Timed Cell-DEVS: modelling and simulation of cell spaces

GABRIEL WAINER AND NORBERT GIAMBIASI

DEVS and Cellular Automata formalisms are applied to define a modelling paradigm for cellular models. Different delay functions to specify the timing behavior of each cell, allowing the modeler to represent the timing complex behavior in a simple fashion. Implementation models for the formalism are presented according with the modeler and developer points of view. As a result, efficient and cost-effective development of cellular models simulators could be achieved.

INTRODUCTION

In recent years, a wide number of artificial systems has become commonplace (i.e., computer networks, traffic controllers, flexible manufacturing plants, embedded applications, etc.). The development cost of such systems is crucial for their successful implementation, and their complex analysis has been attacked using simulated models. It is well known that the use of a formal modelling approach can produce important cost reductions. Fortunately, several modeling paradigms have been developed.

As the specified models are analyzed through simulation, their timing information becomes crucial. Hence, it is needed a paradigm allowing *timed models*, representing the event dates in the system. The DEVS formalism (Discrete EVent systems Specification) proposed by Bernard Zeigler [22, 19], allows this kind of specifications. Here, the model's timing is described as a lifetime for each state variable. DEVS is a discrete event paradigm that allows a hierarchical and modular description of the models. Each DEVS model can be behavioral (atomic) or structural (coupled), consisting of inputs, outputs, state variables, and functions to compute the next states and outputs. Object Oriented approaches have been incorporated to the basic concepts [20, 21]. The paradigm improves the security of the simulations, reducing the testing time and increasing productivity.

We are interested in modelling systems that can be represented as executable cell spaces. The Cellular Automata formalism [18] has been widely used to describe complex systems with these characterictics. These automata evolve by executing a global transition function that updates the state of every cell in the space. The behavior of this function depends on the results of a function that executes locally in each cell.

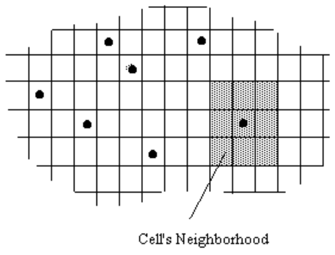


Figure 1. Sketch of a Cellular Automaton

Conceptually, these local functions are computed synchronously and in parallel, using the state values of the present cell and its neighbors. This discrete time paradigm constrains the precision and efficiency of the simulated models. Furthermore, it is usual that several cells do not need to be updated in every step, wasting computation time. These problems can be solved using a continuous time base, providing instantaneous events that can occur asynchronously at unpredictable times. This approach was considered in [22, 19], where discrete event cellular models were presented. Discrete event cellular models were applied in real world applications in later works [8, 23].

These ideas served as a basis for the approach presented here, which will be called Timed Cell-DEVS. We present a summary of the efforts done in building this approach, devoted to describe and simulate discrete event cellular models [16, 17, 9, 11, 13, 14]. A main contribution of the work consists in adapting delay constructions and defining them as a functional component of the model defining each cell [7]. The extensions allow to define explicit timing for each cell, providing a simple mechanism to define it. The specifications of the formalism were used to build a set of tools for modelling and simulation of cell spaces. As a result, the approach allows a modeler to describe complex temporal behavior avoiding the detailed mechanism used for the delays.

The article is organized as follows. First, a description of Timed Cell-DEVS atomic models is introduced. After, coupled cell spaces are considered. Then, several issues related with the implementation models for the formalism are considered. Some examples are used to show implementation issues. Finally, the improvements in the development activities obtained when this approach is used are introduced.

TIMED CELL-DEVS ATOMIC MODELS

Timed Cell-DEVS models are defined as a space composed of individual cells that can be lately coupled to form a complete cell space. This section presents the specification of each cell in a space as a DEVS model with explicit delays. Each cell is a continuous time model, defined by very simple rules and a few parameters. Complex timing definition is overruled due to the use of delay functions. Two kinds of constructions are employed: *transport* and *inertial*. We introduced two kind of delays with different semantics to allow the construction of models at two levels of accuracy. Transport delay has an anticipatory semantics, that is to say that every input event is just delayed. This is an extension of discrete event models with implicit time representation in which event are only ordered. Inertial delays allows to represent more complex temporal behavior because they have preemtive semantics. An event scheduled for a future time will not neccessary executed. For example, this can of delay allows to analyse frequency responses of systems.

An atomic cell can be formally described as:

$$TDC = < \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{I}, V, \theta, E, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

where for $\#T < \infty \wedge T \in \{\boldsymbol{N}, \boldsymbol{Z}, \boldsymbol{R}, \{0,1\}\} \cup \{\phi\}$;

$\boldsymbol{X} \subseteq T$ is the set of external input events;
$\boldsymbol{Y} \subseteq T$ is the set of external output events;
$\boldsymbol{I} = < \eta, \mu^x, \mu^y, P^x, P^y >$. Here, $\eta \in \boldsymbol{N}, \eta < \infty$ is the neighborhood's size,
$\mu^x, \mu^y \in \boldsymbol{N}, \mu^x, \mu^y < \infty$ is the number of other input/output ports, and
$\forall j \in [1, \eta], i \in \{X, Y\}, P_j^i$ is a definition of a port
(input or output respectively), with
$P_j^i = \{ (N_j^i, T_j^i) / \forall j \in [1, \eta+\mu^i], N_j^i \in [i_1, i_{\eta+\mu}]$ (port name), y $T_j^i \in I_i$
(port type)$\}$, where $I_i = \{ x / x \in \boldsymbol{X}$ if X $\}$ or $I_i = \{ x / x \in \boldsymbol{Y}$ if $i = Y \}$;
$\mathbf{V} \subseteq T$ is the set of values that can be used as state variables for the cell;
θ is the definition of the state variables used in each cell, defined as
$\theta = \{ (s, phase, \sigma queue, \sigma) / s \in V$ is the status value for the cell,
$phase \in \{active, passive\}, \sigma queue = \{ ((v_1,\sigma_1),...,(v_m,\sigma_m)) / m \in \boldsymbol{N} \wedge$
$m < \infty) \wedge \forall (i \in \boldsymbol{N}, i \in [1,m]), v_i \in V \wedge \sigma_i \in \boldsymbol{R}_0^+ \cup \infty\}$;
and $\sigma \in \boldsymbol{R}_0^+ \cup \infty \}$; for transport delays, or
$\theta = \{ (s, phase, f, \sigma) / s \in V, phase \in \{active, passive\}, f \in T,$
and $\sigma \in \boldsymbol{R}_0^+ \cup \infty \}$; for inertial delays;
$\mathbf{E} \in V^{\eta+\mu}$ is the set of values of input events;

delay $\in$ {transport, inertial};
d $\in$ $\boldsymbol{R}_0^+$, $d < \infty$ is the transport delay for the cell;
$\boldsymbol{\delta_{int}}$: $\theta \to \theta$ is the internal transition function;
$\boldsymbol{\delta_{ext}}$: Q$\times$$X$ $\to$ θ is the external transition function, where Q is the state values defined as:

$$Q = \{ (s, e) \,/\, s \in \theta \text{ x } E \text{ x } d;\ e \in [0, D(s)]\};$$

τ: E $\to$ V is the local computation function;
λ: $\theta \to Y$ is the output function; and
D: θ x E x d $\to$ $\boldsymbol{R}_0^+ \cup \infty$, is the state's duration function.

The present definition, based on the work presented in [6], is independent of the simulation technique used. Therefore, it allows to specify the system behavior independently of the implementation details.

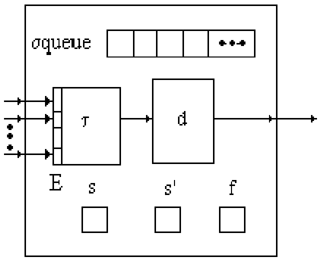


Figure 2. Informal description of an atomic cell.

The cell's interface is composed of a fixed number of ports (P^x, P^y), each connected with a neighbor. A cell can use other inputs and outputs of the interface (μ_x, μ_y) to interchange data with models outside the cell space. Each port in the interface has a name composed by an identifier (**X** for input; **Y** for output) and a natural number (port number). These inputs are stored in the **E** set, whose values are used to compute the future state of the cell. The results obtained when the local function τ executes, can be deferred by using a delay function. To allow this behavior, the cell's state variables (θ) include the cell's present value, the feasible future value for the cell (**f**), and a queue to keep track of the next events (**σqueue**), and the model's phase. The state's lifetime function **D** controls the elapsed time of a cell state, and its evolution is defined by the delay functions. Finally, DEVS transition ($\boldsymbol{\delta_{int}}$, $\boldsymbol{\delta_{ext}}$) and output ($\lambda$) functions are included. In previous works [17, 11,6], the definition of the delay functions was presented as DEVS models. Here, the semantics of the delay functions is presented with detail in the Figure 3.

Each time the external transition function receives a message, the local computing function uses the **E** inputs to obtain the new cell's value. If it is different from the existing, the new value should be sent to the cell's influencees. Otherwise, the neighbors cannot change and the cell remains quiescent [22, 19]. The result is transmitted only after the completion of the delay function associated with the cell.

$$\delta_{\mathbf{int:}} \quad \frac{\sigma = 0; \qquad \sigma\text{queue} \neq \{\varnothing\}; \qquad \text{phase} = \text{active}}{\begin{array}{c}\forall\, i \in [1, m],\ a_i \in \sigma\text{queue},\ a_i.\sigma = a_i.\sigma - \text{head}(\sigma\text{queue}.\sigma); \\ \sigma\text{queue} = \text{tail}(\sigma\text{queue}); \\ s = \text{head}(\sigma\text{queue}.v); \quad \sigma = \text{head}(\sigma\text{queue}.\sigma);\end{array}}$$

$$\frac{\sigma = 0; \qquad \sigma\text{queue} = \{\varnothing\}; \qquad \text{phase} = \text{active}}{\sigma = \infty \qquad \wedge \ \ \text{phase} = \text{passive}}$$

$$\lambda: \quad \frac{\sigma = 0;}{\text{out} = s;}$$

$$\delta_{\mathbf{ext:}} \quad \frac{(s', \text{transport}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } E \text{ x } d); \quad \text{phase=active};}{\begin{array}{c} s \neq s' \Rightarrow (s{=}s' \wedge \forall\, i \in [1,m]\ a_i \in \sigma\text{queue},\ a_i.\sigma = a_i.\sigma\text{-}e \wedge \sigma{=}\sigma - e; \\ \text{add}(\sigma\text{queue},\langle s', d\rangle) \wedge f = s\ )\end{array}}$$

$$\frac{(s', \text{transport}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } E \text{ x } d); \quad \text{phase} = \text{passive};}{s \neq s' \Rightarrow (\ s = s' \wedge \sigma = d \ \wedge\ \text{phase} = \text{active} \ \wedge\ \text{add}(\sigma\text{queue}, \langle s', d\rangle) \ \wedge f = s\ )}$$

$$\frac{(s', \text{inertial}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } E \text{ x } d); \quad \text{phase} = \text{passive};}{s \neq s' \ \Rightarrow \ (\ s = s' \ \wedge\ \text{phase} = \text{active} \ \wedge\ \sigma = d \ \wedge\ f = s\ )}$$

$$\frac{(s', \text{inertial}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } E \text{ x } d); \quad \text{phase} = \text{active};}{s \neq s' \ \Rightarrow \ s = s' \ \wedge \ (f \neq s' \ \Rightarrow \ \sigma\text{queue} = \{\varnothing\} \wedge \sigma = d \ \wedge\ f = s)}$$

Figure 3. Definition of δ_{int}, δ_{ext} and λ for TDC models.

The next events to be transmitted should be queued, because several external events can arrive during a **transport** delay. If the changing cell is passive, it is activated. Instead, if it is active, the values of σ stored in the queue must be updated to reflect the elapsed time since the last event. In both cases, the external transition function schedules an internal event after the time defined by the delay. When the time of an internal event arrives, the first value in the queue is sent to the output ports. The internal transition function removes the first member of the queue recently transmitted. If the queue is not empty, the first element will be used to schedule the internal event. Otherwise, the cell is passivated.

When **inertial** delays are used, the last arrived event can be preempted if a new input arrives before the scheduled time. This only happens if the new external value is different of that one previously stored. If both values are the same, the new external event occurred has the same value than the previous one.

The following figure presents the behavior of both kinds of delay functions. Let us consider a transport delay of 5 time units for a given cell. The Figure 4(a), shows the results delayed for 5 time units. Here, the cell remains active while there are queued values waiting to be transmitted. Oppositely, the behavior of inertial delays can be studied by analyzing the input/output trajectories in the figure 4(b). In this case, an inertial delay function of 5 time units is used. The input values are delayed as in the previous case, but in the simulated time 19, the input of the delay function changes. As this change occurs before the consumption of the delay, the previous event is preempted. The input-output trajectories are piecewise constant for this illustration, in the model, trajectories are transformed in discrete event trajectories.

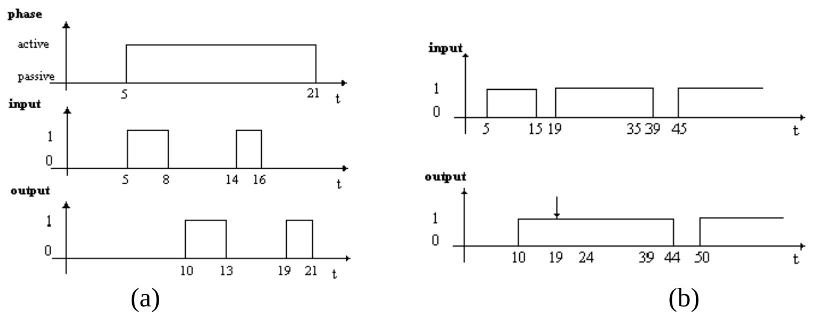


Figure 4. (a) Transport delay behavior; (b) Inertial delay behavior.

COUPLED CELL-DEVS

The atomic cell models presented previously can be coupled with others, forming a multicomponent model. These are defined as a space consisting of atomic cells connected by the neighborhood relationship. After, they can be integrated with other Cell-DEVS or DEVS models.

When modelling a coupled cell space, two different couplings have to be considered. First, the internal coupling defines the connection of a cell with the neighborhood. Then, the external coupling is used to connect certain components in a Cell-DEVS with components in other models. Therefore, we can build complex models consisting of several submodels with different behavior using different paradigms or abstraction levels. These models can be represented as:

$$GCTD = < X, Y, Xlist, Ylist, I, \eta, N, \{m, n\}, C, B, Z, select >$$

where for $\#T < \infty \wedge T \in \{\boldsymbol{N}, \boldsymbol{Z}, \boldsymbol{R}, \{0,1\}\} \cup \{\phi\}$;

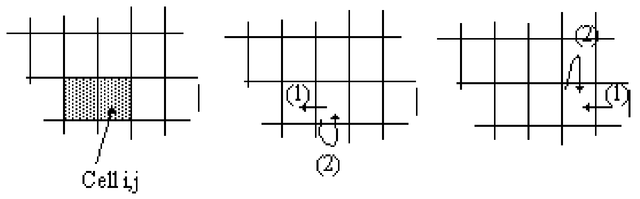
$\boldsymbol{X} \subseteq T$ is the set of external input events;
$\boldsymbol{Y} \subseteq T$ is the set of external output events;
Ylist = $\{ (k,l) / k \in [0,m], l \in [0,n]\}$ is the list of output coupling;
Xlist = $\{ (k,l) / k \in [0,m], l \in [0,n]\}$ is the list of input coupling; and
I = $< P^X, P^y >$ represents the definition of the modular model interface. Here,
for $i = X \mid Y$, P^i is a port definition (input or output respectively), where
$P^i = \{ (N(f,g)^i, T(f,g)^i) / \forall (f,g) \in Xlist, N(f,g)^i = i(f,g)_k$ (port name), and
$T(f,g)^i \in T$ (port type)$\}$;

$\eta \in \boldsymbol{N}$ is the neighborhood size and **N** is the neighborhood set, defined as
$N = \{ (i_p,j_p) / \forall p \in \boldsymbol{N}, p \in [1,\eta] \Rightarrow i_p, j_p \in \boldsymbol{Z} \wedge i_p, j_p \in [-1, 1] \}$;
{m, n} $\in \boldsymbol{N}$ is the size of the cell space;
C defines the cell space, where $C = \{C_{ij} / i \in [1,m], j \in [1,n]\}$, with
$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij} >$$
is a Cell-DEVS component such as those defined in section 3;
B is the set of border cells, where

- $B = \{\emptyset\}$ if the cell space is wrapped; or
- $B = \{C_{ij} / \forall (i = 1 \vee i = m \vee j = 1 \vee j = n) \wedge C_{ij} \in C\}$, where
$$C_{ij} = < I_{ij}, X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij} >$$

is a Cell-DEVS component, such as those defined in section 3, if the atomic border cells have different behavior than the rest of the cell space.
Z is the translation function, defined by:
Z: $P_{kl}{}^{Y_q} \rightarrow P_{ij}{}^{X_q}$, where $P_{kl}{}^{Y_q} \in I_{kl}$, $P_{ij}{}^{X_q} \in I_{ij}$, $q \in [0,\eta]$ and $\forall(f,g) \in N$,
$k = (i+f) \bmod m$; $l=(j+g) \bmod n$;
$P_{ij}{}^{Y_q} \rightarrow P_{kl}{}^{X_q}$, where $P_{ij}{}^{Y_q} \in I_{ij}$, $P_{kl}{}^{X_q} \in I_{kl}$, $q \in [0,\eta]$ and $\forall(f,g) \in N$,
$k = (i-f) \bmod m$; $l = (j-g) \bmod n$;
select is the tie-breaking selector function, with $select \subseteq mxn \rightarrow mxn$.

The present definition only allows bidimensional cell spaces. The formal specification for n-dimensional spaces can be found in [11]. The coupled model includes an interface **I,** built using two lists. **Xlist** is the group of cells where the model's external events are received. **Ylist** includes the cells whose outputs will be collected to be sent to other models. The cell space **C** is a coupled model defined as a fixed size (**m x n**) array of atomic cells with η neighbors each. The neighborhood set (**N**) is represented by a list defining relative position between the neighbor and the origin cell . The present definition only allows regular neighborhoods in adjacent cells (other kinds of neighborhoods can be found in [11]).



Neighborhood: { (0, -1), (0,0) }
Inverse Neighborhood: {(0,1),(0,0)}

Note: -1: left, up; 1: right, down

$P_{ij}{}^{Y_1} \rightarrow P_{i,j+1}{}^{X_1}$ (1) $P_{ij}{}^{X_1} \leftarrow P_{i,j-1}{}^{Y_1}$ (1)

$P_{ij}{}^{Y_2} \rightarrow P_{ij}{}^{X_2}$ (2) $P_{ij}{}^{X_2} \leftarrow P_{ij}{}^{Y_2}$ (2)

(a) (b) (c)

Figure 5. (a) Neighborhood definition; (b) Output ports; (c) Input ports.

The B set defines the cell's space border. If it is empty, every cell in the space has the same behavior. The space is "wrapped", meaning that the cells in one border are connected with those in the opposite one using the neighborhood relationship. Otherwise, the border cells will have different behavior than those of the rest of the model.

Finally, the Z function allows to define the coupling of cells in the model. This function translates the outputs of the m-eth output port in cell C_{ij} into values for the m-eth input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood [22, 19]. The ports' names are generated using the following notation: $P_{ij}{}^{X_q}$ refers to the q-eth input port of cell C_{ij}, and $P_{ij}{}^{Y_q}$ to the q-eth output port. The cell to be coupled with is obtained by adding the cell's position with the values of the N set.

The definition for DEVS coupled models was extended to include Cell-DEVS, as follows:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

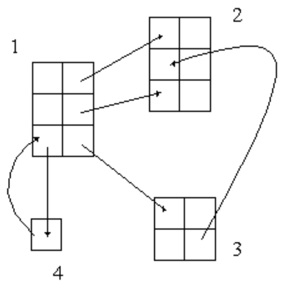
X is the set of input events;
Y is the set of output events;
D is an index for the components of the coupled model, and
$\forall i \in D$, $\mathbf{M_i}$ is a basic DEVS model, where

$$M_i = GCC_i = < I_i, X_i, Y_i, Xlist_i, Ylist_i, n_i, \{m, n\}_i, N_i, C_i, B_i, Z_i, select_i >$$

if the coupled model is Cell-DEVS, and

$$M_i = < I_i, X_i, S_i, Y_i, \delta_{inti}, \delta_{exti}, I_i >$$

otherwise.
$\mathbf{I_i}$ is the set of influencees of model i, and $\forall j \in I_i$, and
$\mathbf{Z_{ij}}$ is the i to j translation function, where
Z_{ij}: $Y_i \rightarrow X_j$ if none of the models involved are Cell-DEVS, or
Z_{ij}: $Y(f,g)_i \rightarrow X(k,l)_j$, with $(f,g) \in Ylist_i$, and $(k,l) \in Xlist_j$ if the models i and j are Cell-DEVS.
Finally, **select** is the tie-break selector.



(a)

$Xlist_1 = \{ (3,1) \}$
$Ylist_1 = \{ (1,2), (2,2), (3,2), (3,1) \}$
$Xlist_2 = \{ (1,1),(2,1),(3,1)\}$
$Ylist_2 = \{\varnothing\}$
$Xlist_3 = \{(1,1)\}$
$Ylist_3 = \{(2,2)\}$

(b)

$Y(1,2)_1 \rightarrow X(1,1)_2$
$Y(2,2)_1 \rightarrow X(3,1)_2$
$Y(3,2)_1 \rightarrow X(1,1)_3$
$Y(3,1)_1 \rightarrow X_4$
$Y_4 \rightarrow X(3,1)_1$
$Y(2,2)_3 \rightarrow X(2,1)_2$

(c)

Figure 6. Model interconnection (a) Basic models; (b) Xi and Yi lists for each model; (c) Zij coupling.

The specifications defined in this section allows to define complete cell spaces in a parametric fashion. A modeler only has to define the behavior for the local computing function and the duration and kind of the delay. After, a complete cell space is specified by defining the neighborhood shape, the size of the space, and the cells chosen as inputs/outputs for the cellular model. Finally, the connection with other models is defined using the translation lists. This approach reduces the development efforts for this kind of models, thus providing a simple framework to develop complex cellular spaces.

IMPLEMENTATION MODELS FOR DEVS-CELLS

This section is devoted to analyze several issues related with the development of implementation models following the conceptual framework recently introduced. A tool that allows the definition of those aspects was defined in [16, 1]. This environment was built using the conceptual definitions previously studied, and has been extended to execute n-dimensional Cell-DEVS [9].

This section presents two points of view related with the implementation models that can be built using the tools. First, a modeler should be able to define a conceptual model using the formal specifications previously defined. This specification should be executable, improving model's validation and verification. Besides, the point of view of the designer of a modeling environment is presented. Previous existing definitions for DEVS models' simulation have been used to define a Cell-DEVS environment.

Definition of Cell-DEVS implementation models.

The previous Cell-DEVS specifications shows that a modeler should be able to represent three basic aspects: **dimension** (size and shape of the cell space), **influencees** and **behavior**.

The **influencees** are specified by defining the cell's neighborhood. Input/output ports for each cell are created following the formal definitions. The cells are coupled applying the procedure presented earlier. When the cell space is to be coupled with other DEVS models, the external connections are specified using the contents of the $X_l$list and the $Y_l$list sets. The specification for the influencees is completed by defining the border cells (otherwise, the cell space is wrapped).

These aspects are defined using the following syntax:

- **Components**: it describes the DEVS and Cell-DEVS models integrating the coupled model. The format is **model_name@class_name**. The model name is used to allow more than one instance of the same atomic model, defined as an existing base model.
- **Out**: It describes the output ports names.
- **In**: It describes the input ports' names.
- **Link**: It describes the internal and external (input and output) coupling scheme. The format is: **origin_port[@model] destination_port[@model]**. The model name is optional and, if not included, it is considered as corresponding to the coupled model being specified.

Cell-DEVS specifications are completed by adding the following parameters:

- *type*: [cell | flat].
- *width*: INTEGER.
- *height*: INTEGER.
- *link*: in this case it must use the name of the cell space and the corresponding input/output cell (Model(x,y)).
- *border*: [WRAPPED | NOWRAPPED].
- *delay*: [TRASPORT | INERTIAL].
- *neighbors*: Cell-DEVS_name(x_1, y_1), ..., Cell-DEVS_name(x_n, y_n).
- *localTransition*: It defines the description for the behavior specification used for the local computation function.
- *zone*: transitionName {range_1..range_n}. It associates a behavior specification with the cells included into the rage defined by the sentence. In this way, different ranges can provide different behavior.

The remaining parameters of the specification are related with the cell's **behavior**. The local computing function is defined using a simple specification language (presented in [11]), and it is translated into an internal behavior representation for the space. The functions are built as a set of logical expressions, providing results from the present state of the cell and its neighborhood. As explained earlier, the timing behavior is specified by the duration and kind of the delay for the cell. The following figure presents a part of the specification language used to define the local transition functions. Further details will be presented in several examples in the following section.

```
rule : result delay { condition }
result: Float
delay: Float
condition:  The  condition  is  a  boolean  expression  using  the
following BNF grammar:
BoolExp :=  Relexp | NOT BoolExp | BoolExp AND BoolExp | BoolExp OR
BoolExp
Relexp := IdRef | Exp OpRel Exp
Exp :=  IdRef | Exp Oper Exp
IdRef :=  CellRef  | '(' BoolExp ')'  | Constant  | Function
Constant :=  Float | Int | Bool
Function := TRUECOUNT | FALSECOUNT | UNDEFCOUNT
CellRef :=   '(' Exp ',' Exp ')'
OpRel := = | != | > | < | >= | <=
Oper := + | - | * | /
Int := [Sign] Digit {Digit}
Float  :=  [Sign]  Digit  {Digit}  [.  Digit  {Digit}  [E  [Sign]  Digit
{Digit} ]  ]
Bool := 0 | 1 | t | f | ?
Sign := [+] | -
Digit ::= 0 | 1 | ... | 9
```

Figure 7. Basic syntax of the implemented specification language.

The specification of a cellular model is translated into an executable model. The local computing function scans the specification, verifying the logical expressions included and computing the new state value for the cell. Several errors of the specification can be found at runtime, allowing the detection of inconsistencies in the model definition:

- Ambiguous models: a cell with the same precondition can produce different results;
- Incomplete models: no result exists for a certain precondition;
- Non-deterministic models: different preconditions are satisfied simultaneously. If they produce the same result, the simulation can continue, but the modeler is notified. Instead, if different results are found, the simulation should stop because the future state of the cell cannot be determined.

Implementation of a Cell-DEVS simulation framework.

This section includes several aspects that should be considered by the developer of a Cell-DEVS simulation framework. This point of view will be exemplified considering our development experiences. A generic DEVS environment was first released [1], and it was extended to allow the simulation of timed cell spaces (using the ideas defined in [17] and [14]). As a final step, the tool was extended to include n-dimensional Cell-DEVS [9].

Basic structure of the tool.

The tool (called CD++) consists of a set of basic classes extending those defined in [20]. The two base classes, *Models* and *Processors* provide the constructors for DEVS models. The *Models* base class provides the basic methods to manage DEVS models. The *Atomic-Model* class is used to represent the behavior of atomic models, by using the *internalFunction*, *externalFunction*, and *outputFunction* methods. The *CoupledModel* class implements the hierarchical constructions. It is responsible to add and manage components, recording the dependencies between them.

The *Processors* implement the abstract simulation mechanisms. These classes should manage the connection between a *processor* with its corresponding *model* and its *parent* processor. *Simulators* and *Coordinators* are specializations that manage the activation of atomic and coupled models respectively. *Root Coordinator* represents the root of the processor's tree, and it is used to start and finish the simulation. It also should manage its global aspects including the maintenance of an *ExternalEvents* list and the global time (*clock*).

The tools follow the basic simulation mechanisms defined in [14], extending the basic classes presented. *AtomicCell* is a specialization of *Atomic* that represents the behavior of a cell and its delay function (transport or inertial). It must execute the local computing function (*localFunction*) depending on the *neighborhood* values. It also defines the interconnection with other models (*outport, NeighborPort*), and keeps the *value* and *delay* for the cell.

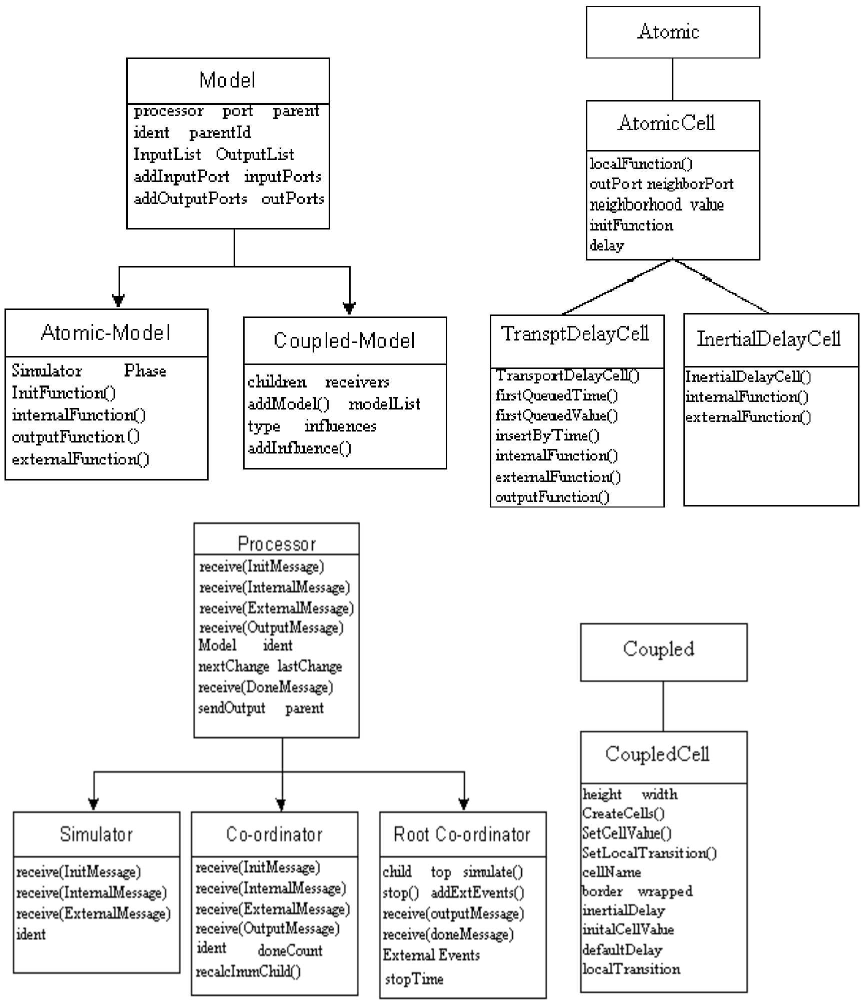


Figure 8. Basic classes defined by the tool [1].

The *CoupledCell* class is in charge of managing all the cells as children models. In a cell space, all the children of a coupled model are alike. Therefore, when the cells are created, the specified behavior is assigned to allow them and they are linked with the models defined by the neighborhood relationship.

A *CellCoordinator* class, specialized in the management of cell spaces has been defined. It is in charge of creating the set of processors associated with the cell space. As the coupled model consists of several atomic ones (the cells), one simulator is associated with each cell. The simulators are created and their names defined using the parameters defined earlier. The definitions of the input/output lists are used to couple the output ports in the interface of a Cell-DEVS with the input ports in the other. The internal coupling is set up as presented earlier, and a coordinator is associated with each coupled model.

CD++ message interaction

Inter-process interaction is carried out through message passing. Each message includes information of the source (or destination), the event simulated time, and a content (consisting of a port and a value). *Message* is the base class that defines the different messages. There are four different messages: * (internal event), X (external event), Y (model's output), and done (a model has finished with its task).

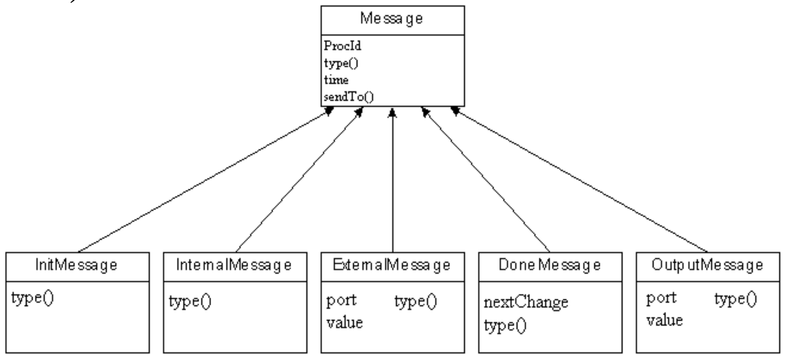


Figure 9. Message class hierarchy.

The basic ideas of message interaction defined in [19, 20] have been extended to allow the simulation of Cell-DEVS models. When an external message arrives, a X-message is consumed and the external transition function executed. The local computing function is activated, and its output is delayed, scheduling and internal transition. In this case, the imminent model will be a cell in the space. When a *-message is received by the cell space coordinator, the imminent cell is selected. Then, the simulator associated with that cell activates the model's output and internal transition functions, executing the procedures presented earlier.

The simulators return done-messages and Y-messages that are converted to new *-messages and X-messages, respectively. These messages are translated using the coupling mechanisms previously defined. The main task of the cell space coordinator is to translate the outputs in inputs by using the internal coupling and the external lists.

Support for N-dimensional models

The real systems that can be studied using cellular models are usually represented by using models in two or three dimensions. Several theoretical problems can be defined as cellular models with four or more dimensions. The original version of the tool only allowed to define two-dimensional cell spaces, constraining its use in more general problems. An extension, following the formal specification of [11], allowed the definition of n-dimensional models.

CD++ was implemented by storing the cell states in a two-dimensional array of $d_1 \cdot d_2$, where the element (x_1, x_2), $x_i \in [0, d_i -1]$, is in the position $x_1 + x_2 \cdot d_1$. In an analogous fashion, N-CD++ uses an array of $\Pi_{i=1\ldots n} d_i$ to store the states for the cellular automata with dimension $(d_1, d_2, \ldots, d_n)$, and in this case $(x_1, x_2, \ldots, x_n)$ occupies the position $\Sigma_{i=1\ldots n}\ x_i \cdot (\Pi_{k=1\ldots i-1} d_k)$.

The specification language was adapted to include references to cells in n-dimensional cell spaces. The definition of *zones* in the cell space was extended. Each zone now is defined by a set of cells determined by the cell range $\{(x_1, x_2, \ldots, x_n)\ldots(y_1, y_2, \ldots, y_n)\}$. Using this capability, different zones into the same cellular model can present different behavior.

Flat simulation of the cell spaces.

DEVS allows hierarchical module definition, and the proposed simulation mechanism has hierarchical nature. Therefore, intermodule communication produces a high degree of overhead, moreover in Cell-DEVS simulations consisting of a large number of cells. One way to avoid this interaction (reducing the related overhead) is by flattening the Processor's hierarchy. This is the goal of the *Flat_Coordinator* class.

Here, the message passing overhead is avoided by creating a unique processor including the values for the cell space, and executing a specialized simulation method [14]. Therefore, the *FlatCoupledCell* creates a set of cells and records the local computing function for each one.

The *FlatCoordinator* is in charge of the flat execution of the cell spaces. This coordinator is implemented as a bidimensional array of records associated with the cell space. Each record includes information of the state, delay, and a neighborhood list for the cell. When this approach is used, multiple intermediate processors are eliminated, as it can be seen in the Figure 11.Each cell in theflat simulation mechanism can use inertial or transport delays.

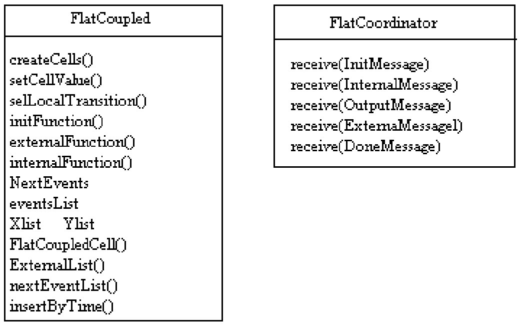


Figure 10. Flat models definition.

A Next-Events list records the cells scheduled to execute their transition functions. A cells' array is used to record the present cell's space state and to detect changes in the model. When a change is detected, the Next-Events list is updated. The results produced by the imminent cells are stored in a New-States list. In this way, the flow of the global transition function of the cell space can be reproduced.

A flat coordinator starts the simulation by detecting quiescent states for the initial state of the cell space. Non-quiescent cells (v.g., those whose state can change) are added in the Next-Events list. When an instance of the *FlatCoordinator* class receives a X-message, it is also inserted in this list. After, the coordinator removes the first element of the Next-Events list, and it invokes the *externalFunction* method.

The coupled models still compute the local transition function for the desired position, and apply the delay algorithm, changing the Next-Events list. If the cell is in the Ylist, the coordinator will create a Y-message containing the cell state value, and will transmit it to the upper level coordinator. When it is finished, it sends a *DoneMessage* to the parent coordinator with the event date of the next imminent child. When an *internalMessage* is received, the *FlatCoordinator* invokes to the *internalFunction* method of the coupled model. If the cell state has changed, its value is returned and the next event time for the cell is computed. This one iterates through all the imminent cells, and generates their output, adding all the influenced cells in the next-events list. Then, the *externalFunction* method is executed for all the virtual cells in this list.

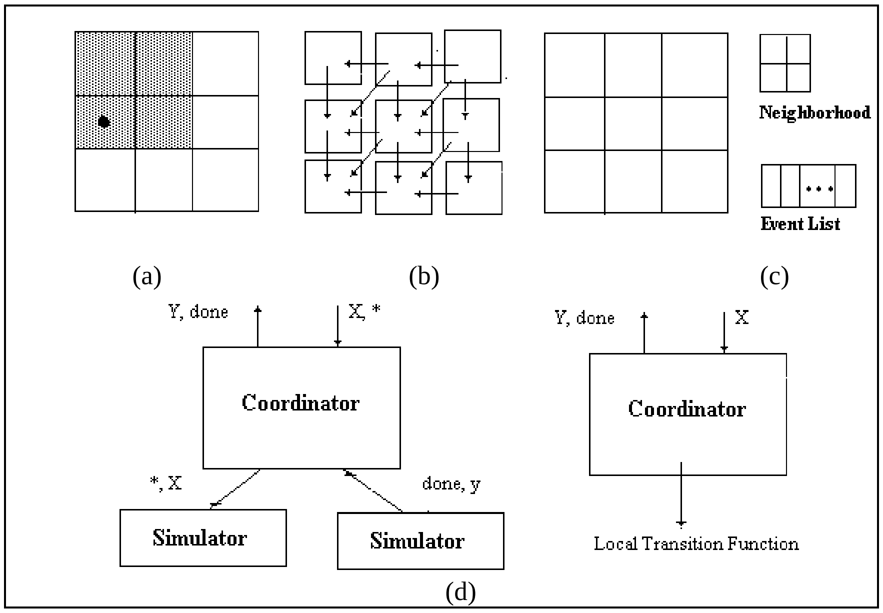


Figure 11. Cell space structures. (a) Basic cell space and its neighborhood. (b) Coupling using hierarchical coordinators. (c) Definition using flat models. (d) Coordinator's reaction to messages.

DEVELOPMENT EXPERIENCES WITH CELLULAR MODELS

This section will focus on some development experiences done with Cell-DEVS. The first one, presented in the following figure, shows one specification for the Life game [5] using the tool CD++. Here, the coupled model consists only of one Cell-DEVS, called "Life". The parameters of the specification defined earlier are included here: model's dimension, kind and length of the delay, shape of the neighborhood, and the local transition function. The local function says that a cell can have a live/dead (1/0) state. A living cell remains alive only if it has three or four living neighbors. Otherwise, it dies. Instead, a new born cell appears when there are exactly three living neighbors of a dead cell.

```
[top]
components : life

[life]
width : 10                    height : 10
delay : inertial              defaultDelayTime : 100
border : wrapped              localtransition : life-rule
neighbors : life(-1,-1) life(-1,0) life(-1,1) life(0,-1) life(0,0)
neighbors : life(0,1) life(1,-1)  life(1,0) life(1,1)

[life-rule]
rule : 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) }
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { t }
```

Figure 12. Life game specification using CD++.

The following figure shows the model's execution using inertial delays. The first row of the figure presents the execution of the model given an initial configuration. Instead, the second row shows the influence of inputs to certain cells using inertial delays. It can be seen that, for instance, a new value is inserted in the position (8,7) in simulated time 15. Therefore, the internal events scheduled for 20 in the cells (7, 7) and (8, 6) are preempted, and these cells die. This happens because these cells have now five living neighboring cells. Therefore, as an inertial delay is being used, the state changes are preempted.

```
Time: 00:00:10:000      Time: 00:00:20:000      Time: 00:00:30:000
+--------------------+  +--------------------+  +--------------------+
|         *          |  |                    |  |         *          |
|         *          |  |       * * *        |  |       * * *        |
|       *   *        |  |     *       *      |  |     * * * * *      |
|     * * * * *      |  |   *           *    |  |   * *       * *    |
| * *   *   *   * *  |  |   *           *    |  | * * *       * * *  |
|     * * * * *      |  |   *           *    |  |   * *       * *    |
|       *   *        |  |     *       *      |  |     * * * * *      |
|         *          |  |       * * *        |  |       * * *        |
|         *          |  |                    |  |         *          |
|                    |  |                    |  |                    |
+--------------------+  +--------------------+  +--------------------+
Time: 00:00:15:000      Time: 00:00:20:000      Time: 00:00:25:00
+--------------------+  +--------------------+  +--------------------+
|         *          |  |                    |  |                    |
|         *          |  |       * * *        |  |       * * *        |
|       *   *        |  |     *       *      |  |     *       *      |
|     * * * * *      |  |   *   *       *    |  |   *           *    |
| * *   *   *   * *  |  |   *       *   *    |  |   *           *    |
|     * * * * *      |  |   *           *    |  |   *           *    |
|       *   *        |  |     *              |  |     *              |
|         *   *      |  |       * *   *      |  |       * *          |
|         *          |  |                    |  |                    |
|               *    |  |               *    |  |                    |
+--------------------+  +--------------------+  +--------------------+
```

Figure 13. Life game execution with inertial delays.

The following figure shows that extensions to three dimensional models can be easily implemented. This example is an extension of the previous one, using a population of active cells distributed in an area of 7x7x3. Likewise, extensions to models of higher dimensions can be defined. We can see that the coupled model is specified by the dimension, kind of delay and neighborhood (3x3x3 adjacent cells). The local function defines a new born being when the cell has more than 9 living neighbors. A cell remains alive when the neighborhood contains 8 or 10 living neighbors. Otherwise, the cell dies.

```
[3d-life]
type : cell           dim : (7,7,3)          border : wrapped
delay : transport      defaultDelayTime : 100
neighbors: (-1,-1,-1) (-1,0,-1) (-1,1,-1) (0,-1,-1) (0,0,-1) ...
neighbors: (1,-1,1) (1,0,1) (1,1,1)

[3d-life-rule]
rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
rule : 0 100 { t }
```

Figure 14. Description of a variation of the Life game.

The following figure shows the results obtained when this model is executed (each plane separately). The execution starts with a high number of living cells, but the execution result is not stable. The number of living cells turns to be reduced, and, finally, in the instant 00:00:01:000, the population is extinguished.

```
Time: 00:00:00:000                   Time: 00:00:00:100
   0123456     0123456     0123456       0123456     0123456      0123456
  +-------+   +-------+   +-------+     +-------+   +-------+    +-------+
 0|*      |  0|       |  0|*      |    0| *    *|  0|**    *|   0| *    *|
 1|* *  **|  1|**   **|  1|  ***  |    1|* *   *|  1|*     *|   1|* **  *|
 2| *   * |  2|   ** *|  2| * **  |    2|**  * *|  2|*    * |   2|**   **|
 3|       |  3|  *  **|  3|     **|    3|    ***|  3|  * * *|   3|    * *|
 4|  *  **|  4|  * *  |  4| *   **|    4|       |  4|     **|   4|       |
 5|  **  *|  5|   * * |  5| **  * |    5|*  *** |  5|* *** *|   5|*  ** *|
 6|*  *  *|  6| *   * |  6| * ** *|    6|       |  6| *     |   6| *  *  |
  +-------+   +-------+   +-------+     +-------+   +-------+    +-------+
Time: 00:00:00:200                   Time: 00:00:00:900
   0123456     0123456     0123456       0123456     0123456      0123456
  +-------+   +-------+   +-------+     +-------+   +-------+    +-------+
 0|*     *|  0|      *|  0|*     *|    0|       |  0|       |   0|       |
 1| **  * |  1| *   * |  1| **  * |    1|       |  1|       |   1|       |
 2|    ** |  2| *    *|  2|       |    2|       |  2|       |   2|       |
 3|*   *  |  3|*   ** |  3|*   ** |    3|       |  3|       |   3|       |
 4|   ****|  4|   **  |  4|   ****|    4|*    * |  4|*    **|   4|*    * |
 5|*  *   |  5|*  *  *|  5|*  *  *|    5|       |  5|       |   5|       |
 6|**   **|  6|**   **|  6|**  ***|    6|       |  6|       |   6|       |
  +-------+   +-------+   +-------+     +-------+   +-------+    +-------+
```

Figure 15. Execution results for the modified Life game.

The next example represents a three dimensional heat diffusion model. Each cell contains a temperature value, computed as the average of the values of the neighborhood. In addition, a heater is connected to the cells (2,2,1) and (3,3,0). On the other hand, a cooler is connected to the cells (1,3,3) and (3,3,2).

```
[top]
components : room Heater@Generator Cooler@Generator
link : out@Heater HeatInput@room
link : out@Cooler ColdInput@room

[Heater] [Cooler]
distribution : exponential           mean : 10

[room]
type : cell           dim : (4, 4, 4)          border : wrapped
delay : transport     defaultDelayTime : 100
neighbors : room(-1,0,-1) room(0,-1,-1)
neighbors : room(0,0,-1) room(0,1,-1)
...
neighbors : room(0,0,-2)  room(0,0,2) room(0,2,0)
neighbors : room(0,-2,0)  room(2,0,0) room(-2,0,0)
initialvalue : 24
in : HeatInput ColdInput
link : HeatInput in@room(3,3,0) in@room(2,2,1)
link : ColdInput in@room(3,3,2)
link : ColdInput in@room(1,3,3)
localtransition : heat-rule
portInTransition : in@room(3,3,0) in@room(2,2,1) setHeat
portInTransition : in@room(3,3,2) in@room(1,3,3) setCold

[heat-rule]
Rule: {( (-1,0,-1)+(0,-1,-1)+(0,0,-1)+ (0,1,-1) + (1,0,-1) +
       (-1,-1,0) + (-1,0,0) + (-1,1,0)  + (0,-1,0) +(0,0,0)+
       (0,1,0)+(1,-1,0)+(1,0,0) + (1,1,0) + (-1,0,1) + (0,-1,1) +
       (0,0,1)+(0,1,1)+(1,0,1)+(0,0,-2)+(0,0,2)+(0,2,0)+
       (0,-2,0)+(2,0,0) + (-2,0,0) ) / 25 } 1000 { t }

[setHeat]
rule : { uniform(24,80) } 1000 { t }

[setCold]
rule : { uniform(-45,10) } 1000 { t }
```

Figure 16. Definition of the heat diffusion model.

Here, the upper level model is composed by three basic components: the room, a heater and a cooler. The last two models are DEVS, defined as random generators. The heater simulator generates a flow of temperatures between 24º C and 80º C with uniform distribution. The cooler creates random values with uniform distribution in the range [-45, 10]. Both generators create values every *x* time units, where *x* has exponential distribution with mean of 10 time units. The model representing the room is composed by a Cell-DEVS of 10x10x4 cells. The function computes the present value for the cell as an average of the neighbors. The model has two input ports (HeatInput, ColdInput) connected to the input ports of the corresponding cells. Whenever a value is received through these ports, the *portInTransition* rules are activated. Here, *setHeat* generates a temperature value in the range [24, 80]. Likewise, *setCold* generates temperatures in the range [-45, 10]. The values of the corresponding cells will be updated using these functions.

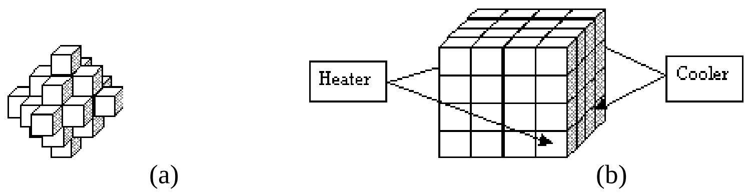
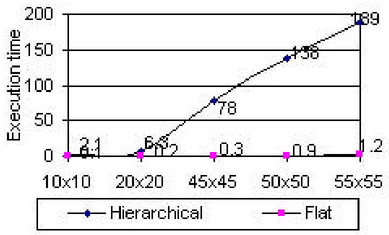
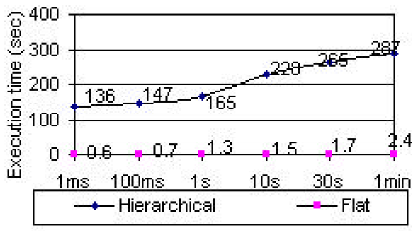


Figure 17. Heat diffusion model. (a) Neighborhood shape. (b) Coupling scheme.

These examples show the implementation of the formalism: the models are specified following a formal description, and the implementation models executes them. The hierarchical and flat simulation mechanisms produced different execution performance for these examples. The number of messages involved in the flat simulation is reduced because interaction only occurs between the higher level coordinator and the Root coordinator. The following figures show the differences for both cases. Those results were obtained for the Life game, and a one-way traffic model, but similar behavior was obtained for most implemented models. The test starts with more than 75% of active cells. The first test shows the influence of increasing the size of the cell space. The second test used a fixed size space (2500 cells), and the duration of the simulation was increased.





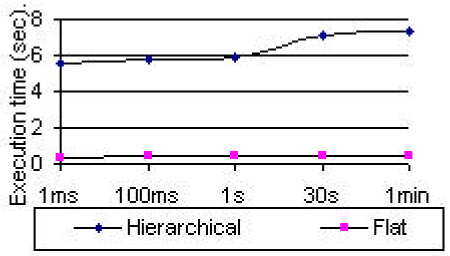


Figure 18. Life game and traffic model, increasing the size and simulation length.

A main goal of the new formalism was to reduce the development times for the simulators. The results obtained were promising though the developed experiences were simple prototypes used to check the use of the tools and the formalism. Several data were recorded relating the development times for the different solutions, classifying the different users and their development activities. These results are presented following.

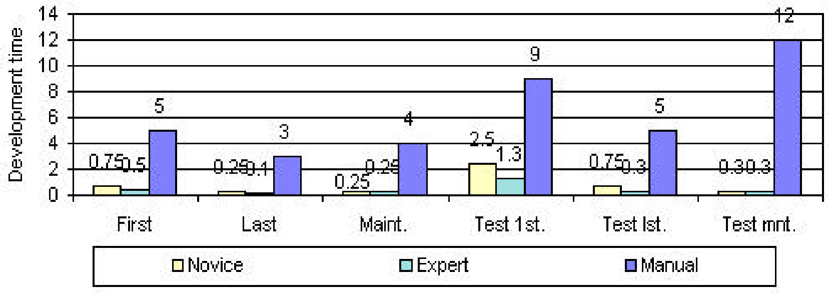


Figure 19. Comparison of development times for the Life game.

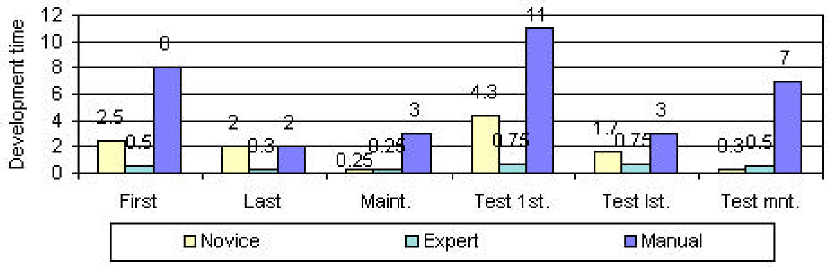


Figure 20. Comparison of development times for the traffic simulation.

The development activities were classified according with the experience of the modelers, and the kinds of activities being considered. The use of the tool was compared to the development of the same problems by hand. Several groups of developers were analyzed, and their first and last developed applications were recorded. The maintenance times for the applications was also registered, considering development times and testing times sepparately.

It can be seen that the results obtained highly improved the development times of the simulations. The main gains have been reported in the testing and maintenance phases, the most expensive for these systems. It also showed performance improvements for the flat models, providing speedups from 2 to 7 times in the execution for the cellular models. Ten-fold improvements could be achieved for expert users although the tested models were simple prototypes of simple cellular models.

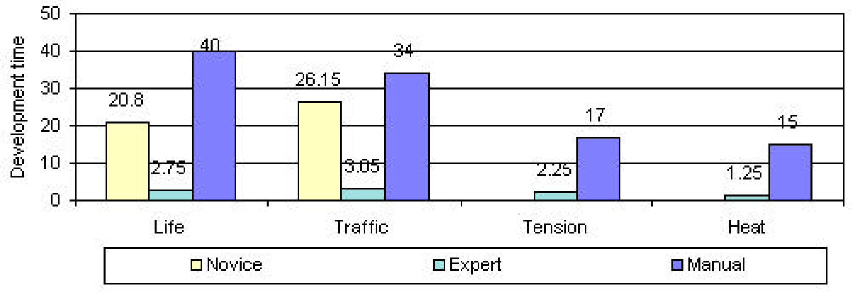


Figure 21. Comparison between total development times for different applications.

PRESENT WORK

This section briefly presents a set of the results derived from the initial specifications of Cell-DEVS models.

Parallel Cell-DEVS

As stated in [2], if we call **e** to the elapsed time since the occurrence of an event, a model can exist in the DEVS structure at either e=0 or e=D(s). A modeler can use the *select* function to solve the conflicts of simultaneous scheduled events in coupled models. In these cases, ambiguity arises when a model scheduled for an internal transition receives an event. The problem here is how to determine which of both elapsed times should be used. The *select* function solves the ambiguity by choosing only one of the imminent models. This is a source of potential errors, because the serialization may not reflect the simultaneous occurrence of events. Moreover, the serialization reduces the possible exploitation of parallelism among concurrent events.

These problems were solved defining the Parallel DEVS [2]. Cell-DEVS models could be coupled with these traditional DEVS submodels. Also, it was proved that a bag is needed for the Cell-DEVS with zero-time transitions [12]. Considering these factors, the Cell-DEVS models were redefined to include parallel behavior [15].

A general environment for parallel simulations was built, considering the use of optimist/pessimist synchronization approaches. A mapping between the Cell-DEVS simulators and these algorithms were defined, and at present, they are being used to build a parallel extension of the tools. After, an extension to the HLA standard will be faced. This approach will enhance the production of results. By introducing a parallel coordinator, execution speedups of several orders of magnitude can be achieved without touching the specifications. Furthermore, a parallel implementation of each problem hand-coded could produce ten-fold delays in the development.

Extensions to the cell's specification language

At present, several modifications were done to the specification language used to define the cell's behavior. The first one is related with the topology of the cellular models. At present, the defined models include rectangular meshes. Nevertheless, several existing cellular models have triangular or hexagonal patterns. Therefore, the tool is being extended to run these approaches. In a first stage, a translator is being used to define rectangular rules derived from the triangular and hexagonal ones. Then, the extended topologies will be included in the tool.

A set of new delay constructions has been defined [12]. These ones allow to define complex timing behavior for inertial delays, improving the definition of timing behavior for cellular models. These constructions are being combined with the new definitions of parallel Cell-DEVS. The new local confluent functions should be included in the specification language, allowing the modeler to define the cell's behavior under simultaneous events. The simulation mechanisms are being extended to include a theory of quantized DEVS models [23].

The specification of cellular models can be impoved using a specialized graphical interface. The models will be defined using a graphical extension of the specification language, and a graphical output will be defined. These tools will be integrated in a web-based simulation framework, providing a Cell-DEVS simulation server.

Applications

Several applications are being considered to apply Cell-DEVS models. A first group of models are related with the definition of physical problems. The first ones include surface tension analysis, lattice gases and studies of echological systems. Some of the results obtained can be seen in the following figure. A second group of applications include the analysis of crystal growth [10]. In the latter case, different isotropies will be studied, including triangular, rectangular and hexagonal meshes. Cellular models has also been used to create chaotic patterns to be used in one-time-pad applications in cryptography. The kind of pattern obtained can be seen in the following figure. The cellular models are also being considered as fields of neurons, to be applied as a workbench to build artificial neural networks.

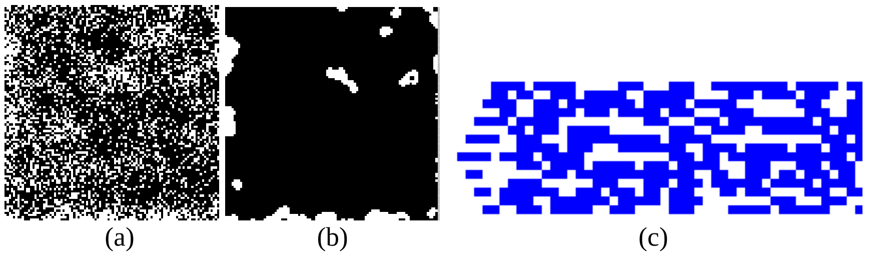


Figure 22. Surface tension models (a: initial; b: final); chaotic pattern for cryptographic application (c).

Several multimodel applications are being faced. First, a detailed study of development times is being extended to consider the integration of mixed DEVS and Cell-DEVS models. The goal is to characterize the development activities and the cost reductions in more complex applications. Also, multidimensional models are being studied. They are being used to represent different aspects of the same three-dimensional system as a cubic zone of a higher dimensional space.

A final group of applications include a specification language used to define traffic simulations as cell spaces. The streets can be defined, analyzing the traffic direction, number of tracks, etc. Once the urban section is defined, the traffic flow is automatically set up. Therefore, a modeler can concentrate in the problem to solve, instead of being in charge of defining a complex simulation system.

A city section is specified by a set of streets connecting two crossings. The vehicles advance in a right line (surpassing slower vehicles), up to their arrival to a crossing. The speed of each vehicle is represented through a random transport delay. Each street is represented as a sequence of segments. These represent a section of one block of length, where every track has the same traffic direction (one way). Consequently, to build a two-way street is necessary to define one segment for each direction. Several models were defined, depending on the number of lanes, their direction, and maximum speeds in the streets.

Once defined the basic behavior for a city section, different components can be defined. These are also part of the specification language, and include definitions for traffic lights, railways, men at work, street holes, transit signals, parked cars, and so on. Finally, special behavior has been defined for special vehicles: trucks, vans and high priority cars (ambulances, policemen, firefighters) [4, 3].

CONCLUSION

The present work presented a description for cell spaces modeling and simulation. The paradigm is based on the DEVS and Asynchronous Cellular Automata formalisms, using transport or inertial delays. These concepts allows to specify complex behavior in a simple fashion, independently of the quantitative complexity of the models.

Cell-DEVS models were described formally, considering specifications for one cell and for general Cell-DEVS coupled models. The methods presented allow automatic definition of cell spaces using the DEVS formalism. Integration of multiple views for each submodel is possible, letting to combine different models in an efficient fashion. The use of a formal approach allowed to prove properties regarding the cellular models. It also provided a sound basis to build simulation tools related with the formal specifications.

One of the main contributions is related with the definition of complex timing behavior for the cells in the space using very simple constructions. Transport and inertial delays allow the modeler to make easier the timing representation of each cell in the space.

An implementation of the paradigms was presented. The modeler and the developer point of view were considered, allowing efficient and cost-effective development of simulators. Two descriptions for the simulation mechanisms were included. The first one considered a hierarchical simulation mechanism. Subsequently, a method to flatten the hierarchical description of the cell spaces was given.

The approach here presented also permits including a parallel coordinator, achieving execution speedups of several orders of magnitude without changes in the specifications. Therefore, this approach can provide important reduction in the implementation of parallel applications for cellular models.

Finally, the formalism allow to improve the security and cost in the development of the simulations. As shown by the experimental application results, the formalism showed ten-fold improvements for expert developers.

ACKNOWLEDGEMENTS

This work was partially funded by ANPCYT Project 11-04460 and UBACYT Projects TX04 and JW10.

REFERENCES

1. BARYLKO, A.; BEYOGLONIAN, J.; WAINER, G. "GAD: a General Application DEVS environment". *Proceedings of IASTED Applied Modelling and Simulation 1998*. Hawaii, U.S.A.
2. CHOW, A.; ZEIGLER, B. "Revised DEVS: a parallel, hierarchical, modular modeling formalism". *Proceedings of the SCS Winter Simulation Conference*. 1994.
3. DAVIDSON, A.; WAINER, G. "Specifying control signals in traffic models". *Proceedings of AIS'2000*. Tucson, Arizona. U.S.A. 2000.
4. DAVIDSON, A.; WAINER, G. "Specifying truck movement in traffic models using Cell-DEVS". *Proceedings of the 33rd Annual Conference on Computer Simulation*. Washington, D.C. U.S.A. 2000.
5. GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'.". *Scientific American*. 23 (4). pp. 120-123. April 1970.
6. GHOSH, S.; GIAMBIASI, N. "On the need for consistency between the VHDL language constructions and the underlying hardware design". *Proceedings of the 8th. European Simulation Symposium*. Genoa, Italy. Vol. I. pp. 562-567. 1996.
7. GIAMBIASI, N.; MIARA, A. "SILOG: A practical tool for digital logic circuit simulation". *Proceedings of the 16th. D.A.C.*, San Diego, U.S.A. 1976.
8. MOON, Y.; ZEIGLER, B.; BALL, G.; GUERTIN, D. P. "DEVS representation of spatially distributed systems: validity, complexity reduction". *IEEE Transactions on Systems, Man and Cybernetics*. pp. 288-296. 1996.
9. RODRIGUEZ, D.; WAINER, G. "New Extensions to the CD++ tool". *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1999.
10. TOFFOLI, T.; MARGOLUS, N. "Cellular Automata Machines". *The MIT Press*, Cambridge, MA. 1987.
11. WAINER, G. “Discrete-events cellular models with explicit delays”. Ph.D. Thesis, Université d'Aix-Marseille III. 1998.
12. WAINER, G. "Improved cellular models with parallel Cell-DEVS". To appear in *Transactions of the Society for Computer Simulation*. 2000.
13. WAINER, G.; BARYLKO, A.; BEYOGLONIAN, J.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modelling and simulation.". *Internal Report. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.* Submitted for publication. 1998.
14. WAINER, G.; GIAMBIASI, N. "Specification, modeling and simulation of timed Cell-DEVS spaces". *Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.* 1998.
15. WAINER, G.; GIAMBIASI, N. “Specification of timing delays in parallel Cell-DEVS models”. *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1999.

16. WAINER,G.; FRYDMAN, C.; GIAMBIASI, N. "An environment to simulate cellular DEVS models". *Proceedings of the SCS European Multiconference on Simulation*. Istanbul, Turkey. 1997.
17. WAINER,G.; GIAMBIASI, N.; FRYDMAN, C. "Cell-DEVS models with transport and inertial delays". *Proceedings of the 9th. European Simulation Symposium and Exhibition*. Passau, Germany. 1997.
18. WOLFRAM, S. “Theory and applications of cellular automata”. Vol. 1, Advances Series on Complex Systems. World Scientific, Singapore, 1986.
19. ZEIGLER, B. "Multifaceted Modeling and discrete event simulation". Academic Press, 1984.
20. ZEIGLER, B. "Object-oriented simulation with hierarchical modular models". Academic Press, 1990.
21. ZEIGLER, B. "Object-oriented simulation with hierarchical modular models. Revised to include source code for DEVS-C++." *Department of Electrical and Computer Engineering. University of Arizona.* 1995.
22. ZEIGLER, B. "Theory of modeling and simulation". Wiley, 1976.
23. ZEIGLER, B. P., CHO, H.; LEE, J.; SARJOUGHIAN, H. *The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering*. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.