fact:[n,client] =>
                        become self ;
                        if (n=0)
                        then        send result:[1] to client
                        else        let c = new factCust with [n,client]
                                          in { send fact:[n-1,c] to self }.

def factCust with [n,c] accept result:[k] => send result:[n*k] to c.

def factClient accept result:[n] => skip.

initially    let f = new recFact
             and c = new factClient
                   in { send fact:[5,c] to f ; send fact:[3,c] to f }.
```

The complete computation is rather tedious to follow (there are over 400 events!) but it is instructive to see the computation state (of one possible execution trace) after the first few events. Here we have grouped together event sequences that start with a visible actor event (i.e., message sending or actor creation). Notice that the factorial actor services the two requests concurrently:

```
[create,recFact,[ ]]→ [created,[id,0]]→ [val,new recFact,[id,0]]→

[create,factClient,[ ]]→ [created,[id,1]]→ [val,new factClient,[id,1]]→ [env,f,[id,0]]→
[env,c,[id,1]]→ [val,5,5]→ [val,c,[id,1]]→ [val,[ ],[ ]]→ [val,[c],[[id,1]]]→ [val,[5,c],[5,[id,1]]]→

[msg,[id,0],fact:[5,[id,1]]]→ [env,f,[id,0]]→ [env,c,[id,1]]→ [val,3,3]→ [val,c,[id,1]]→ [val,[ ],[ ]]→
[val,[c],[[id,1]]]→ [val,[3,c],[3,[id,1]]]→

[msg,[id,0],fact:[3,[id,1]]]→ [quit,l:top]→ [endscope,l:top]→ [nextmsg,fact:[5,[id,1]]]→ mqnext→
[replacement,self]→ [nextmsg,fact:[3,[id,1]]]→ mqnext→ [env,n,5]→ [val,n,5]→ [val,0,0]→
[val,n=0,false]→ [env,n,5]→ [env,client,[id,1]]→ [val,n,5]→ [val,client,[id,1]]→ [val,[ ],[ ]]→
[val,[client],[[id,1]]]→ [val,[n,client],[5,[id,1]]]→

[create,factCust,[5,[id,1]]]→ [created,[id,2]]→ [val,new factCust with[n,client],[id,2]]→
[env,self,[id,0]]→ [env,n,5]→ [env,c,[id,2]]→ [val,n,5]→ [val,1,1]→ [val,n-1,4]→ [val,c,[id,2]]→
[val,[ ],[ ]]→ [val,[c],[[id,2]]]→ [val,[n-1,c],[4,[id,2]]]→
```

```
a:[msg,[id,0],fact:[4,[id,2]]]→ [quit,l:top]→ [endscope,l:top]→ done→ [quit,args]→ [quit,alist]→
[replacement,self]→ [nextmsg,fact:[4,[id,2]]]→ mqnext→ [env,n,3]→ [val,n,3]→ [val,0,0]→
[val,n=0,false]→ [env,n,3]→ [env,client,[id,1]]→ [val,n,3]→ [val,client,[id,1]]→ [val,[ ],[ ]]→
[val,[client],[[id,1]]]→ [val,[n,client],[3,[id,1]]]→


[create,factCust,[3,[id,1]]]→ [created,[id,3]]→ [val,new factCust with[n,client],[id,3]]→
[env,self,[id,0]]→ [env,n,3]→ [env,c,[id,3]]→ [val,n,3]→ [val,1,1]→ [val,n-1,2]→ [val,c,[id,3]]→
[val,[ ],[ ]]→ [val,[c],[[id,3]]]→ [val,[n-1,c],[2,[id,3]]]→


a:[msg,[id,0],fact:[2,[id,3]]]→ [quit,l:top]→ [endscope,l:top]→ done→ [quit,args]→ [quit,alist]→
[replacement,self]→
```

The state we reach at this point in the computation is:

```
((head(fact:[2,[id,3]])~tail([id,0]))\:q
      & (env(alist,[self],[[id,0]])
      & env(args,[n,client],[4,[id,2]])
      & expr(n)
      & expr(0)
      & [val,n,V1]?[val,0,V2]?bool(n=0,V1=V2)
      & [val,n=0,true]?cmd(send result:[1]to client,top,done!nil)
      +[val,n=0,false]?cmd(let c=new factCust with[n,client]
                                    in{send fact:[n-1,c]to self},
                              top,done!nil)
      & atend(nil)
      & b:(beh([id,0],recFact,[ ]))\:b)\:a
& actor([id,1],factClient,[ ])
& actor([id,2],factCust,[5,[id,1]])
& actor([id,3],factCust,[3,[id,1]])
& factory(4)
```

that is, we have one factorial actor with mail address 0 and a request in its mail queue to compute the factorial of 2 and send the result to mail address 3, one client with mail address 1, two customers and an actor factory. The factorial actor has just created its replacement so it can concurrently begin processing the next message while it evaluates the **if** command.

8. Concluding Remarks

We have presented the syntax, semantics and usage of Abacus by means of a series of progressively more advanced examples of concurrency specifications, concluding with a specification of a small actor-based concurrent programming language. We have shown how two new operators, label prefixing and filtering, can be useful for encapsulating concurrent systems, and we have introduced *patterns* as a means of specifying higher-level constructs that evaluate to agents.

Our goal is to provide a platform for prototyping executable language specifications for concurrent object-based languages. Although we have not discussed class inheritance, it turns out not to be very difficult to model with patterns. One can either follow the approach of Cook [5] and construct a pattern for a class by means of *generators*, or one can directly simulate method lookup by forwarding messages to superclass agents à la *delegation* [19]. We explore the possibilities of modelling objects as communicating agents in [17].

One limitation of our current approach is that it only deals with the translation of *valid* programs; it does not provide any means for expressing what programs may be syntactically sound but semantically defective. For example, there is nothing to prevent SAL actors from attempting to use unbound names in expressions, or numerical values as mail addresses. In such cases, actors will simply deadlock as they wait for an event that can never occur. Syntactically correct but semantically invalid SAL programs will be translated to agents that just stop functioning when the error is encountered. We are presently working on a type theory for active objects that allows one to specify basic safety and liveness constraints for well-behaved agents in terms of the expected possible interactions between an agent and its clients[17][18].

There are several interesting directions in which Abacus could evolve. One is to develop a general-purpose pattern mechanism whose semantics can be defined directly by translation to Abacus. Although we use patterns in a disciplined way so that the mapping from a pattern to the agent that realizes it is always well-defined, it is not clear in general what algebraic properties patterns may exhibit. In particular, patterns make it possible to specify systems with *dynamically varying linkage* [15], for example, actors may become dynamically acquainted with new actors. In such cases it may be difficult to reason about the behaviour of systems from the properties of their parts.

At present we are experimenting with Prolog to determine what are the minimal requirements to be able to conveniently express solutions to real problems using patterns. A related aspect is the convenient support of *syntactic* patterns. Prolog supports only infix, prefix and postfix operators, and there are some subtle restrictions on how expressions are parsed. We feel a better solution would be to provide either a fixed set of generally useful syntactic patterns that may be overloaded, or a grammar-based tool that allows one to specify arbitrary syntactic patterns.

Another direction is to better support the execution of specifications by providing finer monitoring control, or even by generating simple compilers so that larger examples can be tested. The current implementation though reasonably fast is not blindingly so – the factorial example takes over a minute when using a Prolog compiler – mainly because event searching is exhaustive for every step of the computation. It is an open issue whether acceptable compilers could actually be generated automatically from Abacus specifications.

Appendix: A Minimal Prolog Implementation

What follows is a minimal, but complete implementation of Abacus in Prolog. The full implementation provides a form of “garbage collection” by re-writing behaviour expressions to simpler equivalent forms and by removing instances of dead nil agents. Simple pretty-printing of behaviour expressions is provided to help isolate the individual agents of a behaviour expression. The full implementation also supports options to print intermediate states of a computation and to force the interpreter to search for all possible computation paths.

An earlier implementation of Abacus [16] did not take advantage of Prolog’s ability to define new operators, and thus exhibited little of the flexibility and compactness of the present approach. In the implementation given below, there is a one-to-one mapping between rules of the transition semantics given earlier and the Prolog rules that implement them: for every visible

transition there is an offer rule, and for every invisible transition there is a tau rule. The rules for each operator are largely independent, making it very easy to add new operators, such as prefixing, filtering and linking. To our knowledge, the only other attempt to develop a similar interpreter for a CCS-based specification language is an interpreter for LOTOS [12][14]. The focus there, however, is on the specification of distributed systems rather than on the specification of concurrent programming languages.

```
:- op(690,xfx,:=).              % Naming
:- op(670,xfy,&).               % Composition
:- op(660,xfy,~).               % Linking
:- op(500,yfx,+).               % Summation
:- op(460,xfy,[!,?]).           % Output/Input
:- op(440,xfy,:).               % Prefixing
:- op(400,yfx,\:).              % Filtering
:- op(400,yfx,\).               % Restriction
:- op(400,yfx,/).               % Relabelling

offer(E?R,(E,?),R).
offer(E!R,(E,!),R).
offer(B+_,O,R)                  :- offer(B,O,R).
offer(_+B,O,R)                  :- offer(B,O,R).
offer(B&X,O,R&X)                :- offer(B,O,R).
offer(X&B,O,X&R)                :- offer(B,O,R).
offer(B\E,O,R\E)                :- offer(B,O,R), not(match(O,E,_)).
offer(B/F,FO,R/F)               :- offer(B,O,R), relabel(F,O,FO).
offer(B,O,R)                    :- B := BE, offer(BE,O,R).
offer(F:B,(F:E,G),F:R)          :- offer(B,(E,G),R).
offer(B\:F,O,R\:F)              :- offer(B,FO,R), exports(FO,F,O).
offer(B1~B2,O,R1~R2)            :- offer(B1&B2\:x,O,R1&R2\:x).

tau(B+_,O,R)                    :- tau(B,O,R).
tau(_+B,O,R)                    :- tau(B,O,R).
tau(B1&B, E, B1&R)              :- tau(B,E,R).
tau(B&B2, E, R&B2)              :- tau(B,E,R).
tau(B1&B2, E, R1&R2)            :- offer(B1,O1,R1), offer(B2,O2,R2), match(O1,E,O2).
tau(B\H,E,R\H)                  :- tau(B,E,R).
tau(B/F,E,R/F)                  :- tau(B,E,R).
tau(B,E,R)                      :- B := BE, tau(BE,E,R).
tau(F:B,E,F:R)                  :- tau(B,E,R).
tau(B\:F,E,R\:F)                :- tau(B,E,R).
tau(B1~B2,E,R1~R2)              :- tau(B1&B2\:x,E,R1&R2\:x).

match((E,?),E,(E,!)).
match((E,!),E,(E,?)).
relabel([FE/E|_],(E,G),(FE,G)).
relabel([_|F],O,FO)             :- relabel(F,O,FO).
relabel([ ],O,O).
exports((F:E,G),F,(E,G)).
exports((FF:E,G),F,(FF:E,G)).

abc(B)                          :- write('Initial configuration: '), write(B), nl, path(B,P,F),
                                       write('Final configuration: '), write(F), nl.

path(B,(E->P),F)                :- tau(B,E,R), write(E), write(' -> '), nl, path(R,P,F).
path(F,F,F).
```

References

[1] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[2] G.R. Andrews, "Synchronizing Resources," ACM TOPLAS, vol. 3, no. 4, pp. 405-430, Oct 1981.

[3] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," ACM Computing Surveys, vol. 15, no. 1, pp. 3-43, March 1983.

[4] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," ACM Computing Surveys, vol. 21, no. 3, pp. 323-357, Sept 1989.

[5] Wm. Cook, "A Denotational Semantics of Inheritance," ACM SIGPLAN Notices, Proceedings OOPSLA '89, vol. 24, no. 10, pp. 433-443, Oct 1989.

[6] E.W. Dijkstra, "Co-operating Sequential Processes," in *Programming Languages*, ed. F. Genuys, pp. 43-112, Academic Press, New York, 1968.

[7] M.J.C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[8] A.N. Habermann, "Synchronization of Communicating Processes," Communications of the ACM, vol. 15, no. 3, pp. 171-176, March 1972.

[9] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no. 3, pp. 323-364, June 1977.

[10] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Communications of the ACM, vol. 17, no. 10, pp. 549-557, Oct 1974.

[11] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[12] ISO8807, "Information Processing Systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour," International Standard ISO 8807, 1989.

[13] B. Liskov, M. Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," 13th Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, Jan 13-15, 1986.

[14] L. Logrippo, A. Obaid, J.P. Briand and M.C. Fehri, "An Interpreter for LOTOS, A Specification Language for Distributed Systems," Software – Practice and Experience, vol. 18, no. 4, pp. 365-385, April 1988.

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

[16] O.M. Nierstrasz, "Abacus: a Notation for Describing Concurrent Computations," in *Object Oriented Development*, ed. D.C. Tsichritzis, pp. 247-275, Centre Universitaire d'Informatique, University of Geneva, July 1989.

[17] O.M. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents," Proceedings OOPSLA '90, 1990, (to appear).

[18] O.M. Nierstrasz and M. Papathomas, "Towards a Type Theory for Active Objects," in *Object Management*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, July 1990, (Working Paper).

[19] L.A. Stein, "Delegation is Inheritance," ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 138-146, Dec 1987.