The Reduction of the Number of Nullmessages in Conservative LP-Simulation Engines

Viviana Mascardi
D.I.S.I. - Università di Genova
via Dodecaneso 35, 16146 Genova, Italy

mascardi@disi.unige.it

November 5, 1998

Abstract

This document deals with techniques used to reduce the number of nullmessages in a distributed discrete–event simulation. Nullmessages are used to avoid deadlocks which can occurr when simulating physical systems (*networks*) consisting of one or more *physical processes*. In the *brute–force* simplest algorithm [10] the number of nullmessages is very high. Reducing the nullmessages number assures improvements to the simulation performance, and many different solutions have been proposed for this purpose.

The document is composed by a survey where these different solutions are presented followed by a comparison between them. A first section introducing the distributed discrete–event simulation general issues makes the document as more self-contained as possible.

1 Distributed discrete–event simulation

This section takes its main contibution by [10], where the distributed approach to discrete–event simulation is described in detail.

As explained in the paper, discrete–event simulation have been traditionally faced in a sequential manner. A variable *clock* holds the time up to which the physical system has been simulated. A data structure, called the *event list*, maintains a set of messages, with their associated times of transmission, that are scheduled for the future. Each of these messages is guaranteed to be sent at the associated time in the physical system, provided the sender receives no message before this message transmission time. At each step, the message with the smallest associated future time is removed from the event list, and the transmission of the corresponding message in the physical system is simulated. Sending this message may cause other messages to be sent in the future (which then are added to the event list) or cause previuosly scheduled messages to be canceled (which are removed from the event list). The clock is advanced to the time of the message transmission that was just simulated. Since the events (i.e. message transmission) in the physical system are simulated chronologically, and the simulation clock is advanced after simulation of an event to the time of the next event, this form of simulation is called *event driven*.

The algorithm for sequential event–driven simulation can not be easily adapted for concurrent execution on a number of processors, since the event list cannot be effectively partitioned for such executions.

Distributed simulation [4, 5, 2] offers a radically different approach to simulation. Shared data objects of a sequential simulation, namely the clock and the event list, are discarded: no shared variables are considered in this algorithm. In the algorithm proposed in [10] one machine may simulate a single physical process; messages in the physical system are simulated by message transmission among machines. The synchronous nature of the physical system is captured by encoding time as part of each message transmitted between machines. These machines may operate concurrently as long as their physical counterparts operate autonomously; they must wait for message receptions to simulate interactions of the corresponding physical processes. Distributed simulation offers many other advantages in addition to the possible speedup of the entire simulation process. It requires little additional memory compared with sequential simulation. There is little global control exercised by any machine. Simulation of a system can be adapted to the structure of the available hardware; for instance, if only a few machines are available for simulation, several physical processes may be simulated (sequentially) on one machine.

1.1 A model of asynchronous distributed computation

A distributed system consists of a finite number of processes and *directed channels* connecting some pairs of processes. To distinguish these processes from physical ones, they are called *logical processes* or *lp*'s, while the physical processes they simulate are called *pp*'s. Each lp may execute sequantial code and two special commands: *receive* and *send*. We assume that messages sent along a channel are delivered in the sequence in which they are sent. The implication of this *chronology* requirement is that if lp_i receives <m@t> (i.e., the message *m* sent at time *t*) from lp_j, then it knows *all* the messages that the corresponding physical process pp_i receives from the physical process pp_j up to and including time *t*, because any future message will have a higher *t* component.

The *channel clock value* of a channel is the *t* component of the last message received along the channel; the channel clock value is 0 if no message has been received along that channel.

A set of lp's is deadlocked at some point in the computation if all the following condition hold:

1. every lp in the set is either waiting to receive or is terminated;
2. at least one lp in the set is waiting to receive;
3. for any lp_i in the set which is waiting to receive from lp_j, lp_j is also in the deadlocked set and there is no message in transit from lp_j to lp_i.

It follows that none of the lp's in the deadlocked set will carry out any further computation since they will remain waiting for each other.

A logical process also has an *internal event list* EVT where internally generated events are kept and are managed when their associated timestamp becomes lower than all the channel clock values.

The *Local Virtual Time* of lp_i, LVT_i, is defined as the minimum between the channel clock values and the timestamp of the first event in EVL. An lp_i can safely simulate the corresponding pp_i up to LVT_i; that is, it can deduce every message that pp_i sends up to time LVT_i. Also, lp_i *may* be able to deduce pp_i's message transmission beyond LVT_i. In any case, lp_i will send messages it can deduce for pp_i. The basic simulation algorithm followed by lp_i is sketched next; the assumption is made that all messages are sent at *t* ¿ 0 in the physical system.

1.1.1 Basic distribution algorithm for a logical process

Firstly we need to name the components of a logical process and the operations allowed over them. Where the name of an operation is self-explaining, no further explanation is given. The notation proposed here will be mantained (and eventually extended) throughout the document, thus allowing an easier comparison between the given algorithms.

LVT is the *local virtual time* of the logical process.

LVTH is the *local virtual time horizon*, namely the minimum value of the input buffers channel clock values (min_iCC[i]).

EVL is the *event list* containing internal events. The operations allowed over EVL are *chronological_insert*, *remove_first*, *remove* (with an event as argument).

S is the *state* of the logical process; the operation *modified_by_occurence_of(e)* updates the state according to the event *e*. The *initialstate* operation returns the initial state for the logical process.

IB is an array of input buffers from which external messages are taken; IB[i] is the i-th input buffer. Over an input buffer it is possible to perform the operations *empty*, *not_empty*, *first*, *remove_first*.

OB is an array of output buffers, where messages to be delivered are put. OB[i] is the i-th output buffer. The allowed operations are *empty*, *not_empty*, *deposit* and *send_out_contents*. A meta-operation *corresponding* chooses the output buffer corresponding to a given message (namely, the output buffer on which the given message has to be deposited).

CC is an array containing the channel clock values for incoming channels; CC[i] is equal to the timestamp of the top message of IB[i], if IB[i] is not empty, otherwise it is equal to the timestamp of the last message extracted from IB[i].

ee, ie, e denote respectively external, internal and general events (internal or external).

<message@timestamp> is the form of messages; *ts(message)* returns the message timestamp.

endtime is a constant defining the time up to which the simulation must be performed.

The following algorithm describes the basic distribution algorithm in a quite informal way, using the given notation.

program basic_conservative_lp_simulation
  LVT = 0; EVL = {}; S = initialstate();
  for all CC[i] **do** CC[i] = 0 **end do**;
  for all ie_i caused by S **do** chronological_insert($< ie_i@occurence\_time(ie_i) >$, EVL) **end do**;
  while LVT $\leq$ endtime **do**
    for all IB[i] **do await** not_empty(IB[i]) **end do**;
    for all CC[i] **do** CC[i] = ts(first(IB[i])) **end do**;
    LVTH = min_iCC[i];
    min_channel_index = i such that CC[i] == LVTH;
    if ts(first(EVL)) $\leq$ LVTH
      then e = remove_first(EVL);
      else e = remove_first(IB[min_channel_index]);
    bf end if;
    LVT = ts(e);
    S = modified_by_occurence_of(e);
    for all ie_i caused by S **do** chronological_insert($< ie_i@occurence\_time(ie_i) >$, EVL) **end do**;
    for all ie_i preempted by S **do** remove(ie_i, EVL) **end do**;
    for all ee_i caused by S **do** deposit($< ee_i@LVT >$, corresponding(OB[j])) **end do**;
    for all OB[i] **do** send_out_contents(OB[i]) **end do**;
  end do while

It is quite easy to understand that this scheme can lead to deadlocks: if logical process lp_1 is blocked on the *await* statement, waiting for a message coming from lp_2, and lp_2 is either waiting for a message coming from lp_1, a deadlock occurs since neither lp_1 nor lp_2 can go on.

1.2 A basic scheme for avoiding deadlocks

A basic scheme for avoiding deadlocks is the following: when an lp has terminated of managing a message, it checks its output queues; if an output queue contains a real message, no synchronization message is added. Otherwise, if the output queue is empty, a message of the form $< null@new\_timestamp >$ (a so-called *nullmessage*) is posted on the output queue.

A nullmessage is not related to the simulated model and only has synchronization purposes; it is sent on any empty output channel as a promise not to send any other message with smaller timestamp in the future. The receiver of the nullmessage can use the nullmessage timestamp to extend its LVTH and so become unblocked; the nullmessage has only this purpose, and it does not require any management.

The *new_timestamp* at which the nullmessage is sent is evaluated by adding to the lp LVT a *lookahead* value. The *lookahead* value gives a degree of how much the lp can predict future events. For example, if an lp knows that processing the next event will certainly increment LVT by t, then nullmessages $<$

$null@(LVT + t) >$ can be sent over empty output channels as further improvement of the LVTH of the receivers.

Lookahead must come directly from the underlying simulation model and enhances the prediction of future events, which is necessary to determine when it is safe to process an event.

In the following algorithm we assume that a function *lookahead()* returns the lookahead for the lp, according to its simulation model and its state.

The algorithm is known as "Chandy–Misra–Bryant (CMB) algorithm".

program cmb_conservative_lp_simulation
    LVT = 0; EVL = {}; S = initialstate();
    for all CC[i] **do** CC[i] = 0 **end do**;
    for all ie_i caused by S **do** chronological_insert($< ie_i@occurence\_time(ie_i) >$, EVL) **end do**;
    while LVT $\leq$ endtime **do**
        for all IB[i] **do await** not_empty(IB[i]) **end do**;
        for all CC[i] **do** CC[i] = ts(first(IB[i])) **end do**;
        LVTH = min_iCC[i];
        min_channel_index = i such that CC[i] == LVTH;
        if ts(first(EVL)) $\leq$ LVTH
            then e = remove_first(EVL);
            else e = remove_first(IB[min_channel_index]);
        end if;
        LVT = ts(e);
        if not nullmessage(e) **then**
            S = modified_by_occurence_of(e);
            for all ie_i caused by S **do** chronological_insert($< ie_i@occurence\_time(ie_i) >$, EVL) **end do**;
            for all ie_i preempted by S **do** remove(ie_i, EVL) **end do**;
            for all ee_i caused by S **do** deposit($< ee_i@LVT >$, corresponding(OB[j])) **end do**;
        end if;
        for all empty(OB[i]) **do** deposit($< null@(LVT + lookahead()) >$, OB[i]) **end do**;
        for all OB[i] **do** send_out_contents(OB[i]) **end do**;
    end do while

Using the nullmessages deadlock avoidance strategy the simulator never deadlocks; if the physical system deadlocks, the simulator continues the computation by transmitting nullmessages with increasing timestamp values. This correctly simulates the corresponding physical situation, in that, while time progresses, no messages are transmitted in the physical system.

The simplicity of this scheme is one of its most attractive points, and empirical studies have shown that this scheme is quite efficient for acyclic networks [Seethalakshmi1979]. The simple scheme can be slightly modified by adding time-out mechanism to prevent nullmessage transmission; since a nullmessage has no effect if it is followed by another message, it may be efficient to delay transmission of nullmessages in the hope that future messages received by an lp would make it unnecessary to transmit them at all. The amount of time τ that an lp waits before transmitting a nullmessage is of importance. If $\tau = 0$, the algorithm is the one described in this section. If $\tau = \infty$, nullmessages are never transmitted and the algorithm is the *basic_conservative_lp_simulation* presented in last section, which can lead to deadlock.

The main problem with the nullmessage approach is that it generates an enormous amount of message traffic. The following section describes different approaches for reducing the number of nullmessages, and a comparison between them is carried out in Section 3.

2 Some approaches for reducing the nullmessages number

In this section different techniques for reducing the number of synchronization messages are presented. We start by describing a class of *demand driven* techniques, in which a process requests time information from its predecessor before advancing its time. The first algorithm [10] detects and recovers deadlocks: it has the disvantage that the size of messages grows in length with the size of the connection graph and that deadlock recovery requires additional information to be mantained in messages, which can become too complex and large for an efficient use. The second proposed *demand driven* algorithm [1] is deadlock-free; it allows small, fixed size messages to be exchanged, even if it does not guarantee that one request for time information be sufficient to advance a process's time.

The second class of described algorithms involves the *carrier null message* approach by W. Cai and S. Turner [3]. This approach employs "carrier-nullmessages" for both advancing simulation times and acquiring global knowledge of the simulation network in order to improve the lookahead ability based on it.

2.1 The demand driven approach

Misra's algorithm

In the query-reply scheme two types of synchronization messages are considered:

$< \mathbf{query}([(\mathbf{p}, \mathbf{cc_p}, \mathbf{ne_p}), \ldots, (\mathbf{y}, \mathbf{cc_y}, \mathbf{ne_y}), (\mathbf{z}, \mathbf{cc_z}, \mathbf{ne_z})])@\mathbf{ts} >$ is sent by lp **y** to lp **z** to demand if **z** can advance the **(z,y)** channel clock value of lp **y**. Such a message is sent when lp **y** has the **(z,y)** channel empty, and the channel clock value of such a channel is equal to **y**'s local virtual time. This means that, until **y** does not receive a message with higher timestamp over **(z,y)** channel, it cannot advance its local virtual time. In this case **y** sends a *query* to **z**. When an lp **z** receives a *query* two conditions can be verified: or the lp can advance the channel clock (its LVT is higher than cc_z) or it cannot (since its LVT is equal to cc_z). In the first case, **z** sends a *reply* to **y** including the query list and the new channel clock; in the latter case it forwards a query to all lp's whose channel clock value is equal to **z**'s LVT. This strategy can lead to deadlocks, thus the query list must contains the path it has traversed to detect deadlocks: when lp **z** receives a query, and appears more then once in the query list, a deadlock is detected. Deadlock recovery is quite difficult, and uses the ne_x field as explained below.

$< \mathbf{reply}(\mathbf{query}([(\mathbf{p}, \mathbf{cc_p}, \mathbf{ne_p}), \ldots, (\mathbf{y}, \mathbf{cc_y}, \mathbf{ne_y}), (\mathbf{z}, \mathbf{cc_z}, \mathbf{ne_z})]))@\mathbf{ts} >$ is sent by a process which receives a *query*, and can advance the channel clock of the query sender. When an lp receives a *reply* it must update the channel clock value to the *reply* timestamp value, and must forward the reply to the lp (if any) which requested a channel clock advance.
The ne_x fields in the query list components is the "time of next event" for lp **x**: it is evaluated as the timestamp of the first event which should be processed if **x** shouldn't receive any message with lower timestamp, and it is used for recovering from deadlocks. When a deadlock is detected, the lp with minimum "time of next event" is forced to advance its local virtual time to its next event time.
Over $< query@ts >$ and $< reply(query)@ts >$ events the *synch_event(event)* boolean function returns *TRUE*. A special kind of event, the *empty_event*, is used in the algorithm for making simpler the execution flow. The boolean function *empty_event(e)* returns true if and only if e is the empty event.
Now we describe Misra's algorithm using the homogeneous style as for the previous algorithms. The algorithm is not exactly the same as described in [10] since here a distributed deadlock recovery mechanism is explicitly inserted in the algorithm, while in Misra's paper it was only sketched. We need some more constants and data structures for implementing the algorithm:

<awake@ts> is a variable which, when enabled, causes the lp to advance its LVT to ts. It can be enabled by any other lp in the system by calling a *enable_awake(lp_to_awake, ts)* procedure. *disable_awake(lp_to_awake)* disables this variable. The *awake_enabled()* boolean functions returns true if the variable is enabled, and the *awake_ts()* function returns the timestamp value of the awake variable.

timeout is the amount of time an lp **y** is willing to wait for an empty input channel to fill with a message. After the timeout, the channel (z, y) is considered empty and this may cause the dispatching of a *query* to **z**.

SENT_QUERIES is the set of already sent queries for an lp. It is necessary for avoiding to send the same query more than once. The $reset\_sent\_queries([(p, cc_p, ne_p), \ldots, (y, cc_y, ne_y), (z, cc_z, ne_z)])$ procedure, called by whatever lp, allows to dispose the elements in the SENT_QUERIES set related to the lp's in the deadlocked sub-system: when a deadlock is detected and recovered, the previously sent queries have to be forgotten. Other common operations over sets are allowed on SENT_QUERIES.

The core of the algorithm, which shares many instructions with the previoulsy given ones, describes the basic demand driven mechanism.

```
program basic_demand_driven_simulation
    LVT = 0; EVL = {};
    S = initialstate(); SENT_QUERIES = {};
    for all CC[i] do CC[i] = 0 end do;
    for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
    while LVT ≤ endtime do
        if awake_enabled()
            then do LVT = awake_ts(); disable_awake() end do;
        end if
        for all IB[i] do await(timeout) not_empty(IB[i]) end do;
        for all CC[i] do
            if not_empty(IB[i])
                then CC[i] = ts(first(IB[i]))
            end if
        end do;
        LVTH = min_i CC[i];
        min_channel_index = i such that CC[i] == LVTH;
        if ts(first(EVL)) ≤ LVTH
            then e = remove_first(EVL);
            else if not_empty(IB[min_channel_index])
                then e = remove_first(IB[min_channel_index]);
                else do
                query_manage(query([(id, LVT, DEFAULT_VALUE)]));
                e = empty_event end do;
            end if
        end if;
        if ((not empty_event(e)) and (not synch_event(e)))
            then do
            LVT = ts(e);
            S = modified_by_occurence_of(e);
            for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
            for all ie_i preempted by S do remove(ie_i, EVL) end do;
            for all ee_i caused by S do deposit(< ee_i@LVT >, corresponding(OB[j])) end do;
            end do
        else if (synch_event(e))
            then do LVT = ts(e); synch_msg_management(e); end do
            end if
        end if;
        for all OB[i] do send_out_contents(OB[i]) end do;
    end do while
```

Now we define the *synch_msg_management(event)* procedure, with all its auxiliary procedures (included *query_manage(query(list))* already used in the algorithm).

```
procedure synch_msg_management(event e)
    case content(e)
        is equal to query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)])
            do query_manage(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)])) end do
        is equal to reply(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]))
```

```
            do SENT_QUERIES = SENT_QUERIES \query([(p, cc_p, ne_p), ..., (z, cc_z, ne_z)]);
        query_manage(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y)]) end do
    end case
end procedure

procedure query_manage(query(list))
    new_list = correct_next_event_time(list);
    if can_advance_channel_clock(query(new_list)
        then send_reply(new_list)
        else send_queries(new_list)
    end if
end procedure
```

correct_next_event_time(list) simply evaluates the minimum next_event_time for the lp (suppose its identifier is z), and substitutes it to ne_z in the tuple (z, cc_z, ne_z). It is necessary because, when an lp sends a query to another lp, it cannot know the receiver next_event_time value and so it must put it to a default value DEFAULT_VALUE. When the receiver gets the query, it must firstly correct this value to its real next_event_time value.

```
function can_advance_channel_clock(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]))
returns boolean
    if LVT > cc_z
        then return true
        else return false
    end if
end function

procedure send_reply(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]))
    do deposit(
        <reply(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]))@LVT>,
        OB[lp_index_corresponding_to(y)]) end do
end procedure

procedure send_queries(query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]))
    for all OB[i] such that CC[i]==LVT do
        x = lp_identifier_corresponding_to(OB[i]);
        if x appears in [(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]
            then deadlock_recovery([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)])
            else if [(p, cc_p, ne_p), ..., (z, cc_z, ne_z), (x, LVT, DEFAULT_VALUE)] ∉ SENT_QUERIES
                then deposit(
                    < query([(p, cc_p, ne_p), ..., (y, cc_y, ne_y),
                    (z, cc_z, ne_z), (x, LVT, DEFAULT_VALUE)]))@LVT >,
                    OB[i]) end do
end procedure
```

Now we finally describe the deadlock recovery mechanism: the idea is awakening the deadlocked lp with the minimum next event time. Awakening an lp, informing it about the exact awakening time, causes it to advance its local virtual time to the awakening time (as described in the basic_demand_driven_simulation core program). The SENT_QUERIES set of the deadlocked processes must be updated by disposing the queries concerning the deadlocked processes.

```
procedure deadlock_recovery([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)])
    lp_to_awake = lp such that ne_lp = min {ne_p, ..., ne_y, ne_z};
    reset_sent_queries([(p, cc_p, ne_p), ..., (y, cc_y, ne_y), (z, cc_z, ne_z)]);
    awake_enabled(lp_to_awake)
end procedure
```

2.2 Bain and Scott's algorithm

As already noticed, Misra's algorithm is not deadlock free (the deadlock is detected and then recovered), and the size of messages increases with the length of traversed paths. Bain and Scott propose a demand–driven based algortihm which overcomes the limitations of Misra's one.

As usual, before a process can consume and process a message, it must assure itself that no message with timestamp less than the message timestamp will arrive from any other predecessor. It thus guarantees that it will never process messages out of order, which is the essential constraint of time progressive synchronization.

If all the predecessor time exceed the message timestamp ts_m, namely if all CC[i] are greater or equal than ts_m, the event message can be consumed immediately. Otherwise the process creates an entry, called a *time request*, containing the process's id and requested time , and places it into a queue, called the *request queue*. This queue is ordered by request times. The process must then request the time from all predecessors whose channel time is less then ts_m. Note that these predecessors may have advanced their time without the process's knowledge. In the proposed algorithm, a time request message includes a unique process id q from the originating process and the requested time ts_m. The complexity of the algorithm arise from the effort to avoid repeated polling in the presence of cycles within the connection graph and the desire to avoid requiring message lengths to grow with the size of the connection graph. In the algorithm multiple requests may be needed to receive the desired reply. In particular there are the following three reply types:

YES which indicates that the predecessor channel has reached the requested time,

NO which indicates that the channel has not reached the requested time and another request must be made, and

RYES (*reflected yes*) which indicates that the channel has conditionally reached the requested time (as explained below).

As an optimization, the algorithm passes forward with a YES reply the actual time of the predecessor should it exceed the requested time. The successor uses this time and thus avoids subsequent requests in some cases.

The use of a reflected yes reply message is a novel feature of the algorithm. It purpose is to detect cycles within the connection graph and thereby avoid deadlock in time synchronization. It allows us to avoid passing path information within messages and to minimize the number of time requests sent to predecessors.

When a (predecessor) process receives a time request from one of its successors it handles the request in one of three ways. First, if the originating process identifier and request time match any request already in the request queue, the process determines that a cycle has been traversed in the graph. It replies to the requesting successor with type RYES, indicating that a cycle was encountered. Otherwise, if the process's time equals or exceeds the requested time, the process sends a reply to the requesting successor of type YES, indicating this fact and including the process's current time. Finally, if the process's time has not reached the requested time, the request is enqueued in the request queue. The process then recursively makes time requests to all of *its* predecessor with lower channel times than the requested time. In order to detect cycles in the connection graph, it places the originating process id in the time request message instead of its own id.

Within each enqueued request is a boolean array **RY**[k] for all successors $k = 1, \ldots, s_i$. All values are set to FALSE when the request is created. An additional boolean for each request **NI_REQD** is also initially set to FALSE. Now, whenever a RYES reply is sent, the corresponding boolean RY[k] for the requesting successor is set to TRUE in the enqueued request. If the process will subsequently send an event message to this successor at a time lower than the requested time, the NO_REQD boolean will be set to TRUE; the boolean is used when a final reply to the request is sent, as described below. This state information is needed to assure that the detection of cycles does not cause incorrect time information to be sent. That is, if a lower time request is sent towards the originating process behind a reflected yes reply, the NO_REQD boolean assures that the process will learn of this by means of a NO reply along another path.

When a (successor) process receives a reply message to a time request, it notes that the predecessor's channel has responded, and if a reply message of type YES is received, the channel time is updated. When all channels have responded to the request and it reaches the front of the request queue, the request is complete. At this point, if this process is *not* the originating process, a reply message is sent to the successor from which the time request was received. The reply type is determined as follows: type NO is sent if a NO was received from any predecessor or the boolean NO_REQD is TRUE; otherwise, type RYES is sent if a RYES was received from any predecessor; otherwise, YES is sent along with the process's current time, which is first updated to the requested time should the requested time be lower. This is optimized slightly by converting a RYES reply to a YES reply if the process's time has reached the requested time before the reply is sent. The request is then deleted from the request queue. In the algorithm there is the assumption that the handling of a message and the resulting update of the process's time is an atomic action with respect to the handling of time requests and replies.

If this process is the originating process for the request, the process does not send out a reply. Instead, it determines from the incoming replies whether the pending message is ready for consumption. If no replies of type NO were received and the NO_REQD boolean is FALSE, the process may proceed to consume the event message when it reaches the front of the request queue. Other time requests with lower times have arrived pending completion of the request. If a reply of type NO was received, it must restart the time requests to all predecessors with lower channel times.

Now that we have described the algorithm, we can try to better formalize it by individuating the needed data structures, besides the basic ones, and by writing down the algorithm in the already previously used preudo-code.

The only (but quite complex) added data structure is:

REQUEST_QUEUE , which is a queue of requests ordered by request times. Associated to each item in the queue there is a boolean array **RY**[k] (initially each entry is set to FALSE), with k which ranges over the process's successors, $k = 1, \ldots, s_i$, and a boolean **NO_REQD**, initially set to FALSE too. Moreover a structure, which we do not detail here, must be associated to each request for keeping track of the replies arrived to it. When a process cannot manage an event because it needs to advance some channel clocks, or when it receives a time request, it creates a new entry in the REQUEST_QUEUE. This entry is initially "unstarted" since it hasn't yet been managed. Only when all replies to a request have arrived, this request is "completed". So, it makes sense testing if the front of REQUEST_QUEUE is "unstarted" or "completed". Over the REQUEST_QUEUE, besides the *front* operation which returns the front of the queue (without dequeueing it), also *enqueue* and *dequeue* operations are allowed. Finally note that "create an entry in the REQUEST_QUEUE" implies "and properly initialize the auxiliary structures for that entry".

```
program advanced_demand_driven_simulation
    LVT = 0; EVL = {};
    S = initialstate(); REQUEST_QUEUE = {};
    for all CC[i] do CC[i] = 0 end do;
    for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
    while LVT ≤ endtime do
        for all IB[i] do await(timeout) not_empty(IB[i]) end do;
        for all CC[i] do
            if not_empty(IB[i])
                then CC[i] = ts(first(IB[i]))
            end if
        end do;
        LVTH = min_i CC[i];
        min_channel_index = i such that CC[i] == LVTH;
        if ts(first(EVL)) ≤ LVTH
            then e = remove_first(EVL);
            else if not_empty(IB[min_channel_index])
                then e = remove_first(IB[min_channel_index]);
                else e = empty_event;
            end if
        end if;
```

```
      if ((not empty_event(e)) and (not synch_event(e)))
          then do
          LVT = ts(e);
          S = modified_by_occurence_of(e);
          for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
          for all ie_i preempted by S do remove(ie_i, EVL) end do;
          for all ee_i caused by S do deposit(< ee_i@LVT >, corresponding(OB[j])) end do;
          end do
      else if (empty_event(e))
          then do create a new entry in REQUEST_QUEUE with requested time equal to LVTH;
          manage_request_queue; end do
          else if (synch_event(e))
              then do LVT = ts(e); synch_msg_management(e); end do
              end if
          end if;
      end if;
      for all OB[i] do send_out_contents(OB[i]) end do;
  end do while


procedure manage_request_queue
  if front of REQUEST_QUEUE is unstarted
      then if LVT < requested time of REQUEST_QUEUE front
          then for all OB[i] such that CC[i]==LVT do
              deposit(< time_request@LVT >, OB[i]) end do
          else deposit(< reply_YES@LVT >, OB[index_corresponding_to_requestor]) end if
      else [the front of REQUEST_QUEUE is complete]
      if the request is mine
          then if any NO replies received or NO_REQD
              then restart request;
              else do dequeue the request;
              consume event message;
              if a new event message is sent on channel j at time T
                  then mark all requests with time ts > T
                  which have replied RYES on channel j as NO_REQD;
                  end if
              end do
          end if
          else [the request is not mine]
              do dequeue the request;
              if any NO replies received or NO_REQD
                  then deposit(< reply_NO@LVT >, OB[index_corresponding_to_requestor])
                  else if any RYES replies received
                      then deposit(< reply_RYES@LVT >, OB[index_corresponding_to_requestor])
                      else deposit(< reply_YES@LVT >, OB[index_corresponding_to_requestor]);
                  end if
              end if
              end do
          end if [if the request is mine]
      end if [if the front is unstarted]
end procedure

procedure synch_msg_management(e)
  case content(e)
      is equal to time_request
```

```
        do if the request is already in the REQUEST_QUEUE
           then for that REQUEST_QUEUE entry, and
           for the channel k corresponding to the request sender
              do RY[k] = TRUE;
              deposit(< reply_RYES@LVT >, OB[k]) end do
           else if LVT ¡ requested time of time_request
              then create a new entry in the REQUEST_QUEUE
              else deposit(< reply_YES@LVT >, OB[index_corresponding_to_requestor])
              end if
           end if
        end do
     is equal to reply
        do record the reply properly in the REQUEST_QUEUE end do
  end case
end procedure
```

2.3 The carrier-nullmessage approach

The carrier null message algorithm exploits the basic algorithm proposed in [10], as well as a novel synchronization message, the *carrier-nullmessage*, in avoidance of deadlocks during the simulation. Lookahead ability is increased, where possible, by exploring the topology of the simulation network using carrier-nullmessages.

A carrier-nullmessage $< (route.inf, la.inf, carrier.null)@t >$ is used to annouce the absence of real messages in a physical system at time t (in the same way as CMB nullmessages. t is the timestamp of this message, route.inf is route-information and la.inf is the lookahead-information.

As a nullmessage, a carrier-nullmessage is a synchronization message. Deletion of them will not cause any changes in the simulation behaviour. Therefore, a carrier-nullmessage $< (route.inf', la.inf', carrier.null)@t >$, can preempt any previous nullmessage, and the carrier-nullmessage with route.inf equal to route.inf', in input and output queues, and in the EVL. Besides carrier-nullmessages, the algorithm also uses "classical" nullmessages, denoted as usual with $< null@t >$.

In the following we separately describe the *Simulation step*, namely what does it happen when a message is managed (we concentrate in the management of synchronization messages), and the *Output step*, namely how and when synchronization messages have to be put into output queues, for avoiding deadlocks.

Note that the core of the algorithm, described below, is slightly different from the basic Misra's algoritm of Section 1.2. Here, in fact, at any computation cycle all the messages with timestamp equal to the LVTH are managed. The set of messages which are going to be managed is identified by $\{M_k\}, k = 1, \ldots, messages\_to\_manage\_number$. In Misra's algorith, instead, only one message was handled at any computation cycle. The inspection of the event list EVL is postponed, in the original paper [3], after the inspection of input buffers and the corresponding simulation step.

Here, for maintaining a certain uniformity with the basic algorithm of Section 1.2, we inspect the EVL before the *Simulation step*. The resulting algorithm is a bit different from the one originally proposed in [3], but it is more similar to the one referred to throughout this document.

Respecting to the basic algorithm, we need to add some more data structures:

CC_OUT is an array containing the channel clock values for outgoing channels; CC_OUT[i] is equal to the timestamp of the last message put in OB[i]. CC_OUT has, for output channels, a function equal to the CC one for input channels.

NEXT is an array, local to the simulation step computation, containing the arrival time of the next possible real message for any input channel.

2.3.1 Simulation step

The processing of synchronization messages in the algorithm is described as follows. We define Next as the set of links on which the input operations were carried out:
$Next = \{i \mid CC[i] = OLD\_LVTH\}$

where OLD_LVTH is the LVTH evaluated during the previous simulation cycle for that lp. A further division is made into three subsets $S1, S2, S3$ based on the message M_k removed from the input queue IB[k].
$S1 = \{k \mid k \in NextandM_k =< (route.inf, la.inf, carrier.null)@t >\}$
$S2 = \{k \mid k \in NextandM_k =< null@t >\}$
$S3 = \{k \mid k \in NextandM_k =< real\_message@t >\}$
Moreover $route.inf$ partitions $S1$ into mutually disjoint subsets $R_1, \ldots, R_q$ that is,
$S1 = R_1 \cup R_2 \cup \ldots \cup R_q, R_i \cap R_j = \emptyset (1 \leq i, j \leq q, i \neq j)$
$R_j = \{k \mid k \in S1$ and M_k has the same $route.inf$ element, $route.inf_j\}$

We treat all carrier-nullmessages in the same R_j as one synchronization message with $route.inf$ equal to $route.inf_j$ and $la.inf$ equal to $max_k\{M_k.la.inf, k \in R_j\}$ since only the most updated $la.inf$ is useful.

Therefore, messages removed from the input queues of links in $R_j, (1 \leq j \leq q)$, or $S2$ may require special treatment. The condition for generating or passing through a carrier-nullmessage is that the arrival time of the next possible real message with respect to a processed synchronization message has to be greater than the minimum value of all input channel clocks LVTH. If a synchronization message is removed from IB[h] then this arrival time is referred to as NEXT[h] which specifies the arrival time of the next possible real message from the input queues other than IB[h]. Therefore it gives an initial estimation of the possible improvement on lookahead. It is calculated as follows:
If $h \in R_m$
NEXT[m] = LVTH for $\{k \mid k \notin R_m\} = \emptyset$;
$min_k\{CC[k], k \notin R_m\}$ otherwise *equation (6)*
If $h \in S2$
NEXT[h] = LVTH for $\{k \mid k \neq h\} = \emptyset$;
$min_k\{CC[k], k \neq h\}$ otherwise *equation (7)*

In summary, an lp can only generate or pass through a carrier-nullmessage in the following cases:

1. Processing a nullmessage M_h from the input channel $h \in S2$, and the NEXT[h] with respect to this message is greater than LVTH. From equation (7) it is easy to deduce that
$NEXT[h] > LVTH \Rightarrow (CC[h] == LVTH)$ and (number of input channel > 1)

2. Processing a group of carrier-nullmessages with the same $route.inf$, i.e.,
$(R_m \subset S1)$ and $(\mid R_m \mid \geq 1)$, and the NEXT[m] with respect to this group (see equation (6)) is greater than LVTH. Similarly, from equation (6), we have:
$NEXT[m] > LVTH \Rightarrow$
(if(CC[k] == LVTH) then $k \in R_m$) and ($\mid R_m \mid <$ number of input links)
In this case, $la.inf$ of the carrier-nullmessage is changed to:
$la.inf = min(NEXT[m], max_k\{M_k.la.inf, k \in R_m\})$ since $la.inf$ is exploited to reflect the earliest timestamp of the real message which may break the waiting dependency loop.

3. Processing carrier-nullmessages with the same $route.inf$ from all input queues. In this case, the carrier-nullmessage is modified so that its $la.inf$ is equal to the highest lookahead information in all received carrier-nullmessages. Whereas in case 2, the calculation of $la.inf$ also depends on NEXT[m].
Suppose $R_1 = S1$, then we have:
$la.inf = max_k\{M_k.la.inf, k \in R_1\})$ and
if $la.inf$ is modified $\Rightarrow \mid R_1 \mid > 1$

A carrier-nullmessage is generated in case 1, whereas the carrier-nullmessage is modified and passed through in cases 2 and 3. The generated or modified carrier-nullmessage is then put into the event list, which may incur further changes and will be timestamnped when picked up in the *Output step*.

2.3.2 Output step

In this step, the possible messages which should be placed into the output queues are determined. If a real message can be picked up from the event pool, then, as in CMB algorithm, $< m@t >$ is placed into the output queue of channel j (OB[j]). Otherwise, there are two cases:

1. A carrier-nullmessage $< (route.inf, la.inf, carrier.null)@0 >$ is found in the event list EVL. There are three sub-cases:

   (a) if $la.inf$ is less or equal to $(LVTH + lookahead())$ it is not necessary to carry on sending this carrier-nullmessage, since the lookahead will not be improved by it anyway. Instead, a normal nullmessage with timestamp equal to $(LVTH + lookahead())$ may be placed in OB[j];

   (b) if $la.inf$ is greater than $(LVTH + lookahead())$ and lp discovers the *waiting dependency* by finding its identifier in the $route.inf$, a prediction on future behaviour will be made, that is, if the corresponding channel clock CC_OUT[j] is less than $la.inf$, lp will place a normal nullmessage with timestamp equal to $la.inf$ (instead of $(LVTH + lookahead())$) into the output queue;

   (c) if $la.inf$ is greater than $(LVTH + lookahead())$, and lp is not in the route.inf, the carrier-nullmessage may be placed into OB[j] with timestamp equal to $(LVTH + lookahead())$. In addition the $route.inf$ on this carrier-nullmessage will be changed to $route.inf \cup lp\_identifier$, if lp has more than one input channel.

2. There is no carrier-nullmessage in the event list EVL. In this case, depending on the value of CC_OUT[j], a normal nullmessage may be simply placed into output queues with timestamp equal to $(LVTH + lookahead())$, indicating that there are no real messages transmitted in the time interval between CC_OUT[j] and $(LVTH + lookahead())$.

In case (b) the lp sends out $< null@la.inf >$, the lookahead ability will be improved since now the lp can predict the absence of real messages up to the time $la.inf$ which is larger than $(LVTH+lookahead())$. The transmission of $< null@la.inf >$ will not violate the local causality constraint because $la.inf$ represents the earliest timestamp of the real message which may break the waiting dependency loop. It is also easy to realize that the carrier-nullmessage is generated and propagated whenever it is necessary and it will disappear and be replaced by the nullmessage when it looses its meaning (see cases (a) and (b)).

Now we summarize what described in natural language, using the homogeneous style already used in the previous sections.

```
program carrier_nullmessage_simulation
    LVT = 0; LVTH = 0; EVL = {};
    S = initialstate();
    for all CC[i] do CC[i] = 0 end do;
    for all CC_OUT[j] do CC_OUT[j] = 0 end do;
    for all NEXT[k] do NEXT[k] = 0 end do;
    for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
    while LVT ≤ endtime do
        for all IB[i] do await() not_empty(IB[i]) end do;
        for all CC[i] do CC[i] = ts(first(IB[i])) end do;
        OLD_LVTH = LVTH;
        if not is_empty(EVL)
            then LVTH = min(ts(first(EVL)), min_i CC[i]);
            else LVTH = min_i CC[i];
            end if
            LVT = LVTH;
        min_channel_indexes = {i | CC[i] == LVTH};
        if (not is_empty(EVL)) and (ts(first(EVL)) == LVTH)
            then do int_M = remove_first(EVL);
            M = {M_k = remove_first(IB[i]) | i ∈ min_channel_indexes} end do
            else M = {M_k = remove_first(IB[i]) | i ∈ min_channel_indexes} ∪ {int_M};
        end if;
        for all M_k ∈ M | (notsynch_event(M_k)) do
            S = modified_by_occurence_of(M_k);
            for all ie_i caused by S do chronological_insert(< ie_i@occurence_time(ie_i) >, EVL) end do;
            for all ie_i preempted by S do remove(ie_i, EVL) end do;
```

```
            for all ee_i caused by S do deposit(< ee_i@LVT >, corresponding(OB[j])) end do;
            end do
            synch_msg_management;
            put_synchronization_messages_in_output_queues;
            for all OB[i] do send_out_contents(OB[i]) end do;
    end do while


procedure synch_msg_management
    Next = {i | CC[i] = OLD_LVTH}
        for all i ∈ Next do evaluate NEXT[i] end do
        case
            (∃h ∈ S2) and (NEXT[h] > LVTH)
                do enqueue < (∅, NEXT[h], carrier.null)@0 > in EVL end do
            (∃R_m ⊂ S1) and (| R_m |≥ 1) and (NEXT[m] > LVTH)
                do la.inf = min(NEXT[m], max_k{M_k.la.inf, k ∈ R_m});
                enqueue < (route.inf_m, la.inf, carrier.null)@0 > in EVL bf end do
            (R_1 = S1) and ({k | k ∉ R_1} = ∅)
                do enqueue < (route.inf_1, max_k{M_k.la.inf, k ∈ R_1}, carrier.null)@0 > in EVL end do
            otherwise skip
    end case
end procedure

procedure put_synchronization_messages_in_output_queues
    for all OB[j] such that is_empty(OB[j]) do
        if < (route.inf, la.inf, carrier.null)@0 >∈ EVL
            then case
                la.inf ≤ (LVTH + lookahead())
                    do deposit(< null@(LVTH + lookahead()) >, OB[j]) end do
                (la.inf > (LVTH + lookahead())) and (lp_identifier ∈ route.inf)
                    do deposit(< null@la.inf >, OB[j]) end do
                (la.inf > (LVTH + lookahead())) and (lp_identifier ∉ route.inf)
                    do if number_of_input_links > 1
                        then do route.inf = route.inf ∪ lp_identifier;
                        deposit(< (route.inf, la.inf, carrier.null)@(LVTH + lookahead()) >, OB[j]) end do
                        else deposit(< (route.inf, la.inf, carrier.null)@(LVTH + lookahead()) >, OB[j])
                    end if
                    end do
                end case
            else deposit(< null@(LVTH + lookahead()) >, OB[j])
        end if
    end do
end procedure
```

2.4 Some other approaches

2.4.1 Time-of-next-event

In [7] the authors focus upon a particular conservative algorithm for parallel simulation, the Time of Next Event (TNE) algorithm. This algorithm computes, in each individual processor, the greatest lower bound on the time stamps of event messages at each empty link. An empty link is a link which contains no messages. A link time corresponds to the time-of-next-event (TNE), and nodes correspond to lp's. The necessary and sufficient condition for a deadlock to exist is:

> A deadlock exists if and only if there is a cycle consisting of empty links which all have the same link time, and nodes which are all blocked because of these links

It is important to note that the lp's involved in the cycle may have no other empty input links with the same link time as those links included in that cycle. The TNE algorithm explores the directed graph of lp's connected by empty links. In is executed independently in all of the logical processors, avoiding higher communication costs associated with a more global algorithm. There is the assumption that the TNEs for each processor's outgoing links are transmitted to those processors where these links are the input links. The messages carrying this information are nullmessages $< null@ts >$. Provided that the algorithm is executed frequently enough in all of the processors, the estimates on future time stamps can unblock several lp's. Note that it is sufficient to have only one unblocked lp per processor all of the time to keep the simulation running. In the following it is shown that the problem can be mapped to the shortest path problem. Let $G(V, E)$ be a directed graph, where vertices V represent the lp's and edges E represent empty links between the lp's such that $E \subseteq \{(u, v) \mid u, v \in V \text{ and } u \neq v\}$. $G_P(V_p, E_p)$ is a subgraph of G such that all $u \in V_p$ are located in the same processor and $E_p \subseteq \{(u, v) \mid u, v \in V_p \text{ and } u \neq v\}$. The assumption is made that the local virtual time $LVT(u)$ and the smallest time stamp $T_{min}(u)$ are known for each $u \in V_p$. If u has no event messages then $T_{min}(u) = T_{END}$, where T_{END} is the end of the simulation time ("endtime"). For a source v it is essential that $T_{min}(v)$ is always known. If $T_{min}(v) \geq T_{END}$, then $T_{min}(v) = T_{END}$.

Let $\delta(u, v)$ be a lower bound on the sum of service time (at u) and propagation delay for an event sent from u to v where $(u, v) \in E$. $\delta(u, v)$ is the lookahead at u for an event sent to v. $\delta(u, v)$ is assumed to be known for each $u \in V_p$ and $v \in V$. The lookahead can be computed dynamically or determined before the simulation starts. In some simulated systems it is possible to know branching probabilities and service times in advance [11]. In this case $\delta(u, v)$ might include the time an event has to wait before it can be sent from u to v (e.g., the events routed from u to other lp's than v are serviced before the first event is sent to v).

A network N is a graph G where each edge $(u, v) \in E$ has weight $\delta(u, v) \geq 0$. The subgraph G_p has a corresponding network N_p.

At any time in the simulation, $\Theta(u)$ is defined as the earliest (simulation) time at which u may begin to service the next event. For example, $\Theta(u) \geq LVT(u)$ is an invariant. The time-of-next-event $T(u, v)$ for $(u, v) \in E$ is the timestamp of the next possible event to arrive from u to v. It is computed as: $T(u, v) = \Theta(u) + \delta(u, v)$. The problem of computing the TNEs for all the edges can be mapped to the single source shortest path problem by the following procedure. The network N_p is augmented by adding a vertex s and directed edges fro s to each $v \in V_p$. The weights for these edges are: $\delta(s, v) = min_{u \in (V - V_p)}(T_{min}(v), T(u, v))$. $\delta(s, v)$ is thus equal to the LP v's smallest input timestamp or to a smaller timestamp which might arrive from an LP located in another processor. Note that $T(u, v)$, where $u \in (V - V_p)$, is a TNE computed in another processor and sent to the processor with lp v (initially it is 0). The augmented network will be $N_p^{'}$ with vertices in $V_p^{'}$ and edges $E_p^{'}$. A path length in a network $N_p^{'}$ is defined as:

> The path length from vertex s to vertex v in network $N_p^{'}$ is defined recursively as:
> $P(v) = max(P(v) + \delta(u, v), LVT(u))$ where $(u, v) \in E_p^{'} and P(s) = 0$.

The theorem which is at the base of the TNE algorithm is:

> $\Theta(v)$ is equal to the length of the shortest path between vertex s and vertex $v \in V_p^{'}$, in the augmented network $N_p^{'}$.

The proof of this algorithm can be found in [7]. $T(u, v)$ can be computed by definition from $\Theta(u)$ and $\delta(u, v)$ for each (u, v), where $u \in V_p$ and $v \in V$.

Besides the TNE algorithm, also a Limited Time-of-Next-Event (LTNE) is presented which is capable of breaking deadlocks among several processors while the remaining processors continue with the simulation.

2.4.2 Conservative time windows

Conservative LP simulations are distributed in nature since LPs can operate in a totally asynchonous way. One way to make these algorithms more synchronous in order to gain from the availability of fast synchronization hardware in multiprocessors is to introduce a *window* W_i in simulated time for each lp_i, such that events within this *time window* are *safe* (events in W_i are independent of events in $W_j, i \neq j$) and can be processed concurrently across all lp_i [9, 12]. A conservative time window (CTW) *parallel* lp simulation synchronously operates in two phases. In phase *(i) (window identification)* for every lp_i a chronological set of events W_i is identified such that for every event $e \in W_i$, e is causally independent of any $e' \in W_j, j \neq i$. Phase *(i)* is accomplished by a barrier synchronization over all lp's. In phase *(ii) (event processing)* every lp_i processes events $e \in W_i$ sequentially in chronological order. Again, phase *(ii)* is accomplished by a barrier synchronization. Since the algorithm iteratively lock-steps over the two consecutive phases, the hope to gain speedup over a purely sequential DES heavily depends on the efficiency of the synchronization operation on the target architecture, but also on the event structure in the simulation model. Different windows will generally have different cardinality of the covered event set, maybe some windows will remain empty after the identification phase for one cycle. In this case the corresponding lp's would idle for that cycle.

A considerable overhead can be imposed on the algorithm by the identification of when it is safe to process an event within lp_i (window identification phase). Lubachevsy [9] proposes to reduce the complexity of this operation by restricting the *lag* of the lp simulation, i.e. the difference in occurrence time of events being processed concurrently is bounded from above by a known finite constant (*bounded lag* protocol). By this restriction, and assuming a "reasonable" amount of dispersion of events in space and time, the execution of the algorithm on N processors in parallel have one event processed in *O(logN)* time on average. An idealized message passing architecture with a tree-structured synchronization network supporting an efficient realization of the bounded lag restriction is assumed for the analysis.

2.4.3 Other Nicol's works

Besides the work on conservative time windows, Nicol has also proposed many algorithms for simulating particular classes of networks. In [6] a sweep algorithm for massively parallel simulation of circuit-switched networks is presented and its implementation and performance on SIMD (single instruction multiple data) and MIDM (multiple instruction multiple data) parallel machines is discussed. In [13] is presented a conservative synchronization protocol for the parallel simulation of queueing networks having C job priority classes, where a job's class is fixed. The problem has long vexed designers of conservative synchronization protocols because of its seemingly poor ability to compute lookahead. For, a job in service having low priority can be preempted at any time by an arrival having higher priority and an arbitrarily small service time. The proposed solution is to skew the event generation activity so that events for higher priority jobs are generated farther ahead in simulated time than lower priority jobs. Thus when a lower priority job enters service for the first time, all the higher priority jobs that may preempt it are already known and the job's departure time can be exactly predicted. An analysis of the algorithm is given in the paper, demonstrating that good performance can be expected on the simulation of large queueing networks. In [15] the automated parallelization of timed Petri-nets simulations is discussed; the paper addresses the issue of processor synchronization as well as the automated mapping, both static and dynamic, of the Petri-net to the parallel architecture. In [8] the conservative parallel simulation of continuous time Markov chains using uniformization is described; two variations of a conservative parallel simulation algorithm are presented. In any of the two variations, a relative short "presimulation" is performed to identify those times, and only those times, at which the simulation algorithm requires processors pairs to synchronize.
Finally, the cost of conservative synchronization in parallel discrete event simulations is discussed in [14]. This paper analytically studies the performance of a synchronous conservative parallel discrete-event simulation protocol. The class of models considered simulates activity in a physical domain, and possesses a limited ability to predict future behaviour. Using a stochastic model, it is shown that as the volume of simulation activity in the model increases relative to a fixed architecture, the complexity of the average per-event overhead due to synchronization, event list manipulation, lookahead calculations, and processor idle time approaches the complexity of the average per-event overhead of a serial simulation, sometimes rapidly. The method is therefore within a constant factor of optimal. The result holds for the worst case "fully-connected" communication topology, where an event in any portion of the domain can cause an event in any other portion of the domain. The analysis conducted by Nicol demonstrates that on large problems there is often enough parallel workload so that processors are not usually idle. It also demonstrates the viability of the method empirically.

3 A comparison between the proposed approaches

To properly conclude this document it is necessary to make a comparison between the presented approaches. In some papers a comparison with the basic CMB algorithm ("nullmessages protocol") is already given, while in some other it isn't. Where possible a comparison, even if not documented in the original paper from which the algorithm has been taken, will be performed on the basis of measurements conducted by the author. This is the case of the "basic demand-driven approach", which has been implemented, tested and compared with CMB approach by the author. For the "advanced demand-driven approach" neither performance evaluations has been found, nor comparison with other approaches, and since it has not been implemented, it is excluded by this comparison section.

3.1 CMB approach vs. basic demand-driven approach

The comparison between the two approaches has been carried out by observing the behaviour of the following system: The five lp involved show a very simple behaviour: when they receive a real message, they

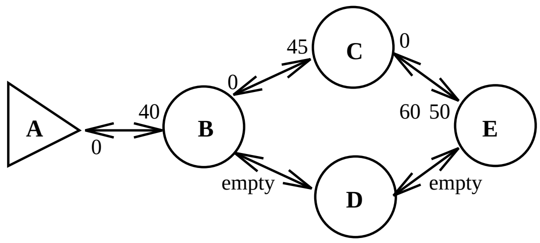

Figure 1: A simple system topology.

answer to all their neighboring (bidirectional arrows represent ongoing and incoming channels) processors with a delay of three time units. For synchronization messages, the lookahead is evaluated by adding one to the current time (this is not always coherent with the model and it can lead in some cases to manage messages out of order, but it was simple to implement and it however allowed a comparison between the two tecniques).

The following table shows the amount of real and synchronization messages in four different cases: *lp_identifier*_CMB refers to messages sent by *lp_identifier* in the basic scheme (CMB algorithm); *lp_identifier*_DD_10 refers to messages sent by *lp_identifier* in the demand-driven basic scheme with timeout of 10 ms; *lp_identifier*_DD_100 refers to the case of demand-driven basic scheme with timeout of 100 ms and finally *lp_identifier*_DD_1000 refers to the case of demand-driven basic scheme with timeout of 1000 ms. NULL stands for sent nullmessages, REPLY stands for sent replies to time queries, QUERY stands for sent time queries, REAL stands for sent real messages, RE\SYN is the number of real messages divided for the number of synchronization messages and TT is the termination time for that processor. The simulation cycle stops when 50 events have been managed. The number of messages and the LVT value when simulation stops are mediated over three simulation runs.

	NULL	REPLY	QUERY	REAL	RE\SYN	TT
a_CMB	1.3	*	*	1.3	1	41
a_DD_10	*	0	1	1	1	9
a_DD_100	*	0	1.3	1.3	1	9
a_DD_1000	*	0	1	1	1	9
b_CMB	3.3	*	*	0.7	0.21	3
b_DD_10	*	6	5.7	5	0.427	10
b_DD_100	*	6.7	6.3	9	0.692	10
b_DD_1000	*	5.7	7.7	7	0.522	10

c_CMB	54.3	*	*	22.3	0.41	24
c_DD_10	*	0	1.3	2	1.538	6
c_DD_100	*	0	1.3	2	1.538	6
c_DD_1000	*	0	1	1.3	1.3	6
d_CMB	51.7	*	*	24.3	0.47	24
d_DD_10	*	12.3	14.3	6	0.225	10
d_DD_100	*	11.7	15.7	6	0.218	10
d_DD_1000	*	13	14.7	4.7	0.169	10
e_CMB	56.7	*	*	24	0.423	25
e_DD_10	*	7.3	7.7	2	0.13	9
e_DD_100	*	8	6.7	2	0.136	9
e_DD_1000	*	12.3	3.7	2	0.125	9

In both the schemes, the input channels are initialized as follows:
(a, b) = $< real@40 >$
(b, a) = $< real@0 >$
(b, c) = $< real@45 >$
(c, b) = $< real@45 >$
(b, d) = empty
(d, b) = empty
(c, e) = $< real@0 >$
(e, c) = $< real@50 >$, $< real@60 >$
(d, e) = empty
(e, d) = empty
In the basic nullmessages protocol there are $< real@0 >$ messages in the channels which are empty in the demand-driven approach. Note that in both schemes a message sent for example from b to c with timestamp 3, will be processed after the message with ts 45 .... This initial configuration has been chosen because it stresses the demand-driven synchronization protocol. Inchoerences due to the presence of messages with $ts > 0$ in the input channels at the beginning of the simulation, and to a rough evaluation of the lookahead can arise. However it is not very simple to choose a model, some initial messages and a function for evaluating the lookahead in a coherent way. The experiments conducted and reported here were more quantitative than qualitative, and had the purpose of verifying the correctness of the implemented synchronization algorithms rather then coherently simulating a real-case application.

These (limited) experiments led us to the following observations:

1. It is very difficult to find a simple but coherent example for conducting experiments: putting initial messages with timestamp > 0 in the input buffer can be interesting (above all for testing the demand-driven algorithm) but usually leads to inconsistencies; evaluating the lookahead is not easy even with a simple model; putting the lookahead $= 0$ usually blocks the simulation (the simulator goes on, but lp's cannot increase their LVTs), but putting it equal to 1 can sometimes cause forwarding of messages with a "too optimistic" timestamp.

2. In the demand-driven approach, the timeout waited on the empty input buffers does not substantially influence the number of real or synchronization messages sent; this may depend on the implementation language (Java), and the language features used for simulating the lp's and their scheduling (the lp's are Java threads, and are scheduled by the Java thread scheduler).

3. There is no evidence of which is the better approach. Globally, the number of real messages divided by the number of synchronization messages is the same for the two approaches (for lp's d and e it seams to be better the CMB approach, while for lp's b and c it seams to be better the demand-driven approach). It is evident that the CMB approach helps in advancing more the LVTs, and in managing a greater number of messages (both real and null). From the example Misra's approach seems less

"fair" (lp a and b manage less than one tenth of messages managed by the other lp's) but this is partly due to the model simulated and its initial conditions.

3.2 CMB approach vs. carrier-nullmessage approach

In [3] a brief performance analysis is conducted, and an important observation arising from this analysis is that

> carrier-null messages are transmitted and processed only when it is necessary.

. The overhead of the carrier-null message algorithm has also been experimentally studied. Experimental results has shown that in simulating a typical feed-forward queueing network both the CMB and carrier-nullmessage algorithms can achieve very good speedup comparing with the sequential event-list approach. Obviously, without considering the overhead caused by transmitting and processing carrier-null messages, the two aloritms should behave exactly the same in simulating such a feed-forward network.

The aim of the carrier-nullmessage algorithm is to use the topological information in order to improve the lookahead ability, and thus reducing the number of synchronization messages required in the deadlock avoidance. It is evident that this goal is achieved in the simulation on certain feed-backward simulation networks. In a simple example of feed-backward simulation network reported in the paper, for the carrier-nullmessage algorithm, only 23 synchronization messages (nullmessages and carrier-nullmessages) were needed in order to break the deadlock, but more than 60 nullmessages were required by the CMB algorithm. Another relatively non-trivial example is the simulation of the queueing network. Previously experimantal studies suggest that the CMB algorithm is not a suitable approach in simulating such a queueing network, because an enormous number of nullmessages are required in the deadlock avoidance, and transmitting and processing of them may severely affect the performance. However the ratio of the number of real messages and the number of synchronization messages (referred to as *RMR*) can be significantly improved by the carrier-nullmessage algorithm, which guarantees that the carrier-nullmessage approach can still achieve a substantial speedup even in simulating such feed-backward queueing networks.

3.3 CMB approach vs. time-of-next-event approach

In order to make the comparison easier we assume that several lp's are clustered onto the same processor. Clustering is not one of the characteristics of the CMB approach. It should be clear that any clustering reduces the number of messages since the transmission of messages among the cluster processes is limited to copying data from one memory location to another. In a fully-interconnected directed lp graph the number of interprocessor communication channels is $n(n - n \setminus k)$ where n is the total number of lp's and k is the number of processors. With decreasing k, the communication cost decreases accordingly.

1. *Zero service times.* In the CMB simulation, the service times (i.e. lookahead) must be strictly larger than zero. Distributions in general do not have a nonzero lower bound. This flaw was remedied in the TNE algorithm.

2. *Communication cost.* The CMB approach does not take advantage of clustering. Every lp is viewed as being independent so it can send a null-message whenever the state of the lp changes. In the TNE approach, the orchestrated generation and delivery of nullmessages reduces their volume. During a single execution of the TNE algorithm, each lp is allowed to send nullmessages to its outgoing links only once. In the CMB approach many more messages are required in order to obtain the same information.

3. *The dissemination of the progress information.* The main objective in the "conservative" distributed simulation is to inform the lp's of the progress of other lp's in the simulation. This information is used to prevent the lp's from receiving any event message from their "past". In the basic CMB approach an lp may send a nullmessage whenever the state of the lp (i.e., the local simulation time) changes. Receipt of a nullmessage may trigger another nullmessage. Soon the lp's are flooded with nullmessages. In the TNE approach the sending of nullmessages proceeds from the lp which possesses a nullmessage with the smallest timestamp. Let the LVT at lp A be larger than the LVT at lp B.

Then, sending a nullmessage from lp A to lp B does not increase the information at B, since A is too far ahead to affect B at the moment. Nonetheless, transmission in the opposite direction increases the information at A a great deal since A may be affected by a message from B at that very moment.

The TNE algorithm is run periodically and asynchronously in each processor: first new events are generated, the algorithm is executed, and then all of the lp's are scheduled until they become blocked. Message transmission between processors takes place after the execution of the algorithm (nullmessages and after all of the lp's are scheduled (simulation messages). Therefore, the algorithm is run more often when lp's block more often.

4. *Reducing the probability of deadlocks.* since the CMB algorithm does not take advantage of the local information when several lp's are mapped to the same processor, it is not very sensitive to different mappings of lp's to processors. In the TNE approach, the mapping of lp cycles within the processor greatly reduces the probability of deadlocks. Lp cycles located in the same processor are efficiently dealt with by the TNE algorithm.

4 Conclusion

We have presented the basic CMB approach, and the most famous approaches for reducing the number of nullmessages, very high in CMB algorithm. A great effort has been devoted to present the different approaches using the same terminology, the same data stuctures and the same pseudo-code, for allowing an easier comparison between them.

A comparison between CMB algorithm and the basic demand-driven algorithm represents the original contibution to the document. Due to time constraints it was not possible to perform experiments over different models, which should have allowed a deeper comprehension of the basic demand-driven algorithm. On the simple model analyzed it was not evident the advantage of adopting the basic demand-driven policy instead of the CMB one.

A comparison between the CMB approach and the carrier-nullmessage one, and the CMB approach and the TNE one as been reported, taken from [3] and [7].

References

[1] W. L. Bain and D. S. Scott. An Algorithm for Time Synchronization in Distributed Discrete Event Simulation. In B. Unger and D. Jefferson, editors, *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 30–33, 1988.

[2] R. E. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical report, Massachusetts Institute of Technology, Cambridge, 1977.

[3] W. Cai and S. Turner. An Algorithm for Reducing Null Message of the CMB Approach in Parallel Discrete Event Simulation. Technical report, 1995. Research Report No 333.

[4] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[5] K. M. Chandy, J. Misra, and V. Holmes. Distributed Simulation of Networks. *Computer Networks*, 3:105–113, 1979.

[6] B. Gaujal, A. Greenberg, and D. M. Nicol. A Sweep Algorithm for Massively Parallel Simulation of Circuit-Switched Networks. *Journal of Parallel and Distributed Computing*, (8):484–500, 1993.

[7] B. Grošelj and C. Tropper. The Distributed Simulation od Clustered Processes. *Distributed Computing*, (4):111–121, 1991.

[8] P. Heidelberger and D. M. Nicol. Conservative Parallel Simulation of Continuous Time Markov Chains Using Uniformization. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):906–921, 1993.

[9] B. D. Lubachevsky. Bounded Lag Distributed Discrete Event Simulation. In B. Unger and D. Jefferson, editors, *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 183–191, 1988.

[10] J. Misra. Distributed Discrete-Event Simulation. *Computer Surveys*, 18(1):39–65, 1986.

[11] D. M. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming, Environments, Applications and Languages*, Yale University, 1988.

[12] D. M. Nicol. Performance Bounds on Parallel Self-initiating Discrete Event Simulations. *ACM Transactions on Modeling and Computer Simulation*, 1(1):24–50, 1991.

[13] D. M. Nicol. Conservative Parallel Simulation of Priority Class Queueing Networks. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):294–303, 1992.

[14] D. M. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Journal of the Association for Computing Machinery*, 40(2):304–333, 1993.

[15] D. M. Nicol and W. Mao. Automated Parallelization of Timed Petri-Net Simulations. *Journal of Parallel and Distributed Computing*, (29):60–74, 1995.