

A LOTOS Interpreter for Simulating Real-Time Behaviour

C.J. Fidge

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

Abstract

A prototype Basic LOTOS interpreter, augmented for modelling real-time systems, is described. Primitive actions are treated as time-consuming, with separate start and end points, and a distinction is made between “true” and interleaved parallelism. Some practical difficulties encountered are discussed.

uses a notion of *time-consuming* actions in which both start and end points are visible.

More information is needed for modelling than is normally found in an abstract specification. Therefore distinct notations are introduced to separate interleaved from “true” parallelism, and time-consuming internal actions from an idle delay.

Introduction

Attempts to add the dimension of time to LOTOS have been constrained by the need to conform with its interleaving semantics. Consequently primitive actions have been treated as instantaneous, typically denoting only the end point of some event of interest (e.g., Quemada et al., 1989).

Although this is a valid abstraction for the purposes of specification we feel that it is less useful when simulating the timing behaviour of proposed real-time systems; our goal is *modelling* rather than requirements specification. This paper describes an experimental LOTOS interpreter which

Examples

Time is global and discrete; interpreter output is given as the absolute time from the instant at which the simulation began.

All *actions*, including the internal action *i*, are assumed to take some user-specified time to be performed. The possibility that an action can take zero time is allowed so that instantaneous events can still be modelled. Each occurrence of an action *a*, followed by a user-specified duration *t*, is bounded by two instantaneous *events*, *+a* and *-a*, denoting the instant at which *a* started and stopped, respectively. These 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) times may express uncertainty. Apart from a single integer a time value may be a set of possible durations, 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.

Despite the introduction of true parallelism, the interpreter is still semantically founded in the concept of interleaved traces. Based on the work of Quemada et al. (1989) trace events are 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.

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

```
(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.,

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

but this option is “off” by default.)

Hidden actions and the special **delay** action similarly cause time to advance. For example, behaviour expression

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

may result in trace

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

A **delay** represents an idle wait and produces no observable behaviour apart from its effect on time. It is distinct from an internal action, however, due to its behaviour when used as an initial action in a choice construct (below).

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 expression

```
(a 2; b 1; exit) [> (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,

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

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

$$a\ 1; (b\ 2; \text{stop}) \ []\ c\ 3; \text{stop}$$

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

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

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

$$a\ 1; (b\ 2; \text{stop}) \ []\ i\ 1; c\ 3; \text{stop}$$

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, the alternative 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 can 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

$$a\ 1; (b\ 2; \text{stop}) \ []\ \text{delay}\ 1; c\ 3; \text{stop}$$

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 (true) parallelism operator allows more than one action to occur at once, e.g.,

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

will generate the trace

$$+a_0^0, +d_0^0, -a_2^2, -d_2^4, +e_0^4, -e_2^6, +b_0^6, -b_3^9, \\ +c_0^9, -c_1^{10}$$

Some actions overlap in time, e.g., **a** and **d**. All processes participating in a shared action must begin and end it at the same instant. The first process is idle from time 2 to 6 due to the need to share action **b**. Where the participants disagree on the duration of a shared action its duration is the maximum of the possibilities so that worst-case behaviour is modelled.

Since knowledge of real-time behaviour makes the “configuration” of a system significant, a new notation is introduced to represent “interleaved” parallelism. For 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. Not surprisingly the simulation shows that the parallel example is faster than the interleaved one.

Implementation

The interpreter is written in Kyoto Common LISP and runs on a SUN-3 workstation. 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 expression

```
(a 5; stop)
[]
(b {1..4}; exit)
```

is typed as

```
([] (; a 5 (stop))
  (; b (1 . 4) (exit)))
```

To perform a behaviour expression in single-stepping mode the **interact** command is used. It issues a prompt at each step, giving the user the power to make nondeterministic choices. 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.

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

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. Using a functional language gave us the ability to quickly develop a prototype system and uncover a number of unexpected problems (below).

Difficulties Encountered

Distributed consensus. Where several processes participate in a shared event they must all agree on the time at which it occurs. This is like the notoriously difficult problem of achieving a “multi-way rendezvous”. The time model used by Quemada et al. (1989) also requires distributed consensus.

Indivisible sequences. Since a start event is always followed by a corresponding end event it would be convenient to have a 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.

“Earliest choice”. The decision to make choice operators select the first avail-

able event was not one of our original goals. Although some authorities advocate this approach (e.g., Liu and Shyamasundar, 1990) we feel that it inextricably links the logical and temporal aspects of the system description; it would be better to keep them orthogonal.

Unfortunately the “earliest choice” 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 it offers were generated by a choice or parallelism operator. Therefore, in guaranteeing the correct behaviour of the parallelism operator, 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”.

Conclusion

The interpreter has just become operational 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 LOTOS actions. The need for the `delay` action was an unforeseen requirement and the semantics of choice continue to be a cause for concern. Another unexpected result was the difficulty of implementing disabling—the inability to interrupt actions in progress seems counter-intuitive and has been strongly criticised.

Acknowledgements. I wish to thank the anonymous 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

- GORRIERI, R., MARCHETTI, S., and MONTANARI, U. (1990): A2CCS: Atomic Actions for CCS, *Theoretical Computer Science*, (72), pp. 203–223.
- 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.
- 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.