An Algorithm for Speculative Parallel Execution of Rendezvous-Synchronized Simulation*

Bruno R. Preiss†
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Abstract

This paper presents an approach for speculative parallel execution of rendezvous-synchronized simulations. Rendezvous-synchronized simulation is based on the notions of processes and gates and on the rendezvous mechanism defined in the the basic process algebra of LOTOS*—a standard formal specification language for temporal ordering[2]. Time is introduced via a mechanism similar to the* `delay` *behaviour annotation provided by the* TOPO *toolset[3–5].*

The algorithm allows speculative gate activations. This increases the available parallelism while ensuring correct execution of the computation.

The model is used to describe closed stochastic queueing network simulations. Analysis of their execution results suggests that the model makes available a promising degree of parallelism.

1 Introduction

This paper presents an approach for speculative parallel execution of rendezvous-synchronized simulations. Rendezvous-synchronized execution is based on a rather simple model of parallel computation. The basic elements of the model are a set of processes and a set of gates. The processes interact with each other using a rendezvous mechanism inspired by the basic LOTOS process algebra[2]. Time is introduced via a mechanism similar to the `delay` behaviour annotation provided by the TOPO toolset[3–5].

The approach taken in this work is different from other concurrent implementations of LOTOS, in that the full LOTOS semantics are not addressed[1, 7]. In this model, the set of processes and the set of gates are static—processes and gates are neither created nor destroyed during the lifetime of the system and their interconnection is fixed. These restrictions admit speculative parallel execution without the complexities that can arise in general LOTOS specifications.

Each of the processes can be thought of as a finite-state machine. The state of the system is the composition of the states of the processes. A state transition is triggered by the activation of a gate. In any given system state, a process may *offer* to participate in a gate activation after a certain time on any number of the gates to which it is connected. A gate is *enabled* if all of the processes attached to it have *offered* to participate after a finite time delay. The rendezvous time for the gate is equal to latest time offered on that gate.

Although in any given system state any number of gates may be enabled, exactly one gate will be activated—the gate having the smallest rendezvous time. The activation of a gate is an *event*. A gate activation causes a state transition in all of the processes attached to that gate. Processes that are not directly connected to an activated gate are unaffected by its activation. The behaviour of the system is simply a sequence of system state transitions triggered by single-gate activations.

This paper presents an algorithm that allows speculative gate activations (i.e., speculative computation). This increases the available parallelism while ensuring correct execution of the simulation.

The remainder of the paper is organized as follows: The execution model is formally specified in Section 2. An example illustrating the model is given in Section 3. A sequential execution algorithm is given in Section 4. Section 5 gives the speculative parallel execution algorithm. The application of the model to the specification of closed stochastic queueing network

*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 at Institutionen för Teleinformatik, Kungl. Tekniska Högskolan, Electrum 204, 164 40 Kista, Sweden.

simulations is presented in Section 6. Some results obtained from the execution of such simulations are given in Section 7. Finally, the contributions of this work are summarized in Section 8.

2 Formal Model

- A system consists of n processes, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$.
- These processes interact via m input/output ports called *gates*, $\mathcal{G} = \{g_1, g_2, \ldots, g_m\}$.
- Virtual time is represented by a set of values $\mathcal{T} = \mathcal{T}_0 \cup \{\infty\}$, where set $\mathcal{T}_0$ is a totally ordered set and is closed under addition. By definition, $t < \infty : \forall t \in \mathcal{T}_0$ and $t + \infty = \infty : \forall t \in \mathcal{T}_0$.
- Each process is a finite-state automaton. E.g., the state of process p_i is $s_i \in \mathcal{S}_i$ where $\mathcal{S}_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,|\mathcal{S}_i|}\}$.
- Each process p_i has associated with it a local virtual time $t_i \in \mathcal{T}$.
- A set of gates is *attached* to each process. A process may be attached to any subset of the gates $\mathcal{G}$, provided it is attached to at least one gate. The set of gates attached to process p_i is $\mathcal{G}_i$ where $\mathcal{G}_i \subseteq \mathcal{G}$ and $|\mathcal{G}_i| > 0$.
- The set of processes *attached* to a gate g_j is $\mathcal{C}_j = \{p_i \in \mathcal{P} : g_j \in \mathcal{G}_i\}$.
- A *event*, $e = (t, g)$, is an element of $\mathcal{E} = \mathcal{T}_0 \times \mathcal{G}$.
- A given process p_i may be involved only with events on gates $g \in \mathcal{G}_i$. Let $\mathcal{E}_i \subseteq \mathcal{E}$ be the set of events in which process p_i may be involved. Thus, $\mathcal{E}_i = \{(t, g) \in \mathcal{E} : g \in \mathcal{G}_i\}$.
- An offer vector, is an element $\mathbf{O} \in \mathcal{O}$ where $\mathcal{O} = \mathcal{T}^m$. I.e., $\mathbf{O} = (t_1, t_2, \ldots, t_m)$ where the element t_i is associated with gate g_i.
- The state of the system is the vector $\mathbf{S} \in \mathcal{S}$ where

  $$\mathcal{S} = (\mathcal{O}_1 \times \mathcal{S}_1 \times \mathcal{T}) \times (\mathcal{O}_2 \times \mathcal{S}_2 \times \mathcal{T}) \times \cdots \times (\mathcal{O}_n \times \mathcal{S}_n \times \mathcal{T}).$$

  I.e.,
  $\mathbf{S} = ((\mathbf{O}_1, s_1, t_1), (\mathbf{O}_2, s_2, t_2), \ldots, (\mathbf{O}_n, s_n, t_n))$.
- Each process p_i is described by a *model*. A model is given by two functions.

  The first function, $\alpha_i : \mathcal{S}_i \times \mathcal{E}_i \mapsto \mathcal{S}_i$, describes *behaviour*—it gives the next state that process p_i enters after processing the given event in the current state.

  The second function, $\beta_i : \mathcal{S}_i \times \mathcal{T}_0 \mapsto \mathcal{O}$, gives the offer vector for process p_i in a given state and at a given time. A process p_i can make offers with finite times only on gates to which it is attached. I.e., since process p_i is attached only to gates in $\mathcal{G}_i$, the result of the function is of the form $\beta_i(s_i, t) = \mathbf{O}$ where the jth element of $\mathbf{O}$ (denoted $\mathbf{O}_{(j)}$) is given by

  $$\mathbf{O}_{(j)} = \begin{cases} t' : t' \geq t, & g_j \in \mathcal{G}_i \\ \infty, & \text{otherwise} \end{cases}$$

  for $1 \leq j \leq m$.

  Note, if a process does not offer at least one gate at a virtual time less than ∞, that process is said to be *stopped*.
- Each model must satisfy the following *causality constraint*: $\beta_i(s, t) = (t_1, t_2, \ldots, t_m)$, $\forall s \in \mathcal{S}_i$, $\forall t \in \mathcal{T}_0$, $t_j \geq t$, $1 \leq j \leq m$.

  In other words, offers cannot be made in the past.
- A model is said to be *time-invariant* if both the offers and behaviour functions are time invariant.

  The behaviour function is time invariant if

  $$\alpha_i(s, (t + \delta, g)) \equiv \alpha_i(s, (t, g))$$

  where $0 \leq \delta < \infty$, $\forall s \in \mathcal{S}_i$, $\forall g \in \mathcal{G}_i$[1].

  The offers function is time invariant if

  $$\begin{aligned} \beta_i(s, t) &= (t_1, t_2, \ldots, t_m) \\ \Longrightarrow \beta_i(s, t + \delta) &= (t_1 + \delta, t_2 + \delta, \ldots, t_m + \delta) \end{aligned}$$

  where $0 \leq \delta < \infty$, $\forall s \in \mathcal{S}_i$.
- Given a system state vector $\mathbf{S} = ((\mathbf{O}_1, s_1, t_1), (\mathbf{O}_2, s_2, t_2), \ldots, (\mathbf{O}_n, s_n, t_n))$, the function $\rho : \mathcal{S} \mapsto \mathcal{O}$, computes the *rendezvous vector* $\mathbf{R} = \rho(\mathbf{S}) = (t'_1, t'_2, \ldots, t'_m)$.

  The elements of the rendezvous vector are obtained from the n offers vectors $\mathbf{O}_i = (t_{i,1}, t_{i,2}, \ldots, t_{i,m})$, as follows:

  $$t'_j = \max_{p_i \in \mathcal{C}_j} t_{i,j}.$$

[1] This definition of time invariance is actually more strict than necessary. The only part of the model visible to an external observer are the gate offers. The constraint $\alpha_i(s, (t+\delta, g)) \equiv \alpha_i(s, (t, g))$ says that the model enters the same state, regardless of the time of the event. However, it is possible that two states are equivalent in the sense that they are indistinguishable from each other on the basis of all possible subsequent gate interactions. Such a model could also be said to be time invariant.

In other words, the jth element of $\mathbf{R}$, which corresponds to gate g_j, is equal to the maximum of the offer times for that gate made by the processes attached to that gate.

If $t_j = \infty, \forall g_j \in \mathcal{G}$, then the system state $\mathbf{S}$ is said to be a *deadlock*[2].

- The scheduling function is $\Gamma : \mathcal{O} \mapsto \mathcal{E}$ where, given $\mathbf{R} = (t_1, t_2, \ldots, t_m)$,

$$\Gamma(\mathbf{R}) = (t_i, g_i) : t_i = \min_{1 \le j \le m} t_j.$$

  I.e., given the rendezvous vector, Γ selects the gate with the smallest rendezvous time.

- The system state transition function $\Theta : \mathcal{S} \times \mathcal{E} \mapsto \mathcal{S}$ applies an event to a system state to produce the new system state. I.e. given $\mathbf{S} = ((\mathbf{O}_1, s_1, t_1), (\mathbf{O}_2, s_2, t_2), \ldots, (\mathbf{O}_n, s_n, t_n))$ and an event $e = (t, g)$, $\mathbf{S}' = \Theta(S, e)$, where

$$\mathbf{S}'_{(i)} = \begin{cases} (\beta_i(s'_i, t), s'_i, t) : s'_i = \alpha_i(s_i, e), & g \in \mathcal{G}_i \\ (\mathbf{O}_i, s_i, t_i), & \text{otherwise} \end{cases}$$

  and $1 \le i \le n$.

- The initial state of the system is comprised of the initial states and initial offers of the individual processes, $\mathbf{S}^0 = \big((\mathbf{O}_1, s_1^0, 0), (\mathbf{O}_1, s_2^0, 0), \ldots, (\mathbf{O}_n, s_n^0, 0)\big)$ where $\mathbf{O}_j = \beta_j(s_j^0, 0)$ for $1 \le j \le n$.

- A *simulation* is an element of $(\mathcal{S} \times \mathcal{E})^+$. (The $+$ denotes positive closure.) The function $\Omega : \mathcal{S} \mapsto (\mathcal{S} \times \mathcal{E})^+$ computes a simulation from the initial state $\mathbf{S}^0$ as follows: $\Omega(\mathbf{S}^0) = \{(\mathbf{S}^0, e^0), (\mathbf{S}^1, e^1), \ldots\}$, where $e^i = \Gamma(\rho(\mathbf{S}^i))$, and $\mathbf{S}^{i+1} = \Theta(\mathbf{S}^i, e^i)$, for $i = 0, 1, 2, \ldots$.

- The length of a simulation is either finite or it is infinite. If it is finite, then the last state (and only the last state) is a deadlock. If the simulation is infinite, then no state in the simulation is a deadlock.

3 An Example

This example makes use of the *behaviour annotations* provided by the TOPO toolset[3–5] to add temporal information to a LOTOS specification. The behavioural annotation takes the form of a LOTOS comment of the form:

```
(*| delay δ |*)
```

where δ is an integer-valued expression. The meaning of the annotation is that the following action denotation will not be offered before δ time units have elapsed[4].

Consider the LOTOS specification given in Figure 1.

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

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

    process B [u, v] : noexit := TypeTwo [u, v]
    endproc

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

    process TypeOne [u] : noexit :=
        (*| delay 1 |*) u; TypeOne [u]
    endproc

    process TypeTwo [u, v] : noexit :=
        (
            u; (*| delay 2 |*) v; exit
        []
            v; (*| delay 3 |*) u; exit
        ) >> TypeTwo [u, v]
    endproc
endspec
```

Figure 1: LOTOS Example with Behavioural Annotations

- There are three processes, $\mathcal{P} = \{A, B, C\}$.

- Process A has one state, $\mathcal{S}_A = \{a_0\}$. Process B has three states, $\mathcal{S}_B = \{b_0, b_1, b_2\}$. Process C has one state, $\mathcal{S}_C = \{c_0\}$.

- There are two gates, $\mathcal{G} = \{\mathtt{g}, \mathtt{h}\}$.

- Process A is attached to one gate, $\mathcal{G}_A = \{\mathtt{g}\}$. Process B is attached to two gates, $\mathcal{G}_B = \{\mathtt{g}, \mathtt{h}\}$. Process C is attached to one gate, $\mathcal{G}_C = \{\mathtt{h}\}$. (See Figure 2 below.)

- The model for process A is given by:

[2] It is sometimes convenient to distinguish between *termination* and *deadlock*. If a particular deadlock state arises because no process has offered a finite rendezvous on any gate, the simulation can be said to have *terminated*. For the purpose of this paper, this distinction is unimportant.

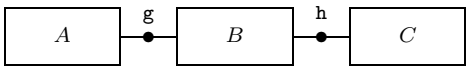

Figure 2:

$s_A \in \mathcal{S}_A$	$e \in \mathcal{E}_A$	$\alpha_A(s_A, e)$
a_0	$(t, \mathtt{g}) : \forall t \in \mathcal{T}_0$	a_0

$s_A \in \mathcal{S}_A$	$(o_\mathtt{g}, o_\mathtt{h}) = \beta_A(s_A, t)$
a_0	$(t+1, \infty)$

Process A satisfies the causality constraint and is time invariant.

- The model for process B is given by:

$s_B \in \mathcal{S}_B$	$e \in \mathcal{E}_B$	$\alpha_B(s_B, e)$
b_0	$(t, \mathtt{g}) : \forall t \in \mathcal{T}_0$	b_1
b_0	$(t, \mathtt{h}) : \forall t \in \mathcal{T}_0$	b_2
b_1	$(t, \mathtt{h}) : \forall t \in \mathcal{T}_0$	b_0
b_2	$(t, \mathtt{g}) : \forall t \in \mathcal{T}_0$	b_0

$s_B \in \mathcal{S}_B$	$(o_\mathtt{g}, o_\mathtt{h}) = \beta_B(s_B, t)$
b_0	(t, t)
b_1	$(t+2, \infty)$
b_2	$(\infty, t+3)$

Process B satisfies the causality constraint and is time invariant.

- The model for process C is given by:

$s_C \in \mathcal{S}_C$	$e \in \mathcal{E}_C$	$\alpha_C(s_C, e)$
c_0	$(t, \mathtt{h}) : \forall t \in \mathcal{T}_0$	c_0

$s_C \in \mathcal{S}_C$	$(o_\mathtt{g}, o_\mathtt{h}) = \beta_C(s_C, t)$
c_0	$(\infty, t+1)$

Process C satisfies the causality constraint and is time invariant.

- The initial state $\mathbf{S}^0$ is of the form $((\mathbf{O}_A^0, a_0, 0), (\mathbf{O}_B^0, b_0, 0), (\mathbf{O}_C^0, c_0, 0))$, where

$$\begin{array}{lclcl} \mathbf{O}_A^0 & = & \beta_A(s_A^0, 0) & = & (1, \infty) \\ \mathbf{O}_B^0 & = & \beta_B(s_B^0, 0) & = & (0, 0) \\ \mathbf{O}_C^0 & = & \beta_C(s_C^0, 0) & = & (\infty, 1) \end{array}$$

The rendezvous vector for this state is $\mathbf{R}^0 = (r_\mathtt{g}, r_\mathtt{h}) = (1, 1)$.

In this state $r_\mathtt{g}$ is 1 because $\mathcal{C}_\mathtt{g} = \{A, B\}$, and because process A offers 1 and process B offers 0 on gate $\mathtt{g}$. Similarly, $r_\mathtt{h}$ is 1 because $\mathcal{C}_\mathtt{h} = \{B, C\}$, and because process B offers 0 and process C offers 1 on gate $\mathtt{h}$.

- The scheduling function, Γ, picks the gate having the smallest offered rendezvous times. In this case, there is a tie. Suppose that in the case of a tie, the scheduler picks a gate at random.

- Let $e^0 = \Gamma(\rho(\mathbf{S}^0)) = (1, \mathtt{g})$.

  Then $\mathbf{S}^1 = ((\mathbf{O}_A^1, s_A^1, 1), (\mathbf{O}_B^1, s_B^1, 1), (\mathbf{O}_C^0, c_0, 0))$ where

$$\begin{array}{lclcl} s_A^1 & = & \alpha_A(a_0, e^0) & = & a_0 \\ s_B^1 & = & \alpha_B(b_0, e^0) & = & b_1 \\ \mathbf{O}_A^1 & = & \beta_A(s_A^1, 1) & = & (2, \infty) \\ \mathbf{O}_B^1 & = & \beta_B(s_B^1, 1) & = & (\infty, 3) \end{array}$$

- The rendezvous vector for state $\mathbf{S}^1$ is $\mathbf{R}^1 = (r_\mathtt{g}, r_\mathtt{h}) = (\infty, 3)$.

  In this state gate $r_\mathtt{g}$ is ∞ because process A offers 2 and process B offers ∞ on gate $\mathtt{g}$. However, $r_\mathtt{h}$ is 3 because process B offers 3 and process C offers 1 on gate $\mathtt{h}$.

- Now the scheduler selects $e^1 = \Gamma(\rho(\mathbf{S}^1)) = (3, \mathtt{h})$.

  Then $\mathbf{S}^2 = ((\mathbf{O}_A^1, a_0, 1), (\mathbf{O}_B^2, s_B^2, 3), (\mathbf{O}_C^2, s_C^2, 3))$ where

$$\begin{array}{lclcl} s_B^2 & = & \alpha_B(b_1, e^1) & = & b_0 \\ s_C^2 & = & \alpha_C(c_0, e^1) & = & c_0 \\ \mathbf{O}_B^2 & = & \beta_B(s_B^2, 3) & = & (3, 3) \\ \mathbf{O}_C^2 & = & \beta_C(s_C^2, 3) & = & (\infty, 4) \end{array}$$

- The simulated system has undergone two state transitions so far:

$$\mathbf{S}^0 \overset{(1,\mathtt{g})}{\rightsquigarrow} \mathbf{S}^1 \overset{(3,\mathtt{h})}{\rightsquigarrow} \mathbf{S}^2$$

  where

$$\begin{array}{lcl} \mathbf{S}^0 & = & \big(((1, \infty), a_0, 0), ((0, 0), b_0, 0), ((\infty, 1), c_0, 0)\big) \\ e^0 & = & (1, \mathtt{g}, 1) \\ \mathbf{S}^1 & = & \big(((2, \infty), a_0, 1), ((\infty, 3), b_1, 1), ((\infty, 1), c_0, 0)\big) \\ e^1 & = & (3, \mathtt{h}, 2) \\ \mathbf{S}^2 & = & \big(((2, \infty), a_0, 1), ((3, 3), b_0, 3), ((\infty, 4), c_0, 3)\big) \end{array}$$

  as shown in Figure 3.

4 Sequential Execution Algorithm

An algorithm for the sequential execution of a rendezvous-synchronized simulation is shown in Figure 4. This algorithm follows directly from the definitions given in Section 2.

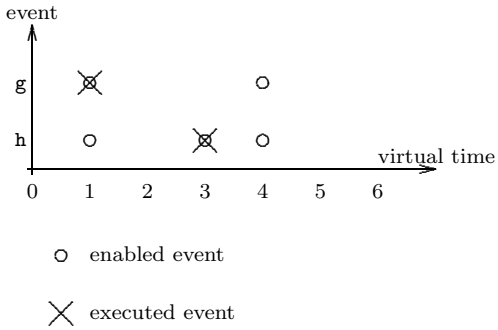

Figure 3:

During the ith iteration, event e^i is obtained by computing $\Gamma(\rho(\mathbf{S}^i))$. The algorithm terminates if the system has deadlocked (i.e., if the time of event e^i is ∞). Otherwise $\mathbf{S}^{i+1}$ is obtained by computing $\Theta(\mathbf{S}^i, e^i)$. The function Θ is completely specified by the model functions α_i and β_i, $1 \leq i \leq n$.

The algorithm in Figure 4 does not exhibit much potential for parallel execution. There is a data dependence from $\mathbf{S}^i$ to e^i and then to $\mathbf{S}^{i+1}$. Furthermore, there is a loop-carried dependence from $\mathbf{S}^{i+1}$ and e^{i+1}. Consequently, it appears that simultaneous event executions are not possible. However, there still is the parallelism which results from executing all of the processes attached to the gate e^i.

function $\Omega : (\mathbf{S}^0 \in \mathcal{S}) \mapsto (\mathcal{S} \times \mathcal{E})^+$
variables
  $e^i \in \mathcal{E} : i = 0, 1, 2, \ldots$
  $\mathbf{S}^i \in \mathcal{S} : i = 0, 1, 2, \ldots$
  $g \in \mathcal{G}$; $i \in \mathbb{Z}$; $t \in \mathcal{T}$
begin
  $i \leftarrow 0$
  loop
    $(t, g) \leftarrow \Gamma(\rho(\mathbf{S}^i))$
    exit when $t > T_{max}$
    $e^i \leftarrow (t, g)$
    $\mathbf{S}^{i+1} \leftarrow \Theta(\mathbf{S}^i, e^i)$
    $i \leftarrow i + 1$
  end loop
end
result $\{(\mathbf{S}^0, e^0), (\mathbf{S}^1, e^1), (\mathbf{S}^2, e^2) \ldots\}$

Figure 4: Sequential Execution Algorithm

5 Speculative Parallel Execution Algorithm

A sequential algorithm for computing a rendezvous-synchronized simulation was given in Section 4. The sequential algorithm does not readily admit parallel execution. In this section an algorithm for parallel execution is proposed. The approach is to use speculative computation to precompute anticipated events.

An speculative parallel execution algorithm is given in Figure 5. This algorithm differs from the sequential one in that during one iteration of the outermost loop, it is possible for several events and states to be computed—not just one. There are two phases, as indicated in Figure 5. The essential idea is that during the first phase, several event activations are computed (in parallel) leading to the tentative system state $\widehat{\mathbf{S}}$. Then, during the second phase, event activations are *committed*, and the correct system states are constructed by merging elements of $\mathbf{S}^i$ and $\widehat{\mathbf{S}}$.

Phase 1: During the first phase, the rendezvous vector $\mathbf{R}$ is determined by computing $\rho(\mathbf{S}^i)$. Next, the function $\Gamma^* : \mathcal{O} \mapsto 2^{\mathcal{G}}$ is used to compute the set A of speculative gate activations. A heuristic algorithm for computing Γ^* is given in Section 5.2. The set A *must* include the next sequential event (given by $\Gamma(\mathbf{R})$). In particular, note that if A contains exactly one event, i.e., $A = \{g : (t, g) = \Gamma(\mathbf{R})\}$, the speculative execution algorithm is semantically equivalent to the sequential algorithm.

Then the tentative system state vector $\widehat{\mathbf{S}}$ is determined by computing $\Theta^*(\mathbf{S}^i, A, \mathbf{R})$. The function $\Theta^* : \mathcal{S} \times 2^{\mathcal{G}} \times \mathcal{O} \mapsto \mathcal{S}$ applies all of the gate activations in A to the system state $\mathbf{S}^i$. The definition of Θ and an algorithm to compute it are given in Section 5.1.

Phase 2: In the second phase, events are *committed* and states are determined by merging elements from $\mathbf{S}^i$ and $\widehat{\mathbf{S}}$. The second phase is a loop—the number of iterations executed is determined by the success of the speculative computation. First, the event (t, g) is determined by computing $\Gamma(\rho(\mathbf{S}^i))$. The loop terminates if $t > T_{max}$ (either simulation is finished or the system has deadlocked) or when the event has not been computed in phase 1 ($g \notin A$ or $\mathbf{R}_{\#g} \neq t$). Otherwise, the current event is $e^i = (t, g)$ and the next state $\mathbf{S}^{i+1}$ is obtained by merging the elements of $\mathbf{S}^i$ and $\widehat{\mathbf{S}}$. The element of the state vector corresponding to process p, namely $\mathbf{S}_{(\#i)}$, is set to $\widehat{\mathbf{S}}_{(\#p)}$ if p is attached to gate g (i.e., $p \in \mathcal{G}_{(\#g)}$), otherwise it remains the same as the previous iteration, $\mathbf{S}^i_{(\#p)}$.

```
function Ω : (S^0 ∈ S) ↦ (S × E)^+
variables
    e^i ∈ E : i = 0, 1, 2, ...
    S^i ∈ S : i = 0, 1, 2, ...
    A ∈ 2^G; g ∈ G; i ∈ Z; R ∈ O; Ŝ ∈ S; t ∈ T
begin
    i ← 0
    loop
        phase 1:
        R ← ρ(S^i)
        A ← Γ*(R)
        Ŝ ← Θ*(S^i, A, R)
        phase 2:
        loop
            (t, g) ← Γ(ρ(S^i)
            exit when t > T_max or g ∉ A or R_(#g) ≠ t
            e^i ← (t, g)
            for each p ∈ P in parallel
                if p ∈ C_#g then
                    S^(i+1)_(#p) ← Ŝ_(#p)
                else
                    S^(i+1)_(#p) ← S^i_(#p)
            i ← i + 1
        end loop
        exit when t > T_max
    end loop
end
result {(S^0, e^0), (S^1, e^1), (S^2, e^2) ...}
```

Figure 5: Parallel Execution Algorithm

5.1 Speculative Execution Phase

An algorithm to compute the function $\Theta^* : \mathcal{S} \times 2^{\mathcal{G}} \times \mathcal{O} \mapsto \mathcal{S}$ is given in this Figure 6. This function applies all of the gate activations in the set A to the initial state $\mathbf{S}$ to produce the tentative system state vector $\widehat{\mathbf{S}}$. For each processor p attached to each gate $g \in A$, the appropriate model functions, $\alpha_{\#p}$ and $\beta_{\#p}$, are used to update the corresponding element of the state vector $\widehat{\mathbf{S}}_{(\#p)}$.

In order for the result of $\Theta^*(\mathbf{S}, A, \mathbf{R})$ to be well defined, it is required that the elements of the set A satisfy the following constraint:

$$\mathcal{C}_{\#g} \cap \mathcal{C}_{\#h} = \emptyset : \forall g, h \in A, g \neq h$$

In addition to ensuring that the result is well defined, the preceding constraint also permits the parallel execution of all of the enabled $\alpha_{\#p}$ and $\beta_{\#p}$ functions. It is for this reason that this phase is called the *parallel execution phase*.

```
function Θ* : (S ∈ S, A ∈ 2^G, R ∈ O) ↦ S
variables
    Ŝ ∈ S
begin
    Ŝ ← S
    for each g ∈ A in parallel
        for each p ∈ C_#g in parallel
            variables
                s, s' ∈ S_#p; t ∈ T
            begin
                t ← R_(#g)
                (·, s, ·) ← S_(#p)
                s' ← α_#p(s, (t, g))
                Ŝ_(#p) ← (β(s', t), s', t)
            end
end
result Ŝ
```

Figure 6: Speculative Execution Phase

5.2 Speculative Scheduling Function

A possible speculative scheduling function is given in this section. The particular function given in this section is based on the following two heuristics:

1. It is assumed that the rendezvous vector does not change substantially between consecutive system states.

2. It is assumed that the most likely gates to be activated are those with the smallest rendezvous times.

The scheduling function, $\Gamma^* : \mathcal{O} \mapsto 2^{\mathcal{G}}$, computes a set of gates whose activations may be computed in parallel. This set *must* include the next sequential event. I.e., $e = \Gamma(\mathbf{R}) \Longrightarrow e \in \Gamma^*(\mathbf{R})$. The function must also satisfy the constraint required by the speculative execution phase:

$$\mathcal{C}_{\#g} \cap \mathcal{C}_{\#h} = \emptyset : \forall g, h \in \Gamma^*(\mathbf{R}), g \neq h$$

A heuristic algorithm to compute Γ^* is given in Figure 7. The algorithm proceeds as follows: It begins with a copy $\mathbf{R}'$ of the rendezvous vector $\mathbf{R}$. At each iteration, it selects an event by computing $\Gamma(\mathbf{R}')$. As long as the event time is less than or equal to T_{max} and up to a maximum number of iterations I_{max}, the

gate g is added to the result set. Then, the offers in $\mathbf{R}'$ for all the gates in the set $\mathcal{H}_{\#g}$ are set to ∞. The set $\mathcal{H}_{\#g}$ is defined as the set of gates attached to the processes attached to gate g (including g itself). I.e., $\mathcal{H}_{\#g} = \bigcup_{p \in \mathcal{C}_{\#g}} \mathcal{G}_{\#p}$. This latter operation ensures that the speculative execution phase constraint that permits parallel execution is satisfied.

function $\Gamma^* : (\mathbf{R} \in \mathcal{O}) \mapsto 2^{\mathcal{G}}$
variables
  $A \in 2^{\mathcal{G}}$; $g \in \mathcal{G}$; $i \in \mathbb{Z}$; $\mathbf{R}' \in \mathcal{O}$; $t \in \mathcal{T}$
begin
  $\mathbf{R}' \leftarrow \mathbf{R}$
  $A \leftarrow \emptyset$
  $i \leftarrow 0$
  loop
    $(t, g) \leftarrow \Gamma(\mathbf{R}')$
    exit when $t > T_{max}$ **or** $i = I_{max}$
    $A \leftarrow A \cup \{g\}$
    for each $h \in \mathcal{H}_{\#g}$
      $\mathbf{R}'_{(\#h)} \leftarrow \infty$
    $i \leftarrow i + 1$
  end loop
end
result A

Figure 7: Speculative Scheduling Function

5.3 Discussion

The success of the speculative parallel execution algorithm depends on a number of factors. An underlying assumption is that the bulk of the computation is involved in computing the behaviour of the models—the model α and β functions. Thus, the primary goal is to enable execution of the model functions for as many processes in parallel as possible.

There are several ways in which the ability of the proposed approach to enable parallel execution can be measured:

Committed vs. speculative gate activations The speculative scheduling function Γ^* (Figure 7) computes a set of speculative gate activations. All of these gate activations are computed, however, not all of them can be merged into the final simulation. The ratio of the number of *committed gate activations* to the number of speculative gate activations ($|\Gamma^*(\cdot)|$) measures the fraction of gate activations that are merged into the final result.

Committed vs. speculative process activations In general, a gate may be attached to more than one process. Consequently, a gate activation may enable parallel execution. The ratio of the number of *committed process executions* to the number of speculative process executions ($|\bigcup_{g \in \Gamma^*(\cdot)} \mathcal{C}_{\#g}|$) measures the degree of useful parallelism (i.e., the parallelism efficiency).

6 Example: Closed Queueing Network Simulations—The Nicol Suite

This section presents a rendezvous-synchronized simulation model for closed stochastic queueing networks. The networks modelled are based on those described in [6]. The following is a very brief description of the queueing networks. For more detailed descriptions see [**?**, **?**, **?**].

The system to be simulated is a static network of nodes. Figure 8 shows the elements of a node. Each node in the network has one or more inputs, one or more outputs, a single queue, and a single server.

The network is populated by a fixed number of customers. A customer arrives at a node; waits for service; is serviced; and then, departs for another node. The queueing discipline is first-come, first-served (FCFS). Service is non-preemptive. The service time distribution is a geometrically distributed random variable. The output arc on which the customer departs is a uniformly distributed random variable.

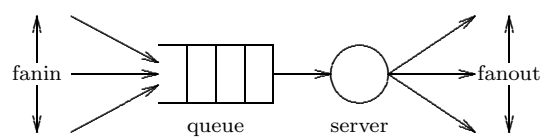

Figure 8: Queue and Server

6.1 The Node Model

This section presents the components of the node model—the elements of the state of a node, and the two functions giving the behaviour (α) and the offers vector (β) of a model. These components are presented as C++ code fragments extracted from the implementation of the simulation.

6.1.1 State

The essential elements of the state of a node are shown below in Figure 9 There are two random number generators, `serviceTimeRV` and `outputPortRV` which are used by the model to determine the service time for

a customer and the output port from which the customer departs (respectively). The variable `queue` is a *deque* (double-ended queue) of departure records. A departure record specifies the departure time and output port for each customer at the node. The element at the head of the deque corresponds to the customer currently being serviced. The remaining elements of the deque correspond to customers in the queue.

```
class Node : public LogicalProcess
{
    Time const meanServiceTime;
    unsigned short int const fanin;
    unsigned short int const fanout;
    unsigned short int const load;
    GeometricRV serviceTimeRV;
    NaryRV outputPortRV;
    Deque <Departure> queue;
public:
    void ModelAlpha (OfferVector&) const;
    void ModelBeta (PortNumber);
};
```

Figure 9: State code

6.1.2 Behaviour

Figure 10 gives the code for the function `ModelAlpha` which performs the state transition for an event on the given `port`. If the event is a customer arrival (`port` < `fanin`), the departure time and output port for that customer is enqueued. Otherwise, the event is a customer departure. In this case, the customer at the head of the queue is dequeued.

6.1.3 Offers

Figure 11 gives the code for the function `ModelBeta` which computes the offers made by a node. A node is attached to `fanin` + `fanout` gates. The first `fanin` ports correspond to customer arrivals, and the last `fanout` ports correspond to customer departures. A node always offers to accept any customer arrivals at any time greater that the current time (`time`). Therefore, the first `fanin` offers are set to `time`.

If there is a customer being serviced, an offer is made on the appropriate output port with the time of the departure. The remaining ports retain their default offer—∞.

7 Results

In this section we present results obtained from the execution of the rendezvous-synchronized simulation model of a closed stochastic queueing network given in Section 6. The network topology considered is a 64-node hypercube populated with 256 customers.

```
void Node::ModelAlpha (PortNumber port)
{
    if (port < fanin)
    {
        Time const serviceTime =
            serviceTimeRV .Sample ();
        PortNumber const outputPort =
            outputPortRV .Sample ();
        Time startTime = time;
        if (!queue .IsEmpty ())
            startTime =
                queue .TheTail () .TheTime ();
        queue .Enqueue (Departure (startTime +
            serviceTime, outputPort));
    }
    else
        queue .Dequeue ();
}
```

Figure 10: Behaviour code

```
void Node::ModelBeta (OfferVector& result) const
{
    for (PortNumber i = 0; i < fanin; ++i)
        result [i] = Offer (time);
    if (!queue .IsEmpty ())
    {
        Departure const& departure =
            queue .TheHead ();
        result [departure .ThePort ()] =
            Offer (departure .TheTime ());
    }
}
```

Figure 11: Offers code

The execution in this case was purely sequential—the purpose was to measure the potential parallelism exposed by the algorithms given in Section 5. The following parameters were measured after each iteration of the parallel execution algorithm (Figure 6): number of speculative gate activations, number of committed gate activations, number of speculative process activations, number of committed process activations, and parallelism efficiency.

7.1 Committed vs. Speculative Gate Activations

Figure 12 shows the effect of the parameter I_{max} on the number of speculative and committed gate activations. The parameter I_{max} bounds the degree of optimism in the speculative scheduling algorithm (Figure 7).

As I_{max} is increased from 1, the number of speculative gate activations increases linearly until it saturates at the mean value of 24. Below saturation, the parameter I_{max} is highly correlated with the number of speculative gates. Therefore, I_{max} can be used to control $|\Gamma^*(\cdot)|$.

The top portion of Figure 12 shows that the number of committed processes gates as the number of speculative gates increases from 1. However, the number of committed gates saturates at 5 for $I_{max} = 10$. In other words, on average speculative gate set sizes beyond size 10 do not produce any additional potential parallelism.

7.2 Committed vs. Speculative Process Activations

In the example of closed queueing networks, all of the gates are attached to exactly two processes. Therefore, the number of processes can be read from Figure 12 simply by doubling the gate numbers. I.e., the average number of committed process activations saturates at the value of 10 for $I_{max} = 10$. Note that for $I_{max} = 10$, the number of speculative process activations is 20.

7.3 Parallelism Efficiency

Figure 13 shows the parallelism efficiency vs. the parameter I_{max}. Recall that the parallelism efficiency is the fraction of the speculative parallelism is committed during each cycle of the parallel execution algorithm.

At the point where the degree of committed parallel execution saturates, $I_{max} = 10$, the parallelism efficiency is 50%. I.e., on average, 20 parallel process activations are enabled by the speculative scheduling algorithm, and 10 of those activations are committed.

The parallelism efficiency can be used to predict a bound on the maximum possible speedup that can be achieved for a particular value of I_{max}. E.g., given 20 processors, the maximum possible speedup is 10, when $I_{max} = 10$.

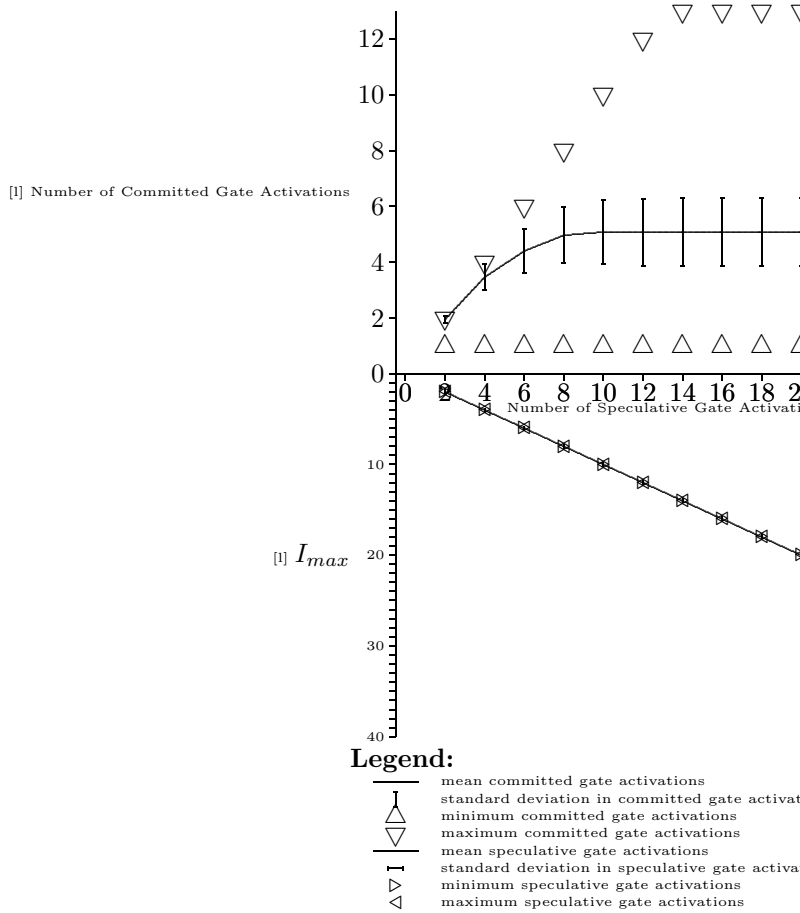

Figure 12: Committed vs. Speculative Gate Activations, hypercube.

8 Summary

This paper has presented an algorithm for the speculative parallel execution of rendezvous-synchronized simulations. The essential idea is to compute a set of non-interacting speculative gate activations; to apply all of the gate activations, in parallel, to produce a compound system state; and then to commit those gate activations which are consistent with the sequential execution of the simulation.

As an example, a closed queueing network simulation model was presented. Execution of the simulation shows that the speculative algorithm can expose a significant degree of parallelism. E.g., as many as 20 processes on average, in a 64 process hypercube network can execute simultaneously—a significant result for what is otherwise a strictly sequential paradigm.

References

[1] Q. Gao and G. V. Bochmann. A virtual ring algorithm for the distributed implementation of multi-rendezvous. Publication 675, Département

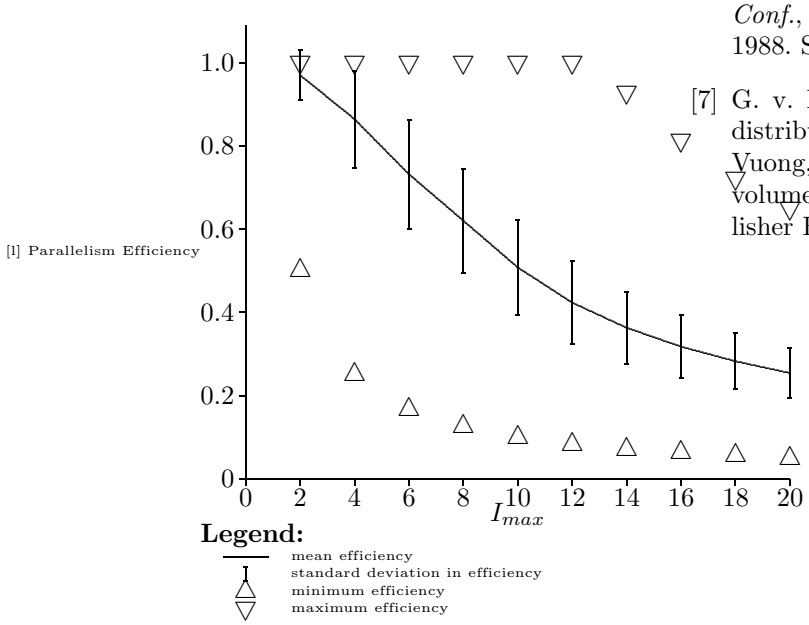

Figure 13: Parallelism Efficiency vs. I_{max}, hypercube.

d'informatique et de recherche opérationnelle, Université de Montréal, Jan. 1989.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] D. M. Nicol. High performance parallelized discrete event simulation of stochastic queueing networks. In M. A. Abrams, P. L. Haigh, and J. C. Comfort, editors, *Proc. 1988 Winter Simulation Conf.*, pages 306–314, San Diego, California, Dec. 1988. Society for Computer Simulation.

[7] G. v. Bochmann, Q. Gao, and C. Wu. On the distributed implementation of LOTOS. In S. T. Vuong, editor, *Formal Description Techniques*, volume II, pages 133–146. Elsevier Science Publisher B. V. (North-Holland), 1990.