The Cellular Automata Formalism
and its Relationship to DEVS

Hans L.M. Vangheluwe[†,‡] and Ghislain C. Vansteenkiste[‡]

[†] School of Computer Science
McGill University, Montréal, Québec, Canada
e-mail: hv@cs.mcgill.ca

[‡] Department of Applied Mathematics,
Biometrics, and Process Control (BIOMATH)
Ghent University (RUG), Gent, Belgium

Keywords: multi-formalism modelling, simulation, Cellular Automata, DEVS

Abstract

Cellular automata (CA) were originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behaviour of complex, spatially distributed systems. Cellular Automata constitute a *dynamic, discrete space, discrete time* formalism where space is usually discretized in a grid of cells. Cellular automata are still used to study, from first principles, the behaviour of a plethora of systems.

The Discrete EVent system Specification (DEVS) was first introduced by Zeigler in 1976 as a rigourous basis for discrete-event modelling. At the discrete-event level of abstraction, state variables are considered to change only at event-times. Furthermore, the number of events occurring in a bounded time-interval must be finite. The semantics of well known discrete-event formalisms such as Event Scheduling, Activity Scanning, and Process Interaction can be expressed in terms of DEVS, making it a common denominator for the representation of discrete-event models.

Due to its roots in traditional sequential formalisms, DEVS offers little potential for parallel implementation. Furthermore, conflicts between simultaneous internal state transitions and external events are resolved by ignoring the internal transitions. In 1996, Chow introduced the parallel DEVS (P-DEVS) formalism which alleviates these drawbacks.

Since the end of 1993, the European Commission's ESPRIT Basic Research Working Group 8467 (SiE), has identified key areas for future research in modelling and simulation. The most urgent need is to support multi-formalism modelling to correctly model complex systems with components described in diverse formalisms. To achieve this goal, formalisms need to be related. The Formalism Transformation Graph (FTG) shown in Figure 1 depicts behaviour-conserving transformations between formalisms.

In this article, both the Cellular Automata and the DEVS and parallel DEVS formalisms are introduced. Then, a mapping between Cellular Automata and parallel DEVS is elaborated. This fills in the $CA \rightarrow DEVS$ edge in the FTG. The mapping describes CA semantics in terms of parallel DEVS semantics. As such, it is a specification for automated transformation of CA models into parallel DEVS models, thus enabling efficient, parallel simulation as well as meaningful coupling with models in other formalisms.

1 The Cellular Automata Formalism

Cellular automata (CA) were originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behaviour of complex, spatially distributed systems [11]. Cellular Automata constitute a *dynamic, discrete space, discrete time* formalism. Space in Cellular Automata is partitioned into discrete volume elements called *cells* and time progresses in discrete steps. Each cell can be in one of a finite number of states at any given time. The "physics" of this logical universe is *deterministic* and *local*. *Deterministic* means that once a local physics and an initial state of a Cellular Automaton has been chosen, its future evolution is *uniquely* determined. *Local* means that the state of a cell at time $t+1$ is determined only by its own state and the states of neighbouring cells at the previous time t. The *operational semantics* of a CA as prescribed in a *simulation procedure* and implemented in a CA solver dictates that values are updated *synchronously*: all new values are calculated simultaneously.

The local physics is typically determined by an *explicit mapping* from all possible local states of a predefined neighbourhood template (*e.g.,* the cells bordering on a cell, including the cell itself), to the state of that cell after the next time-step. For example, for a 2-state ($\{0,1\}$), 1-D Cellular Automaton with a neighbourhood template that includes a cell and its immediate neighbours to the left and right, there will be 2^3 possible neighbourhood states $\{000,\ldots,111\}$. For each of these, we must prescribe whether a transition of the center cell state to a 1 or to a 0 will occur. For an 8-state nearest neighbour 2-D Cellular Automaton, there will be 8^5 possible neighbourhood states, and a choice of 8 states to map to for each of those.

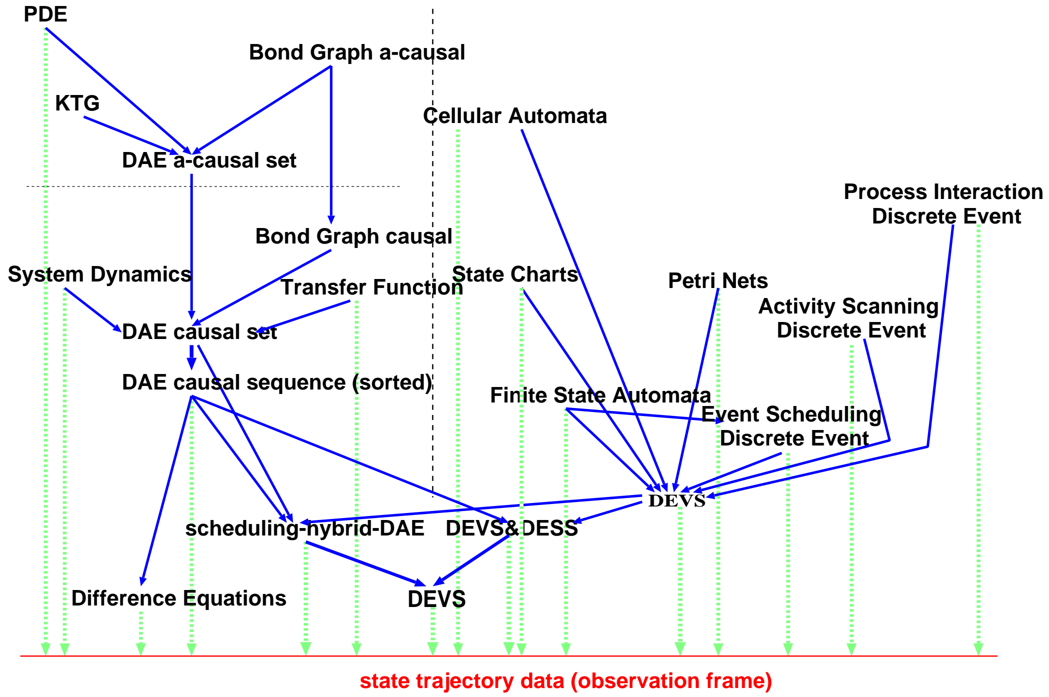

Figure 1: Formalism Transformations

The Cellular Automata formalism CA fits the general structure of deterministic, causal systems in classical systems theory [13, 16]:

$$CA \equiv \langle T, X, \Omega, S, \delta, Y, \lambda \rangle.$$

The formalism is specified by elaboration of the elements of the CA 7-tuple, as described below:

The *discrete* time base T:

$$T = \mathbb{N} \text{ (or isomorphic with } \mathbb{N}).$$

The input set X:

$$X = \{TIME_TICK\}.$$

The Cellular Automata formalism can easily be extended to non-trivial inputs (see further comments on extensions).

The set of all input segments ω:

$$\Omega.$$

An input segment ω may be restricted to a domain ($\subseteq T$) such as $]n, n+1]$:

$$\begin{aligned} \omega &: T \rightarrow X, \\ \omega_{]n,n+1]} &:]n, n+1] \rightarrow X. \end{aligned}$$

The state set S is the product of all the *finite* state sets V_i (also called *cell value sets*) of the individual cells. C is the *cell index set*.

$$S = \times_{i \in C} V_i.$$

In the usual case of Cellular Automata realized on a D-dimensional grid, C consists of D-tuples of indices from a coordinate set I:

$$C = I^D.$$

In the standard Cellular Automata formalism, the cell space is assumed *homogeneous*: all cell value sets are identical:

$$\forall i \in C, V_i = V.$$

The cell value function v maps a cell index i onto its value $v(i)$:

$$v : C \rightarrow V.$$

The total transition function δ is constructed from the transition functions δ_i for each of the cells.

$$\begin{aligned} \delta &: \Omega \times S \rightarrow S, \\ & (\omega_{]n,n+1]}, \times_{i \in C} v(i)) \rightarrow \times_{i \in C} \delta_i(i). \end{aligned}$$

The *uniformity* of Cellular Automata requires the δ_i to be based on a *single* local transition function δ_l for all cells i:

$$\forall i \in C, \delta_i(i) = \delta_l(v(\sigma_{NT}(i))),$$

where the various operators and quantities are explained below.

A neighbourhood template NT, a vector of *finite* size ξ containing *offsets* from an "origin" 0, encodes the relative positions of neighbouring cells influencing the future state of a cell. Usually, the *nearest (adjacent)* neighbours (including the cell itself) are used. For one-dimensional Cellular Automata, a cell is connected to r local neighbours (cells) on either side, where r is a parameter referred to as the *radius*

(thus, there are $2r+1$ cells in each neighbourhood). For two-dimensional Cellular Automata, two types of neighbourhoods are usually considered:

- The *von Neumann* neighbourhood of radius r

$$NT = \{(k,l) \in C |\ |k| + |l| \leq r\}.$$

  For $r = 1$, this yields 5-cells, consisting of a cell along with its four immediate non-diagonal neighbours.
- The *Moore* neighbourhood of radius r

$$NT = \{(k,l) \in C |\ |k| \leq r \text{ and } |l| \leq r\}.$$

  For $r = 1$, this yields 9-cells, consisting of the cell along with its eight surrounding neighbours.

The *local* nature of the Cellular Automata models lies in the fact that *only* near neighbours influence the behaviour of a state, *not all* cells as the general form of δ allows. From a modelling point of view, physical arguments in disciplines ranging from physics to biology and artificial life support this assumption [12]. From a simulation point of view, efficient solvers for the Cellular Automata formalism can be constructed which take advantage of locality. Note how the $NT[j]$ are offsets between C elements which are again elements of C (with an appropriate C such as $\mathbb{N}^D$).

$$NT \in C^{\xi}.$$

The function σ_{NT} *shifts* the neighbourhood template NT to be centered over i:

$$\begin{array}{lllll} \sigma_{NT} & : & C & \rightarrow & C^{\xi}, \\ & & i & \rightarrow & \tau \\ & & & & \text{where } \forall j \in \{1,\ldots,\xi\} : \tau[j] = i + NT[j]. \end{array}$$

For all possible combinations of size ξ of cell values, the local transition function δ_l prescribes the transition to a new value:

$$\begin{array}{llll} \delta_l & : & V^{\xi} & \rightarrow & V, \\ & & [v_1,\ldots,v_{\xi}] & \rightarrow & v_{new}. \end{array}$$

Thanks to the uniformity of Cellular Automata, the same δ_l is used for each element of the cell space. The number of possible combinations of cell values is $\#V^{\xi}$ and the number of distinct results of δ_l is $\#V$ ($\#$ is the cardinality function).

In the above, δ was constructed for an elementary time advance $n \rightarrow n+1$. The composition (or semigroup) requirement for deterministic system models:

$$\forall t_x \in [t_i, t_f]; s_i \in S,$$

$$\delta(\omega_{[t_i,t_f]}, s_i) = \delta(\omega_{[t_x,t_f]}, \delta(\omega_{[t_i,t_x]}, s_i))$$

is satisfied by *construction* (*i.e.,* the definition, combined with iteration over n).

It is possible to "observe" the Cellular Automaton by projecting the total state S onto an output set Y by means of an output function λ

$$\lambda : S \rightarrow Y,$$

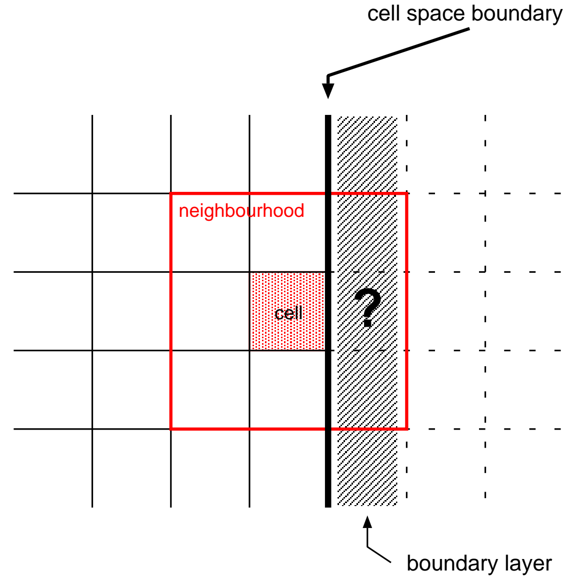

Figure 2: Boundary Conditions for Finite Cell Space

where Y commonly has a structure similar to that of S.

We shall now discuss the *structure* of the cell space. Usually, a D-dimensional *grid*, centered around the origin, is used. Most common choices are $D \in \{1,2,3\}$. In one dimension, a linear array of cells is the only possible geometry for the grid. In two dimensions, there are three choices:

- **triangular grid**: has a small number of nearest neighbours but is hard to visualize on square grid oriented computers.
- **square grid**: is easy to visualize, but (computationally) *anisotropic* (*i.e.,* a wave propagates faster along the primary axes than along the diagonals).
- **hexagonal grid**: has the lowest anisotropy of all grids but computer visualization is hard to implement.

Arbitrary cell space structures are possible (and corresponding cell shapes when visualizing), though not practical.

Although the above mathematical formalism is perfectly valid, it can not be simulated *in practice*. For simulation to become possible, the cell space needs to be *finite*. In particular, the cell index set C must be finite. L, the length of the grid becomes finite, leading to a cell space of L^D cells.

When a finite cell space is used, the application of the transition function at the edges poses a problem as values are needed outside the cell space. As shown in Figure 2 for a 2-D cell space, *boundary conditions* need to be specified in the form of cell values *outside* the cell space. Two common approaches –also used in the specification and solution of partial differential equations [4]– are

- **explicit** boundary conditions. Extra cells outside the perimeter of the cell space hold boundary values (*i.e.,* C and v are extended). The number of extra border

cells is determined by the size of (the perimeter of) the cell space as well as by the size (and shape) of the neighbourhood template. Boundary conditions may be time varying.

- **periodic** boundary conditions. The cell space is assumed to "wrap around": the cells at opposite ends of the grid act as each other's neighbours. In 1-D, this results in a (2-D) circle, in 2-D, in a (3-D) torus. By construction, the boundary conditions are time-varying.

1.1 Implementation of a CA Solver

1.1.1 The Solver Structure

In Figure 3 the backbone algorithm of any cellular automata

$\forall i \in C$: initialise $v(i)$ {Initialize Cell Space}
if explicit boundary conditions **then**
  $\forall i \in$ boundary(C): initialize $v(i)$ {Boundary extension of $v()$}
end if
if periodic boundary conditions **then**
  $\forall i \in C \cup$ boundary(C): $v(i) \leftarrow v(i \bmod L)$ {Modulo extension of $v()$; assume $0 \ldots L-1$ indexing}
end if
for $n := n_s$ to n_f **do**
  $\forall i \in C$: $v_{new}(i) \leftarrow \delta_l(v(\sigma_{NT}(i)))$ {One-step state transition computation}
  $v \leftarrow v_{new}$ {Switch value buffers}
  $n \leftarrow n+1$ {Time Advancement}
end for

Figure 3: CA Simulation Procedure

solver is given.

As the definition requires synchronous calculation, whereby new values only depend on old values (and not on new values) of neighbouring cells, a second value function v_{new} is needed to hold copies of the previous value. Note how a value function is usually efficiently implemented as a lookup in a value *array*.

1.2 Examples

The Santa Fe Institute [1] hosts a plethora of Cellular Automata examples. The site is mainly devoted to the study of Artificial Life, one of the prominent uses of Cellular Automata. Artificial Life research tries to explain and reproduce, ab-initio, many physical and biological phenomena.

1.2.1 Simple 2-state, 1-D Cellular Automaton

Figure 4 demonstrates the simulation procedure for a simple 2-state ($\{0,1\}$), 1-D Cellular Automaton of length 4 with periodic Boundary Conditions and initial condition 1011. The local transition function (a 1-D version of the

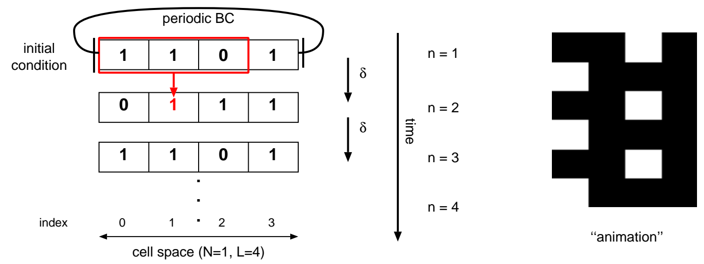

Figure 4: 2-state, 1-D Cellular Automaton of length 4

```
# 2 dimensional game of life

2 dimensions of 0..1

sum := [ 0,  1] + [ 1,  1] + [ 1,  0] +
       [-1,  1] + [-1,  0] + [-1, -1] +
       [ 0, -1] + [ 1, -1]

cell := 1 when (sum = 2 & cell = 1) | sum = 3
     := 0 otherwise
```

Figure 5: "cellang" specification of Conway's game of Life

Game of Life) is

$$\delta_l: \quad \begin{array}{ll} 000 \rightarrow 0 & 100 \rightarrow 0 \\ 001 \rightarrow 0 & 101 \rightarrow 1 \\ 010 \rightarrow 0 & 110 \rightarrow 1 \\ 011 \rightarrow 1 & 111 \rightarrow 0 \end{array}$$

For a 1-D Cellular Automaton, "animation" of the cell space can be visualized by colour coding the cell values (here: 0 by white and 1 by black) and by mapping t onto the vertical axis which leads to the 2-D image in Figure 4.

1.2.2 The Game of Life

Developed by Cambridge mathematician John Conway and popularized by Martin Gardner in his Mathematical Games column in Scientific American in 1970 [5], the game of Life is one of the simplest examples of a Cellular Automaton. Each cell is either alive (1) or dead (0). To determine its status for the next time step, each cell counts the number of neighbouring cells which are alive. If the cell is alive and has 2 or 3 alive neighbours, then the cell is alive during the next time step. With fewer alive neighbours, a living cell dies of loneliness, with more, it dies of overcrowding. Many interesting patterns and behaviours have been investigated over the years. An example of a high-level *cellang* [3] specification (*i.e.,* δ_l is written implicitly) of the Cellular Automata model is given in Figure 5. In combination with boundary conditions and an initial condition, this specification allows for model solving.

1.3 Formalism Extensions

The Cellular Automata formalism can easily be extended in different ways:

1. Addition of *inputs*. The formalism as presented above is *autonomous*: there are no (non-trivial) inputs into the system. An intuitively appealing way of adding inputs is to associate an input with each cell. The input set X will thus have a structure similar to that of the state set S.

2. The requirement of having the same cell value set for each cell can be relaxed to obtain *heterogeneous* Cellular Automata whereby not necessarily $\forall i \in C : V_i = V$. The homogeneous case can always be emulated by constructing V as the union of all individual cell value sets:
$$V = \bigcup_{i \in C} V_i.$$

3. The requirement of having the same local transition function δ_l for each cell can be relaxed to obtain *non-uniform* Cellular Automata.

4. As is demonstrated in Figure 5, a modelling language may allow for high-level representations. Agents are a typical example of such a high-level construct. Here, δ_l is no longer specified explicitly.

The "grid of cells" discretized representation of space can also be used for continuous models. In particular, the local dynamics of cells can be described by System Dynamics models rather than finite state automata. Simulation will be carried out by transforming the model to a flat DAE model and subsequently (numerically) solving that continuous model, given initial conditions.

2 The DEVS Formalism

The DEVS formalism was conceived by Zeigler [14, 15] to provide a rigorous common basis for discrete-event modelling and simulation. For the class of formalisms denoted as *discrete-event* [7], system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-span, only a *finite number* of relevant *events* occur. These events can cause the state of the system to change. In between events, the state of the system does *not* change. This is unlike *continuous* models in which the state of the system may change continuously over time.

Just as Cellular Automata, the DEVS formalism fits the general structure of deterministic, causal systems in classical systems theory mentioned in section 1. DEVS allows for the description of system behaviour at two levels. At the lowest level, an *atomic DEVS* describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input (events) and how it generates output (events). At the higher level, a *coupled DEVS* describes a system as a *network* of coupled components. The components can be atomic DEVS models or coupled DEVS in their own right. The connections denote how components influence each other. In particular, output events of one component can become, via a network connection, input events of another component. It is shown in [14] how the DEVS formalism is *closed under coupling*: for each coupled DEVS, a *resultant* atomic DEVS can be constructed. As such, any DEVS model, be it atomic or coupled, can be replaced by an equivalent atomic DEVS. The construction procedure of a resultant atomic DEVS is also the basis for the implementation of an *abstract simulator* or solver capable of simulating any DEVS model. As a coupled DEVS may have coupled DEVS components, *hierarchical* modelling is supported.

In the following, the different aspects of the DEVS formalism are explained in more detail.

2.1 The atomic DEVS Formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system:
$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle.$$
The *time base* T is continuous and is not mentioned explicitly
$$T = \mathbb{R}.$$
The *state set* S is the set of admissible *sequential states*: the DEVS dynamics consists of an ordered sequence of states from S. Typically, S will be a *structured* set (a product set)
$$S = \times_{i=1}^{n} S_i.$$
This formalizes multiple (n) *concurrent* parts of a system. It is noted how a structured state set is often synthesized from the state sets of concurrent components in a coupled DEVS model.

The time the system *remains* in a sequential state before making a transition to the next sequential state is modelled by the *time advance function*
$$ta : S \rightarrow \mathbb{R}^{+}_{0,+\infty}.$$
As time in the real world always advances, the image of ta must be non-negative numbers. $ta = 0$ allows for the representation of *instantaneous* transitions: no time elapses before transition to a new state. Obviously, this is an abstraction of reality which may lead to simulation *artifacts* such as infinite instantaneous loops which do not correspond to real physical behaviour. If the system is to stay in an end-state s *forever*, this is modelled by means of $ta(s) = +\infty$.

The internal transition function
$$\delta_{int} : S \rightarrow S$$
models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; ta adds the progression of time.

It is possible to *observe* the system output. The output set Y denotes the set of admissible *outputs*. Typically, Y will be a *structured* set (a product set)

$$Y = \times_{i=1}^{l} Y_i.$$

This formalizes multiple (l) output ports. Each port is identified by its unique index i. In a user-oriented modelling language, the indices would be derived from unique port *names*.

The output function

$$\lambda : S \rightarrow Y \cup \{\phi\}$$

maps the internal state onto the output set. Output events are *only* generated by a DEVS model at the time of an *internal* transition. At that time, the state *before* the transition is used as input to λ. At all other times, the non-event ϕ is output.

To describe the *total state* of the system at each point in time, the sequential state $s \in S$ is not sufficient. The *elapsed* time e since the system made a transition to the current state s needs also to be taken into account to construct the total state set

$$Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$$

The elapsed time e takes on values ranging from 0 (transition just made) to $ta(s)$ (about to make transition to the next sequential state). Often, the *time left* σ in a state is used:

$$\sigma = ta(s) - e$$

Up to now, only an *autonomous* system was described: the system receives no external inputs. Hence, the *input set* X denoting all admissible input values is defined. Typically, X will be a *structured* set (a product set)

$$X = \times_{i=1}^{m} X_i$$

This formalizes multiple (m) input ports. Each port is identified by its unique index i. As with the output set, port indices may denote *names*.

The set Ω contains all admissible input segments ω

$$\omega : T \rightarrow X \cup \{\phi\}$$

In discrete-event system models, an input segment generates an input *event* different from the *non-event* ϕ only at a finite number of instants in a bounded time-interval. These *external events*, inputs x from X cause the system to interrupt its autonomous behaviour and react in a way prescribed by the external transition function

$$\delta_{ext} : Q \times X \rightarrow S$$

The reaction of the system to an external event depends on the sequential state the system is in, the particular input *and* the elapsed time. Thus, δ_{ext} allows for the description of a large class of behaviours typically found in discrete-event models (including synchronization, preemption, suspension and re-activation).

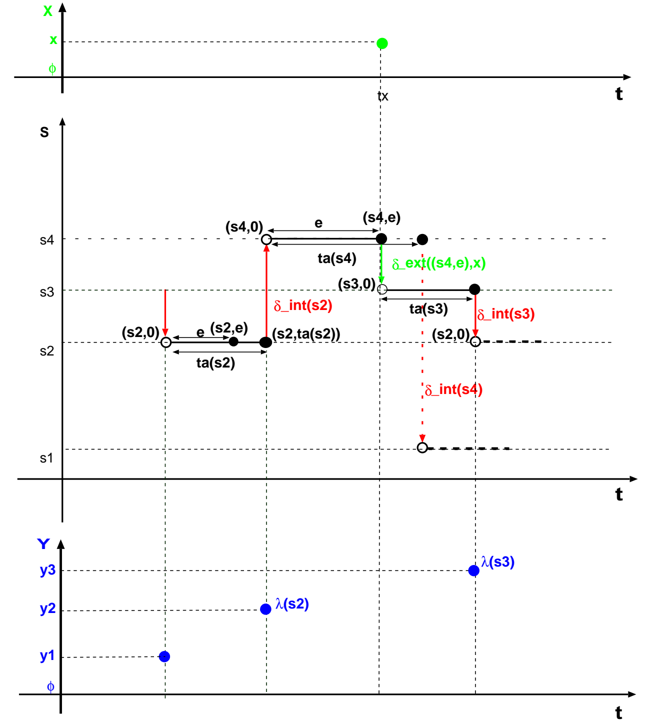

Figure 6: State Trajectory of a DEVS-specified Model

When an input event x to an atomic model is not listed in the δ_{ext} specification, the event is *ignored*.

In Figure 6, an example state trajectory is given for an atomic DEVS model. In the figure, the system made an internal transition to state $s2$. In the absence of external input events, the system stays in state $s2$ for a duration $ta(s2)$. During this period, the elapsed time e increases from 0 to $ta(s2)$, with the total state $= (s2,e)$. When the elapsed time reaches $ta(s2)$, first an output is generated: $y2 = \lambda(s2)$, then the system transitions instantaneously to the new state $s4 = \delta_{int}(s2)$. In autonomous mode, the system would stay in state $s4$ for $ta(s4)$ and then transition (after generating output) to $s1 = \delta_{int}(s4)$. Before e reaches $ta(s4)$ however, an external input event x arrives. At that time, the system forgets about the scheduled internal transition and transitions to $s3 = \delta_{ext}((s4,e),x)$. Note how an external transition does not give rise to an output. Once in state $s3$, the system continues in autonomous mode.

2.2 The coupled DEVS Formalism

The coupled DEVS formalism describes a discrete-event system in terms of a network of coupled components.

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

The component $self$ denotes the coupled model itself. X_{self} is the (possibly structured) set of allowed external inputs to the coupled model. Y_{self} is the (possibly structured) set of

allowed (external) outputs of the coupled model. D is a set of unique component references (names). The coupled model itself is referred to by means of $self$, a unique reference not in D.

The set of *components* is

$$\{M_i | i \in D\}.$$

Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

The set of *influencees* of a component, the components influenced by $i \in D \cup \{self\}$, is I_i. The set of all influencees describes the coupling network structure

$$\{I_i | i \in D \cup \{self\}\}.$$

For modularity reasons, a component (including $self$) may not influence components outside its scope –the coupled model–, rather only other components of the coupled model, or the coupled model $self$:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}.$$

This is further restricted by the requirement that none of the components (including $self$) may influence themselves directly as this could cause an instantaneous dependency cycle (in case of a 0 time advance inside such a component) akin to an algebraic loop in continuous models:

$$\forall i \in D \cup \{self\} : i \notin I_i.$$

Note how one can always encode a self-loop ($i \in I_i$) in the internal transition function.

To translate an output event of one component (such as a departure of a customer) to a corresponding input event (such as the arrival of a customer) in influencees of that component, *output-to-input translation functions* $Z_{i,j}$ are defined:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$\begin{aligned} Z_{self,j} &: X_{self} \to X_j \quad &&, \forall j \in D, \\ Z_{i,self} &: Y_i \to Y_{self} \quad &&, \forall i \in D, \\ Z_{i,j} &: Y_i \to X_j \quad &&, \forall i, j \in D. \end{aligned}$$

Together, I_i and $Z_{i,j}$ completely specify the coupling (structure and behaviour).

As a result of coupling of concurrent components, multiple state transitions may occur at the same simulation time. This is an artifact of the discrete-event abstraction and may lead to behaviour not related to real-life phenomena. A logic-based foundation to study the *semantics* of these artifacts was introduced by Radiya and Sargent [9]. In sequential simulation systems, such transition *collisions* are resolved by means of some form of *selection* of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages. The coupled DEVS formalism explicitly represents a *select* function for *tie-breaking* between simultaneous events:

$$select : 2^D \to D$$

select chooses a unique component from any non-empty subset E of D:

$$select(E) \in E.$$

The subset E corresponds to the set of all components having a state transition simultaneously.

2.3 Implementation of a DEVS Solver

The algorithm in Figure 7 is based on the closure under coupling construction and can be used as a specification of a –possibly parallel– implementation of a DEVS solver or "abstract simulator" [14, 6]. In an atomic DEVS solver, the last event time t_L as well as the local state s are kept. In a coordinator, only the last event time t_L is kept. The next-event-time t_N is sent as output of either solver. It is possible to also keep t_N in the solvers. This requires consistent (recursive) initialization of the t_Ns. If kept, the t_N allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator involves handling four types of messages. The $(x, from, t)$ message carries external input information. The $(y, from, t)$ message carries external output information. The $(*, from, t)$ and $(done, from, t_N)$ messages are used for scheduling (synchronizing) the abstract simulators. In these messages, t is the simulation time and t_N is the next-event-time. The $(*, from, t)$ message indicates an internal event $*$ is due.

When a coordinator receives a $(*, from, t)$ message, it selects an imminent component i^* by means of the tie-breaking function *select* specified for the coupled model and routes the message to i^*. Selection is necessary as there may be more than one imminent component (with minimum next remaining time).

When an atomic simulator receives a $(*, from, t)$ message, it generates an output message $(y, from, t)$ based on the old state s. It then computes the new state by means of the internal transition function. Note how in DEVS, output messages are only produced while executing internal events. When a simulator outputs a $(y, from, t)$ message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation, to each of the influencees of i^* (if any). If the coupled model itself is an influencee of i^*, the output, after appropriate output-to-output translation, is sent to the the coupled model's parent coordinator.

When a coordinator receives an $(x, from, t)$ message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components.

When an atomic simulator receives an $(x, from, t)$ message, it executes the external transition function of its associated atomic model.

After executing an $(x, from, t)$ or $(y, from, t)$ message, a simulator sends a $(done, from, t_N)$ message to its parent co-

message m	**simulator**	**coordinator**
$(*, from, t)$	simulator correct only if $t = t_N$	
	$y \leftarrow \lambda(s)$ **if** $y \neq \phi$: **send** $(\lambda(s), self, t)$ to *parent* $s \leftarrow \delta_{int}(s)$ $t_L \leftarrow t$ $t_N \leftarrow t_L + ta(s)$ **send** $(done, self, t_N)$ to *parent*	**send** $(*, self, t)$ to i^*, where $i^* = select(imm_children)$ $imm_children = \{i \in D \mid M_i.t_N = t\}$ $active_children \leftarrow active_children \cup \{i^*\}$
$(x, from, t)$	simulator correct only if $t_L \leq t \leq t_N$ (ignore δ_{int} to resolve a $t = t_N$ *conflict*)	
	$e \leftarrow t - t_L$ $s \leftarrow \delta_{ext}(s, e, x)$ $t_L \leftarrow t$ $t_N \leftarrow t_L + ta(s)$ **send** $(done, self, t_N)$ to *parent*	$\forall i \in I_{self}$: **send** $(Z_{self,i}(x), self, t)$ to i $active_children \leftarrow active_children \cup \{i\}$
$(y, from, t)$		$\forall i \in I_{from} \setminus \{self\}$: **send** $(Z_{from,i}(y), from, t)$ to i $active_children \leftarrow active_children \cup \{i\}$ **if** $self \in I_{from}$: **send** $(Z_{from,self}(y), self, t)$ to *parent*
$(done, from, t)$		$active_children \leftarrow active_children \setminus \{from\}$ **if** $active_children = \emptyset$: $t_L \leftarrow t$ $t_N \leftarrow min\{M_i.t_N \mid i \in D\}$ **send** $(done, self, t_N)$ to *parent*

Figure 7: DEVS Simulation Procedure

$t \leftarrow t_N$ of topmost coordinator
repeat until $t = t_{end}$
  send $(*, meta, t)$ to topmost coupled model top
  wait for $(done, top, t_N)$
  $t \leftarrow t_N$

Figure 8: DEVS Simulation Procedure Main Loop

ordinator to prepare a new schedule. When a coordinator has received $(done, from, t_N)$ messages from all its components, it sets its next-event-time t_N to the minimum t_N of all its components and sends a $(done, from, t_N)$ message to its parent coordinator. This process is recursively applied until the top-level coordinator or *root coordinator* receives a $(done, from, t_N)$ message.

As the simulation procedure is synchronous, it does not support a-synchronously arriving (real-time) external input. Rather, the environment or Experimental Frame should also be modelled as a DEVS component.

To run a simulation experiment, the *initial conditions* t_L and s must first be set in *all* simulators of the hierarchy. If t_N is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop described in Figure 8 is executed.

3 The parallel DEVS Formalism

As DEVS is a formalization and generalization of sequential discrete-event simulator semantics, it does not allow for drastic parallelization. In particular, simultaneously occurring internal transitions are serialized by means of a tie-breaking *select* function. Also, in case of *collisions* between simultaneously occurring internal transitions and external input, DEVS ignores the internal transition and applies the external transition function. Chow [2] proposed the parallel DEVS (P-DEVS) formalism which alleviates some of the DEVS drawbacks. In an atomic P-DEVS

$$atomic\ P-DEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, \delta_{conf}, Y, \lambda \rangle,$$

the model can explicitly define collision behaviour by using a so-called *confluent* transition function δ_{conf}.

Only δ_{ext}, δ_{conf}, and λ are different from DEVS.

The external transition function

$$\delta_{ext} : Q \times X^b \rightarrow S$$

now accepts a *bag* of simultaneous inputs, elements of the input set X, rather than a single input.

The confluent transition function

$$\delta_{conf} : S \times X^b \rightarrow S$$

describes the state transition when a scheduled internal state transition and simultaneous external inputs collide.

An atomic P-DEVS model can generate multiple simultaneous outputs

$$\lambda : S \rightarrow Y^b$$

in the form of a bag of output values from the output set Y.

As conflicts are handled explicitly in the confluent transition function, the *select* tie-breaking function can be eliminated from the coupled DEVS structure:

$$coupled\ P-DEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle.$$

In this structure, all components M_i are atomic P-DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, \delta_{conf,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

For the proof of closure under coupling of P-DEVS as well as for the description of an efficient parallel abstract simulator, see [2].

4 Formalism Transformation

Cellular Automata are a simple form of discrete-event models, of DEVS in particular. Describing a Cellular Automaton as a simple atomic DEVS is thus straightforward. Thanks to the general nature of DEVS, all extensions of the Cellular Automata formalism mentioned before can also be mapped onto DEVS.

It is more rewarding however to map a CA onto a coupled model, whereby every CA cell's dynamics is represented as an atomic model and the dependency between one cell and its neighbourhood is represented by the coupled model's coupling information. In what follows, a CA is mapped onto a coupled P-DEVS as this mapping is more elegant than that onto the original DEVS. In addition, the resultant parallel DEVS holds more potential for parallel implementation. The coupled parallel DEVS representation presented here, corresponds to the Cellular Automaton specification in section 1:

$$P-DEVS-CA = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} \rangle.$$

As the Cellular Automaton in section 1 did not incorporate external input,

$$X_{self} = \emptyset.$$

Typical output of the Cellular Automaton consists of all the cell values

$$Y_{self} = \times_{i \in D} V.$$

Components in the coupled parallel DEVS model correspond to CA cells. Indexing uses the CA index set:

$$D = C.$$

The $\{M_i\}$ are atomic P-DEVS components, described in more detail later.

The set of influencees of a component/cell i is constructed by means of the CA's neighbourhood template NT. The NT contains influencer rather than influencee information.

Thus, the offset information needs to be mirrored with respect to the origin (*i.e.,* its inverse with respect to addition of offsets needs to be calculated) to obtain the influencees of component i:

$$I_i = \{j \in C | j = i - offset,\ offset \in \{NT[k] | k = 1, \ldots, \xi\} \setminus \{0\}\}$$

As DEVS does not allow $i \in I_i$, the offset 0, if present in NT, is not included. A state transition of a CA cell usually does depend on the old state of the cell itself. This is encoded in the atomic P-DEVS's (confluent) transition function rather than by means of an external self-coupling. Note how $j = i - offset \in C$ may need to be relaxed depending on the particular boundary conditions. Cells outside C may need to be considered (and initialized).

As there is no external input, the input-to-input translation $Z_{self,i}$ is not needed.

The i, j output-to-input translation converts the output of a cell (*i.e.,* the cell's value) into a tuple containing that value and the offset between the two cells:

$$\begin{array}{rcl} Z_{i,j} : & Y_i & \rightarrow & X_j \\ & s_i & \rightarrow & (s_i, i - j). \end{array}$$

The output-to-output translation translates the output of each cell into a tuple with the output in the position corresponding to the cell's index:

$$\begin{array}{rcl} Z_{i,self} : & Y_i & \rightarrow & Y_{self} \\ & s_i & \rightarrow & (\ldots, s_i, \ldots). \end{array}$$

Individual cells are mapped onto atomic P-DEVS components:

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, \delta_{confl,i}, Y_i, \lambda_i \rangle, \forall i \in D.$$

Values of the components/cells are those from the cell value set

$$S_i = V.$$

The time advance is set to the same arbitrary non-zero value Δ for all cells to allow for synchronous operation:

$$ta_i(s_i) = \Delta.$$

The internal transition function does not modify the component's state

$$\delta_{int}(s_i) = s_i.$$

The output function sends out a set containing only the cell value:

$$\lambda(s_i) = \{s_i\},\ \text{where } s_i \in Y_i = V.$$

The external transition function is not used as there is no global external input into the CA. Due to the synchronous operation, whereby internal transitions and external inputs will always collide, *only* the confluent transition function is used.

$$\delta_{conf,i}(s_i, e_i, x_i^b) = \delta_l(v_i),$$

where v_i is a vector with the same dimensions as the neighbourhood template NT with values

$$\begin{array}{rcl} v_i[\eta] & = & s_i,\ \text{for the } \eta \text{ for which } NT[\eta] = 0; \\ v_i[proj_{offset}(x)] & = & proj_{value}(x),\ \forall x \in x_i^b. \end{array}$$

Messages communicate values of neighbouring cells, as well as offsets from the current cell:

$$X_i^b = \{(v, offset) | v \in V, offset \in C\}.$$

As an example of the above mapping, Murato coded the Game of Life CA in his `a-DEVS-0.2` [8] parallel DEVS implementation.

5 Conclusions

Since the end of 1993, the European Commission's ESPRIT Basic Research Working Group 8467 (SiE) [10], has identified key areas for future research in modelling and simulation. The most urgent need was deemed for multi-formalism modelling, to correctly model complex systems with components described in diverse formalisms. To correctly model such systems, formalisms need to be related. The Formalism Transformation Graph (FTG) shown in Figure 1 depicts meaningful mappings between formalisms. In this article, a mapping between Cellular Automata and parallel DEVS was elaborated. This fills in the $CA \rightarrow DEVS$ edge in the FTG. The mapping describes CA semantics in terms of parallel DEVS semantics and is a basis for automated transformation of CA models into parallel DEVS models, thus enabling efficient, parallel simulation as well as meaningful coupling with models in other formalisms.

References

[1] CAweb. Artificial life. Cellular automata website, The Santa Fe Institute, 1999. `http://alife.santafe.edu/alife/topics/ca/caweb/`.

[2] A.C.-H. Chow. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International*, 13(2):55–68, June 1996.

[3] J. Dana Eckart. A *cellular* automata simulation system. Technical report, Radford University, Radford, VA 24242, August 1998. `http://www.cs.runet.edu/~dana/ca/tutorial.ps`.

[4] Stanley J. Farlow. *Partial Differential Equations for Scientists and Engineers*. Dover Publications, New York, 1993.

[5] Martin Gardner. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[6] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical

DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, September 1996.

[7] Richard E. Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

[8] James Nutaro. aDEVS-0.2. C++ library for parallel DEVS, University of Arizona, Tucson, 1999. www.ece.arizona.edu/~nutaro/.

[9] Ashvin Radiya and Robert G. Sargent. A logic-based foundation of discrete event modeling and simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):3–51, 1994.

[10] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. SiE: Simulation for the Future. *Simulation*, 66(5):331 – 335, May 1996.

[11] John von Neumann. *Theory of Self-reproducing Automata*. University of Illinois Press, 1966. Edited and completed by Arthur W. Burks.

[12] Stephen Wolfram. *Theory and applications of cellular automata*. World Scientific, 1986.

[13] A. Wayne Wymore. *A Mathematical Theory of Systems Engineering – the Elements*. Wiley Series on Systems Engineering and Analysis. Wiley, 1967.

[14] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[15] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Robert E. Krieger, Malabar, Florida, 1984.

[16] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition, 2000.