

# A LOTOS Interpreter for Simulating Real-Time Behaviour\*

*C.J. Fidge*

Key Centre for Software Technology  
The University of Queensland  
Queensland, 4072  
AUSTRALIA

26th September 1990

## Abstract

A prototype Basic LOTOS interpreter, augmented for modelling real-time systems, is described. Unlike previous attempts to add time to the process algebras, which have treated primitive actions as instantaneous, the interpreter is designed to model actions as time-consuming, with separate start and end points. A distinction is made between “true” and interleaved concurrency, and between timeouts and time-consuming internal actions. This experimental system encountered a number of unforeseen difficulties which suggest directions for future research.

## 1 Introduction

Attempts to add the dimension of time to the process algebras have been constrained by the need to conform with the interleaving semantics used to formally define these languages. Consequently primitive actions have been treated as instantaneous, denoting either the start or end point of some behaviour of interest (e.g., Quemada et al., 1989; Gerber et al., 1988; Žic, 1990).

Although this approach is a valid abstraction for the purposes of specification we feel that it is inconsistent with the needs of simulating the timing behaviour of existing, or proposed, real-time systems; our goal is *modelling* rather than requirements specification. This paper

---

\*A much shorter version of this paper was published in J. Quemada, J. Mañus, and E. Vazquez, editors, *Formal Description Techniques III (FORTE'90)*, North-Holland, 1991, pages 523–526.

describes an experimental LOTOS interpreter which incorporates a notion of *time-consuming* actions inspired by the Real Time Logic formalism (Jahanian and Mok, 1986).

Development of the prototype interpreter quickly demonstrated the need for more information than is found in an abstract specification. To accommodate this information distinct notations are introduced to separate interleaved from “true” concurrency, and time-consuming internal actions from an idle delay.

Although by no means a polished product, the experimental interpreter brought to light a number of fundamental problems that hinder the introduction of such a time model into the process algebraic languages.

## 2 Operators

The interpreter implements a superset of the standard Basic LOTOS operators (Bolognesi and Brinksma, 1987). The meaning of the operators with respect to time requires some explanation, however. Examples are given in section 4.

Time is global and discrete; all output from the interpreter is given in terms of the absolute time from the instant at which the simulation of the LOTOS specification began.

All *actions*, including the special internal action *i*, are assumed to take some finite time to be performed. The possibility that an action can take zero time is allowed so that instantaneous events can still be modelled, however. Each occurrence of an action *a* is bounded by two instantaneous *events*, *+a* and *-a*, denoting the instant at which *a* started and stopped, respectively.

The primitive behaviour expressions remain unaffected by the time semantics (although see section 7):

<code>stop</code>	inaction
<code>exit</code>	successful termination

All uses of the action prefix operator must be accompanied by the duration of the action:

<code><i>g t</i>; <i>B</i></code>	perform action <i>g</i> with duration <i>t</i> , then behave like <i>B</i>
<code><i>i t</i>; <i>B</i></code>	perform an internal action with duration <i>t</i> , then behave like <i>B</i>

This means that action *g* (or *i*) will take *t* time units to be performed, i.e., *+g* and *-g* will be separated by exactly *t* units of global time. Primitive actions are thus atomic, i.e., non-interruptible.

As in Quemada et al. (1989) *t* may express uncertainty. Apart from a single integer value *t* may be a set of time values, or an integer range. The time taken to perform the action will be one of these values, but it cannot be predicted which, and it may be different each time the same action is instantiated.

Hiding a timed action has the obvious meaning:

$\text{hide } g_1 \dots g_n \text{ in } B$       action hiding

Behaviour expression  $B$  will perform an internal action whenever it would normally perform one of the named actions  $g_i$ , using the duration specified for  $g_i$  in  $B$ .

The enabling operator has the usual meaning, i.e., the initial event(s) of the second expression can occur only after the final event(s) of the second (assuming that the first expression terminates successfully):

$B_1 \gg B_2$       sequential composition (execute  $B_2$  after  $B_1$ )

Choice has a special meaning with respect to time:

$B_1 \sqcap B_2$       choose the first available behaviour expression

Choice selects whichever behaviour expression first becomes ready to perform an event. If both arguments can simultaneously perform an initial event the choice is nondeterministic.

The disabling operator,

$B_1 \sqsupset B_2$       disabling (let  $B_2$  interrupt  $B_1$ )

makes the initial event(s) of  $B_2$  available whenever an action of  $B_1$  may be performed. However, due to the indivisibility of atomic actions,  $B_1$  cannot be interrupted while performing an action.

The standard parallel composition notation is defined to represent “true” concurrency:

$B_1 \mid [g_1 \dots g_n] \mid B_2$       general parallelism  
 $B_1 \parallel B_2$       pure parallelism  
 $B_1 \parallel\parallel B_2$       fully synchronised parallelism

In practical terms we think of these operators as defining a set of processes each of which executes on a different processor. This means that two or more primitive actions may be performed at exactly the same time. All participants in a shared action in  $\mid \square \mid$  or  $\parallel$  must begin and end at exactly the same instant. The start point of the shared action will be delayed until all participating processes are ready (but no later); a “processor” may thus be forced to be idle. Where the participants disagree on the duration of the shared action its duration is the maximum of the possibilities so that worst-case behaviour is modelled.

To model two or more processes executing on the same processor we introduce a new notation for “interleaved” concurrency:

$B_1 / [g_1 \dots g_n] / B_2$       general interleaving  
 $B_1 \text{ /// } B_2$       pure interleaving  
 $B_1 \text{ // } B_2$       fully synchronised interleaving

In all cases one action only may be performed at any time. Again shared actions must begin and end simultaneously, and their duration is the maximum of the given values. Interestingly  $||$  is equivalent to  $//$ , except where internal events are involved.

In modelling a “real life” system it is normally meaningful to only nest interleaving operators within parallelism operators, and not vice versa. (Of course, from the viewpoint of a requirements specification, we should not be concerned with the distinction between interleaving and parallelism. The goal herein is to purposely introduce necessary low-level detail from which timing behaviour can be modelled.)

Finally, the possibility that a process is “idle” for a period of time can be explicitly modelled with the `delay` action:

`delay t; B`                      perform  $B$  after a delay of duration  $t$

This expression does not produce any observable behaviour except to delay the beginning of behaviour expression  $B$ . It is distinct from performing an internal action for time  $t$ , however, due to its effect when used as an initial behaviour in a  $[\ ]$  alternative (see section 4). Again time  $t$  may be an “uncertain” value.

### 3 Implementation

This section briefly outlines the implementation of the “timed” LOTOS interpreter. It is written in Kyoto Common LISP and runs on a SUN-3 workstation.

Each LOTOS operator is represented by a LISP macro which, when presented with an event name, either rejects the event or accepts it and returns the subsequent behaviour expression and the global time at which the event occurred. LOTOS process definitions define new LISP macros; in effect each new process defined by the user is a new LOTOS operator and interpreted using the same code as the built-in operators. All LOTOS specifications are entered as LISP expressions (see appendix A), although the conversion to standard LOTOS concrete syntax is straightforward.

The interpreter allows LOTOS specifications to be “executed”. It is not a “trace generator” like that of Patel et al. (1989), although it is potentially capable of exercising all possible traces defined by a given specification. Interaction is normally through a single-stepping function that prompts the user with all events that may be performed next and allows one to be selected. For long-lived computations, where this would become tedious, an “automatic” function may be used to run the LOTOS specification to termination. Internal actions are normally not displayed, but an interpreter option makes them observable as a specification debugging aid.

When presenting the user with possible next events, the interpreter allows selection among only those events which may occur “earliest”. This is necessary to prevent the user from selecting “future” actions before more recent ones executing in parallel. Effectively this means

that the user behaves like a process running in parallel with the specification being performed, sharing all observable events, and immediately capable of performing any event. The user can thus exercise influence over the computation only when two or more events are ready to occur at exactly the same instant.

The passage of time is modelled by wrapping the subsequent behaviour expression in a call to the `delay` macro whenever a time-consuming action is performed. This is invisible to the user, but it allows the specification expression to “remember” how much time has passed as it evolves. Uncertain timing, like nondeterminism, is simulated using the built-in LISP random number generator to arbitrarily select a time from the set of possibilities.

After performing a start event for its named action the prefix macro transforms into an internal representation which is then only prepared to perform the corresponding end event. Similarly the interleaving operators ensure that after performing a start event the only possible next event for the behaviour expression is the corresponding end. Other “inactive” processes have an appropriate delay inserted. An additional complication is the need to delay participating processes until all are ready, when a shared action is to be performed.

When processes perform events in isolation the general parallelism operator must cause time to pass in all other parallel processes. This was done using a function based on the *Old* operator of Quemada et al. (1989). In addition it was necessary to allow for the possibility that a process is “idle” while waiting to perform a shared action. This caused considerable inefficiency since it was first necessary to determine how long the idle period must be, and then to re-execute the delayed process with an appropriate `delay` in place. This complication is a manifestation of the need to achieve a “distributed consensus” on the time at which a shared event takes place.

Both parallelism and interleaving need to update the time in other processes even when one process is capable of performing an action in isolation. This unfortunate complication was first observed by Quemada et al. (1989) and one of the original goals in introducing time-consuming actions was to avoid it, but this did not prove to be possible.

Like its predecessor (Fidge, 1989), the interpreter is plagued by inherent efficiency problems and is suitable for executing only relatively small examples. Nonetheless it is an adequate platform for experimentation with the semantics of our timed LOTOS calculus. Using a functional language has given us the ability to quickly develop a prototype interpreter and uncover a number of unexpected problems (section 6).

## 4 Examples

This section illustrates the behaviour of the timed operators during simulation. Standard LOTOS syntax is used rather than the less familiar interpreter input (appendix A).

Despite the introduction of true concurrency, the interpreter is still semantically founded in

the concept of interleaved traces. Based on the work of Quemada et al. (1989) traces are simply augmented with time values which may include a zero separation to indicate simultaneous events. The interpreter generates traces as triples consisting of the global time at which an event occurred, the event name, and the elapsed time since the previous event. For instance, let

$$-a_2^{10}$$

represent the event marking the end of action **a** occurring at time 10, 2 time units after the last observable event. Unless otherwise stated it is assumed throughout this section that each example is executing in an environment that is immediately prepared to participate in any of the observable events offered, and that each example starts at time 0.

Action prefix generates pairs of start/end events. For instance the possible traces of specification

```
(a 4; b 2; i {5,7}; c {2..5}; stop)
```

include

$$+a_0^0, -a_4^4, +b_0^4, -b_2^6, +c_5^{11}, -c_3^{14}$$

Other traces are possible due to the “uncertain” durations in braces. Action **a** always takes 4 time units but **c** may take any time between 2 and 5 inclusive. The end point of **a** and the start point of **b** are identical in time.

The internal action, which may take either 5 or 7 time units, is observable only in its effect on the passage of time. (Alternatively an interpreter option allows internal actions to be observed, i.e.,

$$+a_0^0, -a_4^4, +b_0^4, -b_2^6, +i_0^6, -i_5^{11}, +c_0^{11}, -c_3^{14}$$

This option is “off” by default.)

Hidden actions and explicit delays similarly cause time to suddenly advance. For example, behaviour expression

```
hide b in (a 2; b 3; c 1; delay {2..4}; d 2; stop)
```

may result in trace

$$+a_0^0, -a_2^2, +c_3^5, -c_1^6, +d_2^8, -d_2^{10}$$

The disabling operator allows one of the interrupting start events to occur whenever the first behaviour expression is ready to start performing an action. However an action in progress cannot be interrupted until it is complete. For instance the traces of specification

(a 2; b 1; exit) [ $\triangleright$ ] (c 4; stop)

include

$+a_0^0, -a_2^2, +c_0^2, -c_4^6$

but not

$+a_0^0, +c_1^1, -c_4^5$

since the latter trace treats action **a** as divisible.

Sequencing two observable behaviours has the obvious effect, i.e., the start point of the second behaviour expression is delayed by the time taken to reach the end point of the first. For instance,

(a 4; exit)  $\gg$  (b 2; stop)

generates trace

$+a_0^0, -a_4^4, +b_0^4, -b_2^6$

Less familiar is the behaviour associated with choice. Where the alternatives both offer initial events at the same instant it has the usual semantics. For example

a 1; (b 2; stop [ $\square$ ] c 3; stop)

offers **+b** and **+c** at the same instant (time 1). When placed in an environment prepared to immediately participate in both **b** and **c** this expression can generate trace

$+a_0^0, -a_1^1, +b_0^1, -b_2^3$

or

$+a_0^0, -a_1^1, +c_0^1, -c_3^4$

Similarly when one of the alternatives begins with an internal action, e.g.,

a 1; (b 2; stop [ $\square$ ] i 1; c 3; stop)

the random decision to select the internal action is made when the choice first begins execution (at time 1 in this case). This example will offer the environment **+b** at time 1 or **+c** at time 2, but the environment cannot control which, as usual for a choice between an observable and internal action. Possible traces (in an environment immediately prepared to accept either initial event) are thus

$+a_0^0, -a_1^1, +b_0^1, -b_2^3$

and

$$+a_0^0, -a_1^1, +c_1^2, -c_3^5$$

However, where an alternative cannot be selected immediately because it is shared with a process which is not yet ready to participate in the action, whichever alternative is ready first is always selected. For instance, if the environment of the above example is only prepared to start executing **b** at time 3 then the second alternative will always be selected because the internal action is *always* prepared to be performed immediately (i.e., at time 1).

Although necessary for modelling the possibility that a process arbitrarily selects among time-consuming internal and observable actions these semantics are incompatible with the need to model timeouts since a timeout defers the choice until a given period has expired. This situation can be modelled by using the special `delay` action. The following example

```
a 1; (b 2; stop [] delay 1; c 3; stop)
```

will only generate the first of the two traces above in an environment immediately ready to execute **b** and **c**. However, if the environment will not offer **+b** until time 3 or later, then only the second choice is possible. If the environment offers **+b** at time 2 then either alternative is possible.

The distinction between parallelism and interleaving is easily illustrated by comparing their traces when applied to the same arguments. For instance, the following interleaved example,

```
(a 2; b 3; c 1; exit) / [b] / (d 4; e 2; b 2; exit)
```

has traces including

$$+d_0^0, -d_4^4, +a_0^4, -a_2^6, +e_0^6, -e_2^8, +b_0^8, -b_3^{11}, +c_0^{11}, -c_1^{12}$$

Only one action is ever active at any instant. The shared action **b** is performed for the maximum of its two possible durations.

By contrast the same expressions executed in parallel, i.e.,

```
(a 2; b 3; c 1; exit) | [b] | (d 4; e 2; b 2; exit)
```

will generate the trace

$$+a_0^0, +d_0^0, -a_2^2, -d_4^4, +e_0^4, -e_2^6, +b_0^6, -b_3^9, +c_0^9, -c_1^{10}$$

Some actions overlap in time, e.g., **a** and **d**. The first process is idle from time 2 to 6 due to the need to share action **b**. Not surprisingly the simulation showed that the parallel example is faster than the interleaved one.

Finally, as a more substantial example, we present the stop-and-wait protocol from Quemada et al. (1989):

```

process LINK :=
  ((TRANSMITTER | [] | RECEIVER)
   | [SendInfo,SendAck,RecInfo,RecAck] | LINE)
endproc

process TRANSMITTER :=
  delay {0..infinity}; get 0; SENDING
where
  process SENDING :=
    SendInfo 8; (RecAck 0; TRANSMITTER
                [] delay 30; SENDING)
  endproc
endproc

process RECEIVER :=
  RecInfo 0; give 0; SendAck 2; RECEIVER
endproc

process LINE :=
  (SendInfo 0; (RecInfo 10; LINE [] i 10; LINE)
   []
   SendAck 0; (RecAck 10; LINE [] i 10; LINE))
endproc

```

Superficially this specification looks similar to the original one but it is important to realise that the time values above are durations whereas those used by Quemada et al. (1989) are end points given relative to some “initial instant” (apparently the time at which the encompassing process was instantiated).

The constant `infinity` is used to represent an indefinite wait for a new frame to be made available. In the interpreter implementation this value is the largest available positive integer.

The original example used internal actions in two distinct ways. In process `LINE` they represented a “transmission error”, i.e., the loss of a frame, an action that takes 10 time units. In `SENDING` the internal action represented a “time-out” of length 30 while waiting to receive an acknowledgement. Quemada et al. (1989) note these different uses of `i` in comments preceding the example. Interestingly, our calculus requires us to explicitly differentiate between these two unobservable behaviours. The `delay` action must be used in `SENDING` because `i` would allow it to “actively” select the timeout alternative, i.e., start executing the internal action, as soon as `SendInfo` finishes whereas the choice should be “passive”, i.e., dependent upon the availability of shared actions. In `LINE`, however, an “active” decision is immediately made to

either transmit the information or corrupt it, and hence `i` is used.

Notice that the timeout is relative to the start, rather than the end, of the `RecAck` action. This approach can be readily criticised from the point of view of real-time predictability since it cannot be known in advance whether the recovery action or `RecAck` is performed immediately after the timeout period has expired. Nevertheless this represents the, perhaps unfortunate, realities of existing real-time programming practice. Only the “timed statements” planned for a future upgrade of Ada allow timeouts relative to the end of an action (Baker et al., 1990).

Another notable difference with Quemada et al. (1989) is that they use a range such as `{0..no_limit}` in those cases where it is not known how long a shared action will take, e.g., for `SendInfo` in `LINE` where the actual duration is controlled by `SENDING`. In our calculus this concept is represented by giving these actions a duration of 0 since their actual duration will then always be defined by the other participating process(es). For instance, `SendAck` always takes 2 time units due to its definition in `RECEIVER` even though `LINE` does not know its duration.

The example is infinitely recursive so we cannot enumerate its possible traces. However the following trace achieved with the interpreter exercises all of the control paths:

```
+get100100, -get0100, +SendInfo0100, -SendInfo8108, +SendInfo30138, -SendInfo8146, +RecInfo0146,
-RecInfo10156, +give0156, -give0156, +SendAck0156, -SendAck2158, +SendInfo18176, -SendInfo8184,
+RecInfo0184, -RecInfo10194, +give0194, -give0194, +SendAck0194, -SendAck2196, +RecAck0196,
-RecAck10206, +get11,28811,494, -get011,494, ...
```

The chaotic values for the beginning of `get` are due to the vaguely defined `delay` preceding it. The reader is invited to compare this trace with the action tree given by Quemada et al. (1989) to confirm the equivalence of the two protocol specifications.

In both cases the timing requirements restrict the possible behaviours of the specification. To illustrate this the following trace was generated when the timing information was removed from the example and it was executed using the untimed LOTOS interpreter:

```
get, SendInfo, give, SendInfo, SendAck, ...
```

This trace cannot occur in the timed version because the second occurrence of `SendInfo` is delayed until after the first frame has been transmitted through the link.

## 5 Relationship to Standard LOTOS

It has already been observed that the possible behaviours of a timed specification are more restrictive than an equivalent untimed specification. It can be further noted that the possible traces generated by the timed LOTOS interpreter can all be translated into valid standard LOTOS traces simply by removing all end point (or start point) events and removing the “+” (or “-”) annotation from the remaining events. For instance the trace given for the “parallel” example in section 4 can be systematically converted into the following trace,

a, d, e, b, c

a perfectly valid trace of the equivalent untimed behaviour expression,

(a; b; c; exit) | [b] | (d; e; b; exit)

“Timed” traces are always extensions to valid untimed traces.

## 6 Lessons Learned

Problems encountered during the development of the experimental interpreter point to a number of fundamental difficulties with using the process algebras to model timing behaviour in this way.

- Where several processes participate in a shared event they must all agree on the time at which it occurs. In essence this is a variant of the classical problem of achieving a “multi-way rendezvous”, a notoriously difficult problem to implement efficiently. Although described as having “operational semantics” (Bolognesi et al., 1990), the time model used by Quemada et al. (1989) also requires distributed consensus.
- Since a start event is always followed by a corresponding end event it would be convenient to have a fundamental mechanism for treating two sequential events as an atomic unit. Recently Gorrieri et al. (1990) addressed a similar need for “atomic actions”, i.e., indivisible sequences of two or more lower-level actions, in CCS.
- The decision to make choice operators select the first available event was not one of our original goals. Although some authorities (e.g., Liu and Shyamasundar, 1990) advocate this approach we feel that it inextricably links the logical and temporal aspects of the specification; it would be better to keep them orthogonal. Unfortunately the “earliest event” rule is necessary to ensure that time only goes forward. Consider the following two examples,

(a 1; exit) [] (delay 2; b 1; exit)

and

(a 1; exit) ||| (delay 2; b 1; exit)

With conventional untimed semantics both offer their environment the possibility of executing actions **a** or **b**. However, when the time values are considered, the second example must initially only offer event **+a** to preserve correct interleaving with regard to the passage of time (**+a** can occur at time 0, but **+b** cannot occur until time 2). Unfortunately,

when evaluating a behaviour expression, the interpreter does not know whether the set of events being offered were generated by a choice or parallelism operator. Therefore, in guaranteeing the correct behaviour of the parallelism operators, it was necessary to make an “across the board” decision to only offer the earliest events, thus inadvertently affecting the behaviour of choice as well. Ideally a rule like that of Quemada et al. (1989) is needed, stating that an event is offered to the environment only if other parallel processes have stopped or can offer an event at a later time. Unfortunately a practical implementation of this rule has proven elusive due to its need for global knowledge.

For further experimentation the interpreter currently has an option that allows the user to select events other than the earliest ones available but, when applied to a behaviour expression containing parallelism operators, this carries with it the danger that time may appear to go backwards temporarily due to parallel events being selected “out of order”.

## 7 Future Work

The third problem identified in section 6 continues to be a cause for concern. A solution would appear to require making the difference between alternatives offered by choice and parallelism operators explicit to their environment, a substantial alteration to the operational interleaving semantics traditionally used by process algebras (c.f., Degano et al., 1988).

A feature of the interpreter not discussed so far is the ability to associate a (possibly “uncertain”) delay with each operator. For instance, a delay of 2 time units associated with the `>>` operator causes

```
(a 3; exit) >> (b 4; stop)
```

to have trace

$$+a_0^0, -a_3^3, +b_2^5, -b_4^9$$

Although already implemented this feature is presently not in use until its applicability is determined (all such delays are currently set to 0). The original intention of the feature was to model expensive activities such as the creation of parallel processes. However it is not clear if such a delay is meaningful for other operators, e.g., `stop`, `[]` and `[>`.

Of course simulating the behaviour of a real-time design is only the first part of evaluating its timing correctness. Next the output must be analysed to ensure that the observed timing behaviour conforms with our expectations. An advantage of programming the interpreter in a functional language is that this activity can be easily automated. For instance the list generated by the `execute` function (appendix A) could be used as input to a post-mortem analysis function to check that safety assertions are not violated.

## 8 Conclusion

At the time of writing the interpreter has just become operational (all of the examples given herein were tested using it) but continues to be enhanced.

Nevertheless its development to date has already provided us with considerable insight into the difficulties of associating a “duration” with the actions of a process algebra. The inability to avoid an equivalent to the *Old* operator of Quemada et al. (1989) in parallel composition was a disappointing outcome. The need for the *delay* action was an unforeseen requirement. Another unexpected result was the difficulty of implementing disabling—the inability to “interrupt” actions in progress seems counter-intuitive and has been strongly criticised. Hopefully future development will ease some of these qualms about the precise behaviour of LOTOS with time-consuming actions.

## Acknowledgements

I wish to thank the anonymous FORTE'90 referees for their insightful criticisms of a draft of this paper. This work was supported by an Australian Postdoctoral Research Fellowship and an Australian Telecommunications and Electronics Research Board Project Grant.

## References

- BAKER, T., BALDO, J., LOCKE, C. D., and WEIDERMAN, N. (1990): Time Issues Working Group, *Ada Letters*, X(4), pp. 119–143.
- BOLOGNESI, T. and BRINKSMA, E. (1987): Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems*, 14(1), pp. 25–59.
- BOLOGNESI, T., LUCIDI, F., and TRIGILA, S. (1990): From Timed Petri Nets to Timed LOTOS, *Proc. 10th International Symposium on Protocol Specification, Testing and Verification*, Ottawa, pp. 377–406.
- DEGANO, P., DE NICOLA, R., and MONTANARI, U. (1988): A Distributed Operational Semantics for CCS Based on Condition/Event Systems, *Acta Informatica*, 26, pp. 59–91.
- FIDGE, C. J. (1989): *A Basic LOTOS Interpreter*, Technical Report 140, Key Centre for Software Technology, University of Queensland.
- GERBER, R., LEE, I., and ZWARICO, A. (1988): *A Complete Axiomatization of Real-Time Processes*, Technical Report MS-CIS-88-88, University of Pennsylvania.
- GORRIERI, R., MARCHETTI, S., and MONTANARI, U. (1990): A2CCS: Atomic Actions for CCS, *Theoretical Computer Science*, (72), pp. 203–223.

- JAHANIAN, F. and MOK, A. (1986): Safety Analysis of Timing Properties in Real-Time Systems, *IEEE Transactions on Software Engineering*, SE-12(9), pp. 890–904.
- LIU, L. Y. and SHYAMASUNDAR, R. K. (1990): Static Analysis of Real-Time Distributed Systems, *IEEE Transactions on Software Engineering*, 16(4), pp. 373–388.
- PATEL, S., ORR, R. A., NORRIS, M. T., and BUSTARD, D. W. (1989): Tools to Support Formal Methods, *Proc. 11th International Conference on Software Engineering*, Pittsburgh, Pennsylvania.
- QUEMADA, J., AZCORRA, A., and FRUTOS, D. (1989): A Timed Calculus for LOTOS, *Proc. Second International Conference on Formal Description Techniques (FORTE'89)*, Vancouver, pp. 245–264.
- ŽIC, J. (1990): Some Thoughts on Communication System Performance Specification, In Hoang, D. B. and Chew, E., editors, *Proc. Open Distributed Processing Workshop*, Sydney.

## A User Interface

All process definitions, behaviour expressions and interpreter commands are entered using Cambridge Polish notation, i.e., a bracketed list consisting of a function name and a space-separated list of arguments. For example, the process

```
process AorB [a,b] :=
  a 5; stop
  []
  b {1..4}; exit
endproc
```

is typed as

```
(process AorB (a b) :=
  ( [] (; a 5 (stop))
    (; b (1 . 4) (exit)))
endproc)
```

To perform a behaviour expression in single-stepping mode the `interact` command is used. It prompts the user at each step, giving the user the power to make “externally” nondeterministic choices. For example (user inputs are in italics):

```
> (interact (AorB c d) (c d))
0: - started -           Next (+c +d)? +d
0(0): +d                 Next (-d)? -d
```

```
3(3): -d
3(0): - ended -
*finished*
```

After each event has been selected it is displayed, along with the global time at which it occurred and the elapsed time since the last observable event. Other traces are possible for this particular process, of course.

Alternatively, “hands off” execution until termination, or some maximum number of events have been performed, is possible using the `execute` command. It operates without user-guidance and returns a list of time/difference/event triples:

```
> (execute (AorB c d) (c d))
((0 0 +c) (5 5 -c) *stopped*)
```

Where a choice is possible `execute` randomly selects the next event.

The second argument to these two commands is the alphabet of the expression to be performed. Options are available for making internal actions observable and allowing events other than the earliest available ones to be selected.