An Environment for Advanced Parallel Simulation of Cellular Models

Shafagh Jafer Gabriel A. Wainer

Department of Systems and Computer Engineering, Carleton University, Ottawa, ON K1S 5B6 Canada (phone: 613-520-2600 x 1957; e-mail: {sjafer,gwainer@sce.carleton.ca}).

Abstract— **DEVS is a sound formal modeling 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 environments. These techniques are modifications to the original Time Warp mechanism offered by WARPED kernel. Time Warp functionalities are revised to include two new algorithms namely, Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM). The resultant simulator is used as new simulation engine for CD++, a M&S toolkit that implements DEVS and Cell-DEVS theories. The results obtained allowed us to achieve considerable speedups due to the reductions that LRFM and GRFM protocols perform on number of rollbacks and anti-messages.**

Index Terms— **Cellular Automata, Parallel Simulator, Cell-DEVS, Optimistic Simulator.**

I. Introduction

MODELING and simulation (M&S) methodologies have become crucial for implementing, designing, and analyzing a broad verity of systems. Among the existing simulation techniques, **DEVS** (Discrete Event System Specification) formalism [Zei00] provides a discrete-event M&S approach which allows construction of hierarchical models in a modular manner. DEVS is a sound formal framework based on generic dynamic systems concepts that allows model reuse, and reduction in development and testing time due to its hierarchical approach in constructing models. **Cell-DEVS** [Wai01] is an extension to DEVS which integrates DEVS and cellular automata by presenting each cell as an atomic DEVS model. Cell-DEVS introduced a novel mechanism for computation based on asynchronous cellular models with explicit timing constructions. The technique has been used to develop a wide variety of models in different field, ranging from environmental sciences, traffic, biology and physics.

When large complex models are defined, the computing power of a single computer degrades. In these cases, a parallel simulator can improve execution times. Here, we present new techniques for executing DEVS and Cell-DEVS models in parallel and distributed environments based on the WARPED kernel [Mar99], an implementation of the Time Warp protocol [Jef85]. Our optimistic simulator, called as PCD++, is built as a new simulation engine for CD++ [Wai02], a M&S toolkit that implements the DEVS and Cell-DEVS formalisms. Algorithms in CD++ and the WARPED kernel are redesigned based on Near Perfect State Information technique to carry out optimistic simulations using a non-hierarchical approach that reduces the communication overhead. Two new algorithms namely, Local Rollback Frequency Model

(LRFM) and Global Rollback Frequency Model (GRFM) have been implemented and used by our PCD++ simulator. These two algorithms have been tested using different Cell-DEVS models. Here we present an evacuation model of a ship and a model of the Synapsin-Vesicle reaction in neurons.

II. BACKGROUND

DEVS [Zei00] is a formalism for modeling and simulation of DEDS (Discrete Events Dynamic Systems) which provides a framework for the definition of hierarchical models in a modular way by decomposing the real system into behavioral (atomic) and structural (coupled) components. DEVS theory provides a rigorous methodology for representing models, and it does present an abstract way of thinking about the world with independence of the simulation mechanisms, underlying hardware and middleware. A DEVS atomic model is formally defined by:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

S is the set of sequential states;

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

$\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function, where

$Q = \{(s,e) \mid s \in S, 0 < \in < ta(s)\}$ is the total state set, e is the time elapsed since the last state transition;

$\lambda: S \rightarrow Y$ is the output function;

$ta: S \rightarrow R^{+}_{0,\infty}$ is the time advance function.

The semantics for this definition is given as follows. At any time, a DEVS coupled model is in a state $s \in S$. In the absence of external events, the model will stay in this state for the duration specified by ta(s). When the elapsed time e = ta(s), the state duration expires and the atomic model will send the output $\lambda(s)$ and performs an internal transition to a new state specified by $\delta_{int}(s)$. Transitions that occur due to the expiration of ta(s) are called internal transitions. However, state transition can also happen due to arrival of an external event which will place the model into a new state specified by $\delta_{ext}(s,e,x)$; where s is the current state, e is the elapsed time, and x is the input value. The time advance function ta(s) can take any real value from 0 to ∞. A state with ta(s) value of zero is called transient state, and on the other hand, if ta(s) is equal to ∞ the state is said to be passive, in which the system will remain in this state until receiving and external event.

A DEVS coupled model is composed of several atomic or coupled submodels, which is formally defined by:

$$CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle,$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names, and the following requirements are imposed on the components, which must also be DEVS models:

For each $d \in D$, $M_d = (X_d, Y_d, S_d, \delta_{int}, \delta_{ext}, \lambda, ta)$ is a DEVS with

$X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$,

and $Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}$.
The component couplings are subject to the following requirements:

External input coupling (EIC) connects external inputs to component inputs,
$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N e\ IPorts, d \in D, ip_d \in IPorts_d\}$;
External output coupling (EOC) connects component outputs to external outputs,
$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$;
Internal coupling (IC) connects component outputs to component inputs,
$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$;
Select: $2^D - \{\} \rightarrow D$ is the tie-breaking function for imminent components.

Due to the closure property, a coupled model is regarded as a new DEVS model [Zei00]. This property clarifies that the overall behavior of a coupled model is equivalent to a basic atomic model, and therefore allows hierarchical model construction.

Cell-DEVS [Wai01] is an extension to DEVS which integrates DEVS and cellular automata by presenting each cell as an atomic DEVS model. Two types of timing delays can be used, namely *transport* and *inertial* [Wai00]. When transport delay is used, the future value is added to queue sorted by output time, allowing the previous values that were scheduled for output but have not yet been sent to be kept. On the other hand, inertial delays allow a preemptive policy at which any previous scheduled output value will be deleted and the new value will be scheduled. Cell-DEVS formalism is defined by:

$$TDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

where

X	is a set of external input events;
Y	is a set of external output events;
I	represents the model's modular interface;
S	is the set of sequential states for the cell;
θ	is the cell state definition;
N	is the set of states for the input events;
d	is the delay for the cell;
δ_{int}	is the internal transition function;
δ_{ext}	is the external transition function;
τ	is the local computation function;
λ	is the output function; and
D	is the state's duration function.

By integrating atomic Cell-DEVS, coupled models can be constructed representing the cell space. A coupled Cell-DEVS model is formally defined as follows:

$$GCC = < Xlist, Ylist, I, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z, select >$$

where

Xlist	is the input coupling list;
Ylist	is the output coupling list;
I	represents the definition of the model's interface;
X	is the set of external input events;
Y	is the set of external output events;
n	is the dimension of the cell space;
$\{t_1,...,t_n\}$	is the number of cells in each of the dimensions;
N	is the neighborhood set;
C	is the cell space;

B is the set of border cells;
Z is the translation function; and
select is the tie-breaking function for simultaneous events.

The above formalism explains that a coupled model is composed of an array of atomic cells with given size and dimensions where each cell is connected through standard DEVS input/output ports to the cells defined in the neighborhood. Since the cell space is finite, the borders of the cells are either connected to a different neighborhood than the rest of the space, or they are "wrapped" in which they are connected to those in the opposite one using the inverse neighborhood relationship. However, border cells have a different behavior due to their particular locations, which result in a non-uniform neighborhood. A Cell-DEVS coupled model is informally presented in Figure 1.

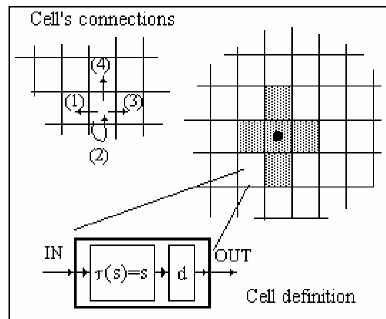


Figure 1. Description of a Cell-DEVS atomic model[Wai00]

CD++ [Wai02] is a modeling tool that implements the DEVS and Cell-DEVS theories by applying the original formalisms. The toolkit includes facilities to build DEVS and Cell-DEVS models. CD++ toolkit also includes an interpreter for Cell-DEVS models [Wai00]. The language is based on the formal specifications of Cell-DEVS. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and the type of cell's bordering. The cell's local computing function is defined using a set of rules with the form *POSTCONDITION DELAY { PRECONDITION }*. These indicate that when the *PRECONDITION* is met, the state of the cell will change to the designated *POSTCONDITION* after the duration specified by *DELAY*. If the precondition is not met, then the next rule is evaluated until a rule is satisfied or there are no more rules. The next section will present two Cell-DEVS models generated with CD++ toolkit.

In parallel and distributed environments the entire task of simulation is divided among the processors or nodes (Logical Process (**LP**)) and therefore each one of them handles a smaller chunk of the simulation while the whole process of execution takes place in parallel and as a result in a significantly reduced time. In sequential simulations, events are executed base on timestamp order, therefore the correctness of the result is automatically guaranteed. In contrast, parallel and distributed simulations require a mechanism to ensure that the result of concurrent execution is identical to that of sequential one. To obtain this correctness, *Local Causality Constraint* [Fuj00] must be satisfied. This requirement is said to be met if and only if each process executes events in non-decreasing timestamp order. Therefore, synchronization among LPs is the most challenging problem of parallel and distributed simulation. There exist three different types of synchronization strategies for event driven simulations:

1. No synchronization at all: synchronization is ensured by the application.
2. Pessimistic (conservative) synchronization [Bry77]: causality violations are strictly avoided.
3. Optimistic synchronization [Jef85]: causality errors are fixed by the notion of *rollbacks*.

Conservative parallel discrete event simulation: This synchronization approach disallows any occurrence of causality errors. The essential for this technique is the *lookahead* which provides the smallest time stamp of the new events that a process can schedule in the future. Null messages are responsible to carry out the lookahead information among LPs. This way each LP, based on the lookahead information that it receives from all other LPs can derive a lower bound on the time stamp (LBTS) of the events that it will receive in future. As a result, the LP would know which event is safe to process. The biggest drawback of the conservative synchronization approach is the time wasting flow of null messages which degrade the simulation performance significantly. Having the fact that optimistic approaches lack in terms of causality errors avoidance, however, they offer two important advantages over conservative techniques:

1. Optimistic approaches have a higher degree of parallelism unlike the conservative approaches where they are overly pessimistic and force the simulation to behave sequentially when in is not necessary.
2. Conservative approaches rely very much on application-specific information when making run-time decisions on whether it is safe to process the event or not. But optimistic mechanism are less reliant on the application for correct execution, therefore allow a simplified software development and more transparent synchronization.

Optimistic parallel discrete event simulation: In this technique which was first proposed by Jefferson's Time Warp mechanism [Jef85], each LP has a *Local Virtual Time* (**LVT**) which advances every discrete step as events are executed on the process. Therefore, time warp processes execute their own portion of the simulation based on LP's LVT. Since every LP has its own LVT, causality errors occur when LPs send messages to each other. This way, an LP may receive a message with time stamp smaller than its current LVT. Such events are referred to as *straggler events*. Once a straggler event is received the process will *rollback*. Rollback is the operation performed upon reception of a straggler event, where the process recovers from the causality error by undoing the effects of all the events that were processed and had timestamp greater than the time stamp of the straggler event. Therefore, these messages were falsely sent to other processes and now must be canceled. This cancellation is performed by sending *anti-messages*. The anti-message has exactly the same format as the original message (the positive message) except for a negative flag to indicate it is an anti-message.

Our PCD++ optimistic simulator implements the Parallel DEVS and Cell-DEVS formalisms and provides the frame work for building and executing DEVS and Cell-DEVS models in distributed environments using the Time Warp protocol. PCD++ implements a *flattened* structure for the simulation framework. Two types of CD++ processors exist on PCD++: *Flat Coordinator* (FC) and *Node Coordinator* (NC). This approach reduces the communication overhead by flattening the structure of the simulation framework.

III. SHIP EVACUATION MODEL

The first Cell-DEVS model we represent here is the illustration of an emergency ship evacuation scenario [Klu01]. The model consists of 20×20 cell space in CD++. The rules defining the model are based on the following restrictions:

1. Each cell representing a person on the ship, calculates its shortest path toward the exit. During initialization phase, people are placed randomly in any empty cell to imitate real ship evacuation scenario.
2. People run in their initial direction until they encounter another person or an obstacle (e.g. wall).

At the end of simulation, there should be no one left on the ship, i.e. all people must have been left through the exit doors.

The neighborhood of each cell consists of 10 cells which will affect the cell's movement (i.e. they can be walls, exit doors, people, or empty cells) as shown in

Figure 2.

	UU (-2,0)		
UL (-1, -1)	U (-1,0)	UR (-1, 1)	
L (0, -1)	(0,0)	R (0, 1)	RR (0, 2)
DL (1, -1)	D (1,0)	DR (1, 1)	

Figure 2. Cell neighborhood

From the above figure we can see that the neighborhood consists of 11 cells: {(-2,0), (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0, 1), (0, 2), (1, -1), (1,0), (1, 1)} where UU is the upper's upper cell, UL is the upper's left cell, U is the upper cell, UR is the upper's right cell, L is the left cell, R is the right cell, RR is the right's right cell, DL is the down's left cell, D is the down cell and DR is the down's right cell. Each value on the cell space defines a distinct state, such as the type of the cell: wall, empty, exit door, a moving person. Also each type of movement is given a state value in order to identify the next position of the person. Table 1 summarizes these values.

Name	Value	Comments
N/A	0	Unknown Empty cell.
Wall	1	Represents an obstacle or a wall.
Exit	2	Represents an exit (e.g. stairs, door).
ED	3	Empty cell and its down (D) cell is the shortest path to the nearest exit.
ER	5	Empty cell and its right (R) cell is the shortest path to the nearest exit.
EU	7	Empty cell and its up (U) cell is the shortest path to the nearest exit.
EL	9	Empty cell and its left (L) cell is the shortest path to the nearest exit.
FD	4	A Full cell (cell with person) and its down (D) cell is the shortest path to the nearest exit.
FR	6	A Full cell (cell with person) and its right (R) cell is the shortest path to the nearest exit.
FU	8	A Full cell (cell with person) and its up (U) cell is the shortest path to the nearest exit.
FL	10	A Full cell (cell with person) and its left (L) cell is the shortest path to the nearest exit.

Table 1. State values and their description

Based on these values, we define different rules for the movement of people in the vessel. The first four rules initialize the model by calculating the shortest path for each undefined cell and placing people on the cell space randomly.

Result	Precondition
3 or 4 (ED or FD)	(0,0) = Undefined and (1,0) is defined.
5 or 6 (ER or FR)	(0,0) = Undefined and (0,1) is defined.
7 or 8 (EU or FU)	(0,0) = Undefined and (-1,0) is defined.
9 or 10 (EL or FL)	(0,0) = Undefined and (0, -1) is defined.

The algorithm works as follows: when a cell detects that one of its attached cells has changed its state to "defined", it would know that the attached cell is the shortest path. The above four rules are implemented as the following:

```
rule: {3+randInt(1)} 0 {(0,0)=0 and (1,0)>1 and (1,0)<11}
rule: {5+randInt(1)} 0 {(0,0)=0 and (0,1)>1 and (0,1)<11}
rule: {7+randInt(1)} 0 {(0,0)=0 and (-1,0)>1 and
(-1,0)<11}
rule: {9+randInt(1)} 0 {(0,0)=0 and (0,-1)>1 and    (0,-1)<11}
```

Then the second four rules define the case when a cell knows that a person will move towards it. The cell knows it will soon be occupied by a person if it is empty and it is the shortest path to at least one cell with a person occupying it.

Result	Precondition
4 → FD state	(0,0) = ED and ((0,1) = FL or (-1,0) = FD or (0,-1) = FR)
6 → FR state	(0,0) = ER and ((1,0) = FU or (-1,0) = FD or (0,-1) = FR)
8 → FU state	(0,0) = EU and ((1,0) = FU or (0,1) = FL or (0,-1) = FR)
10 → FL state	(0,0) = EL and ((1,0) = FU or (0,1) = FL or (-1,0) = FD)

The third four rules define when a cell occupied with a person is attached to the exit. Then, the cell knows that a person will leave it and exit.

Result	Precondition
3→ ED state	(0,0) = FD and (1,0) is exit
5→ ER state	(0,0) = FR and (0,1) is exit
7→ EU state	(0,0) = FU and (-1,0) is exit
9→ EL state	(0,0) = FL and (0,-1) is exit

Then the fourth four rules define when a cell knows that a person will leave it when it is not near an exit. The cell knows that a person will leave it when it is already occupied by a person and its shortest path cell is empty. However, only one person can move to the empty cell when more than one person is trying to move to the same cell. In this case, the priority is first with the person who is in the upper cell, second the one in the right cell, third the one in the down cell, and finally the one in the left cell has the lowest priority.

Result	Precondition
3→ ED state	(0,0) = FD and down (D) cell is empty.
5→ ER state	(0,0) = FR and right cell (R) is empty and UR,RR, and DR cells don't have a person moving to R.
7→ EU state	(0,0) = FU and upper cell (U) is empty and UU and UR cells don't have a person moving

Result	Precondition
	to U.
9→ EL state	(0,0) = FL and left cell (L) is empty and UL doesn't have a person moving to L.

Finally if none of the rules are evaluated, the following rule which serves as a default case, will evaluate. A cell executing this line will remain unchanged and stay as before.

```
rule : {(0,0)} 100 { t }
```

Figure 3 shows an extract of the model's definition in CD++.

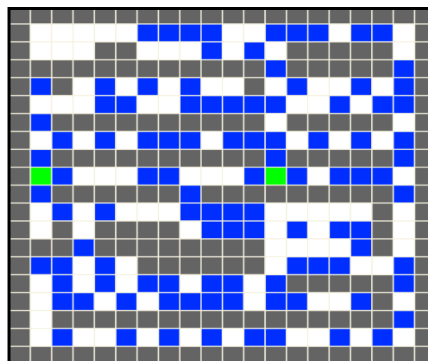
```
[top]
components : ship

[ship]
type : cell dim : (20,20) delay : transport
defaultDelayTime : 20 border : nowrapped

neighbors : (-2,0) (-1,-1) (-1,0) (-1,1) (0,-1)
neighbors : (0,0) (0,1) (0,2) (1,-1) (1,0) (1,1)
…
localtransition : ship-rule
[ship-rule]
rule : {3 + randInt(1)} 0 {(0,0)=0 and (1,0)>1 and (1,0)<11}
…
rule : 4  100 {(0,0)=3 and ((0,1)=10or(-1,0)=4 or (0,-1)=6)}
…
rule : 3 100 { (0,0) = 4 and (1,0) = 2}
…
rule : 3  100 { (0,0) = 4 and odd((1,0)) }
…
rule : {(0,0)} 100 { t }
```

Figure 3. Definition of ship evacuation model in CD++

The ship evacuation model can be modified by adding more exit doors or changing the position of these cells. As presented in Figure 4 initially four different types of cells appear on the grid: empty spaces, walls, people, and exit doors, while the final result of the simulation shows no presence of people, i.e. the ship is evacuated.



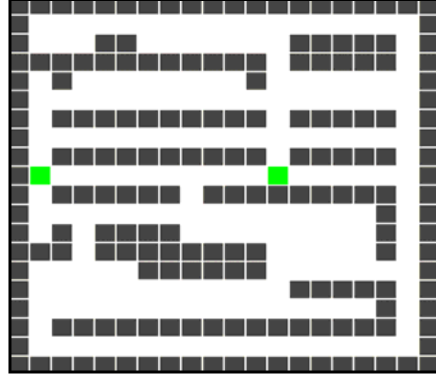


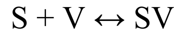
Figure 4. Model Execution Results; initial values; final execution

IV. SYNAPSIN-VESICLE REACTION MODEL

The second model we built was the reserve pool of synaptic vesicles in a presynaptic nerve terminal, predicting the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at differing frequencies. Time series amounts for the components are obtained using rule-based methods (the rules defined by Cell-DEVS) [Ala07].

Synapsin is a neuron-specific phosphoprotein that binds to small synaptic vesicles and actin filaments in a phosphorylation-dependent pattern. Microscopic models have demonstrated that synapsin inhibits neurotransmitter release either by forming a cage around synaptic vesicles (cage model) or by anchoring them to the F-actin cytoskeleton of the nerve terminal [Ben90].

We modeled the molecular interaction of *synapsin* (**S**) with *vesicles* (**V**) which occur inside a nerve cell. The model describes the behavior of synapsin movements until reaching a vesicle and binding to it. Once binding has occurred, depending on *offrate* V and S can again go apart and break their bindings. The *onrate* and *offrate* describe how often bindings occur or break then after. The following formula describes the nature of the reaction:



$$S + V \leftrightarrow SV$$

From the above formula, the left hand side of the equation demonstrates the binding scenario where *synapsin* and *vesicles* perform a bind at a rate specified by *onrate*, while the right hand side of the equation illustrates the bind-break scenario where an *synapsin-vesicle* at an *offrate* which is always smaller than *onrate* breaks apart and again *synapsin* and *vesicles* get released. Then, *synapsin* and *vesicles* can again perform binding and break apart then after. This equation shows an on-going process of “binding” and “breaking apart” which depends on *offrate/onrate*. The larger the *offrate* is, the more bindings get broken apart. Similarly, the larger the *onrate* is, the more V-S binds are produced. Three different scenarios are modeled: 1) V is stationary (with a fixed position on cell space), and S is mobile, 2) V is mobile and S is stationary, and 3) V and S are both mobile (leads to maximum number of total movements and therefore bindings).

The coupled Cell-DEVS model for this application is described as follows.

M=<I,X,Y,Xlist,Ylist, η, N,{m,n}, C, B, Z, select>

Xlist=Φ Ylist=Φ η=9 I=<P^X,P^y>,with P^X={Φ},P^y={Φ};

N={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1)(1,0) (1,1)};

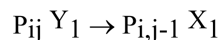
X={0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};

Y={0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};

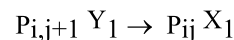
m=26; n=22; B={Φ} ; C={Cij/i ε [1,26], j ε [1,22]}

select ={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1) };

Z:



$$P_{ij}\,{}^{Y_1} \rightarrow P_{i,j-1}\,{}^{X_1}$$



$$P_{i,j+1}\,{}^{Y_1} \rightarrow P_{ij}\,{}^{X_1}$$

$$P_{ij}\,Y_2 \rightarrow P_{i+1,j}\,X_2 \qquad P_{i-1,j}\,Y_2 \rightarrow P_{ij}\,X_2$$
$$P_{ij}\,Y_3 \rightarrow P_{i,j+1}\,X_3 \qquad P_{i,j-1}\,Y_3 \rightarrow P_{ij}\,X_3$$
$$P_{ij}\,Y_4 \rightarrow P_{i-1,j}\,X_4 \qquad P_{i+1,j}\,Y_4 \rightarrow P_{ij}\,X_4$$
$$P_{ij}\,Y_5 \rightarrow P_{ij}\,X_5 \qquad P_{ij}\,Y_5 \rightarrow P_{ij}\,X_5$$

The cell space, the value 1 was used to represent V, and the value 2 was used to represent S. The number 0 represents an empty cell for which a mobile S can occupy. To give direction to the V (although the model assumes fixed V) or S, a two digit number was used. For example, the following represent:

11	"up" moving V	21	"up" moving S
12	"right" moving V	22	"right" moving S
13	"down" moving V	23	"down" moving S
14	"left" moving V	24	"left" moving S

As we can see, Cell-DEVS provides great support for defining these models, for having independent cell states and random mobility of cells, provide an excellent environment to simulate the process of synapsin-vesicles interactions of a nerve. As mentioned earlier, the model constructed can be further extended to include the movement of both synapsin (S) and vesicles (V) as well as defining different off and on rates. Aside from V-S reactions, the model can also include *Actins, which* bind to *synapsins*. Actins can be represented as a string of cells being fixed at their cell space position. A summarized version of the model's definition in CD++ is as follows:

```
[top]
components : chemCell

[chemCell]
type : cell dim : (26,22)  delay : transport
defaultDelayTime : 100    border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1)
neighbors :  (1,-1)  (1,0)  (1,1)
localtransition : chemCell-rule

[chemCell-rule]
rule : {round(uniform(11,14))} 100 {  (0,0) = 1  }
…
rule : {round(uniform(31,34))} 100 {((0,0)=21 or (0,0)=22 or (0,0)=23 or
(0,0)=24) and(((-1,0)-10=1 or (-1,0)-10=2 or…}
…
%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
…
%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or
(0,0)=31 or (0,0)=34) and random < 0.10}
…
rule : { (0, 0) } 100 { t}
```

Figure 5. Synapsin-Vesicle Reaction model in CD++

In the following code, we explain in details each part of the model definition.

```
initialrowvalue : 0       0010201202201012020100
```

```
initialrowvalue : 1       0001020120101020120100
initialrowvalue : 2       0000000000000000000000
initialrowvalue : 3       0010112010120220220100
initialrowvalue : 4       0002010001120220111200
   ...
initialrowvalue : 25      0001202020111202201000

   ...
rule : {round(uniform(11,14))} 100 {  (0,0) = 1  }
rule : {round(uniform(21,24))} 100 {  (0,0) = 2  }
```

In the above two rules, the cells are first initialized with 11-14 (for Vesicles) and 21-24 (for Synapsin) to show the scenario at time = 0, where bindings have not yet been performed. Once bindings occur, cells change their values; 11-14 get replaced with 31-34, and 21-24 get replaced with 41-44. Also for Synapsins, four intermediate values 91-94 are used to represent a moving cell that has not yet being settled down. Once it settles down its value changes back to 21-24 (depending on its direction of movement) and gets ready to bind to a vesicle in its neighborhood.

```
rule : {round(uniform(31,34))} 100 {((0,0)=21 or (0,0)=22 or (0,0)=23 or
(0,0)=24) and
( ((-1,0)- 10 = 1 or (-1,0)- 10 = 2 or (-1,0)- 10 = 3 or (-1,0)- 10 =4 ) or
  ((1,0)- 10 = 1 or (1,0)- 10 = 2 or (1,0)- 10 = 3 or (1,0)- 10 = 4)     or
  ((0,-1)- 10 = 1 or (0,-1)- 10 = 2 or (0,-1)- 10 = 3 or (0,-1)- 10 = 4) or
  ((0,1)- 10 = 1 or (0,1)- 10 = 2 or (0,1)- 10 = 3 or (0,1)- 10 = 4 )    or
  ((-1,1)- 10 = 1 or (-1,1)- 10 = 2 or (-1,1)- 10 = 3 or (-1,1)- 10 = 4) or
  ((1,-1)- 10 = 1 or (1,-1)- 10 = 2 or (1,-1)- 10 = 3 or (1,-1)- 10 = 4) or
  ((1,1)- 10 = 1 or (1,1)- 10 = 2 or (1,1)- 10 = 3 or (1,1)- 10 = 4)     or
  ((-1,-1)- 10 = 1 or (-1,-1)- 10 = 2 or (-1,-1)- 10 = 3 or (-1,-1)- 10 = 4))
and random > 0.10}
```

The above rule describes the following scenario: if there exists a synapsin having the value 21, 22, 23, or 24 (a synapsing that can move up/right/down/left) and there is a vesicle in its neighboring which could be an adjacent cell or a diagonal cell, then the synapsin (red cells) will move toward this vesicle and a binding will occur soon, the value of the synapsin gets changed to 31, 32, 33, or 34 (i.e. 21 changes to 31, 22 changes to 32, 23 changes to 33, and 24 changes to 34) to represent a synapsin that is bonded to a vesicle.

```
rule : {round(uniform(41,44))} 100 {((0,0)=11 or (0,0)=12 or (0,0)=13 or
(0,0)=14) and
( ((-1,0)- 30 = 1 or (-1,0)- 30 = 2 or (-1,0)- 30 = 3 or (-1,0)- 30 = 4) or
((1,0)- 30 = 1 or (1,0)- 30 = 2 or (1,0)- 30 = 3 or (1,0)- 30 = 4) or
((0,-1)- 30 = 1 or (0,-1)- 30 = 2 or (0,-1)- 30 = 3 or (0,-1)- 30 = 4)  or
((0,1)- 30 = 1 or (0,1)- 30 = 2 or (0,1)- 30 = 3 or (0,1)- 30 = 4 ) or
((-1,1)- 30 = 1 or (-1,1)- 30 = 2 or (-1,1)- 30 = 3 or (-1,1)- 30 = 4) or
((1,-1)- 30 = 1 or (1,-1)- 30 = 2 or (1,-1)- 30 = 3 or (1,-1)- 30 = 4) or
((1,1)- 30 = 1 or (1,1)- 30 = 2 or (1,1)- 30 = 3 or (1,1)- 30 = 4) or
((-1,-1)- 30 = 1 or (-1,-1)- 30 = 2 or (-1,-1)- 30 = 3 or (-1,-1)- 30 = 4))
and random > 0.10}
```

Similarly, the above rule describes the following: if there exists a vesicle having the value 11, 12, 13, or 14 (a vesicle that can move up/right/down/left) and there is a synapsin in its neighboring which could be an adjacent cell or a diagonal cell, then since the synapsin will come toward this vesicle and a binding will occur soon, the value of the vesicle gets changed to 41, 42, 43, or 44 (i.e. 11 changes to 41, 12 changes to 42, 13 changes to 43, and 14 changes to 44).

For the movement of synapsin the following four rules are implemented: (each movement is performed in three steps)

```
%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
rule : 00 0 {(0,0)=91}
```

step 1: checking to see if there is an empty cell so the synapsin can move into it, for example if the synapsin's direction is upward (value = 21), then at first we need to check if there is an empty cell right above it. (91 is used as an intermediate value to occupy the empty cell)
step 2: once an empty cell is found, it gets occupied by the synapsin (i.e. the cell's value changes from 0 to a random number 21-24).
step 3: the previous position of the synapsin that just moved to an empty cell gets cleared by setting the value of the cell to 0.

Same procedure is used for right, left, and down movement.

```
%moving right
rule : 92 100 {(0,0)=22 and (0,1)=00 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (0,-1)=92}
rule : 00 0 {(0,0)=92}

%moving down
rule : 93 100 {(0,0)=23 and (1,0)=00 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (-1,0)=93 }
rule : 00 0 {(0,0)=93}

%moving left
rule : 94 100 {(0,0)=24 and (0,-1)=00 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (0,1)=94 }
rule : 00 0 {(0,0)=94}

%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or (0,0)=31 or
(0,0)=34) and random < 0.10}
```

The above rule is used to break the S-V bindings using an offrate = 0.10. According to this rule, 10% of the bindings get broken and as a result synapsins get released and will be given another direction and they will move around until finding a vesicle and binding to it.

Figure 6 shows the grid at the initial case where S and V have not yet interacted to perform a bound. Then, Figure 7 will show how bounds are formed and the corresponding cells change their values to represent the binding.

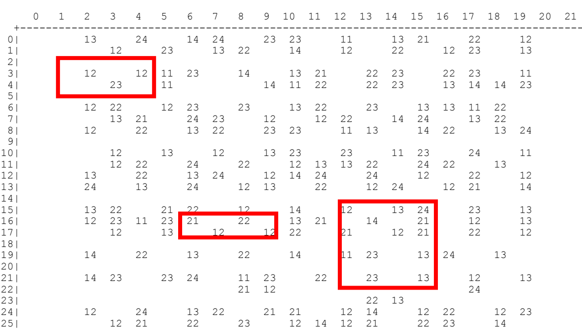


Figure 6. V and S before binding at Time: 00:00:00:100 (bold boxes represent examples of binding structures)

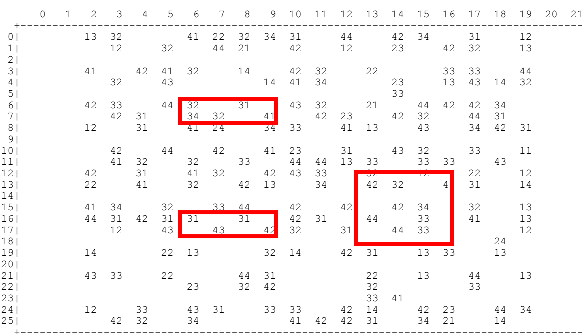


Figure 7. V and S after binding at Time: 00:00:00:300

As illustrated on the above figures, the bold boxes show bindings between synapsin (31-34) and vesicle (41-44). The first illustration (Figure 6) represents the initial scenario where synapsins (21-24) and vesicles (11-14) are free and have not yet performed bindings. Once synapsins walk toward vesicles, the values of the corresponding cells change to 31-34 (bonded synapsins) and 41-44 (bonded vesicles). It is shown that vesicles can be surrounded by more than one synapsin, but each synapsin can bind to only one vesicle at any time.

From the above figure we can see the following possible binding scenarios:

12 22 → 42 33 corresponds to: V– S

12 21 21 → 42 31 31 corresponds to: S – V
|
S

Several initial parameters were tested in order to see the running process of cell nerve with different *offrate*.

V. Parallel and Distributed Simulation

As was mentioned earlier, P-DEVS and Parallel Cell-DEVS extend the standard formalisms of their type to allow a higher degree of parallelism in parallel and distributed environments. In our research, we have modified CD++ sequential simulator to enable parallel and distributed simulations by implementing optimistic synchronization protocol that was first proposed by Jefferson as Time Warp mechanism [Jef85]. We have built an optimistic parallel CD++ simulator (optimistic PCD++) that executes simulation via several ***Time Warp processes*** [Mar99] by exchanging time-stamped event messages using MPI [Gro96]. The Time Warp protocol used by PCD++ simulator consists of two parts: the *local control mechanism* and the *global control mechanism*. The local control mechanism which is provided in each Time Warp process is in charge of rollback operations which include: sending anti-messages, restoring the state of the LP, readjusting Local Virtual Time (**LVT**), etc. On the other hand, the global control mechanism takes care of global issues such as memory management, I/O operations, and termination detection.

Our optimistic PCD++ simulator employs a layered architecture, where each layer depends only on the layers below it. Figure 8 represents these layers.

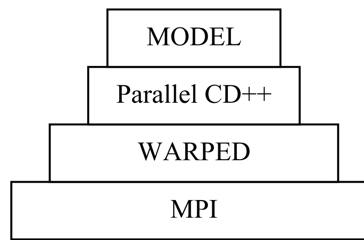


Figure 8. Layered architecture of PCD++ [Gli04]

On the bottom of the architecture the operating system resides. We have chosen Linux Operating System as the underlying platform on which our simulator runs. Above the Operating System lies the Message Passing Interface (MPI). MPI is a standard specification of message-passing library for high-performance communications on both massively parallel machines and on workstations clusters. The Operating System with the use of MPI provides the communication infrastructure for the PCD++ simulator. We have used MPICH [Gro96] portable implementation of MPI which provides a vehicle for MPI implementation research and for developing parallel and distributed applications. The WARPED [Rad98] simulation kernel is our next layer which serves as a configuration middleware that implements the Time Warp mechanism and a verity of optimization algorithms. On top of the WARPED kernel we have our PCD++ simulator implementing the Parallel DEVS and Cell-DEVS formalisms which provides the frame work for building and executing DEVS and Cell-DEVS models in distributed environments using the Time Warp protocol.

PCD++ implements a *flattened* structure for the simulation framework. Two types of CD++ processors exist on PCD++: *Flat Coordinator* (FC) and *Node Coordinator* (NC). This approach reduces the communication overhead by flattening the structure of the simulation framework. The class hierarchies in the modeling and the simulation frameworks are shown in Figure 9.

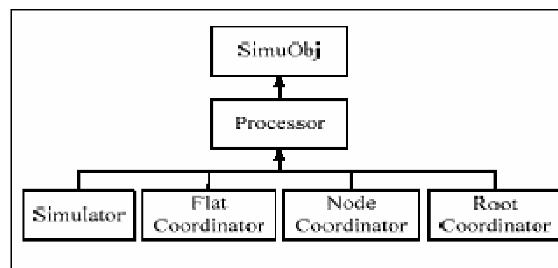


Figure 9. Processor hierarchy

As seen on the diagram, there are four types of PCD++ processors during the simulation: *Simulator*, *FC*, *NC*, and *RC*. When DEVS and Cell-DEVS models are executed over multiple machines, a distributed processor structure is constructed in PCD++ to carry out the simulation. Lets consider the following example to see how partitioning takes place when simulating a coupled DEVS or Cell-DEVS model on two machines using PCD++ simulator. Figure 10 represents this scenario. In this example there are four atomic models (A1, A2, A3, and A4) where A1 and A2 are grouped by the coupled model C1, and C1, and the other two atomic models A3, and A4 are then grouped by the TOP coupled model. The whole model is referred to as TOP model. Since we will execute the simulation on two machines, there will be two partitions encapsulating the atomic models two by two. Partition 0 will take care of A1 and A2, and partition 1 will be responsible for A3 and A4. By *partition* we mean the machine that will run the simulation.

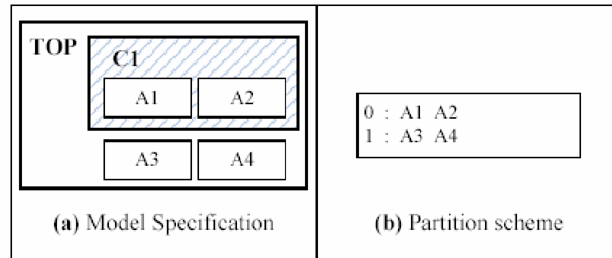


Figure 10. Example model and partition definition

Moreover, a graphical representation of the distributed processors structure of this example is illustrated by Figure 11 .

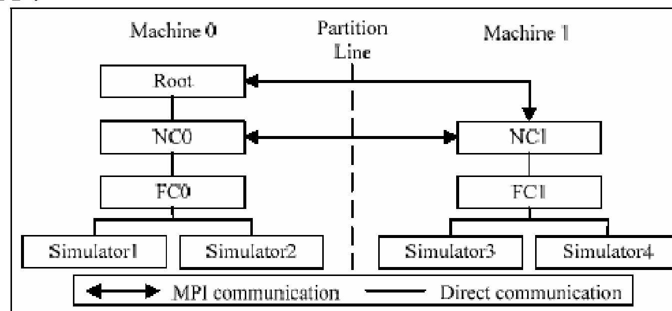


Figure 11. Distributed processor structure for the example

For this example, two logical processes are created, one per each machine: LP0 and LP1. LPs group together the PCD++ processors on the machine they belong to. Two types of messages exist: *remote messages* and *local messages*. Local messages are exchanged among those Simulators which reside on the same LP, while remote messages are exchanged among simulators residing on LPs other than the one they are originating from. Local messages are handled by the FC, and the remote messages are handled by the NC and then sent to the appropriate Simulator through the destination FC. The *root coordinator* is created only on machine 0. It starts the simulation and handles I/O operations. The NC which exists on each machine is the local central controller on each LP and the end point of inter-LP communications. The FC residing between the NC and the Simulators is responsible for synchronizing the execution of its child Simulators. Finally, the Simulator is responsible for executing DEVS abstract functions defined in the atomic models. When a Simulator sends a message to another Simulator sitting on a remote machine, the message is first directed to the FC, then to the local NC through direct communication. Once the message gets to the NC, it will be forwarded to the destination NC through MPI communication. On the receiving end, the NC will then forward the message to the destination Simulator through the child FC.

In PCD++, two types of communications exit among LPs: *synchronous* intra-LP communications which are carried out by all types of PCD++ processors (i.e. Simulator, NC, FC, RC), and *asynchronous* inter-LP communications carried out by NCs. Since inter-LP communications are asynchronous, the NCs require a special structure named as *NC Message Bag* to handle the message passing between LPs who have different local virtual times. The following properties hold for *NC Message Bag*:

1. Messages inside a *Message Bag* can have different timestamps.
2. The time of a *Message Bag* is equal to the minimum timestamp among the contained messages. If the *Message Bag* is empty, then its time is set to infinity.
3. Messages inside a *Message Bag* are processed based on their timestamp in an increasing order. That is the message whose timestamp is equal to the *Message Bag*'s time is processed first. Once the message is processed, it is then removed from the bag, and the bag's time is advanced to the next minimum value among the timestamps of the

remaining messages. Once all the messages are processed and removed from the bag, the *Message Bag*'s time is restored back to infinity implying that the bag is empty.

In contrast, synchronous intra-LP communications are handled by the Simulators and the FC since they are local to the LP and do not pass the boundary of the LP. Similar to the *NC Message Bag*, for intra-LP messages the FC holds a *message bag*. In this case, when two local Simulators (i.e. sitting on the same LP) need to communicate to each other, they send the message to the local FC, and then the message will be directed to the destination local Simulator by the FC. There is no direct communications between Simulators, even the ones sitting on the same LP. Local Simulators can only communicate to each other through their FC. This is the purpose of having *FC message bag*. PCD++ messages are in form of data objects which are dynamically allocated and deleted by the PCD++ processors.

PCD++ processors exchange two categories of messages: *content messages* and *control messages*. The first category includes the external message (*x*) and the output message (*y*), and the second category includes the initialization message (*I*), the collect message (@), the internal message (*), and the done message (*D*). To describe these messages, external and output messages are used to exchange simulation data between the models, initialization messages start the simulation, collect and internal messages trigger the output and the state transition functions respectively in the atomic DEVS models, done messages handle synchronization by carrying the model timing information. The simulation is executed in a message-driven manner.

Each type of PCD++ processor, define its own receive functionality for each type of messages. Let's what happens at each PCD++ processor considering the scenario of reception of different types of messages:

Simulator: upon receiving (*I*, 0) from the parent FC, two variables are used to record the current simulation time (t_L) and the value of *sigma* (t_a). Upon receiving the initialization message, (*I*, 0), the Simulator resets t_L to the timestamp of the message, therefore the Simulator's virtual time now is equal to zero. Then the simulator initializes the variables defined in its associated atomic model, and after that, it informs its parent FC of the value of t_a by sending a *done* message stamped with time 0. When a (*@, t*) message is received, the Simulator invokes the output function (λ) of the atomic model and as a result an output message (*y, t*) is sent to the FC. After this, the Simulator will send (*D, t*) to the FC with $t_a = 0$ to indicate that it is imminent. Following the collect message, a (**, t*) will arrive to trigger internal/external/confluent function of the atomic model depending on the timing of the message and the status of the Simulator's message bag. The last message that may arrive at the Simulator is (*x, t*) which is simply inserted into the Simulator's message bag.

Flat Coordinator: when (*I*, 0) is received, the FC records the total number of its children in a variable named as *doneCount* then forwards the (*I*, 0) message to each child. After this, the FC waits for all its children to respond to this initialization by sending back a (*D*, 0). The FC will only pass the control over to the NC if all its children have finished their previous computation and have sent done messages as notification messages. Upon receiving a (*@, t*) message, the FC forwards it to all imminent Simulators and will keep a record of this for later use (to know which children need to do state transitions when (**, t*) is received). Moreover, when (*y, t*) is received, the FC searches the model coupling information to find out the correct destination. The destination is either an input port on an atomic model, or an output port on the topmost coupled model. In case of receiving (*x, t*) message, the FC will simply insert the message into its message bag. Upon receiving (**, t*) message, the external messages inside the FC's message bag are flushed to the local receiving Simulators. This will trigger the imminent Simulators to perform a state transition. Finally, when a (*D, t*) message is received from a child Simulator, the FC updates the child's t_N to the sum of the current simulation time and the *sigma* value carried by the received (*D, t*) message.

Node Coordinator: upon receiving (*I*, 0), the NC simply forwards it to the child FC. In case of receiving (*x, t*), NC will insert this message into the *NC Message Bag*. These external messages contain values sent from remote Simulators to local ones. When (*y, t*) is received the NC simply forward it the Root (it has to be sent to the environment). Reception of a (*D, t*) message by the NC from a child FC indicates that this is a response to a control message that was previously sent out by the NC.

Root Coordinator: this processor only handles environment-oriented output messages during the simulation. Output to the environment is done through a test file called as output file or OUT file.

Aside from the functionalities of each of the PCD++ processors, we have modified the WARPED [Mar99] kernel in order to run simulations under different protocols. These protocols are modifications of the optimistic one that WARPED implements. The idea is to reduce the number of rollbacks by suspending the LP that has large number of rollbacks and therefore stopping it from flooding the net with anti-messages. However, the LP will still be able to receive input events and they will be inserted into the corresponding message bags. After a predefined duration, the suspend LP is released and will go on simulating. These two protocols [Szu00], namely *Local Rollback Frequency Model* (LRFM) and *Global Rollback Frequency Model* (GRFM) are based on the "Near Perfect State Information - NPSI" protocol [Sri98]. The NPSI protocol implements the *Elastic Time* mechanism. Briefly, Elastic Time is composed of two parts:

1. Identifying the NPSI of the simulation.
2. Translating the NPSI in optimism on the simulation objects.

Each part can be implemented in many ways. The main concept is to associate each LP with a *potential error* (**PE**) to control the optimism of LP_i. During the simulation run, the value of each PE is kept updated by evaluating a function called *M1* which uses state information that is received from the feedback system. Then, the function *M2* translates dynamically every update of PE_i in delays in the execution events.

VI. Local Rollback Frequency Model

The Local Rollback Frequency Model (LRFM) protocol is only based on local information of the logical processes. That is, the simulation object within a LP will be suspended or allowed to continue simulating only based on the number of rollbacks it had. First M1 and M2 functions must be defined:

- *Function M1*: The potential error of a simulation object is the number of rollbacks that the object had from a time *T1* until the actual time *T2*, having that $T2 - T1 <= T$, where *T* is the interval after which the local number of rollbacks of the simulation object gets restarted back to zero.

- *Function M2*: If the number of rollbacks for a simulation object at the interval *T* is greater than a specified value, then the object is suspended, adopting a conservative behavior. By suspending the simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the chance to resumes. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (Time Warp).

To implement this protocol each LP has to be informed about two values: *max_rollback*, and *period*. Where *max_rollback* is the maximum number of allowed rollbacks before suspension of the simulation object, and *period* is the duration for which the simulation object will stay suspended. The algorithm is presented in Figure 12.

```
1. In each LP, at the beginning predefine:
  max_rollbacks and period
2. In each simulation object, at the simulation start:
  previous_time = 0
3. In each object, when the LP is scheduled to run:
  actual_time = Warped.TotalSimulationlTime ()
     if (actual_time - previous_time >= period)
          simulateNextEvent()
          previous_time = actual_time
```

```
        rollbacks = 0
    else
      if  (rollbacks < max_rollbacks)
        simulateNextEvent()
      /* else, SUSPEND the simulation object */
```

Figure 12. LRFM Algorithm

VII. Global Rollback Frequency Model

In Global Rollback Frequency Model (GRFM) protocol each simulation object uses global information in such a way that among all the simulation objects residing on all LPs, the one with greatest number of rollbacks must be suspended for the duration of time defined at the beginning of the simulation. Therefore, at each simulation cycle all the LPs must broadcast the information regarding the rollback counts of all of their simulation objects. As in LRFM, M1 and M2 functions must first be defined:

Function M1: The potential error of a simulation object is the number of rollbacks that the object had minus the maximum number of rollback of the other simulation objects of the simulation, from a time $T1$ until the actual time $T2$, having that $T2 - T1 <= T$, where T is the interval after which the local number of rollbacks of the simulation object gets restarted back to zero.

Function M2: If the number of rollbacks for a simulation object at the interval T is greater than other number of rollbacks of the other simulation objects, then the object is suspended, adopting a conservative behavior. By suspending the simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the chance to resumes. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (Time Warp).

This algorithm is implemented as follows:

```
1. In each LP, at the beginning predefine: period
2. In each simulation object, at the beginning predefine:
      previous_time  = 0
      max_rollbacks = 0
3. In each simulation object, when the LP is scheduled to run:
     actual_time = Warped.TotalSimulationlTime ()
if (actual_time - previous_time >= period)
   simulateNextEvent()
   previous_time  = actual_time
   rollbacks = 0
else
   if  (rollbacks < max_rollbacks)
        simulateNextEvent()
/* else, SUSPEND the simulation object */
4. For i from 1 until the number of LPs
if (i is NOT this PL id)
send to LP i the number of rollbacks of the objects of the LP id
Subroutine that receives the number of rollbacks from other LP:
For j from 1 until the numbers received
If (rollbacks[j] > max_rollbacks)
max_rollbacks = rollbacks[j]
```

Figure 13. GRFM Algorithm

With LRFM and GRFM different simulation results can be collected since the RFM *period* (and in case of LRFM the *max_rollbacks*) can be modified very easily at the beginning of the simulation. This is done by changing these values in the configuration files right before the

simulation starts and therefore, there is no need to rebuild the whole simulator in order for these modifications to have effect.

VIII. Simulation Results

After modifying WARPED kernel of PCD++ simulator to include LRFM and GRFM, the Ship Evacuation model and Synapsin-Vesicle Reaction model were executed and results were collected. To study the performance of our optimistic PCD++ simulator, the experiments were first carried out on standalone CD++ on a single machine, and then on a cluster. The cluster consisted of 32 compute nodes (dual 3.2 GHz Intel Xeon processors, 1GB PC2100 266MHZ DDR RAM) running Linux WS 2.4.2.1 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6.

The metric used to measure the performance of PCD++ simulator is the *Execution Time* which is the total execution time of the simulation collected from the execution environment. During the experiment, results of execution of both protocols; the LRFM and GRFM were conducted and the *Overall Speedup* which is defined as follows was calculated.

$$\text{Overall Speedup} = T(1) / T(N)$$

Table 2 represents the execution of both models on single machine using the standalone sequential CD++ simulator.

Model	Total Execution Time (sec)
Ship Evacuation	6.4327
Synapsin-Vesicle Reaction	3.7621

Table 2. Results of standalone CD++ simulator

Then, simulations were run for both models on 1 to 8 nodes and for each node five trials were collected. The values shown on the graph (Figure 14, Figure 15) are the average of these five trials for each node which are within a confidence interval of 95%.

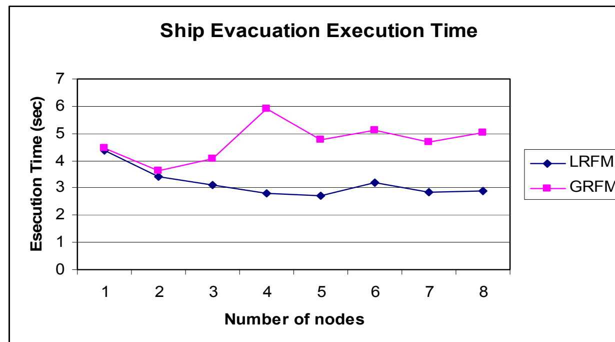


Figure 14. Execution time of Ship Evacuation model with LRFM and GRFM protocols

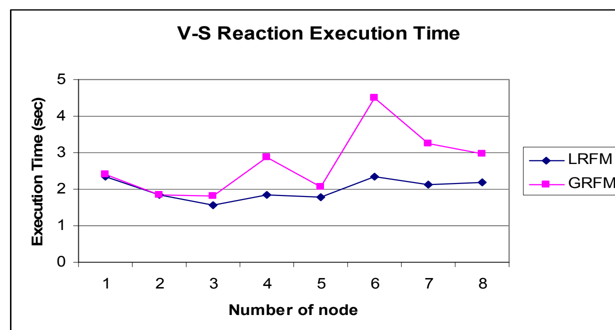


Figure 15. Execution time of Synapsin-Vesicle Reaction model with LRFM and GRFM protocols

The following figures show the speedups for both models:

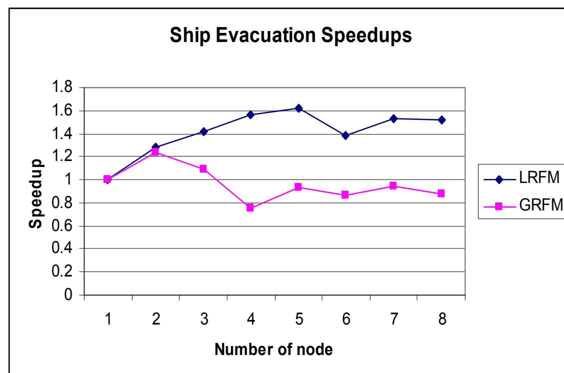


Figure 16. Speedups of Ship Evacuation model with LRFM and GRFM protocols

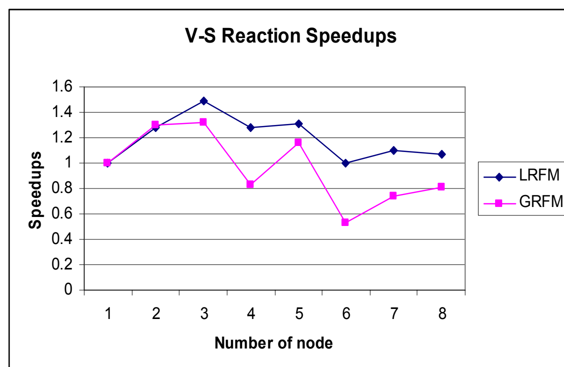


Figure 17. Speedups of Synapsin-Vesicle Reaction model with LRFM and GRFM protocols

IX. Conclusion

We have introduced two new simulation techniques for P-DEVS and Cell-DEVS optimistic simulator by modifying 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-based mechanism to reduce the number of rollbacks and anti-messages. Our two algorithms, namely Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) are implemented as modifications of the optimist one that Time Warp implements. The use of LRFM and GRFM enable achieving higher speedups and lower execution times. Under our new protocols, during the simulation objects with a large number of rollbacks are suspended for a predefined period (although the objects will continue receiving input events). The idea is to stop the objects with large number of rollbacks from flooding the simulation with anti-messages and only allowing the rest of objects to advance. These new protocols are based on the Near Perfect State Information protocol. The execution results (based on two Cell-DEVS models) showed better performance than stand-alone execution. Using more complex and larger models will show considerable speedups.

X. References

[Ala07] Al-aubidy, B.; Dias, A.; Bain. R.; Jafer, S.; Dumontier, M.; Wainer, G.; Cheetham, J. ”Advanced DEVS models with applications to biomedicine”. *Artificial Intelligence, Simulation and Planning*. Buenos Aires, Argentina. AIS 2007.

[Ben90] Benfenati, F., Valtorta, F., Greengard, P.; “Computer Modelling of Synapsin 1 Binding to Synaptic Vesicles and F-actin”. Implications for Regulation of Neurotransmitter Release. 1990.

[Bry77] Bryant, R.E. Simulation of Packet Communication Architecture Computer Systems. Massachusetts Institute of Technology, Cambridge, MA. USA. 1977.

[Fuj00] Fujimoto, R. M. “Parallel and Distributed Simulation Systems”. A Wiley-Interscience publication. ISBN 0-471-18383-0. 2000.

[Gli04] Glinsky, E. “New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models in CD++”. M. A. Sc. Thesis. Carleton University. Canada. 2004.

[Gro96] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. “A high-performance, portable implementation of the MPI message-passing interface standard”. Parallel Computing. Vol. 22, pp. 789-828. 1996.

[Jef85] Jefferson, D. “Virtual Time”. ACM Transactions on Programming Languages and Systems. 7(3):405-425. 1985.

[Klu01] Klüpfel, W.; Meyer-König, T.; Wahle, J.; Schreckenberg, M. “Microscopic Simulation of Evacuation Processes on Passenger Ships”, Theoretical and Practical Issues in Cellular Automata, pp. 63-71, Springer-verlag 2001.

[Mar99] Martin, D. E.; McBrayer, T. J.; Radhakrishnan, R.; Wilsey, P. A. “WARPED – A Time Warp Parallel Discrete Event Simulator (Documentation for version 1.0)”.

[Rad98] Radhakrishnan, R.; Martin, D. E.; Chetlur, M.; Rao, D. M.; Wilsey, P.A. “An Object-Oriented Time Warp Simulation Kernel”. Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE’98). Vol. LNCS 1505, pp. 13-23. Springer-Verlag. 1998.

[Sri98] Srinivasan, S.; Reynolds, J., “Elastic Time”, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 2. 103-139. April 1998.

[Szu00] Szulsztein, E.; Wainer, G. “New Simulation Techniques in WARPED Kernel” (in Spanish). Proceedings of JAIIO, Buenos Aires, Argentina, 2000.

[Wai01] Wainer, G.; Giambiasi, N. “Timed Cell-DEVS: modeling and simulation of cell spaces “. In “Discrete Event Modeling & Simulation: Enabling Future Technologies”, Springer-Verlag. 2001.

[Wai02] Wainer, G. “CD++: a toolkit to develop DEVS models”. Software – Practice and Experience. Vol. 32, pp. 1261-1306. 2002.

[Zei00] Zeigler, B.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.