Parallel Simulation and LOTOS: Research In Progress*

Bruno R. Preiss†
Institutionen för Teleinformatik
Kungl. Tekniska Högskolan
Electrum 204, 164 40 Kista, Sweden

December 16, 2001

*This work was supported in part by Svenska institutet (The Swedish Institute), Stockholm, Sweden; Axel Wenner-Gren Stiftelse (The Axel Wenner-Gren Foundation for Scientific Research), Stockholm, Sweden; the Information Technology Research Centre (ITRC) of the Province of Ontario (Canada); and by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

†This report was prepared while the author was on sabbatical from the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

Revision 1.18. Copyright © 2001 by Bruno R. Preiss.

1 First Attempt

Consider the following LOTOS[1] specification [1]:

```
specification Demo1 [g, h] : noexit
behaviour
    A [g] |[g]| B [g, h] |[h]| C [h]

where
    process A [g] : noexit := TypeOne [g] endproc

    process B [g, h] : noexit := TypeTwo [g, h] endproc

    process C [h] : noexit := TypeOne [h] endproc

    process TypeOne [g] : noexit :=
        g; TypeOne [g]
    endproc

    process TypeTwo [g, h] : noexit :=
        (g; h; exit [] h; g; exit) >> TypeTwo [g, h]
    endproc
endspec
```

The system consists of three processes, A, B, and C. Processes A and B synchronize on gate g. Processes B and C synchronize on gate h. (See Figure 1.)

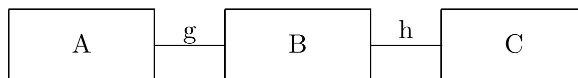

Figure 1:

The LOTOS semantics require that events happen one at a time. Initially, either event g or h can occur. A mechanism is needed to select an event and cause it to execute. The following solution, based to a large extent on Shipp's work, comes to mind[5]:

Figure 2 shows the use of three layers of logical processes. The process layer has an LP for each LOTOS process. When a process offers a gate, it sends a message up to the gate layer. The gate layer captures the synchronization constraints imposed by the LOTOS gates. If and only if all of the required processes has offered a gate, that gate is enabled. This results in a message sent to the arbitration layer. The arbiter selects one gate from the enabled ones, and broadcasts this to all of the processes.

Clearly, this approach has very limited parallelism. However, lets give *a* `Yaddes` *implementation* a try.

[1] All of the LOTOS specifications given in this report have been tested using the TOPO toolset[4, 2, 3].

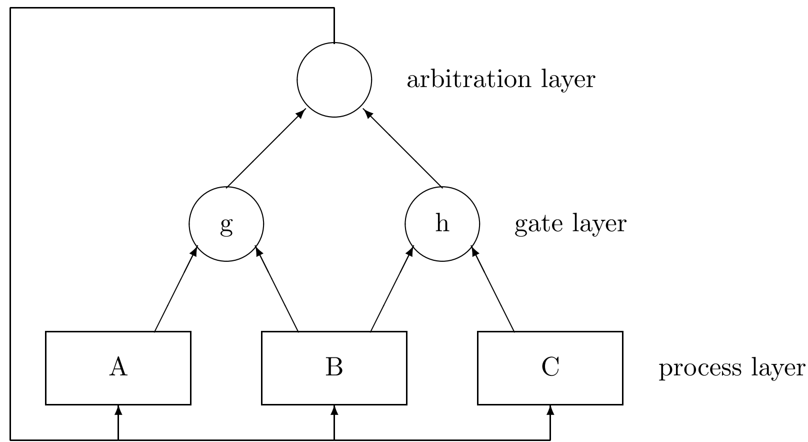

Figure 2:

1.1 A Yaddes Implementation

In this section, I describe an implementation of the simulation shown in Figure 2 using `Yaddes`[?].

1.1.1 Arbiter

Here is the `Yaddes` code for the Arbiter. This arbiter randomly selects one of the enabled input gates: (In the case of Chandy-Misra simulation, null messages are ignored.)

```
preamble
{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
}

constant maxArbiterFanIn = 8

model Arbiter
    inputs in [maxArbiterFanIn]
    outputs out
    state
    {
        int fanIn;
        RNG_TYPE rng;
    }
    initial state
    {
```

```
        0,
        RNG_INITIAL_VALUE
    }
    action initial and not (final or in [0 or .. maxArbiterFanIn - 1] )
    {
        int i;

        for (i = $state->fanIn; i < $maxArbiterFanIn; ++i)
            $deactivate ($in [i]);

        InitializeRNG ($pid + 1, &($state->rng));
        $settimelimit (3000);
    }
    action final and not (initial or in [0 or .. maxArbiterFanIn - 1] ) {}
    action in [0 or .. maxArbiterFanIn - 1] and not (initial or final)
    {
        int i;
        int count;
        int list [$maxArbiterFanIn];
        long n;

        count = 0;
        for (i = 0; i < $state->fanIn; ++i)
            if ($eventoccurred ($in [i]) && $event [$in [i]])
                list [count++] = $event [$in [i]];

        assert (count != 0);
        n = (Random (&($state->rng))>> 3) % count;

        (void) printf ("enabled");
        for (i = 0; i < count; ++i)
            (void) printf (" %c", 'g' + list [i] - 1);
        (void) printf (" choosing %c\n", 'g' + list [n] - 1);

        $output ($out, $time + 1, list [n]);
    }
end model
```

1.1.2 Gate

Here is the `Yaddes` code for a gate in the gate layer. A gate only sends an output message if it has received an input message on all of its inputs. (In the case of Chandy-Misra simulation, null messages are ignored.)

```
preamble
```

```
{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
}

constant maxGateFanIn = 4

model Gate
    inputs in [maxGateFanIn]
    outputs out
    state
    {
        int fanIn;
        int event;
    }
    initial state
    {
        0,
        0
    }
    action initial and not (final or in [0 or .. maxGateFanIn - 1] )
    {
        int i;

        for (i = $state->fanIn; i < $maxGateFanIn; ++i)
            $deactivate ($in [i]);
    }
    action final and not (initial or in [0 or .. maxGateFanIn - 1] ) {}
    action in [0 or .. maxGateFanIn - 1]
    {
        int i;
        int enabled = 1;

        for (i = 0; i < $state->fanIn; ++i)
            if (!$eventoccurred ($in [i]) || $event [$in [i]] == 0)
                enabled = 0;

        if (enabled)
            $output ($out, $time + 1, $state->event);
        else
            $nulloutput ($out, $time + 1, 0);
    }
end model
```

1.1.3 `TypeOne` Process

Here is the `Yaddes` code for a `TypeOne` process. This is the code corresponding to processes A and C in Figure 2. This process always offers a gate.

```
preamble
{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
}

model ProcessTypeOne
    inputs in
    outputs out
    state
    {
        int event;
    }
    initial state
    {
        0
    }
    action initial and not (final or in)
    {
        $output ($out, $time + 1, $state->event);
    }
    action final and not (initial or in) {}
    action in and not (initial or final)
    {
        if ($event [$in] == $state->event)
            (void) printf ("process %s next state\n", $name);
        $output ($out, $time + 1, $state->event);
    }
end model
```

1.1.4 `TypeTwo` Process

Here is the `Yaddes` code for a `TypeTwo` process. This is the code corresponding to process B in Figure 2. The specification is a little more complicated, because we have to specify a finite state machine corresponding to the behaviour given earlier.

```
preamble
{
#include <stdlib.h>
```

```
#include <stdio.h>
#include <assert.h>
}

constant typeTwoFanOut = 8

model ProcessTypeTwo
    inputs in
    outputs out [typeTwoFanOut]
    state
    {
        int state;
        int fanOut;
        int event [$typeTwoFanOut];
    }
    initial state
    {
        -1, 0, 0
    }
    action initial and not (final or in)
    {
        int i;

        for (i = 0; i < $state->fanOut; ++i)
            $output ($out [i], $time + 1, $state->event [i]);
    }
    action final and not (initial or in) {}
    action in and not (initial or final)
    {
        int i;
        int nextState;

        switch ($state->state)
        {
            case -1:
                for (i = 0; i < $state->fanOut; ++i)
                    if ($event [$in] == $state->event [i])
                    {
                        nextState = i;
                        break;
                    }
                break;
            default:
                nextState = -1;
```

```
                break;
        }
        for (i = 0; i < $state->fanOut; ++i)
        {
            if (nextState >= 0 && i != nextState)
                $nulloutput ($out [i], $time + 1, 0);
            else
                $output ($out [i], $time + 1, $state->event [i]);
        }
        $state->state = nextState;
        (void) printf ("process %s next state %d\n", $name, nextState);
    }
end model
```

1.1.5 Processes and Connections

The processes and the interconnections are specified as follows:

```
preamble
{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
}

import Gate from "gate.y"
import Arbiter from "arb.y"
import ProcessTypeOne from "type1.y"
import ProcessTypeTwo from "type2.y"

constant eventG = 1
constant eventH = 2

process A on 0 : ProcessTypeOne {$eventG}
process B on 0 : ProcessTypeTwo { -1, 2, $eventG, $eventH}
process C on 0 : ProcessTypeOne {$eventH}
process G on 1 : Gate {2, $eventG}
process H on 1 : Gate {2, $eventH}
process Z on 2 : Arbiter {2}

connect A .out to G .in [0]
connect B .out [0] to G .in [1]
connect B .out [1] to H .in [0]
connect C .out to H .in [1]
```

```
connect G .out to Z .in [0]
connect H .out to Z .in [1]

connect Z .out to A .in, B .in, C .in
```

1.2 Simulation Results

So, what happens when you run the simulation? Well, if you run the simulation for 1000 gate activations, you get the following statistics:

```
number of events posted 8004
number of events not posted 1000
number of events processed 8000
number of generations 3002
average generation size 2.00266
number of model calls 6012
average model calls per generation 2.00266
number of predictions 8004
average prediction 1
```

The interesting number is that the average generation size is 2. Not surprising since the way the simulation works is that, first all three processes in the process layer fire, then both processes in the gate layer fire, and then the arbiter fires—average is 2.

However, this number is high, because all of the processes in the process layer fire every time through the cycle—even if they are not involved with the particular gate selected by the arbiter. (They need to execute, so they can know that their gate was not activated!)

Well, this approach seems like a dead end.

2 Second Attempt

The problem with the previous approach seems to be that it does not capture the desired parallelism. LOTOS is supposed to be good at specifying systems with inherent parallelism. Yet the previous approach imposes a kind of sequentiality on the execution of the simulation.

So, lets come up with an example that has trivial (embarrassing?) parallelism.

```
specification Demo2 : noexit
behaviour
    ProcessOne ||| ProcessTwo

where
    process ProcessOne : noexit :=
        hide a, b in
            (a; exit [] b; exit) >> ProcessOne
    endproc
```

```
    process ProcessTwo : noexit :=
        hide c, d in
            (c; exit [] d; exit) >> ProcessTwo
    endproc
endspec
```

The system consists of two process, `ProcessOne` and `ProcessTwo`. These processes do not interact at all. Therefore, they should be able to execute in parallel.

However, if we follow the approach taken in the previous example, we end up with a simulation that looks like Figure 3.

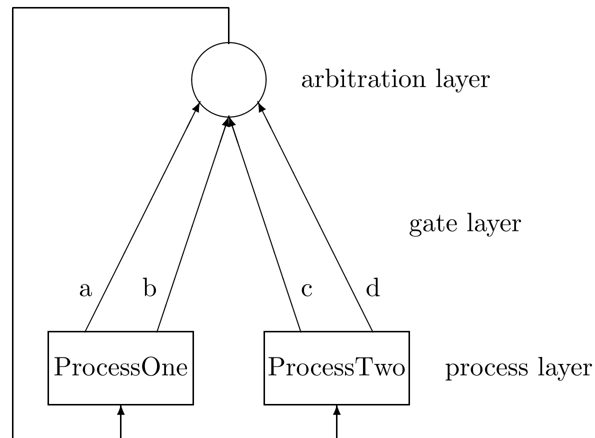

Figure 3:

Clearly, Figure 3 shows that we have overconstrained the synchronization requirements. Even though events could be processed simultaneously, the existence of the central arbiter prevents this possibility. We need *another way of looking at the problem.*

2.1 Another Way of Looking at the Problem

We can represent the evolution of a simulation of the system by a graph as shown in Figure 4.

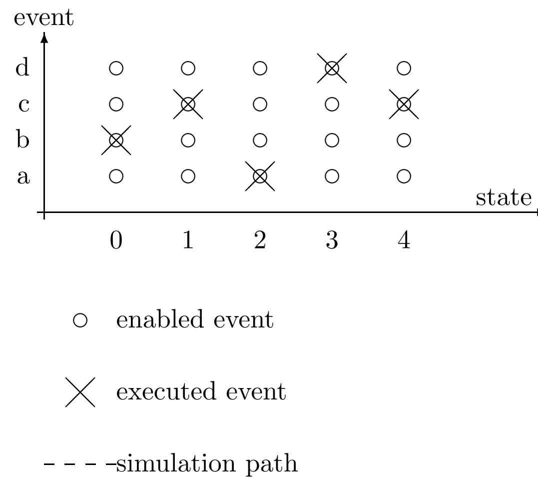

Figure 4:

In this figure, the horizontal axis represent the system state. The vertical axis represents events. A circle indicates enabled events. For example, in system state 0, events a, b, c, and d, are all enabled.

In a given simulation of the system, exactly one event is selected for execution. This is indicated on the graph by an $\times$. For example, in system state 0, event b has been selected for execution. Upon executing event b, the system enters state 1, in which events a, b, c, and d, are all enabled. A particular simulation executes some sequence of events, tracing-out a path in the graph.

How do we make this parallel? Well to answer this question, we need a *model of execution*.

2.2 A Model of Execution

Ok, so here is a model of execution:

> The processing of an event consumes some amount of computation time. Lets assume that this amount is a fixed constant, and that it is the same for all events.

On the basis of this execution model, we can redraw Figure 4 as shown in Figure 5.

In this figure, the horizontal axis represents time, and the vertical axis represents processes. The horizontal lines in the figure represent computation. Each line is labelled with the event being processed. This figure shows the sequential constraints that a strict interpretation of LOTOS places on the execution.

How can we *introduce some parallelism?*

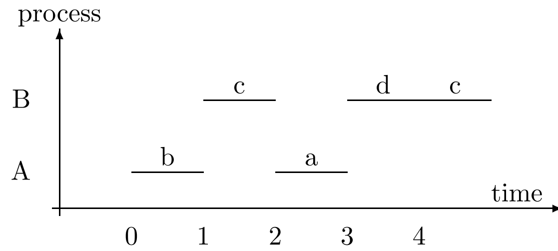

Figure 5:

2.3 Introducing Parallelism

Suppose we had the ability to predict the future. Then, we could compress the execution shown in Figure 5 as shown in Figure 6. This figure shows that with two processors, perfect speedup would result in a speedup of $5/3 = 1.7$.

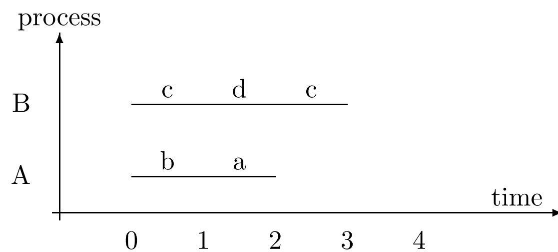

Figure 6: Perfect Speedup

In order to achieve this speedup, we need *a new form of optimism*.

2.4 Another Example[2]

Lets look at the first LOTOS specification given in Section 1. Remember, it had three processes, A, B, and C, synchronizing via two gates, g and h. Figure 7 shows a simulation trace for this example.

2.5 Extend the Model

In order to draw the execution of this case, we need to extend the execution model given in Section 2.2. I.e., we need to address the question of processes synchronizing on a gate—in particular, which process or processes execute an event when a gate fires?

[2](I suppose you thought this section would be called *A New Form of Optimism*. Well, I don't know what that might be, so I will do some more examples.)

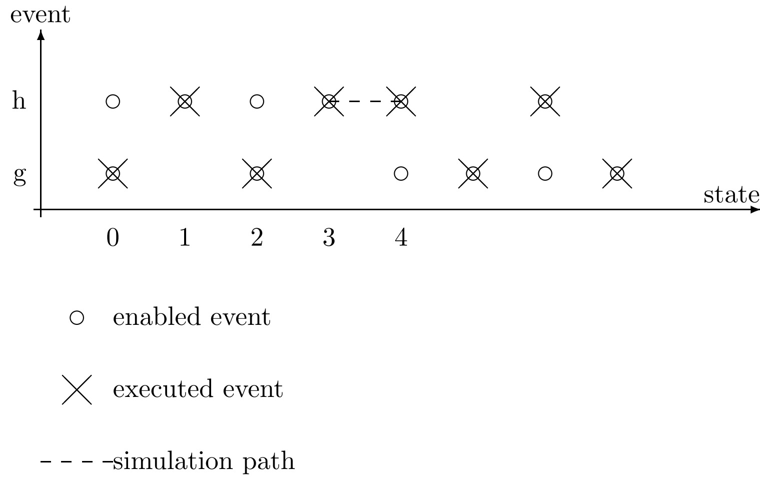

Figure 7:

> The execution time of an event is consumed in each of the processes synchronized on that event. For example, if two processes, say A and B, synchronize on on gate g, then if g is executed, both processes A and B perform computation associated with that event.

On the basis of the preceding assumption, we get the execution profile shown in Figure 8 for the three processes involved in the simulation:

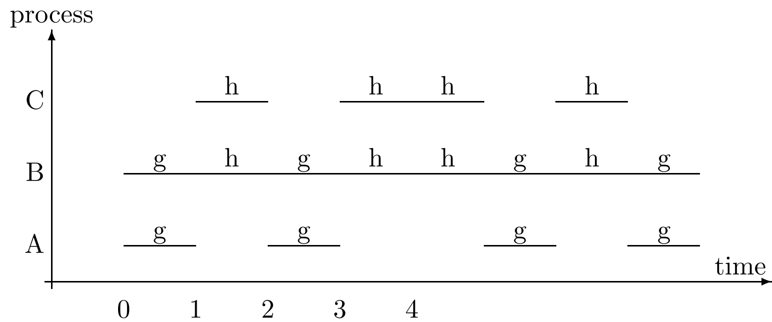

Figure 8:

We can draw two conclusions from this figure. First, the parallel execution already has speedup with respect to a purely sequential execution. For example, in this case, the speedup is 2 on three processors.

The second conclusion is that, in this example, if we assume that the events processed by a single process (process B) cannot be executed in parallel, no further speedup is possible. I.e., even with perfect prediction, we cannot do any better.

Is this just a quirk of this example? Lets look at *yet another example*.

2.6 Yet Another Example

This example is the one worked by Neil Shipp.

```
(* Neil Shipp's example *)
specification test[a,b,c,d] :noexit
    behaviour
        hide g,h in
            A[g,a] |[g]| B[g,h,b,c] |[h]| C[h,d]
    where
        process A[a,b] : noexit :=
            (a; exit [] i; exit) >> b; A[a,b]
        endproc

        process B[a,b,c,d] : noexit :=
            (a; c; exit [] b; d; exit) >> B[a,b,c,d]
        endproc

        process C[a,b] : noexit :=
            (a; exit [] i; exit) >> b; C[a,b]
        endproc
endspec
```

This example has three processes, A, B, and C, and eight gates—external gates a, b, c, and d; hidden g and h; and the internal gates i inside processes A and B.

Figure 9 shows a possible simulation trace for this example.

Using our execution model, we get the execution shown in Figure 10.

Figure 10 shows that the execution model yields an inherent speedup of $7/5 = 1.4$ on three processors. In Figure 11 we make the perfect prediction assumption. I.e., assume we can predict with perfect accuracy which events will be executed in each process.

This figure shows that the ideal speedup is $7/3 = 2.7$ on three processors. In order to achieve this speedup, we need *a new form of optimism*.

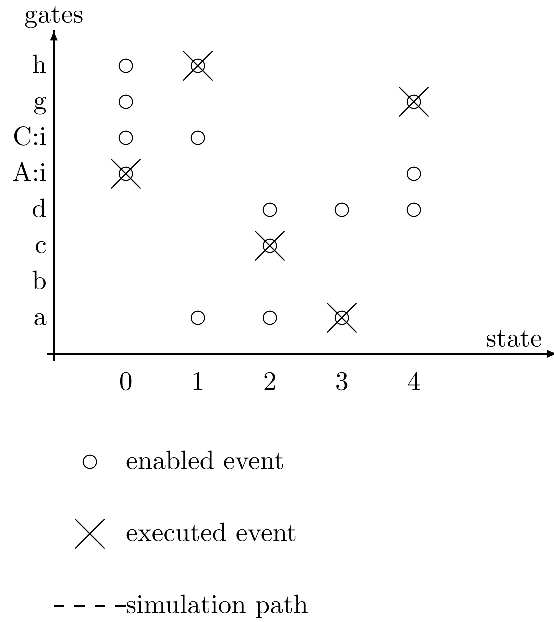

Figure 9:

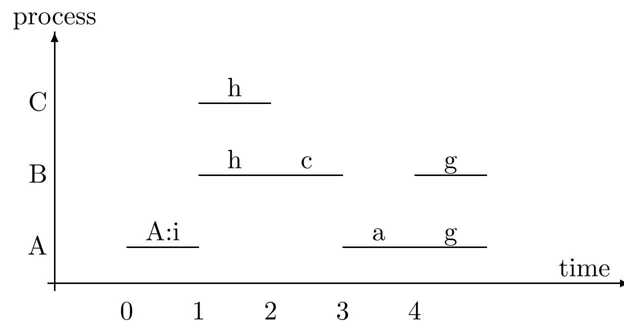

Figure 10:

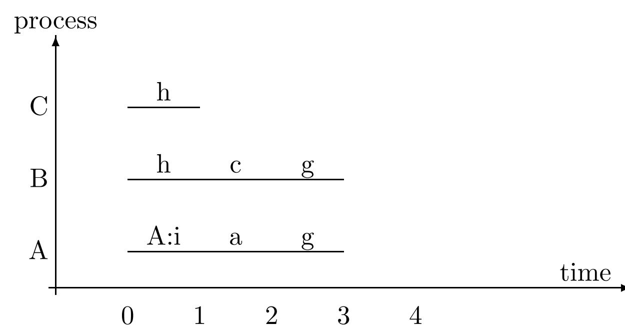

Figure 11:

3 Interesting Questions

The observations outlined in the previous section raise a number of interesting questions:

1. Do LOTOS specifications have sufficient inherent parallelism that it is worthwhile trying to make parallel simulations based upon them?

2. If we find that LOTOS specifications do have sufficient parallelism, how do we implement the parallel simulation. Specifically, what parts of the LOTOS specification give rise to logical processes, and how do you implement distributed gate rendezvous?

3. Can we introduce a new form of optimism that tries to achieve perfect speedup on the basis of speculative computation?

4 Some Thoughts on the New Optimism

In Time Warp, the optimism is *synchronization optimism*. I.e., we are assuming something about the arrival *order* of input messages—namely, that they will arrive in order, and that we can process them as soon as they arrive.

To do LOTOS, we need a different kind of optimism:

1. we need to be able to predict the set of enabled events; and

2. we need to be able to predict which event will be selected from the set of enabled events.

How might we predict the enabled set? One possibility is to assume that the next enabled set is the same as the current enabled set minus the event just fired.

How might we select a possible event from the enabled set? On possibility is to use the same algorithm we would use in the normal case. (Whatever that algorithm might actually be!)

Will it work? It depends on the likelihood of constructing the correct enabled set and selecting the correct event from that set.

What is the likelihood of doing this? That probably depends on the nature of the system being simulated. However, I suspect that the success rate will be very good when the size of the enabled set is large. If the enabled set is large, then I would expect that the typical event would only cause a small number of retractions from the enabled set. So, we would have a good chance of picking the right next event.

5 Proposal for an Experiment

Objective To measure the potential speedup due to optimistic execution inherent in a LOTOS specification assuming the execution model given in Sections 2.2 and 2.5.

Approach Implement (somehow) a simulation (of something) specified in LOTOS. Run the simulation and at each step:

1. begin with the enabled set,
2. select an event for execution,
3. fire the event and compute the next enabled set, and
4. determine "all" optimistic events that might be executed and determine which of these would correspond to correct predictions.

To do this, we need to understand how to construct the correct new system state from the optimistic executions. In this regard, we make the following observations:

- Events only affect the processes that participate in them. Therefore, firing an event only affects a subset of the system state. It should be possible to merge the affected substates into a consistent new system state.
- Each event is associated with a set of processes. We need to select events for execution in such a way as to cause as many processes to be activated as possible, but to only activate any given process once.

A Possible Benchmarks

As a possible source of interesting LOTOS benchmarks, I am considering the paper by Turner et al.[6], in which they present a way to specify logic gates in LOTOS. Although I am not entirely satisfied with their approach, it does give rise to interesting LOTOS examples with large enabled event sets.

Here is a sample specification based on the approach given in Turner's paper:

```
specification Circuit: noexit

library Boolean
endlib

type BoolOp is Boolean
    sorts BoolOp

    opns
        ident: -> BoolOp
        not: -> BoolOp
        and: -> BoolOp
        or: -> BoolOp
        nand: -> BoolOp
        nor: -> BoolOp
```

```
        Apply: BoolOp, Bool -> Bool
        Apply: BoolOp, Bool, Bool -> Bool

    eqns
        forall x, y: Bool

        ofsort Bool
            Apply (ident, x) = x;
            Apply (not, x) = not (x);
            Apply (and, x, y) = x and y;
            Apply (or, x, y) = x or y;
            Apply (nand, x, y) = not (x and y);
            Apply (nor, x, y) = not (x or y);
endtype

behaviour

    hide node00, node01, node02 in
        (Source [node00] (false) ||| Source [node01] (true) )
        |[node00, node01]|
        AndNot [node00, node01, node02]
        |[node02]|
        Sink [node02]

where

    process Source [output] (value: Bool): noexit :=
        output ! value; stop
    endproc

    process Sink [input]: noexit :=
        input ? value: Bool; Sink [input]
    endproc

    process Logic1 [input, output] (operator: BoolOp): noexit :=
        Logic1A [input, output] (operator, false)
    where

        process Logic1A [input, output] (operator: BoolOp,
                prevInput: Bool): noexit :=
            input ? newInput: Bool; Logic1A [input, output] (operator, newInput)
            []
            (
                let newOutput: Bool = Apply (operator, prevInput)
```

```
            in
                output ! newOutput; Logic1B [input, output]
                (operator, prevInput, newOutput)
        )

    endproc

    process Logic1B [input, output] (operator: BoolOp,
            prevInput: Bool, prevOutput: Bool): noexit :=
        input ? newInput: Bool; Logic1B [input, output] (operator,
            newInput, prevOutput)
        []
        (
            let newOutput: Bool = Apply (operator, prevInput)
            in
                output ! newOutput [newOutput ne prevOutput];
                Logic1B [input, output] (operator, prevInput, newOutput)
        )
    endproc

endproc

process Logic2 [input0, input1, output] (operator: BoolOp): noexit :=
    Logic2A [input0, input1, output] (operator, false, false)
where

    process Logic2A [input0, input1, output] (operator: BoolOp,
            prevInput0: Bool, prevInput1: Bool): noexit :=
        input0 ? newInput0: Bool;
            Logic2A [input0, input1, output]
                (operator, newInput0, prevInput1)
        []
        input1 ? newInput1: Bool;
            Logic2A [input0, input1, output]
                (operator, prevInput0, newInput1)
        []
        (
            let newOutput: Bool = Apply (operator, prevInput0, prevInput1)
            in
                output ! newOutput; Logic2B [input0, input1, output]
                (operator, prevInput0, prevInput1, newOutput)
        )

    endproc
```

```
        process Logic2B [input0, input1, output] (operator: BoolOp,
                prevInput0: Bool, prevInput1: Bool, prevOutput: Bool): noexit :=
            input0 ? newInput0: Bool; Logic2B [input0, input1, output] (operator,
                newInput0, prevInput1, prevOutput)
            []
            input1 ? newInput1: Bool; Logic2B [input0, input1, output] (operator,
                prevInput0, newInput1, prevOutput)
            []
            (
                let newOutput: Bool = Apply (operator, prevInput0, prevInput1)
                in
                    output ! newOutput [newOutput ne prevOutput];
                    Logic2B [input0, input1, output]
                        (operator, prevInput0, prevInput1, newOutput)
            )
        endproc

    endproc

    process Inverter [input, output]: noexit :=
        Logic1 [input, output] (not)
    endproc

    process TwoInputAnd [input0, input1, output]: noexit :=
        Logic2 [input0, input1, output] (and)
    endproc

    process AndNot [input0, input1, output]: noexit :=
        hide notInput0 in
            Inverter [input0, notInput0] |[notInput0]|
            TwoInputAnd [notInput0, input1, output]
    endproc

endspec
```

References

[1] International Organization for Standardization. *International Standard ISO 8807: Information processing systems—Open Systems Interconnection—LOTOS—A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[2] J. A. Mañas, T. de Miquel, T. Robles, J. Salvachua, G. Huecas, and M. Veiga. TOPO: Quick reference Auxiliary tools—version 3R2. On-line document accompanying the

TOPO toolset, June 1993.

[3] J. A. Mañas, T. de Miquel, T. Robles, J. Salvachua, G. Huecas, and M. Veiga. TOPO: Quick reference C code generator—version 3R2. On-line document accompanying the TOPO toolset, Oct. 1993.

[4] J. A. Mañas, T. de Miquel, T. Robles, J. Salvachua, G. Huecas, and M. Veiga. TOPO: Quick reference Front-end—version 3R2. On-line document accompanying the TOPO toolset, June 1993.

[5] N. L. Shipp. On the parallelization of LOTOS. Work-term Report, Department of Electrical and Computer Engineering, University of Waterloo, 10 pp., Dec. 1993.

[6] K. J. Turner and R. O. Sinnott. DILL: Specifying digital logic in LOTOS. In *Proc. FORTE '93*, 1993.