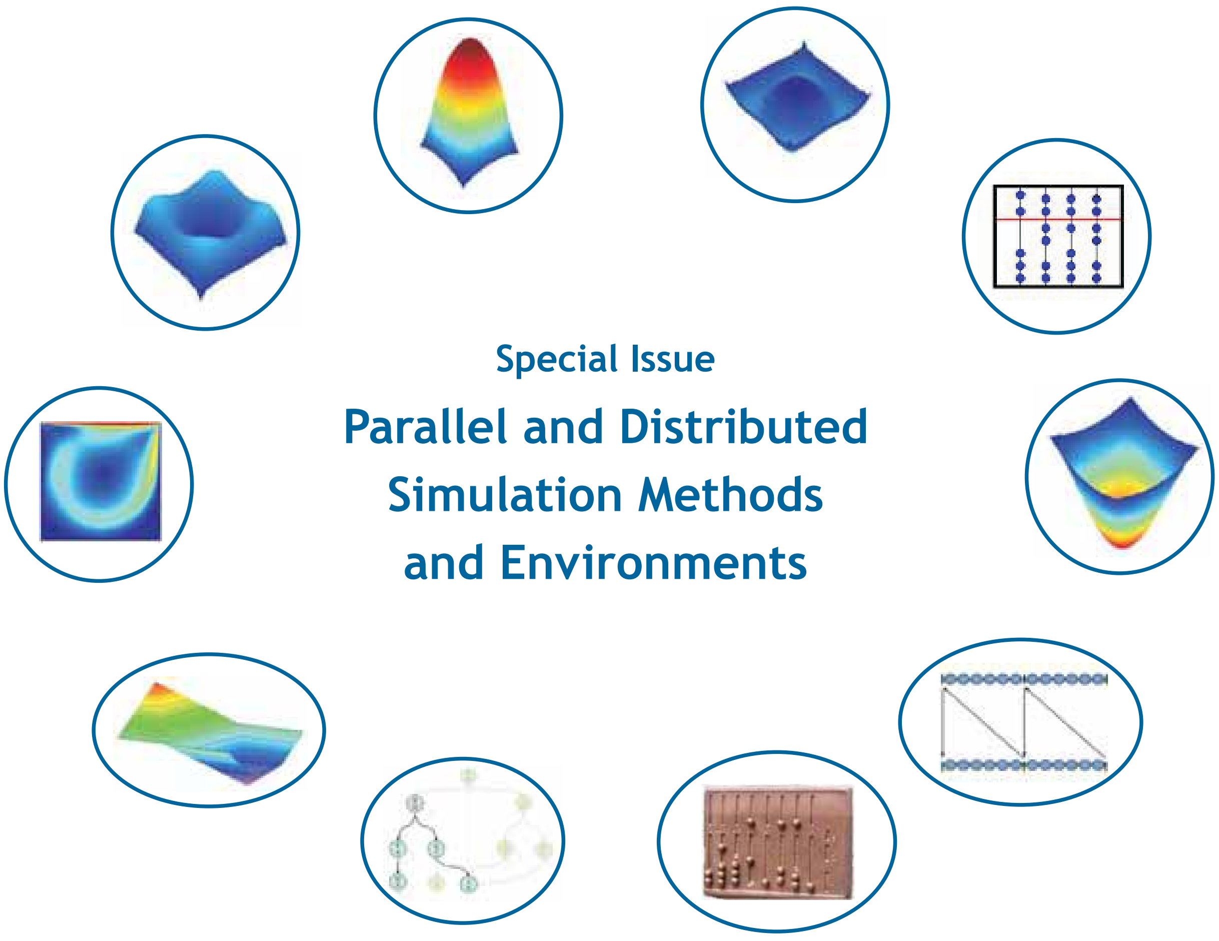
SNE SIMULATION NEWS EUROPE



Special Issue

Parallel and Distributed Simulation Methods and Environments

Volume 16 Number 2

September 2006, ISSN 0929-2268

Journal on Developments and Trends in Modelling and Simulation

Special Issue

ARGESIM

Dear readers,

We are glad to present the first SNE Special Issue - a Special Issue on 'Parallel and Distributed Simulation Methods and Environments'. The idea for special issues was born in ASIM, the German Simulation Society. As there was and as there still is a need for state-of-the-art publications in topics of modelling and simulation, ASIM first tried to publish monographs on this subject. But publication of such books showed disadvantages: too slow production time, too high costs, and lack of publication issues. ASIM, seeking for alternatives, contacted ARGESIM with the idea of SNE Special Issues - while ARGESIM itself thought on Special Issues, because of lack in publication space in the regular SNE issues. Now, one year after the first contact, we can present the first Special Issue, edited by Thorsten & Sven Pawletta from University Wismar, Germany.

The editorial policy of SNE Special Issues is to publish high quality scientific and technical papers concentrating on state-of-the-art and state-of-research in specific modeling and simulation oriented topics in Europe, and interesting papers from the world wide modeling and simulation community. This Special Issue 'Parallel and Distributed Simulation Methods and Environments' (SNE 16/2), will be sent to all ASIM members - together with the regular SNE 16/1 (SNE 46), and sample copies will be sent to other European Simulation Societies; furthermore, it is available on basis of an individual subscription of SNE - SNE Special Issues are open for everybody, for publication and subscription (not only for ASIM). We think also on Special Issues publishing selected papers from EUROSIM conferences.

We hope, you enjoy this Special Issue, which presents state-of-the-art in parallel and distributed simulation, from theory with lookahead formulas via implementation with HLA and other systems to applications in ship desgin and blood flow simulation.

It is planned to publish a SNE Special Issue each year, for 2007 a Special Issue on 'Verification and Validation' (Guest Editor Sigrid Wenzel, University Kassel) is scheduled (SNE 17/2). I would like to thank all people who helped in managing this first Special Issue, especially the Guest Editors, Thorsten and Sven Pawletta from Wismar University.

Felix Breitenecker, Editor-in-Chief SNE; *Felix.Breitenecker@tuwien.ac.*at

Content

SNE Editorial Board

Guest Editors Special Issue *Parallel and Distributed Simulation Methods and Environments*
Thorsten Pawletta, *pawel@mb.hs-wismar.de*
Sven Pawletta, *s.pawletta@et.hs-wismar.de*
Res. Group Computational Engineering and Automation, Wismar University, 23952 Wismar, Germany
WWW.MB.HS-WISMAR.DE/cea

SNE Contact

SNE-Editors / ARGESIM
c/o Inst. f. Analysis and Scientific Computation
Vienna University of Technology
Wiedner Hauptstrasse 8-10, 1040 Vienna, AUSTRIA
Tel + 43 - 1- 58801-10115 or 11455, Fax - 42098
sne@argesim.org; WWW.ARGESIM.ORG

Editorial Info - Impressum
SNE Simulation News Europe ISSN 1015-8685 (0929-2268).
Scope: Development in modelling and simulation, benchmarks on modelling and simulation, membership info for EUROSIM and Simulation Societies.
Editor-in-Chief: Felix Breitenecker, Inst. f. Analysis and Scientific Computing, Mathematical Modelling and Simulation, , Vienna University of Technology, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; *Felix.Breitenecker@tuwien.ac.at*
Layout: A. Breitenecker, ARGESIM TU Vienna / Linz; Anna.Breitenecker@liwest.at
Printed by: Grafisches Zentrum TU, Wiedner Hauptstr. 8-10, A-1040, Wien
Publisher: ARGESIM and ASIM
ARGESIM, c/o Inst. for Analysis and Scientific Computation, TU Vienna, Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria, and
ASIM (German Simulation Society), c/o Wohlfartstr. 21b, 80939München
© ARGESIM / ASIM 2006

Parallel Simulation Techniques for DEVS/Cell-DEVS Models and the CD++ Toolkit

Gabriel Wainer, Ezequiel Glinsky, Carleton University, Ottawa, Canada
WWW.SCE.CARLETON.CA/faculty/wainer

DEVS is a sound formal modelling and simulation (M&S) framework based on generic dynamic system concepts. Cell-DEVS is a formalism for cell-shaped models based on DEVS. This work presents a new simulation technique for execution of DEVS and Cell-DEVS models in parallel/distributed environments. The parallel simulator is based on Time Warp, and developed as a new simulation engine for CD++, an M&S toolkit that implements DEVS and Cell-DEVS theories. The technique uses a non-hierarchical approach that simplifies the structure of the simulator and reduces the communication overhead. The results obtained allowed us to achieve considerable speedups.

Introduction

The widespread use of M&S is leading to execution of larger and more complex systems. One way of handling this complexity is to devote more memory and processor cycles through the use of multiple resources [1]. *Parallel discrete event simulation* (PDES) studies the execution of discrete event models in parallel or distributed computers [1]. The main concern of this community was to reduce execution time of applications by using multiple processors, and a large number of synchronization algorithms were developed [1]. Most of these algorithms are based on Chandy-Misra-Bryant [2, 3] and Time Warp [4], which introduced fundamental ideas that are still used.
Another way to attack these problems considered using the DEVS formalism [5] as the modelling framework for PDES [6, 7, 8, 9]. DEVS is a sound formal framework based on generic dynamic systems concepts that supports provably correct, efficient, event-based simulation. DEVS enables the construction of models in a hierarchical, modular fashion, allowing component reuse and reducing development and testing time.

Cell-DEVS [10] combines cellular automata [11] with DEVS theory, improving timing definition. Individual cells are defined as DEVS models and coupled to form complete cell spaces. *CD*++ [12] is an M&S tool that implements DEVS and Cell-DEVS theory. A hierarchical, conservative parallel simulation mechanism has been implemented in CD++, showed improved results for both DEVS and Cell-DEVS [8]. However, its degree of parallelism and speedups are bounded. Here, we introduce a new technique for optimistic simulation of large, complex DEVS and Cell-DEVS models in CD++. The technique combines the *Time Warp* synchronization mechanism and the DEVS abstract simulators. We introduce two new classes of DEVS processors that carry out the simulation efficiently across multiple processors. In our approach, the hierarchy of the simulation objects is flattened to reduce communication overheads, using a flat simulation approach that eliminates the need for intermediate coordinators [7, 13]. Consequently, it reduces the overhead of message passing, improving the overall performance of the simulation.

1 DEVS and Cell-DEVS

The *DEVS* formalism [5] provides a framework for the definition of hierarchical modular models, allowing for model reuse and development time reduction. A DEVS model is described as a composite of models, each of them being behavioural (*atomic*) or structural (*coupled*). P-DEVS [6] provides a flexible way of dealing with simultaneous events. An atomic DEVS model is defined as:

$$M = < X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$$

At any given time, an atomic model is in state s during a period defined by $ta(s)$. When that time expires, an internal transition takes place; the system outputs the value $\lambda(s)$ and then it changes to the state specified by $\delta_{int}(s)$. If one or more external events (X_M) occur before $ta(s)$, the new state is given by the external transition function, $\delta_{ext}(s, e, X_M)$, which uses a bag of events to allow multiple events to be processed simultaneously. If external and internal transitions are in conflict (an external event is received at this time), the new state is given by $\delta_{con}(s)$.
Coupled models are defined as a set of basic components (atomic or coupled) interconnected through the model's interfaces. The model's coupling defines how to convert the outputs of a model into inputs for the others. A coupled model is:

$$CM = < X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC >$$

X is the set of inputs, Y is the set of outputs, D is a set of the component names; for each $d \in D$, M_d is a basic DEVS model; the external input couplings set (*EIC*) defines how to connect external inputs to components; the external output couplings set (*EOC*) defines how to connect component to external outputs; and the internal couplings set (*IC*) defines how to interconnect components.

Cell-DEVS [10] allows the specification of executable cell spaces with explicit timing delays, which allows easy definition of complex behaviour in physical systems. A parallel Cell-DEVS atomic model [14] can be defined as:

$$TDC = < X^b, Y^b, S, N, d, \tau, \tau_{con}, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, D >$$

Each cell uses a set of N inputs to compute the next state. These values are received through a well-defined interface (X^b and Y^b), activating a local function (τ, τ_{con}), which uses the cell's inputs and present state (S). d defines the kind of delay and D is the state's duration function. The model advances through the activation of δ_{int}, δ_{ext}, δ_{con}, λ, and D, as in other DEVS models.

After the behaviour for a cell is defined, the complete cell space will be constructed by building a coupled Cell-DEVS model:

$$GCC = < X_{list}, Y_{list}, X, Y, n, \{t_1, ..., t_n\}, N, C, B, Z >$$

The cell space is a coupled model composed of an n-sized array of $t_1 \times ... \times t_n$ atomic cells (C). Each of them is connected to the cells defined by the neighbourhood (N). As the cell space is finite, the border (B) can have different behaviour than the rest of the space. X is the set of input events and Y is the set of output events. X_{list} and Y_{list} are the lists of input and output couplings. Finally, the Z function defines the internal and external coupling of cells in the model. CD++ [12] implements DEVS and Cell-DEVS formalisms. Atomic models can be defined in C++ or an interpreted graphical notation, while coupled and Cell-DEVS models are defined using a built-in specification language.

2 Optimistic PDES of DEVS Models

As mentioned earlier, we want to combine advanced DEVS simulators with PDES. In PDES, the simulation is subdivided in smaller parts running on different nodes. Each subpart is a sequential simulation, usually referred to as a *Logical Process* (LP), which groups one or more objects running in a node [1]. Simulation objects communicate through timestamped messages. Objects located on different LPs have to traverse the boundaries of the LPs to interact with each other. *Synchronization* is key in these cases, as the difference in execution speeds can mix events with different timestamps, causing causality problems (i.e., an event in the past affects the present). *Conservative* synchronization algorithms avoid violating causality constraints at all times [2, 3]. Although many conservative algorithms are currently found in real-world applications, they have two main disadvantages ([1]): it is not possible to take full advantage of the concurrency in the application, and the simulator has to be specifically designed to exploit concurrency, leading to a complex, tedious design process.

Optimistic synchronization, instead in [4], allows some causality errors to occur, but provides a detection/ recovery mechanism. Optimistic algorithms enable greater degree of parallelism, and they do not rely on application-specific data.

2.1 The CD++ simulator

CD++ was built as a class hierarchy in C++, where each class corresponds to a simulation entity using the basic concepts defined in [5]. There are two basic abstract classes: *Model* and *Processor*. The former is used to represent the behaviour of the atomic and coupled models, while the latter implements the simulation mechanisms. *Simulators* manage the atomic models. *Coordinators* manage coupled models. The *Root Coordinator* manages global aspects. CD++ was redesigned to provide parallel execution of DEVS and Cell-DEVS [8]. The parallel version of CD++ was built on top of Warped [15], a simulation kernel that provides an implementation of *Time Warp* with different optimizations. Warped uses the MPI message passing standard for communication [16]. Although Parallel CD++ showed speedups in the execution of both DEVS and Cell-DEVS models, a single *Root Coordinator* still acts as a global scheduler for every node in the simulation.

Another problem is that most DEVS simulators are hierarchical, creating a one-to-one correspondence between model components and simulation objects. Since the simulation advances by exchanging messages between simulation objects, communication costs can be considerable. Flat simulation mechanisms, instead, try to reduce this overhead by simplifying the underlying simulator structure, while keeping the model definition and preserving the separation between model and simulator.

Flat simulation approaches have been implemented in distributed [7] and stand-alone [13] environments.

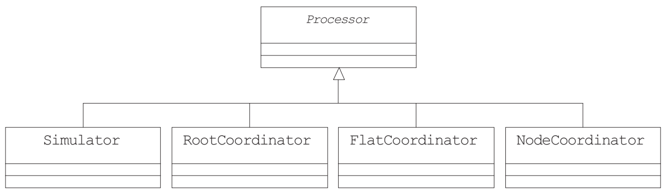


Figure 1: New Processor class hierarchy [17]

There are two basic abstract classes in CD++: *Model* and *Processor*. The former is used to represent the behavior of the atomic and coupled models, while the latter implements the simulation mechanisms. *Simulators* manage the atomic models. *Coordinators* manage coupled models. The *Root Coordinator* manages global aspects (starting/stopping the simulation, communication with environment). This reflects the clear distinction between model and simulator. We took advantage of this separation of concerns by focusing on the processors' class hierarchy only. All classes inheriting from model remain unchanged from those defined in earlier versions of the tool, allowing direct reuse of existing models.

Two new classes are introduced [17]: *Flat Coordinator* (FC) and *Node Coordinator* (NC). Additionally, we modified the *Simulator* and *Root Coordinator* classes (Figure 1). The algorithms we defined are based on those in [6] and [14]. The *Root Coordinator* only handles I/O operations, and starts/stops the simulation. The NC is in charge of synchronization and time management for the LP. The FC is responsible for receiving, translating, and sending messages between its descendants, using a flat data structure with coupling information for every component.

In order to run the model on a distributed environment, we need to indicate the nodes that will participate in the simulation, and how they are allocated to each processor. Figure 2 shows an example where two atomic models run on `Processor 0`, three atomic models run on `Processor 1`, and the remaining two models on `Processor 2`. Node coordinators handle inter-processor communication.

During the creation and registration of each Simulator object, they are associated to the corresponding LP. NCs can communicate with each other using inter-LP messaging. The Root Coordinator executes on one LP, and it forwards messages from the environment to the corresponding NC. On the other hand, when a NC processes an output that must be sent back to the environment, it is sent to the Root Coordinator.

2.2 Abstract simulators in CD++

We will describe the simulation mechanism by presenting the behaviour of each *Processor*. The simulation is message-driven. Different messages can be exchanged among processors: `init` (initialization), `q` (external input), `y` (output), `@` (collect), `*` (internal transition), and `done`.

Simulator

A Simulator is created for each atomic component or cell. It is responsible of executing the functions of the associated model, as follows.

```
1 when (init, 0) message is received
2    initialize model's variables
3    t_L = 0
4    t = ta (s)
5    send (done,t) to parent flat coord.
6 end when
```

When the initialization message is received, variables are initialized (lines 2 and 3) and the simulator informs its parent the time of the next scheduled internal transition (line 5).

When a simulator receives a collect message `(@,t)`, it generates an output, which is sent to the parent flat coordinator (lines 3-5).

When an external message (`q`, `t`) is received, it is stored in the bag of external events (line 12). These messages will be used later, when the external transition is triggered.

```
0:atomic_4
atomic_5

1:atomic_1
atomic_2 atomic_3

2:atomic_6
atomic_7
```

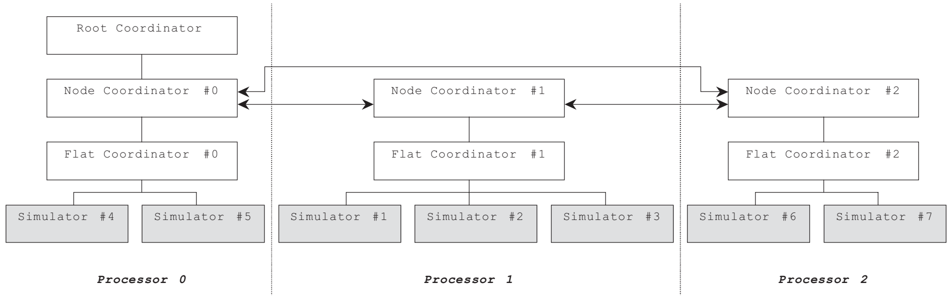


Figure 2. Model partition file for CD++.

```
1  when a (@, t) message is received
2    if t = $t_N$ then
3      y = $\lambda$(s)
4      send (y,t) to parent flat coord.
5      send (done,t) to parent flat coord.
6    else
7      raise error
8    end if
9  end when
10 /**********************************/
11 when a (q, t) message is received
12   add event q to the bag
13 end when
```

An internal message `(*,t)` triggers the execution of a transition function. The simulator executes one of the three transition functions based on `t` (elapsed time since the last transition), t_N (time of the next scheduled transition), and the contents of bag of external events, as showed in the following code fragment. If $t<t_N$ (lines 2-6) and the bag of external events contains at least one element, the external transition is executed. If $t=t_N$ (lines 7-9), it is time for the internal transition. However, if the bag is not empty and $t=t_N$ (lines 10-13) the confluent transition has to be executed. In every case, after executing the corresponding transition, a done message is sent to the parent flat coordinator, indicating the next scheduled transition time (lines 17-19).

```
1  when a (*, t) message is received
2    case $t_L \le t < t_N$
3        e = t - $t_L$
4        s = $\delta_{ext}$ (s, e, bag)
5        empty bag
6    end case
7    case t = $t_N$ and bag is empty
8        s = $\delta_{int}$ (s)
9    end case
10   case t = $t_N$ and bag is not empty
11       s = $\delta_{con}$ (s, bag)
12       empty bag
13   end case
14   case  t > $t_N$ or t < $t_L$
15       raise error
16   end case
17   $t_L$ = t
18   $t_N$ = $t_L$ + ta (s)
19   send (done, $t_N$) to parent flat coord.
20 end when
```

Flat coordinator

A *flat coordinator* has one or more *simulator children* (in charge of the atomic components), and one *parent node coordinator*. The *flat coordinator* uses model coupling information to translate output events into input events. Additionally, it synchronizes models that are imminent in this logical process using a structure called synchronize set.

When the initialization message is received, the f*lat coordinator* forwards it to all its children to complete the initialization phase (lines 3-5). Using the `done` messages received from them, the minimum time of next change is computed and communicated to the parent node coordinator (lines 6-8).

```
1 when (init, 0) message is received from
             parent node coordinator
2    $t_L$ = 0
3    for each child simulator $s_i$
4          send (init, 0) to child $s_i$
5    end for each
6    wait until all done messages receiv.
7    $t_N$ = minimum $t_N$ of all components
8    send (done, $t_N$) to parent node coord.
9 end when
```

When a collect message `(@,t)` is received, it is sent to all dependant simulators with minimum `t` (lines 3-7). Once all the responses are received (line 8), a `done` message is sent to the parent node coordinator. Simulators that have been scheduled for a transition are cached in the synchronize set.

```
1  when a (@, t) message is received from
              parent node coordinator
2  if t = $t_N$ then
3   $t_L$ = t
4   for each imminent child $s_i$ with min. $t_N$
5      send (@, t) to child $s_i$
6      cache i in the synchronize set
7   end for each
8   wait all done messages received
9   send (done, t) to parent node coord.
10 else
11   raise error
12 end if
13 end when
```

```
1  when (y,t) message received from child i
2   if destination of y is the environment
3   send (y, t) to parent node coordinator
4   else
5    for each influencee j of child i
6       q = $z_{i,j}$ (y)
7       if (j is a local processor) then
8          send (q, t) to child j
9          cache j in the synchronize set
10      else
11         send (q,t) to parent node coord.
12      end if
13    end for each
14  end if
15 end when
```

If the destination of the output `(y,t)` message is the environment, the message is sent to the parent node coordinator (lines 2-3; above).

If not, all influencees of the message are computed using the function z_{ij}, and one or more `(q,t)` messages are sent accordingly (lines 5-13).

For destination processors on the same LP, messages are sent directly to the simulator (lines 8-9). Messages for remote simulators are sent to the parent node coordinator (line 11), which will forward them to the corresponding LPs. Local components with scheduled transitions are cached in the synchronize set.

When an external message `(q,t)` is received in a flat coordinator, it is stored in a bag of events.

```
 1 when (q,t) message received from
                 parent node coordinator
 2  if destination of q msg is local then
 3      add event q to the bag
 4   else
 5      raise error
 6   end if
 7 end when
```

Upon receiving an internal message `(*,t)`, the *flat coordinator* sends the external messages stored in the bag to the corresponding components (lines 3 to 8 in the following fragment). All the receivers of these messages are added to the synchronize set. Then, an internal message is sent to all components in the synchronize set. After all done messages are received back from these components, the time of the next event is calculated and a done message is sent to the *node coordinator* (lines 13-17).

```
 1 when (*,t) message is received from
           parent node coordinator
 2   if $t_L \le t \le t_N$ then
 3      for each q ∈ bag
 4         for each local receiver $s_j$ of q
 5           send (q, t) to $s_j$
 6           cache j in the synchronize set
 7         end for each
 8      end for each
 9      empty bag
10      for each i ∈ synchronize set
11         send (*, t) to i
12      end for each
13      wait all done messages received
14      $t_L$ = t
15      $t_N$ = minimum $t_N$ of all components
16      clear the synchronize set
17     send (done,$t_N$) to parent node coord.
18   else
19      raise an error
20   end if
21 end when
```

Node coordinator

One *node coordinator* is located on each LP, and it has one flat coordinator child. Node coordinators drive inter-LP communication, and advance the simulation time in the local LP based on the information received from the *root coordinator* and from its dependant flat coordinator. The algorithms describing its behaviour are described next.

The initialization message, sent by the *root coordinator*, triggers the simulation in each LP. An initialization message `(init,0)` is sent to the flat coordinator (line 2), which will forward it to every simulator.

```
 1 when a (init, 0) message is received
          from root coordinator
 2   send (init, 0) to child flat coord.
 3   wait for done message to be received
          from flat coordinator
 4   sort queue of events by arrival time
 5   t = min ($t_N$ of flat coordinator,
            time of first event in queue)
 6   if t = $t_N$ of queue then
 7      for each q in queue with time t
 8         send (q, t) to flat coordinator
 9      end for each
10   end if
11   send (@, t) to child flat coordinator
12   next-message-type = *
13 end when
```

The first simulation cycle starts after a `(done,t)` message is received. The time for the first collect message is determined by the minimum between the first element in queue of external events and the time of next change reported by the *flat coordinator* (lines 3-5). `next-message-type` is used to determine which type of message has to be sent. If the message to be sent is a collect (lines 6-17), the process is analogous to the initialization phase: minimum time `t` is computed, events with time `t` are sent (if any), the collect message is sent (line 16) and the next message type is set to internal (line 17).

When an internal `(*,t)` message has to be sent to finish the current simulation cycle, the type of the next message is set to collect (line 4).

```
 1 when a (done, t) message is received
             from child flat coordinator
 2   if next-message-type = * then
 3      send (*, t) to child flat coord.
 4      next-message-type = @
 5   else
 6    t= min ($t_N$ of flat coordinator,
           time of first event in queue)
 7      if t > stop simulation time then
 8         stop simulation in this LP
 9      else
10      if t=$t_N$ of first event in queue then
11          for each q in queue with time t
12             send (q, t) to flat coord.
13          end for each
14         end if
15      end if
16      send (@, t) to child flat coord.
17      next-message-type = *
18   end if
19 end when
```

An external message `(q, t)` can be received in a *node coordinator* either from another (remote) *node coordinator* or from its dependant *flat coordinator*. In the first case, this event must be sent to the dependant *flat coordinator* (line 3). This happens when a remote atomic component sends an output through a port connected to an atomic component executing in the local LP. As we have shown earlier, this message is forwarded by the *flat coordinator* to the corresponding simulator.

The timestamp `t` of a message received from a remote *node coordinator* might be lower than the current time in this LP. In such a case, the LP has to recover by performing a rollback. The rollback has to bring that object back to a state whose time is equal or smaller than the time of the straggler. In addition, the messages that were (incorrectly) transmitted from this node coordinator have to be cancelled (anti-messages have to be sent to the destination objects). In the second case, the message must be sent to a remote LP. Thus, it is necessary to determine which *node coordinator* is in charge of that LP, and then to send the message using inter-process communication (lines 5-6).

When a *node coordinator* receives an output message from its child (lines 10-16), a message has to be sent to the environment. The parameter `send-outputs-from-NC` determines whether outputs must be processed directly by the *node coordinator* (line 12), or via the *root coordinator* (line 14). The first alternative reduces the number of messages required to process an output (messages do not have to travel through the *root coordinator*) but requires some post-processing if the outputs of multiple node coordinators have to be merged. The second alternative centralizes the actual processing of outputs in the *root coordinator*; it does not require any post-processing but the overhead is larger.

```
 1 when a (q, t) message is received
 2   if destination q is local
 3      send (q, t) to child flat coord.
 4   else
 5     dest_nc=node coordinator running
            atomic model that must receive q
 6     send (q,t) to node coord. dest_nc
 7   end if
 8 end when
 9 /*********************************/
10 when (y,t) message is received from
          child flat coordinator
11  if send-outputs-from-NC
12     send output (y, t) to environment
13  else
14  send output (y,t) to parent root coord.
15  end if
16 end when
```

When an external event is received from the *root coordinator*, the event is stored in timestamp order. The destination simulator for that event will eventually receive it when that time is reached by the LP.

```
when a (q, t) message is received from
          parent root coordinator
  add q to the sorted queue of events
end when
```

Root coordinator

The *root coordinator* is responsible for starting the simulation, dealing with external events, and sending outputs back to the environment. It starts the simulation by sending initialization messages to every *node coordinator*, located on the different logical processes that form the simulation.

```
for each child node coordinator nc_i
  send (init, 0) to nc_i
end for each
```

External events are received from the environment in the *root coordinator*, which sends an external event to *node coordinators* that have one or more atomic model that should receive that message (lines 3-6 in the next code fragment).

```
 1  when (q,t) is received from environment
 2   t_L = t
 3   for each child node coord. nc_i sharing
 4     LP with destination atomic model of q
 5      send (q, t) to nc_i
 6   end for each
 7  end when
 8  when a (y, t) is received from child NC
 9   t_L = t
10    send (y, t) to environment
11  end when
```

Output messages received by the *root coordinator* are sent back to the environment. This code is never executed if the parameter `send-outputs-from-NC` is set; otherwise, the *root coordinator* consolidates the processing of output messages.

Figure 3 summarizes the flow of messages using the previous algorithms. The *root coordinator* is in charge of starting the simulation process by sending initialization messages. When an output is sent from an atomic component to another, we can identify two different cases: Simulators execute on the same or on different LPs. In the first case, the FC on that LP takes care of the situation. In the second case, a Simulator on LP_i has to send an output to a Simulator on LP_j. FC_i identifies that the destination Simulator is not its descendant.

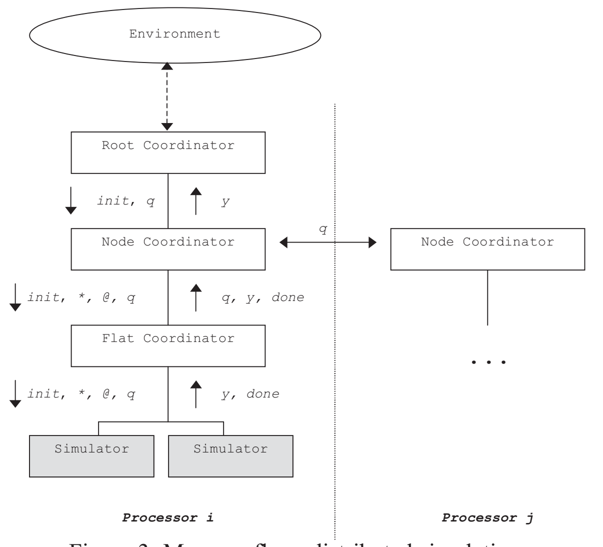


Figure 3: Message flow: distributed simulation.

Thus, it forwards the message to its parent $\mathtt{NC}_i$, which identifies the corresponding `LPj` and forwards the message. Inter-LP communication can lead to violations to the local causality constraint; in that case, a rollback is triggered.

2.3 Rollbacks in CD++

We will show the execution behaviour of the simulator presented in the previous section by showing how to execute a 10 ×10 Cell-DEVS model divided in two LPs. Figure 4 shows the initialization phase. The first simulation cycle is started by the *Root Coordinator*, which sends an initialization message to the NCs in $\mathtt{LP}_0$ and $\mathtt{LP}_1$ (`1` and `2`). When `(init,0)` is received in a NC, it is forwarded to the `FC` (`1.1` and `2.1`). Then, the FCs forward the messages to their Simulators (`1.2`–`1.51`, and `2.2`–`2.51`), triggering an initialization function.

After computing the time for the next change (using the *ta*(*s*) function), every simulator sends a `done` message to its parent FC reporting its time of next change. For example, $\mathtt{S}_1$ indicates that there is an internal transition function to be executed at time `100` (message `1.52`), whereas $\mathtt{S}_2$ reports no scheduled internal transition (message `1.53`, which contains infinity, and represents that the model is in a passive state). After receiving all done messages, the FC sends a `done` message to its parent NC (messages `1.103` and `2.103`) with the minimum time of its components (in this case, `100` for both LPs).

Then, the NC checks for external messages to be sent, and it sends the first collect message to collect the outputs of the imminent components. Figure 5 shows how NCs send the first message to their FCs (`1.1` and `2.1`), which forward a collect message to imminent descendants (Simulators with next change = `100`). For example, in $\mathtt{LP}_0$ it is sent to $\mathtt{S}_1$ (`1.2`) but not to $\mathtt{S}_2$.

When receiving a collect message, Simulators execute their output functions and send the result/done messages to its parent (`1.20` and `2.18`). FC translates output messages and sends external messages to the corresponding local influencees or to the local NC. FC sends a `done` message to the NC completing the collect phase.

At this point, the NC is ready to send an *internal* message (`*`) to start the next phase. The cycle is similar to the *collect* phase: the FC forwards the *internal* message only to Simulators that have a scheduled transition for the current simulation time (`100`). Simulators execute the internal, external, or confluent transition function according to the current time, the time of next change and the state of the bag of events. Done messages are sent to inform the time for the next transition.

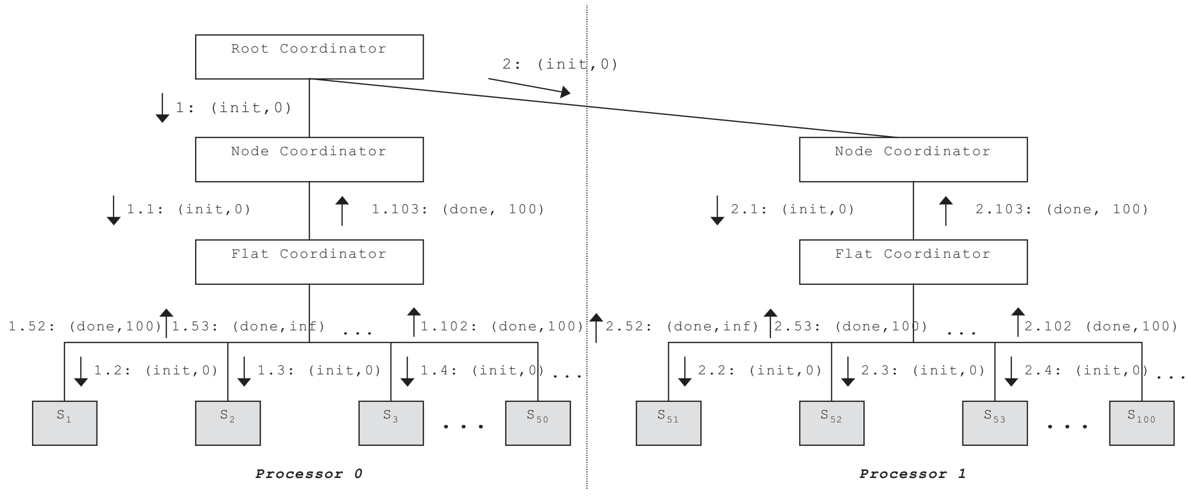


Figure 4: Initialization phase in sample Cell-DEVS model.

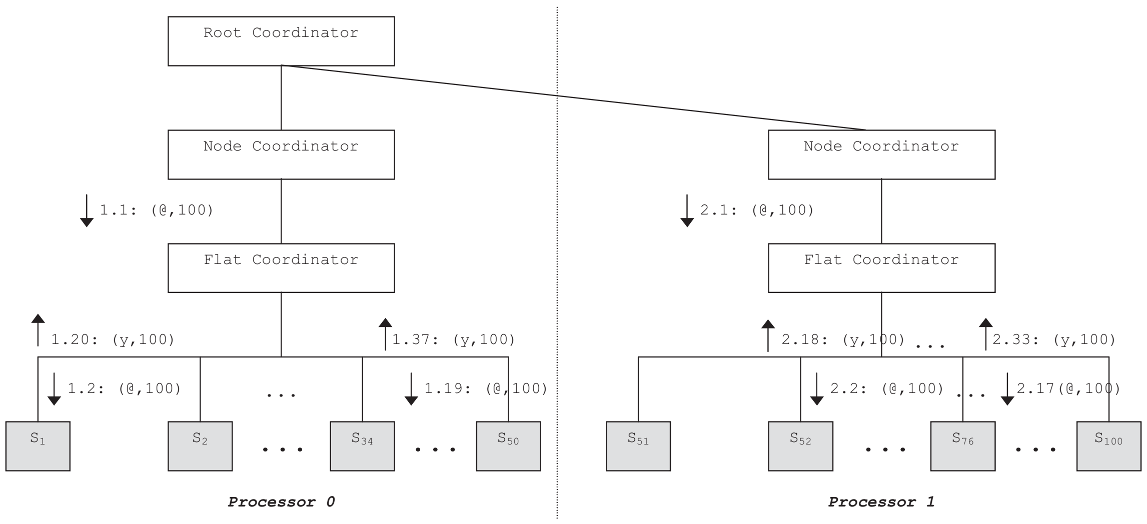


Figure 5: Collect phase in sample Cell-DEVS model.

NCs are in charge of inter-LP communication. If the message has a timestamp greater than the local time in the destination, simulation continues. However, if there is a violation to causality, a rollback has to be executed. Figure 6 shows the scenario for a straggler message in processor P_0. The local times are t_0=280, and t_1=210. In P_0, the NC sent an internal message, which FC forwarded to S_1 (1 and 1.1). In P_1, NC sent a collect message (2), which after being forwarded (2.1 to 2.14) resulted in an *output* from S_{52} (2.15) that has to be sent to S_1. This message is forwarded as an external message, q, from the FC (2.16) to the NC. Then, the NC in P_1 forwards it to the NC in P_0 (2.17). The timestamp of the message, 210, is smaller than the time at the local processor (280), triggering a rollback in P_0.

Figure 7a shows the state of the NC's input, state, and output queues at the moment of receiving a straggler with t=210. Figure 7b depicts the NC's queues after the rollback was completed: then, the NC can return to process events, starting by the one that caused the rollback.

We defined the previous algorithms using different services provided by Warped. Figure 8 shows the new class diagram of the DEVS processors along with some of methods that implement the algorithms previously described. *Processor* is an abstract class that is derived from *TimeWarp* class. *Processor* provides basic functionality and data that are common to all DEVS processors in the application. It defines the methods *initialize*, *executeProcess* and *finalize* as well as other methods and variables.

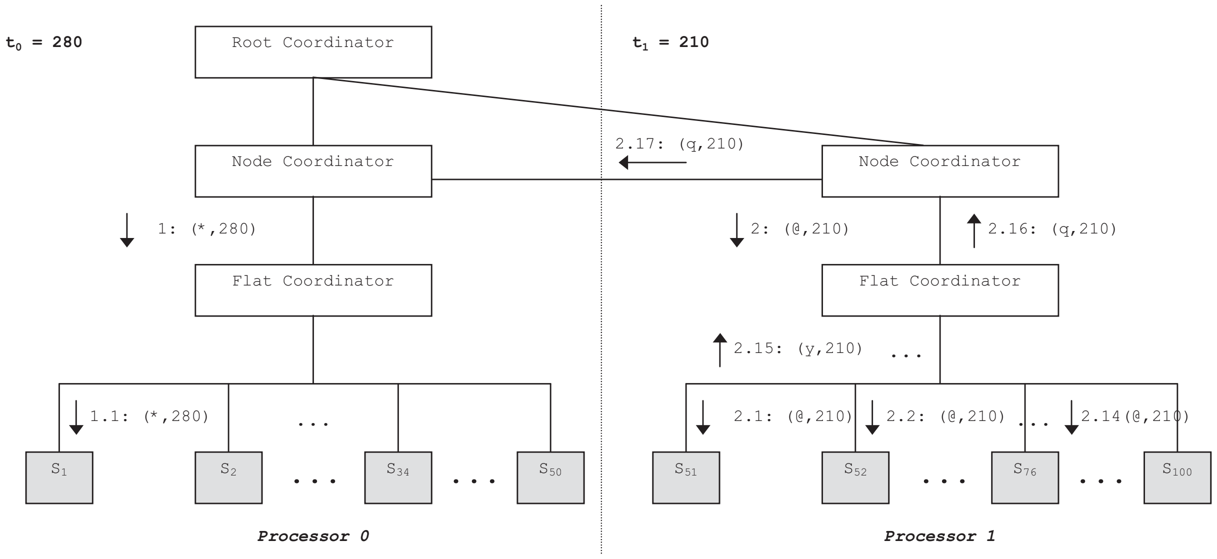


Figure 6: Straggler message received during the simulation of a Cell-DEVS model.

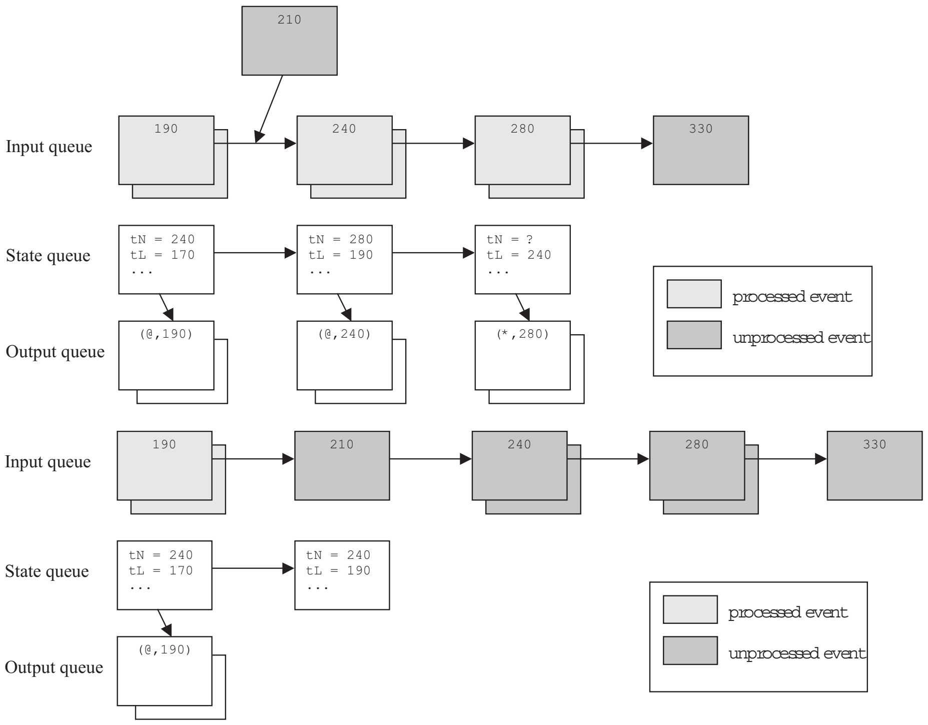


Figure 7: (a - top) Reception of a straggler message in a NC, (b - bottom) State of the NC after the rollback.

In general, processor includes the definition of:

- send methods for each type of message. These methods use, in turn, *sendEvent* in *TimeWarp*.
- Time management methods (e.g., *timeNext()*, *timeLast()*, *timeNext(VTime)*, *timeLast(VTime))*, to report and update the time of the next scheduled change, the time of last change, etc.
- *Initialize, finalize*, and some debugging methods.
- *ExecuteProcess()*, which defines the behaviour of any DEVS processor.
- *rollbackCheck()*, which is called in the receive method, and checks for straggler messages.
- Basic variables, such as the model associated to this processor, its parent, id and descriptors.

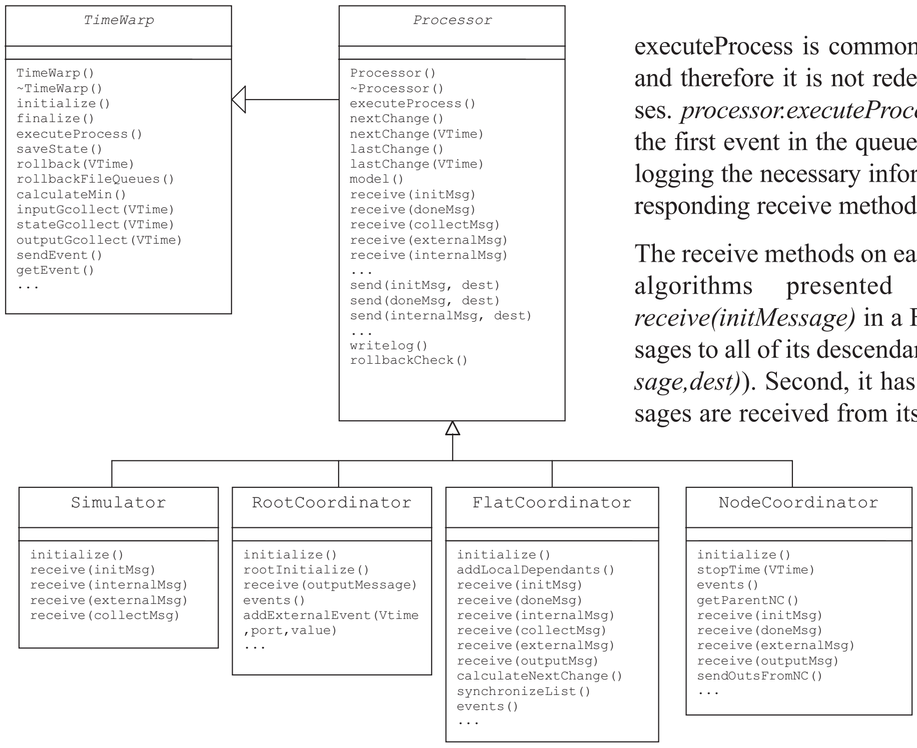


Figure 8: Class diagram for the new DEVS processors.

executeProcess is common to every DEVS processor, and therefore it is not redefined by any of its subclasses. *processor.executeProcess()* is in charge of getting the first event in the queue of events (using *getEvent*), logging the necessary information, and calling the corresponding receive method based on the message type.

The receive methods on each processor implement the algorithms presented earlier. For example, *receive(initMessage)* in a FC sends initialization messages to all of its descendants (using the *send(initMessage,dest)*). Second, it has to wait until all done messages are received from its dependant Simulators.

nodeCoordinator keeps track of the number of done messages it has received (using *doneCount()*). Finally, it determines and updates the time of next change (using *nextChange(VTime)*), and sends this value to its parent NC (using *send (doneMsg,dest)*).

The *receive* (*initMessage*) method on *Simulator*, in contrast, initializes the model variables, computes the time for the next transition (using time advance function, *ta*) and sends a *done* message to its parent.

3 Simulation Experiments

We carried out different performance tests to analyze the results obtained with the new algorithms. To provide uniform means for the overhead, we used the DEVStone benchmark, a synthetic model generator that automatically creates models [18]. DEVStone uses three different types of models with variations in their internal and external structure: LI models, with a low level of interconnections for each coupled model; HI models with a high level of input couplings, HO models with high level of coupling and numerous outputs.
Table 1 shows the parameters we used for different tests. Each model is executed using a different number of levels in the modeling hierarchy (Depth), and different number of submodels on each level (Width). Likewise, different execution times are used for the transition functions, using the DEVStone benchmark.

	Type	Depth	Width	δ_{int}	δ_{ext}
A	LI	3	10	50 ms	50 ms
B	LI	10	3	50 ms	50 ms
C	LI	5	5	50 ms	50 ms
D	LI	10	10	50 ms	50 ms
E	HI	3	6	50 ms	50 ms
F	HI	6	3	50 ms	50 ms
G	HI	5	5	50 ms	50 ms
H	HI	6	6	50 ms	50 ms

Table 1: Simulation parameters.

The following figures show some of the overhead results obtained for these different execution times. The experiments were executed in a single processor, allowing us to measure the pure overhead incurred by our simulator.

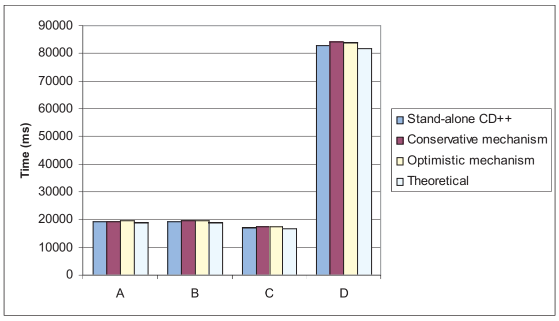


Figure 9: Execution times for LI models.

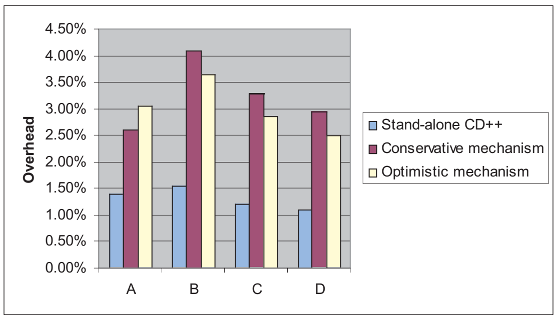


Figure 10: Overhead for LI models

In Figure 9 and Figure 10, we present the total execution time and the overhead for models A-D. We can see that the stand-alone engine outperforms the optimistic one, because the optimistic simulator is more complex. In this case, we need extra synchronization, saving states, input, and output queues, etc. Although the overhead associated with those tasks can be considerable, the optimistic simulator still outperformed the conservative simulator for models B, C, and D. This is a consequence of the reduction in communication overhead incurred by the flat simulator. In model A, the hierarchical conservative engine performs better than the flat, optimistic engine as a consequence of the structure (3x10) of model A. In this case, the reduction in messages exchanged is not that important.
Figure 11 illustrates the results for HI models, which are similar to those obtained for LI.

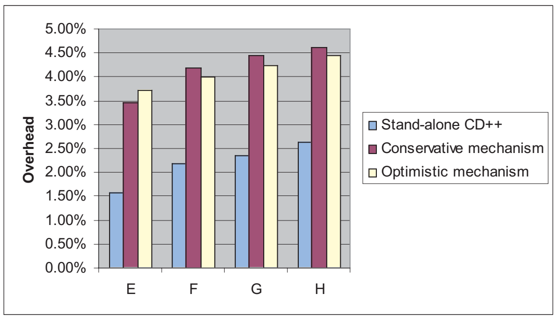


Figure 11: Overhead for HI models

We executed several Cell-DEVS models using different cell spaces on one and four processors. As we can see in Figure 12, the execution time for the model running on one processor varies from 30.7 to 90.8 seconds. When running the model in parallel on 4 processors, the execution time is smaller (between 18.1 and 47.5 seconds); in some cases, the optimistic simulator allows to reduce the execution time in ~50%. Here, the speedup has been affected by the communication costs, as the tests were executed over a relatively slow network, a 10 Mbit/s hub.

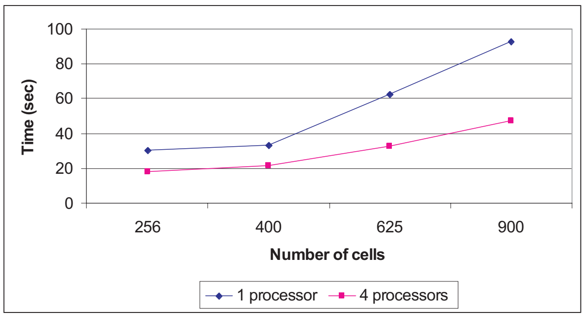


Figure 12: Execution times (1 vs. 4 processors).

We are interested in analyzing the performance of our simulator for larger Cell-DEVS. Figure 13 shows the execution times for different configurations for a cell space of 50x50, using different initial values (*life A-D*). The execution times significantly reduce on 8 processors. When a 50x50 model is executed on a single processor, only one LP is created. Hence, a single instance of a FC is in charge of the 2500 Simulators, and a single NC is in charge of scheduling tasks for the entire model. In contrast, the distribution on 8 processors allows a smaller structure associated with each LP (312 Simulators).

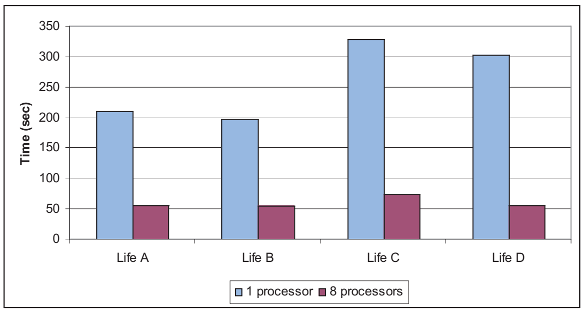


Figure 13: Execution times (50x50 model).

The test uses a sample Cell-DEVS model to study the performance of a firefly model, in which most of the cells change frequently, producing increased processor load. We execute models with 400 and 900 cells, with two initial configurations (models 1 to 4). The optimistic simulator running on a single processor achieves almost the same performance as the conservative simulator running on 4 processors, which shows the increased communication costs.

Figure 14 shows that the optimistic simulator allows significant speedups: 2.91 for 20x20 models, 3.17 for 30x30 models. The speedup factor obtained by executing the simulation on 4 processors using the optimistic approach instead of the equivalent partitioning for the conservative approach is 2.45 for 20x20 and 30x30 models.

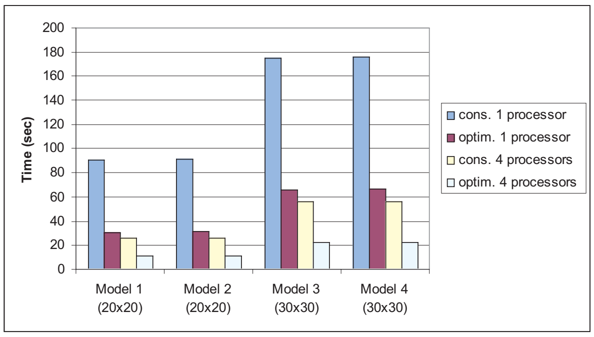


Figure 14: Execution times using conservative and optimistic simulators (1-4 processors).

4 Conclusions

We have introduced a new flat simulation technique for DEVS and Cell-DEVS based on Time Warp, a well-known optimistic synchronization protocol. Our efforts address the need for efficient, fast execution of models using parallel and distributed simulation. We propose an optimistic distributed mechanism that enables achieving higher degrees of parallelism than previous efforts, which only allowed exploiting parallelism in a limited way.

Under our new approach, scheduling tasks are distributed on the Logical Processes; each *Node Coordinator* is in charge of the scheduling tasks for the local simulation objects. Node Coordinators advance the simulation optimistically, assuming that there will be no straggler events. In case of detecting a violation to the local causality constraint, a rollback mechanism allows recovering from it.

Using DEVStone, we compared the overhead of our new technique with the overhead of previous implementations. Although the overhead associated with synchronization tasks implemented by our simulator can be considerable, it still outperformed previous alternatives for some models in single-processor executions. This is a consequence of the flat mechanism implemented in our engine, which outweighs the increased overhead associated with its more complex implementation.
More importantly, we showed that when executing different types of DEVS models, the overhead of Warped/MPI is reasonable small (2.5%-5%). This is a promising result, as the amount of speedup time achievable by these simulators is considerable, and having a constrained overhead in the kernel permits a better utilization of the computing resources.

We showed that the execution times for a particular Cell-DEVS model can be reduced using distributed simulation. Different model sizes where considered, ranging from 256 to 2500 cells.

The execution of the model in a distributed environment allowed achieving better performance than stand-alone execution. Using distributed environments, our simulator outperforms other alternatives and achieves considerable speedups.

References

[1] R. M. Fujimoto: *Parallel and Distribution Simulation Systems*. Wiley. 1999.

[2] R. E. Bryant: *Simulation of Packet Communication Architecture Computer Systems*. MIT, Cambridge, MA. USA. 1977.

[3] K. Chandy, J. Misra: *Distributed Simulation: A Case Study in Design and Verification of Distributed- Programs.* IEEE Transactions on Software Engineering, pp. 440-452. 1979.

[4] D. R. Jefferson: *Virtual time*. ACM Transactions on Programming Languages and Systems. vol. 7(3), pp. 404-425. July, 1985.

[5] B. Zeigler, T. Kim, H. Praehofer: *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. 2000.

[6] A. C. Chow, B. P. Zeigler: *DEVS: A parallel, hierarchical, modular modelling formalism*. Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.

[7] K. H. Kim, Y. R. Seong, T. G. Kim, K. H. Park: *Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally*. Transactions of the Society for Modelling and Simulation International. vol. 13(3), pp. 135-154. 1996.

[8] A. Troccoli, G. Wainer: *Implementing Parallel Cell-DEVS.* Proceedings of the Annual Simulation Symposium. Washington DC, USA. 2003.

[9] B. Zeigler, Y. Moon, D. Kim, G. Ball: T*he DEVS Environment for High-Performance Modelling and Simulation*. IEEE Computational Science and Engineering. 4 (3), pp. 61 -71. 1997.

[10] G. Wainer, N. Giambiasi. *N-Dimensional Cell-DEVS*. In Discrete Events Systems: Theory and Applications, Kluwer. Vol. 12, No. 1. January 2002. pp. 135-157.

[11] S. Wolfram: *A new kind of science*. Wolfram Media, Inc.

[12] G. Wainer, G. *CD++: a toolkit to develop DEVS models*. Software - Practice and Experience. vol. 32, pp. 1261-1306. 2002.

[13] E. Glinsky, G. Wainer: *Performance analysis of DEVS environments.* Proceedings of AI Simulation and Planning. Lisbon, Portugal. 2002.

[14] G. Wainer: I*mproved cellular models with parallel Cell-DEVS*. Transactions of the SCS. vol 17 (2). June 2000.

[15] D. Martin, T. McBrayer, P. Wilsey: *WARPED: Time Warp Simulation Kernel for Analysis and Application Development*. Proceedings of the 29th Hawaii International Conference on System Sciences. 1996.

[16] J. Dongarra, J. et al. *MPI: The Complete Reference*. The MIT Press. 1996.

[17] E. Glinsky, G. Wainer: *New Parallel simulation techniques of DEVS and Cell-DEVS in CD++*. In Proceedings of the 38th IEEE/SCS Annual Simulation Symposium. Huntsville, AL. 2006.

[18] E. Glinsky, G. Wainer: *DEVSTONE: a Benchmarking Technique for Studying Performance of DEVS Modelling and Simulation Environments*. Procedings of IEEE/DS-RT. Montréal, QC. 2005.

Corresponding author: Gabriel Wainer,
Dept. of Systems and Computer Engineering,
Carleton University
4456 Mackenzie Building, 1125 Colonel By Drive
Ottawa, ON. K1S 5B6, CANADA.
WWW.SCE.CARLETON.CA/faculty/wainer

Received: June 5, 2006
Revised: July 1, 2006
Accepted: July 15, 2006