Asymmetric Cell-DEVS Models with the Cadmium Simulator

Román Cárdenas[ab] and Gabriel Wainer[b]

[a]*Laboratorio de Sistemas Integrados, Universidad Politécnica de Madrid, Madrid, Spain;* [b]*Systems and Computer Engineering, Carleton University, Ottawa, Canada.*

CORRESPONDING AUTHOR: Román Cárdenas. Email: r.cardenas@upm.es

ABSTRACT

Cellular models provide a natural approach for describing the behavior, dynamics, and structure of natural systems with spatial features. The classic Cell-DEVS formalism allows us to define discrete-event cellular models as a lattice of cells in which cells states are updated asynchronously over a continuous time base with explicit timing delays. Here, we present asymmetric Cell-DEVS, a generalization that supports the definition of cellular models with irregular topologies that allows us to model more complex scenarios. We include a prototype implementation of formalism using the Cadmium simulator, and illustrate the use of this formalism and tools by presenting different cellular models combined with geographical data to provide realistic results.

KEYWORDS: cellular models; discrete-event modeling and simulation; DEVS

1. Introduction

Numerous natural systems show great complexity in their behavior, dynamics, and structure, making it difficult to describe them mathematically (Wolfram, 1984). Cellular models provide simple abstractions of these systems, enabling the study of geographical and temporal changes on them (Janelle, 2005). Among the different approaches for implementing cellular models, Cellular Automata (CA) (Toffoli and Margolus, 1987) is one of the most popular methods. CA is a discrete-time formalism that is typically conceived as a lattice of cells. Each cell behaves as a Finite State Machine (FSM) and computes its next state according to its previous state and the previous state of nearby cells (i.e., the cell neighborhood). CA has been successfully used in multiple disciplines, such as chemistry (Gerhardt and Schuster, 1989), biology (Hatzikirou et al., 2012), or generative arts (Ashlock and Kreitzer, 2020).

In CA, cells state transitions are computed synchronously in a discrete-time base, constraining the precision of the simulations. Furthermore, multiple cells do not transition to new states in every step, compromising the computation efficiency of the simulated models. Additionally, the CA formalism is not easy to combine with other approaches (Wainer and Giambiasi, 2001). The Cell-DEVS formalism (Wainer, 2009) combines CA with the Parallel Discrete Event System Specification (PDEVS) (Zeigler et al., 2000) to overcome these issues. PDEVS is a revision of the original Discrete Event System Specification (DEVS). PDEVS is currently the prevailing DEVS variant. In the following, the use of the DEVS word implies PDEVS formal specifications.

Cell-DEVS allows us to define discrete-event cellular models in which cells states are updated asynchronously over a continuous time base, removing unnecessary computations and increasing the accuracy of the results. Furthermore, as Cell-DEVS is defined on top of the DEVS formalism, it is easy to integrate with models described with DEVS or other formalisms (Vangheluwe, 2000). The CD++ toolkit (Wainer, 2002) is a simple framework for implementing Cell-DEVS models. These models are executed asynchronously and provide a continuous time base for improving the accuracy of the simulations. CD++ has been successfully used in different research areas (e.g., building evacuation protocols (Wang et al., 2012), CO_2 diffusion models (Khalil et al., 2020), or epidemics (Cárdenas et al., 2020)).

Nonetheless, current tools for simulating Cell-DEVS models present some limitations. For example, they only support typical CA models that divide the scenario in a regular grid of cells. This approach hinders the integration with spatial systems with irregular topologies, for instance Geographic Information Systems (GISs), which follow a vectorial approach to describe spaces with irregular topologies (Chang, 2019). In these scenarios, alternative modeling approaches show better results. For instance, metapopulation models (Hanski, 1994) divide the system into multiple populations with different characteristics that interact with each other. The relationship between every pair of groups is defined separately. Furthermore, current Cell-DEVS tools integrate a low-level interface for describing aspects of the Cell-DEVS models (e.g., cells state transition functions or neighborhoods). This interface makes defining different scenarios for exploring the search space of the system under study cumbersome. Finally, current Cell-

DEVS tools cannot easily use third-party software for more advanced models (e.g., TensorFlow (Abadi et al., 2016) for representing the cells state transitions using complex machine learning models).

In this paper, we present the asymmetric Cell-DEVS formalism, a new flavor of Cell-DEVS. The proposed formalism integrates concepts of metapopulation modeling to support both classic CA models and vectorial scenarios. Vectorial scenarios are suitable for simulating complex systems with irregular topologies and non-uniform relationships between entities. They support the simulation of more complete and realistic scenarios without incurring memory nor performance burdens.

In addition, we present a new version of the Cadmium simulator (Belloli et al., 2019) that supports the implementation of asymmetric and classic Cell-DEVS scenarios. This new implementation is a flexible, header-only C++ (Stroustrup, 2013) library that allows users to implement custom cellular models with complex state transitions and timing behaviors. It also integrates a high-level interface for configuring Cell-DEVS scenarios using JavaScript Object Notation (JSON) files (Bray, 2017). This interface allows reusing a base Cell-DEVS model to execute different setups effortlessly. Furthermore, using JSON instead of any other notation format eases the integration with web-based tools and the readability of the scenario setup. It also simplifies the integration with GIS models that use the GeoJSON format (Butler et al., 2016).

The remainder of this paper is organized as follows. Section 2 presents related work. In Section 3, we present the asymmetric Cell-DEVS formalism, an extension of the classic Cell-DEVS formalism to support scenarios with irregular shapes. Section 4 describes the new version of the Cadmium simulator. We focus on its new Application Programming Interface (API) and the semantics supported by JSON configuration files for defining asymmetric and classic Cell-DEVS models. We illustrate how the asymmetric Cell-DEVS formalism and Cadmium can assist modelers in developing cellular models in Section 5. We also enumerate a set of successful use cases that follow this approach. Section 6 draws the main conclusions of this work.

2. Related Work

The modeling of natural systems aims to capture the behavior of biological processes to understand the relationships and dynamics of these systems (Wolfram, 1984). For example, models for pandemic spread may assist us with the definition of effective contention policies to reduce the impact of the disease. The COVID-19 outbreak has shown that modeling and simulation methodologies are valuable for understanding the disease and assessing the effect of different approaches to diminish its propagation speed (Adiga et al., 2020). Numerous proposals follow the approach presented by Kermack & McKendrick (1927), which classifies the population into three compartments: susceptible (S), infected (I), and can recovered (R). This model is known as the SIR compartmental model. Figure 1 shows a schematic of the SIR model.

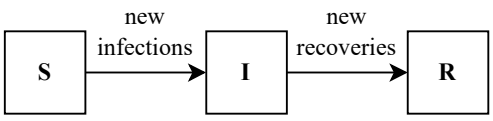


Figure 1. Schematic of the SIR compartmental model.

People that have never been infected remain in the S compartment. Infected individuals are assigned to the I group. People in this compartment can infect the susceptible population. Once infected individuals recover from the disease, they transition to the R compartment. The SIR model describes the dynamics between compartments with a set of Ordinary Differential Equations (ODEs). Different works extended this model with new compartments (e.g., deceased (D)) and new dynamics between compartments (e.g., immunity loss) (Tang et al., 2020). However, these models rely on the law of large numbers and cannot capture interactions between groups across both time and space. Several state-of-the-art works combine SIR-based compartmental models with cellular models to integrate the spatial aspects of the spread of the disease (Bin et al., 2019).

CA is one of the most popular approaches to define cellular models. CA is a discrete space-time formalism built as a lattice network of cells (Wolfram, 2002). In classic cellular models, cells are arranged uniformly in an n-dimensional lattice. A grid cellular scenario is characterized by the following parameters:

- Shape: n-tuple that defines the dimension of the scenario. Usually, $n \in \{1, 2, 3\}$.
- Origin cell: it indicates which cell is considered the first cell of the scenario. The other cells are numbered according to their position with respect to the origin cell.
- Wrap: wrapped scenarios connect the scenario boundaries with their opposite boundary. For instance, a wrapped 2-dimensional scenario connects cells in the left boundary with the cells in the right boundary, and uppermost cells are connected to lowermost cells. Wrapped scenarios can be thought of as n-cubes.

Figure 2 shows an example of a classic cellular model. In this example, the shape of the scenario is ⟨3,4⟩ (i.e., a 3 by 4 2-dimensional cell grid). The origin cell is (1,1). Therefore, cell (2,1) is below the origin cell, and the cell (1,2) is to the right of cell (1,1). The shown scenario is wrapped. Thus, even though cell (1,4) corresponds to the upper-right corner of the scenario, cell (1,1) is to its right, and cell (3,4) is above it.

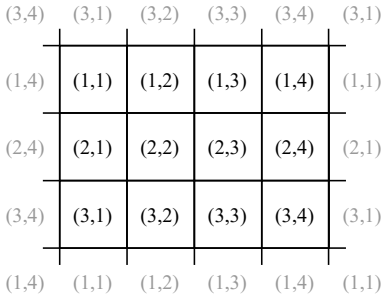


Figure 2. Example of classic cellular model.

CA are formally defined with the following quadruple:

$$\mathrm{CA} = \langle \mathrm{S}^{\mathrm{n}}, \Theta, \mathrm{N}, \tau \rangle, \tag{1}$$

where:

- S^{n} is the n-dimensional working space. Usually, $\mathrm{n} \in \{1, 2, 3\}$.
- Θ is the set of states. At time step t, the state of any cell corresponds to one of these states:

$$\forall \mathrm{t}, \forall \mathrm{i} \in \mathrm{S}^{\mathrm{n}}, \theta_{\mathrm{i}}^{(\mathrm{t})} \in \Theta. \tag{2}$$

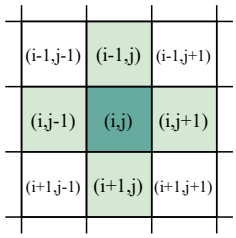
- N is the neighborhood set. It contains relative positions of the neighboring cells concerning a given cell.
- $\tau: \Theta^{|N|} \rightarrow \Theta$ is the local computation function. It comprises a set of rules to obtain the new state of all the cells at every time step:

$$\forall \mathrm{i} \in \mathrm{S}^{\mathrm{n}}, \theta_{\mathrm{i}}^{(\mathrm{t}+1)} = \tau\left(\left\{\theta_{\mathrm{i}+\mathrm{j}}^{(\mathrm{t})} \mid j \in N\right\}\right). \tag{3}$$

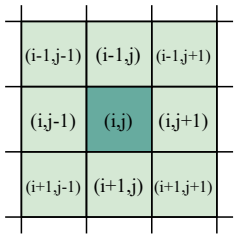
The neighborhood set in CA models is defined as a set of distance vectors. Every cell of the scenario can compute its neighboring cells by summing its location and each distance vector in the neighborhood set. For example, in a 2-dimensional scenario, if the neighborhood set is $\mathrm{N} = \{\langle(1,0)\rangle\}, \langle(0,0)\rangle, \langle(0,1)\rangle, \langle(-1,0)\rangle, \langle(0,-1)\rangle\rangle$, the neighboring cells of cell (i, j) are $(i+1, j)$, (i, j), $(i, j+1)$, $(i-1, j)$, and $(i, j-1)$. This neighborhood set is common in the literature and is known as the von Neumann neighborhood of range 1, or simply the von Neumann neighborhood. For a given cell, the von Neumann neighborhood of range $\mathrm{r} \in \mathbb{N}$ is the set of cells in the scenario which Manhattan distance is less than or equal to r. Another popular cellular neighborhood is the Moore neighborhood. For a given cell, the Moore neighborhood of range $\mathrm{r} \in \mathbb{N}$ is formally described as the cells in the scenario which Chebyshev distance is less than or equal to r. Figure 3 represents these popular neighborhoods for 2-dimensional scenarios.

Usually, the cell space uses a rectangular topology, in which each cell is represented as a rectangle. However, previous works proposed alternative lattice patterns like hexagons (Tariq and Kumaravel, 2016). Furthermore, Baetens & De Baets (2012) extend the CA formalism to support irregular tessellations. CA has been successfully used to describe the dynamics in natural systems, such as the course of braided rivers (Murray and Paola, 1994) or urban development (Ward et al., 2000).

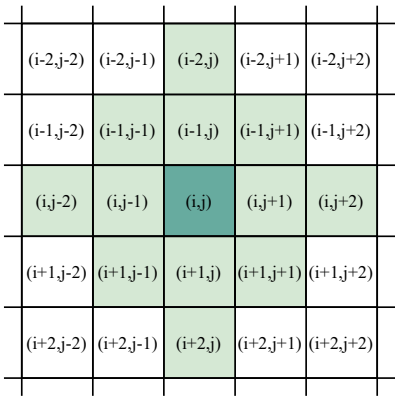
In CA, the state of all the cells in the cell space are recomputed at every time step in synchronous fashion. However, usually, only a few cells update their state at a given time step. Thus, cells which state remains unchanged add an unnecessary computation overhead. This overhead is greater for simulations that need shorter time steps for better time precision.



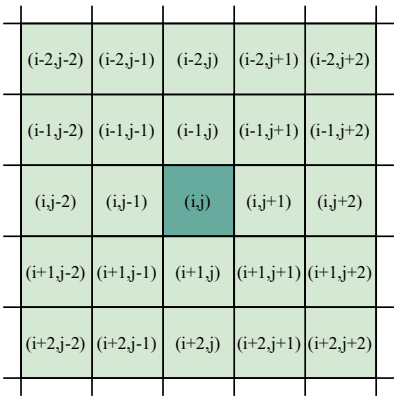
(a) von Neumann neighborhood (r = 1).



(b) Moore neighborhood (r = 1).



(c) von Neumann neighborhood (r = 2).



(d) Moore neighborhood (r = 2).

Figure 3. Schematic of von Neumann and Moore cellular neighborhoods.

Different research have defined asynchronous CA (Ingerson & Buvel, 1984, Fatès, 2013, Dennunzio et al., 2017) to avoid these problems. However, these approaches still present some limitations inherent to the CA formalism (e.g., limited model complexity and closure to external events (Muzy et al., 2005)). Combining CA with the DEVS formalism has shown success in addressing these issues. DEVS (Zeigler et al., 2000) is a hierarchical and modular approach to describe discrete-event systems. The DEVS formalism describes systems at two levels: atomic models, which define the behavior of a system as transitions between states and response to external events; and coupled models, which specify how subcomponents of the system interconnect. The formal definition of an atomic model is described as the following:

$$A = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle, \tag{4}$$

where:

- X is the set of input events. Each element corresponds to a possible input event that may trigger the atomic model's external transition function.
- Y is the set of output events. Each element corresponds to a possible output event that may be triggered by the atomic model's output function.
- S is the states set. At any given time, the state of the atomic model is $s \in S$.

- $\mathrm{ta}: S \to \mathbb{R}_{\geq 0} \cup \infty$ is the time advance function. When the atomic model transitions to the state s, it will remain in this state for $\mathrm{ta}(s)$ time units or until the model receives one or more inputs.
- $\delta_{\mathrm{int}}: S \to S$ is the internal transition function. After spending $\mathrm{ta}(s)$ time units in the state s without receiving any input event, the atomic model transitions to the state $s' = \delta_{\mathrm{int}}(s)$.
- $\lambda: S \to Y$ is the output function. When the atomic is about to change its state due to an internal transition, the atomic model generates $\lambda(s) \subseteq Y$ output events. This function is triggered right before calling the internal transition function.
- $\delta_{\mathrm{ext}}: S \times \mathbb{R}_{\geq 0} \times X \to S$ is the external transition function. It is triggered when the atomic model receives a bag of inputs $x \subseteq X$ after e time units since the atomic model transitioned to its current state s (i.e., $0 \leq e \leq \mathrm{ta}(s)$). When triggered, $\delta_{\mathrm{ext}}(s, e, x)$ determines the new state of the atomic model.
- $\delta_{\mathrm{con}}: S \times \mathbb{R}_{\geq 0} \times X \to S$ is the confluent transition function. This transition function decides the next state in cases of collision between external and internal events (i.e., $e = \mathrm{ta}(s)$). Typically, $\delta_{\mathrm{con}}(s, \mathrm{ta}(s), x) = \delta_{\mathrm{ext}}(\delta_{\mathrm{int}}(s), 0, x)$.

The formal definition of a coupled model is described as follows:

$$M = \langle X, Y, C, \mathrm{EIC}, \mathrm{EOC}, \mathrm{IC} \rangle, \tag{5}$$

where X is the set of inputs; Y is the set of outputs; C is the set of DEVS subcomponents; EIC is the external input coupling relation, from external inputs of M to component inputs of $c_i \in C$; EOC is the external output coupling relation, from component outputs of $c_i \in C$ to external outputs of M; and IC is the internal coupling relation, from components outputs of $c_i \in C$ to component inputs of $c_j \in C$.

In most cases, cellular models described with DEVS require expertise in programming, leading to multiple specific single-use programs which are difficult to reuse for related experiments (Wainer, 2006). The Cell-DEVS formalism (Wainer, 2009) proposes a simple and generic way of defining discrete-event cellular models with advanced timing behavior. Each cell is a continuous-time model in which state transitions are defined by a local computation function, as in CA. It also provides different delay functions that simplify complex timing definitions. This approach allows us to define CA models with advanced timing behavior while using the formal specifications of DEVS. Section 3 provides an in-depth description of the Cell-DEVS formalism.

Cellular models usually divide the space into uniform cells with the same behavior. Additionally, the relationship between these cells is described implicitly with the neighborhood set. However, even though these assumptions are an advantage for formal analysis, they are sometimes overly simplistic (Hanski and Simberloff, 1997).

Metapopulation models propose a different approach to study the dynamics of groups that interact at some level (Hanski, 1994). These models divide the system into multiple populations. Each population has a different location and different characteristics (e.g., number of inhabitants). Additionally, metapopulation models

describe the relationship between every pair of groups separately. Thus, the interaction between groups is heterogeneous and is more accurate than in cellular models. Metapopulation models have shown success in describing epidemics (Watts et al., 2005), biodiversity in natural areas (Muneepeerakul et al., 2007), and demographics of species (Heide-Jørgensen et al., 2013).

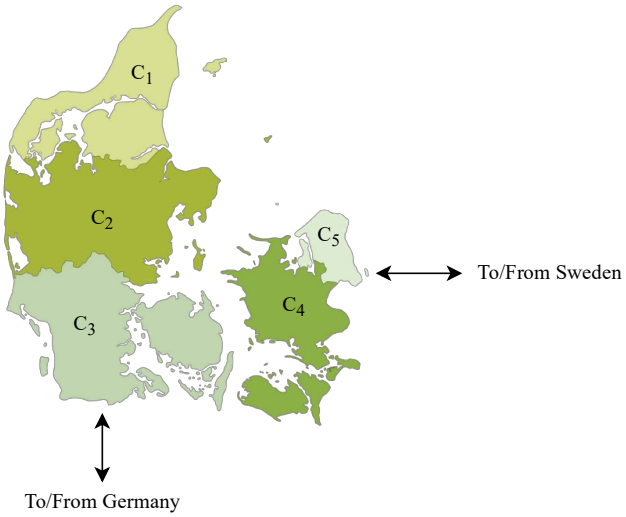
Some works aim to integrate aspects of metapopulation models to CA. For example, Sonnenschein & Vogel (2001) propose the Asymmetric Cellular Automaton (ACA) formalism. ACA substitutes the neighborhood of CA with a topology matrix, which describes the neighborhood degree of two cells. The neighborhood degree represents how strongly a neighboring cell influences another cell. Every cell in the scenario assigns a different neighborhood degree to each neighboring cell, allowing asymmetric neighborhood relations. Additionally, the ACA formalism includes a global state. The global state describes the characteristics of the scenario that affect all cells in the model but, in contrast to the cell state, cells cannot modify directly (e.g., wind direction for fire spreading models). The variables that make up the global state evolve independently of the scenario cells. Zhong et al. (2009) employ this formalism to define a CA model for the spread of infectious diseases where cells are irregular regions of a city, and the neighborhood degree is computed as a function of the shared border width and the number of roads that interconnect these regions. In this paper, we present the asymmetric Cell-DEVS formalism to support discrete-event cellular models based on metapopulation systems. This new formalism is compatible with conventional cellular models and provides mechanisms to define global states for all the cells in the scenario.

3. The Asymmetric Cell-DEVS Formalism

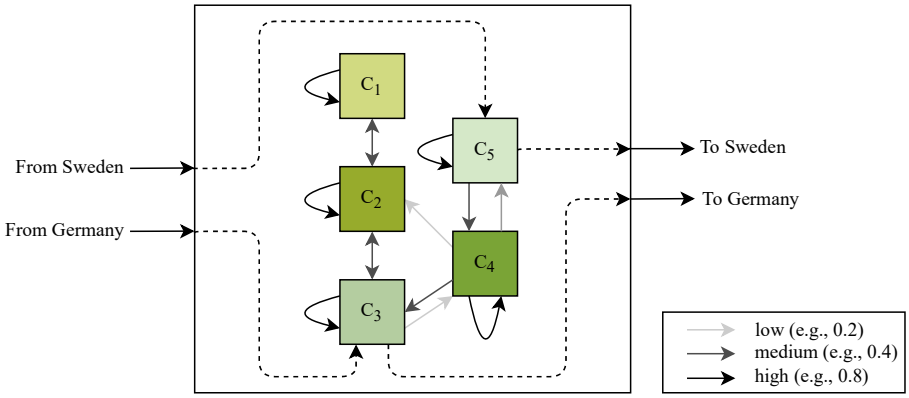
This section defines the asymmetric Cell-DEVS formalism. Then, we illustrate how this formalism contains the classic Cell-DEVS approach based on a uniform lattice of cells.

3.1. Formal Definition of the Asymmetric Cell-DEVS Formalism

In the asymmetric Cell-DEVS formalism, each cell may have different behavior, and they are not necessarily arranged in a regular lattice. Thus, unlike traditional cellular models, neighborhoods can be diverse for every cell. Next, we describe all the elements that comprise the Cell-DEVS formalism. We use the example shown in Figure 4 to illustrate the principal concepts of the proposed asymmetric Cell-DEVS formalism.



(a) Map of the five Regions of Denmark with the borders to other countries.



(b) Representation of an asymmetric Cell-DEVS model for Denmark.

Figure 4. Example of an asymmetric Cell-DEVS model.

Let us assume we want to study the movement of individuals between the five Regions of Denmark presented in Figure 4a (identified as C_1-C_5). We also want to assess the influence of the connections with bordering countries (namely, Germany and Sweden). Figure 4b shows the structure of an asymmetric Cell-DEVS coupled model representing the system under study. Asymmetric Cell-DEVS models are regular coupled DEVS models that are formally described as follows:

$$\mathrm{CD} = \langle \mathrm{X}, \mathrm{Y}, \mathrm{C}, \mathrm{EIC}, \mathrm{IC}, \mathrm{EOC} \rangle, \tag{6}$$

where:

- X is the input set. It defines the possible inputs that the Cell-DEVS model may receive.

- Y is the model output set. It describes every potential output that the Cell-DEVS model may generate.
- C is the subcomponent set. It defines all the cells in the model.
- EIC is the external input coupling set. It maps inputs of type X with inputs in each cell $C_i \in C$ in the model.
- IC is the internal coupling set. It defines the interconnections between the cells in the model. We say that cell $C_i \in C$ is influenced by cell $C_j \in C$ if the IC set maps outputs in cell C_j to inputs of cell C_i. Alternatively, we say that cell C_i is a neighbor of C_j if the IC set maps outputs of C_i to inputs of cell C_j.
- EOC is the external output coupling set. It maps outputs of cells to the Y set. It allows us to forward information about the scenario to other DEVS models.

In the example of Figure 4, there is one cell per Region of Denmark (i.e. $C = \{C_1, C_2, C_3, C_4, C_5\}$). Figure 4b represents the X and Y sets with edges that go from and to the foreign countries (namely, Germany and Sweden), respectively. The EIC set is represented by the dashed edges that link inputs from Sweden and Germany to the cells C_5 and C_3, and the EOC set is depicted with dashed edges that link cells C_3 and C_5 to Germany and Sweden. Finally, Figure 4b represents the IC set with solid edges that link the cells. For instance, cell C_2 is influenced by C_1, C_2, C_3, and C_4. On the other hand, C_3, C_4, and C_5 are neighbor cells of C_4.

Each cell $C_i \in C$ of the scenario is formally defined as follows:

$$C_i = \langle X_i^N, X_i^E, Y_i, S_i, N_i, \tau_i, d_i, \text{delay}_i \rangle, \quad (7)$$

where:

- X_i^N is the neighborhood input set. It defines the possible inputs that the cell may receive from neighboring cells in the Cell-DEVS model.
- X_i^E is the external input set. It enumerates inputs that the cell may receive not from neighboring cells (e.g., inputs from a different DEVS or Cell-DEVS model). We use the cell input set $X_i = X_i^N \cup X_i^E$ to refer to the neighborhood and external input sets together. It represents all the potential inputs that the cell may receive.
- Y_i is the cell output set. It describes all the possible output events of the cell.
- S_i is the cell state set. It enumerates the possible states of the cell.
- N_i is the cell neighborhood set. It enumerates all the neighboring cells of the cell.
- $\tau_i: S_i \times N_i \times X_i \to S_i$ is the local computation function of the cell. When triggered, it computes the new state of cell C_i from its previous state, the neighborhood set, and inputs.
- $d_i: S_i \to \mathbb{R}_{\geq 0}$ is the cell delay function. When the cell transitions to a new state $s_i \in S_i$, it waits for $d_i(s_i)$ time units before sharing with other cells this change.
- delay_i is the cell delay type function. These functions allow us to define complex timing behavior effortlessly. We provide more details about these functions later.

The external input set X_i^E, enables modelers to use state variables that provide global information. For example, in forest fire spread models, an external DEVS model can be used to model the wind flow, and cells in the scenario could use X_i^E to receive inputs notifying wind direction changes.

In cellular models, the state of neighboring cells has an influence on the cell state. In asymmetric Cell-DEVS, the neighboring cells are not necessarily the immediate nearby cells (for instance, C_5 could be connected to C_1), and not all the neighbor cells affect the cell behavior in the same way. For example, when modeling a geography-based cellular model like the one in Figure 4, the length of the border shared with each neighboring cell varies. Asymmetric Cell-DEVS represents this by associating the *vicinity factor* V_i^j, which describes how the state of the neighboring cell C_j affects the state of cell C_i. The neighborhood set N_i is defined as a set of pairs ⟨cell ID, vicinity factor⟩ for each of the neighboring cells of cell C_i:

$$N_i = \{\langle C_j, V_i^j\rangle | C_j \text{ is neighbor of } C_i \text{ and } V_i^j \text{ the vicinity factor of } C_j \text{ over } C_i\}. \quad (8)$$

Figure 4b shows the vicinity factor using different colors for each of the solid edges from neighboring cells. For instance, if an edge from a neighboring cell to the origin cell is black, this represents a *high* vicinity factor. On the other hand, dark and light gray edges correspond to *medium* and *low* vicinity factors, respectively. In this example, *high*, *medium*, and *low* vicinity factors correspond to 0.8, 0.4, and 0.2, respectively. In this way, a cell can use the vicinity factor to ponder the effect of each neighboring cell. For instance, the neighborhood set of the cell C_4 is $N_4 = \{\langle C_3, 0.2\rangle, \langle C_4, 0.8\rangle, \langle C_5, 0.4\rangle\}$. This means that the state of cell C_4 has a high influence on itself (e.g., most of its inhabitants do not leave this cell), while the state of cells C_3 and C_5 have a low and medium influence, respectively (e.g., it is more usual to have individuals moving from cell C_5 to C_4 than from cell C_3). Thus, when transitioning to a new state, cells will pay more attention to the status of neighboring cells with higher associated vicinity factors. Vicinity factors allow modelers to define more complex dynamics between cells, which can potentially increase the accuracy of the simulation results. Asymmetric Cell-DEVS models can be thought of as directed graphs with weighted edges (Gansner et al., 1993) in which cells are the nodes and the edges connect neighboring cells with influenced cells. The weight of an edge from C_j to C_i corresponds to the vicinity factor V_i^j.

Figure 5 represents the behavior of a cell in asymmetric Cell-DEVS.

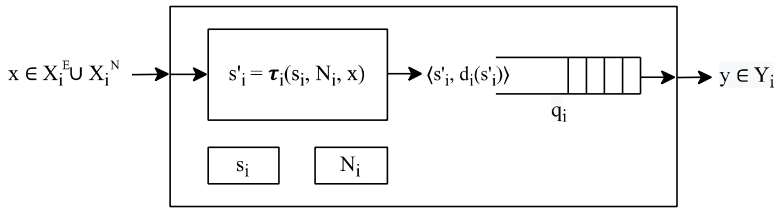


Figure 5. Schematic of a cell in the asymmetric Cell-DEVS formalism.

At any given time, the state of cell C_i is $s_i \in S_i$. When C_i receives input messages $x \subset X_i$, it executes the local computation function $\tau_i(s_i, N_i, x) = s_i'$. The cell new state s_i' depends on its previous state, its neighborhood set, and the received input messages. If the new state is equal to the previous state (i.e., $s_i = s_i'$), the cell passivates and stops computing. Otherwise, it schedules an output of the new state value after the time specified by the cell delay function $d_i(s_i')$. To do so, we might need to use an output queue q_i, which is updated according to the cell delay type function, delay_i. Delay type functions allow us to define complex timing behavior effortlessly. These functions have three input parameters: the previous output queue q_i, the state change to be scheduled s_i', and the delay to wait before sending the corresponding message $d_i(s_i')$; and return a new output queue with the new scheduled message:

$$q_i' = \text{delay}_i(q_i, s_i', d_i(s_i')). \tag{9}$$

The output queue q_i contains tuples $\langle \vartheta, \sigma \rangle$, where $\vartheta \in S_i$ is a previously scheduled output and σ is the time to wait before sending it. If q_i is empty, the cell passivates. Otherwise, it waits until one or more scheduled messages expire and sends them.

The original Cell-DEVS formalism defines two types of delay type functions, both based on logical circuit theory (Huang et al., 2009):

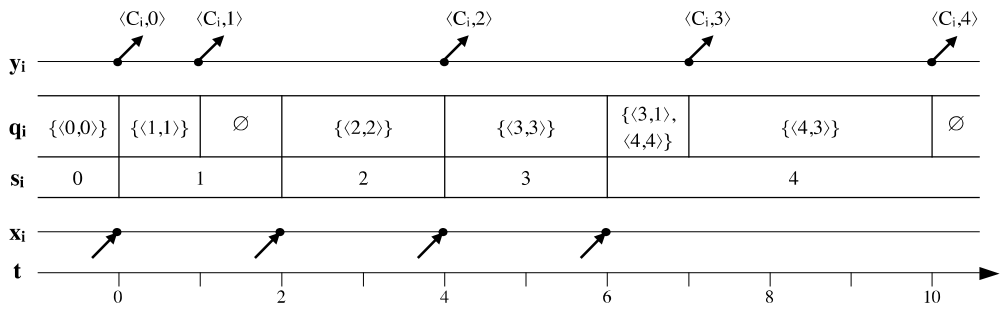
- *Transport delay*. It is based on ideal digital devices with infinite frequency response, in which any input pulse, no matter how short it is, produces an output pulse. It adds to the output queue a new message for every cell phase change:

$$\text{transport}(q_i, s_i', d_i(s_i')) = \{\langle s_i', d_i(s_i') \rangle\} \cup q_i. \tag{10}$$

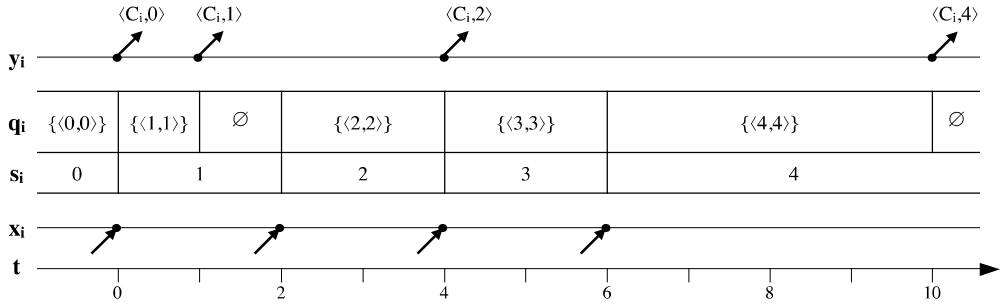
- *Inertial delay*. It is based on more realistic electronic circuits which do not have infinite frequency response, so very short input stimuli are ignored. The inertial delay function presents a preemptive behavior, as the output queue can only contain one message at a given time. Therefore, if the cell changes its state, all the previously scheduled messages are dropped from the queue:

$$\text{inertial}(q_i, s_i', d_i(s_i')) = \{\langle s_i', d_i(s_i') \rangle\}. \tag{11}$$

Figure 6 illustrates the difference between these delay type functions. The abscissae represent the simulation time. In this example, the cell state keeps track of the number of external transitions triggered by the cell. The delay function returns the value of the cell new state (i.e., $d(s_i') = s_i'$).



(a) Transport delay function.



(b) Inertial delay function.

Figure 6. Comparison of transport and inertial delay functions.

In both cases, the initial cell state is $s_i = 0$, and its output queue contains a scheduled message with a delay of 0 (i.e., $q_i = \{\langle 0,0 \rangle\}$). At $t = 0$, the cell outputs the scheduled message and receives input messages with the initial state of its neighboring cells. The local computation function sets s_i to 1, and the output queue schedules a new message with a delay of $d_i(1) = 1$. The cell outputs it at $t = 1$, leaving the output queue empty.

At $t = 2$, the cell receives a new message and transitions to $s_i = 2$. The output queue schedules a message with a delay of $d_i(2) = 2$. At $t = 4$ the scheduled message expires, and the cell outputs it. At the same time, it receives new inputs. Thus, the cell transitions to the phase $s_i = 3$. The delay to be applied is $d_i(3) = 3$. Thus, the cell schedules an output message at $t = 4 + 3 = 7$.

At $t = 6$ (i.e., before outputting the scheduled message), the cell receives new inputs, transitions to a new state $s_i = 4$. The cell must wait for $d_i(4) = 4$ time units. Therefore, the output queue schedules a new message at $t = 6 + 4 = 10$. Here, we can observe the difference between the transport and inertial delay functions. The transport delay function updates the delay to be applied to the message previously scheduled in the queue (i.e., $q_i = \{\langle 3,1 \rangle, \langle 4,4 \rangle\}$). On the other hand, the inertial delay function drops it (i.e., $q_i = \{\langle 4,4 \rangle\}$).

The transport delay function can schedule new events before previously queued messages. For example, if output messages represent individuals moving from one cell to another, people may move at different speeds, and a fast person can overtake a slow person that started moving before. However, in other cases (e.g., epidemic models), messages normally include the sender cell state value. For these models, receiving

messages in reverse order may lead to inconsistencies, as cells might receive outdated copies of the state of their neighboring cells.

To avoid these issues, we now include a *hybrid delay function*. This function behaves as the transport delay, however, when scheduling a new phase change, it preempts all the previously scheduled messages which σ is equal to or greater than the delay to be applied to the new message. By doing so, the hybrid delay function guarantees that the latest scheduled message is always the last message in the queue:

$$\text{hybrid}(q_i, s_i', d_i(s_i')) = \{\langle s_i', d_i(s_i')\rangle\} \cup \{\langle y, \sigma\rangle \in q_i \mid \sigma < d_i(s_i')\}. \quad (12)$$

Defining a Cell as an Atomic DEVS Model

In the asymmetric Cell-DEVS formalism, cells are atomic DEVS models. As discussed in Section 2, an atomic DEVS model is formally defined with the tuple shown in Eq. (4). The input set of the atomic DEVS model corresponding to cell C_i is the union of its neighborhood and external inputs sets, while the output set of the DEVS model is the output set of the cell:

$$X = X_i^N \cup X_i^E \; ; Y = Y_i. \quad (13)$$

The state of the atomic DEVS model includes the current cell state, its output queue, and its neighborhood set (i.e., $s = \langle s_i, q_i, N_i\rangle$). The time advance function of the atomic model returns the minimum delay of the output messages scheduled in the cell output queue:

$$ta(s) = \begin{cases} \min\limits_{\langle \vartheta,\sigma\rangle \in q_i} \sigma, & q_i \neq \emptyset \\ \infty, & \text{otherwise} \end{cases}. \quad (14)$$

After waiting this time, an internal transition occurs, and the cell outputs those scheduled messages for which σ expires. Eq. (15) displays the DEVS output function for the cell C_i:

$$\lambda(s) = \{\langle C_i, \vartheta\rangle \mid \langle \vartheta, \sigma\rangle \in q_i \wedge \sigma = ta(s)\}. \quad (15)$$

The internal transition function of the atomic model leaves the cell state as is, updates the σ of queued messages, and removes those already sent by the previous λ function:

$$\delta_{int}(s) = \langle s_i, \{\langle \vartheta, \sigma - ta(s)\rangle \mid \langle \vartheta, \sigma\rangle \in q_i \wedge \sigma > ta(s)\}, N_i\rangle. \quad (16)$$

When the output queue is empty, the cell passivates. Otherwise, it schedules the next internal transition using the first element of the queue.

If C_i receives inputs $x \subset X$, the external transition function $\delta_{ext}(s, e, x)$ is activated, and it executes the τ function, and computes $s_i' = \tau_i(s_i, N_i, x)$. Only if the new

state is different from the previous state (i.e., $s_i \neq s'_i$), will the cell schedule a new output. Eq. (17) shows the external transition function of the cell.

$$\delta_{ext}(s, e, x) = \begin{cases} \langle s'_i, q_i^e, N_i \rangle & , \text{ if } s'_i = s_i, \\ \langle s'_i, \text{delay}_i(q_i^e, s'_i, d_i(s'_i)), N_i \rangle, & \text{otherwise.} \end{cases} \quad (17)$$

Note that it also updates the σ of messages queued in q_i by subtracting the time elapsed since the previous state transition of the model, and schedules an internal event using the first element in the queue:

$$q_i^e = \{\langle \vartheta, \sigma - e \rangle \mid \langle \vartheta, \sigma \rangle \in q_i\}. \quad (18)$$

The confluent transition function is defined as $\delta_{con}(s, e, x) = \delta_{ext}(\delta_{int}(s), 0, x)$.

At time $t = 0$, the state of cell C_i is set to its initial value, s_i^0, and the output queue schedules a message with a delay of 0 so all the cells being influenced by C_i are aware of its initial state. In the same way, cell C_i receives messages with the initial state of its neighboring cells and triggers its local computation function τ_i. If the cell transitions to a new state, it schedules a new message with the delay provided by the delay function d_i. Otherwise, it passivates and waits for any input. Figure 7 shows a flow chart of the behavior of cells in the asymmetric Cell-DEVS formalism.

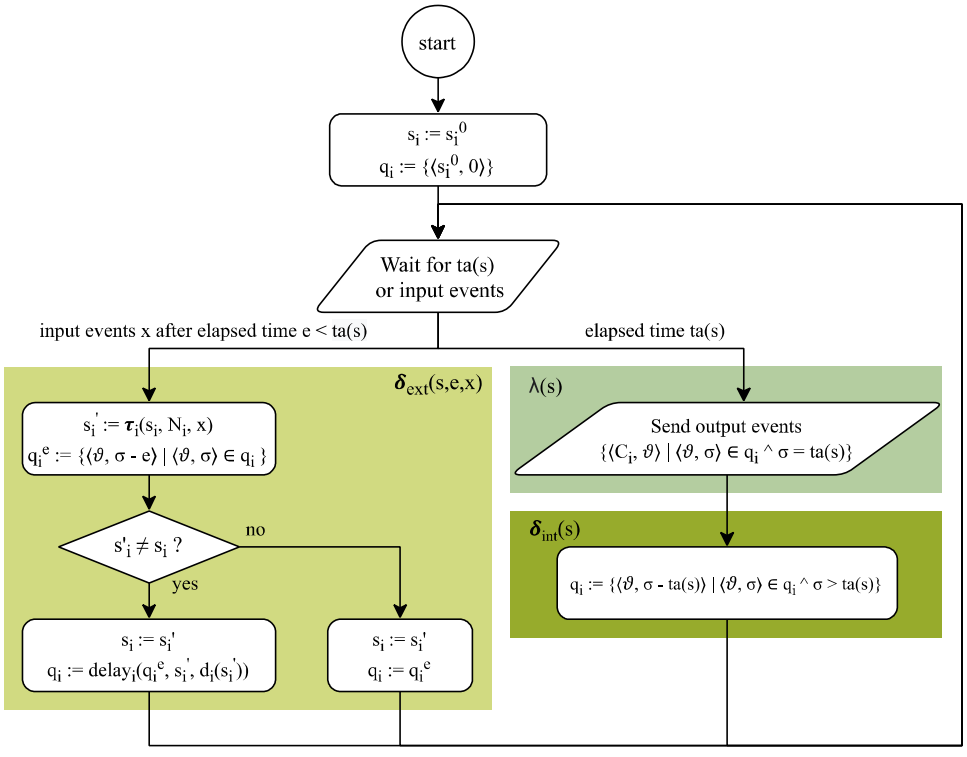


Figure 7. Flow chart of a cell in the asymmetric Cell-DEVS formalism.

3.2. Defining Classic Cellular Models with Asymmetric Cell-DEVS

Asymmetric Cell-DEVS can model classic grid scenarios as follows: first, each cell in the scenario is uniquely identified by its position in the lattice. Then, the vicinity factor of neighboring cells corresponds to the distance vector from the cell to the neighboring cell. Thus, in the asymmetric Cell-DEVS formalism, a 2-dimensional von Neumann neighborhood of range 1 (see Figure 3a) for cell (i,j) corresponds to the set $N_{(i,j)} = \{\langle(i+1,j),(1,0)\rangle, \langle(i,j),(0,0)\rangle, \langle(i,j+1),(0,1)\rangle, \langle(i-1,j),(-1,0)\rangle, \langle(i,j-1),(0,-1)\rangle\}$. Thus, the vicinity factor of one cell over another corresponds to the relative position of the prior from the perspective of the latter.

4. Simulation of Asymmetric Cell-DEVS Scenarios with Cadmium

This section presents a new version of the Cadmium simulator that supports both classic and asymmetric Cell-DEVS models. Cadmium is a header-only library written in C++ that allows the modeling and simulation of computational models based on the DEVS formalism. In this new version of Cadmium, cells are implemented in C++ and the cell space is defined using a JSON configuration file. This approach allows us to study multiple setups by simply modifying the configuration file, thus avoiding recompilations and reducing the overall time required for exploring a scenario. Furthermore, modelers can integrate Cell-DEVS models with other DEVS models implemented with Cadmium. This new version of Cadmium is publicly available on GitHub (Cárdenas and Trabes, 2022). Figure 8 shows the simulation lifecycle to explore cellular models with the asymmetric Cell-DEVS formalism and the Cadmium simulator.

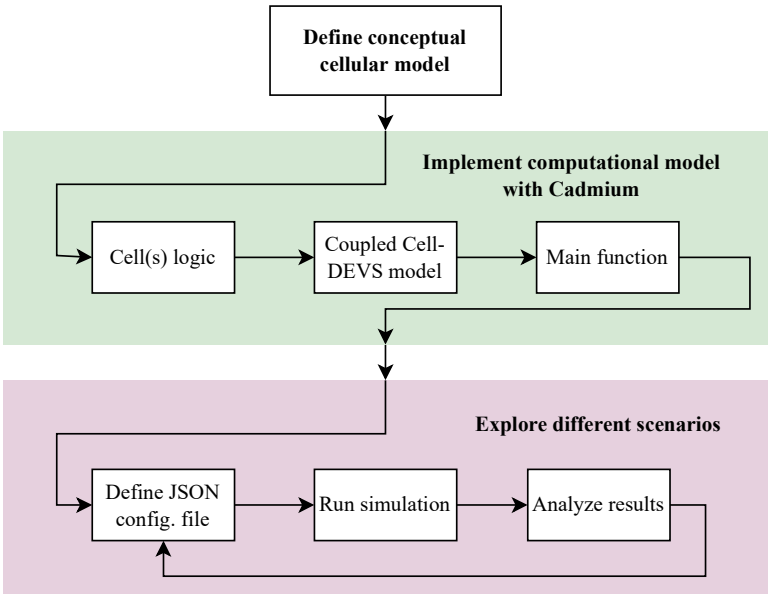


Figure 8. Simulation lifecycle for Cell-DEVS models with Cadmium.

First, we need to define a conceptual cellular model describing the system under study

following the asymmetric Cell-DEVS formalism. The conceptual model is then translated into a computational model using the tools provided by the Cadmium library. Once we have implemented the corresponding computational model, we can run simulations over different scenarios by modifying the JSON configuration file. Then, we analyze the simulation results to gain insight into the system under study.

This section focuses on the API of Cadmium to implement computational models of Cell-DEVS scenarios. We also describe the notation used by JSON configuration files to define a simulation scenario. In Section 5, we present a use case scenario to illustrate how to use Cadmium for developing asymmetric Cell-DEVS models. This Cell-DEVS library is built using the tools provided by the previous version of the Cadmium simulator. Appendix A provides additional information about the API of Cadmium. The remainder of this section describes implementation details of the presented extension of Cadmium to support asymmetric and classic Cell-DEVS models.

4.1. Configuration and Auxiliary Data Structures for Cell-DEVS Scenarios

The Cadmium Cell-DEVS library provides auxiliary and configuration data structures for Cell-DEVS scenarios. Figure 9 shows a Unified Modeling Language (UML) class diagram of these structures.

We use three template arguments. `c` determines the data type used to identify a cell in a scenario. For asymmetric Cell-DEVS models, `c` corresponds to `std::string`. On the other hand, classic (or symmetric) cells use a `std::vector<int>` with their location in the scenario. The `coordinates` data type is just an alias of `std::vector<int>`. `s` corresponds to the data type of the state of the cells, and `v` identifies the data type of the vicinity factor between neighboring cells. All the cells in the scenario must use the same data type for `s` and `v`. If we compare it to the theoretical model explained in Section 3, `s` determines the state set of the cells (i.e., all the possible states of each cell). Alternatively, `v` constrains the values of the vicinity factor of a neighboring cell over an influenced cell.

`CellConfig<C,S,V>` describes the configuration for one or more cells in the scenario. It determines the model and delay type function used by the cells (i.e., inertial, transport, or hybrid). It also determines the initial state of the cells, their neighborhood, any additional cell configuration parameters, EICs, and EOCs of the cells.

`CellConfig<C,S,V>` describes the configuration for one or more cells in the scenario. It determines the model and delay type function used by the cells (i.e., inertial, transport, or hybrid). It also determines the initial state of the cells, their neighborhood, any additional cell configuration parameters, EICs, and EOCs of the cells.

`AsymmCellConfig<S,V>` and `GridCellConfig<S,V>` are class specializations of `CellConfig<C,S,V>` for asymmetric and symmetric Cell-DEVS configurations, respectively. In symmetric Cell-DEVS scenarios, `GridScenario` contains all the information of the scenario (namely, scenario shape, the origin cell, and whether the scenario is wrapped or not). The `GridScenario` class also implements auxiliary methods for scenario-related algebraic functions (e.g., computing the distance between two cells).

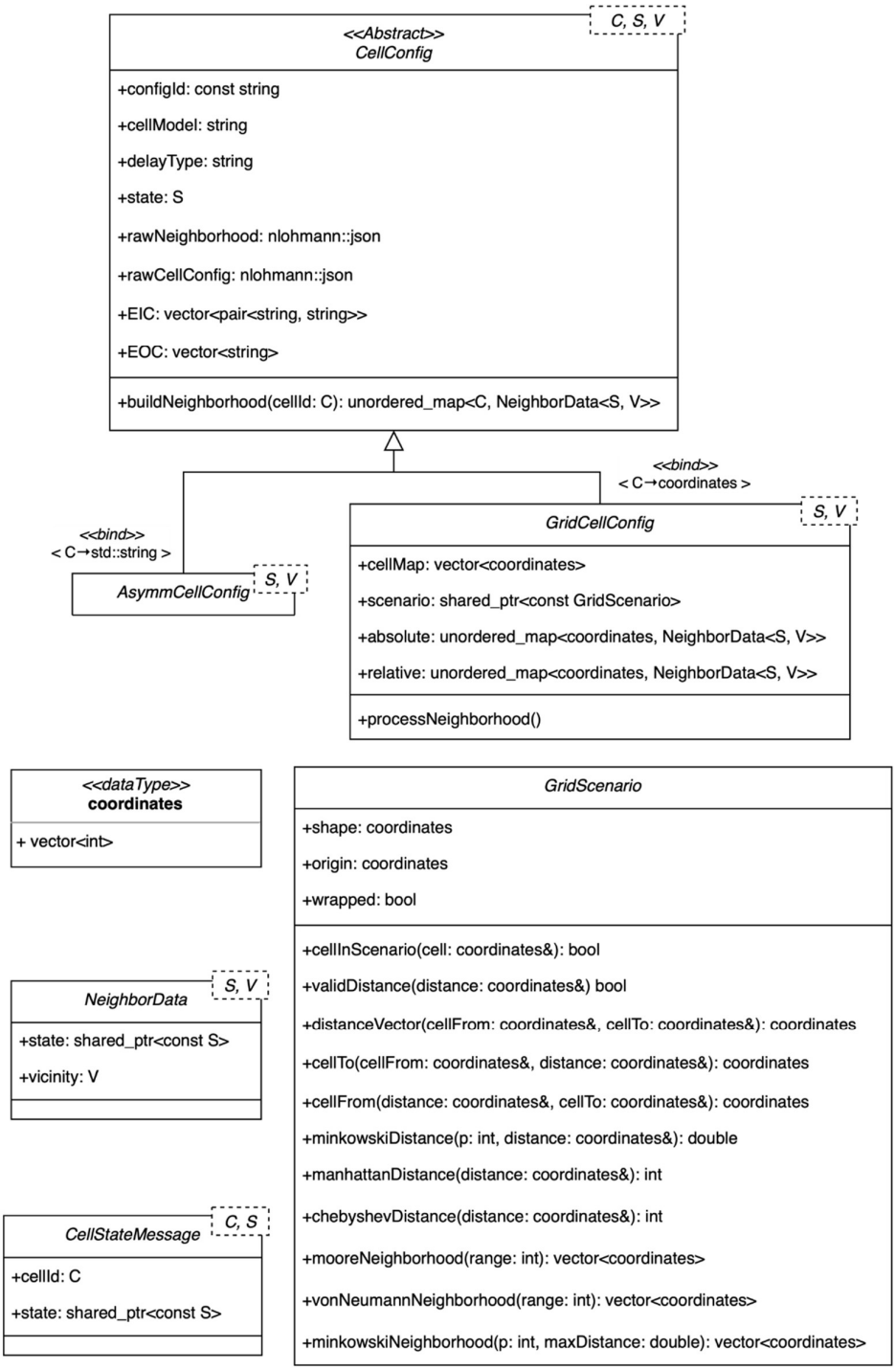


Figure 9. UML class diagram of auxiliary data structures for Cadmium Cell-DEVS.

Cells keep track of the latest known state and the vicinity factor of their neighboring cells. This information is stored in `NeighborData<S,V>` objects. When a cell transitions to a new state, it transmits this change to influenced cells using `CellStateMessage<C,V>` messages.

4.2. Modeling Cells with the Cadmium Cell-DEVS Library

Figure 10 displays an UML class diagram of all the elements related to the implementation of cells in the Cadmium Cell-DEVS library. The `Cell<c,s,v>` class is an abstract implementation of the behavior of cells as regular DEVS models that follows the flowchart shown in Figure 7.

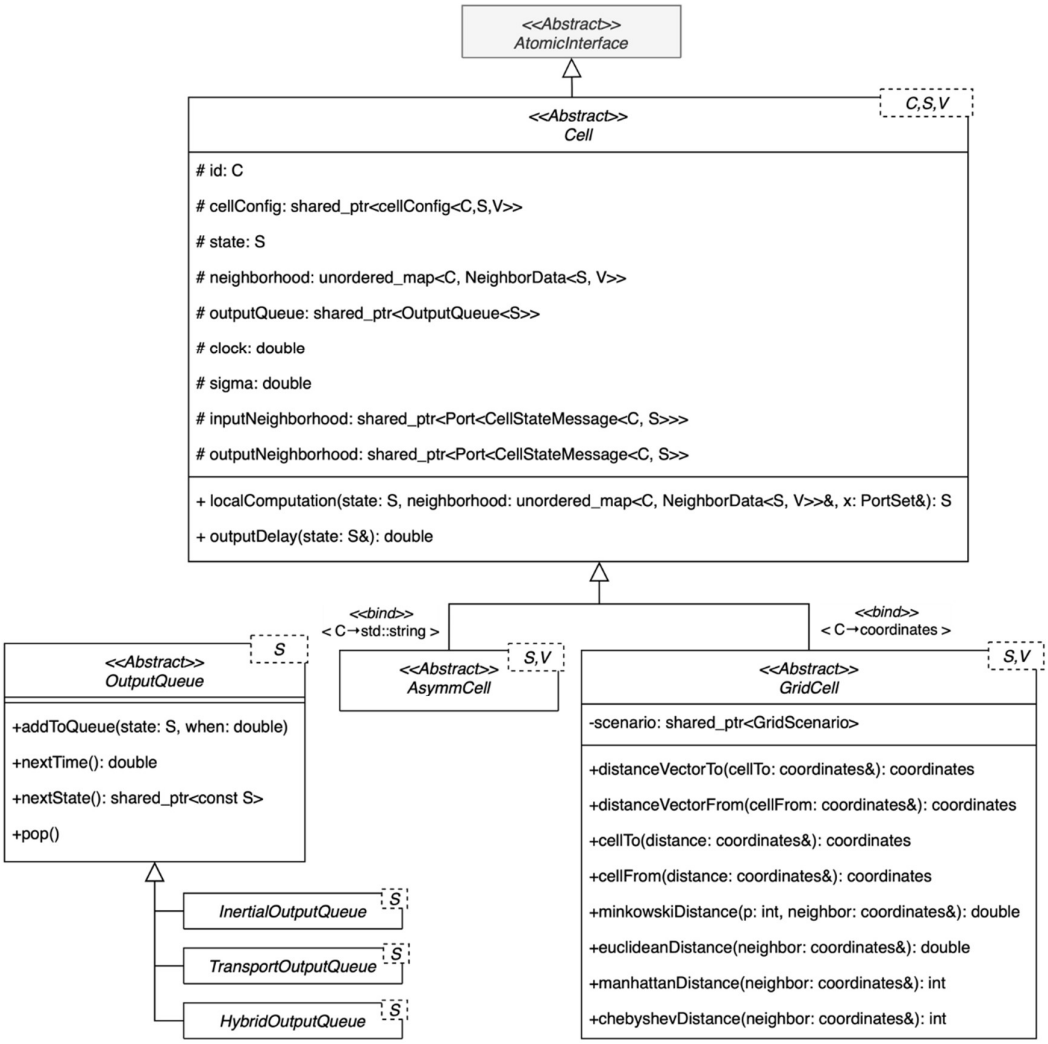


Figure 10. UML class diagram of cell models in the Cadmium Cell-DEVS library.

The `localComputation` and `outputDelay` functions are virtual methods and must be overwritten according to the desired behavior of the cell. Both methods are constant and cannot modify any field of the cells when they are triggered. Instead, the `localComputation` function receives a copy of the latest state of the cell and returns its new state. If the new state is not equal to the previous state, the Cadmium library adds this new state to its output queue.

The `OutputQueue<S>` class represents the cell output queue and output delay function. The `InertialOutputQueue<S>`, `TransportOutputQueue<S>`, and

`HybridOutputQueue<S>` classes implement the behavior of output queues ruled by the inertial, transport, and hybrid output delay type functions, respectively. Cadmium automatically adds to cell models a cell state output port. Messages that leave the output queue are transmitted to influenced cells using this port. Alternatively, cells also have a cell state input port to receive state changes of their neighboring cells. It is possible to add extra input ports for receiving external input events from other DEVS models.

The `AsymmCell<S,V>` and `GridCell<S,V>` classes are class specializations of the `Cell<C,S,V>` class for asymmetric and symmetric Cell-DEVS cells, respectively. `GridCell<S,V>` also implements auxiliary methods for scenario-related algebraic functions, similarly to the `GridScenario` class.

4.3. Implementation of Coupled Cell-DEVS models in Cadmium

In the Cadmium Cell-DEVS library, coupled Cell-DEVS models are defined in a JSON configuration file that is used to create all the cells and couple them according to the neighborhood described in the scenario. Figure 11 presents an UML class diagram of all the elements related to coupled Cell-DEVS models.

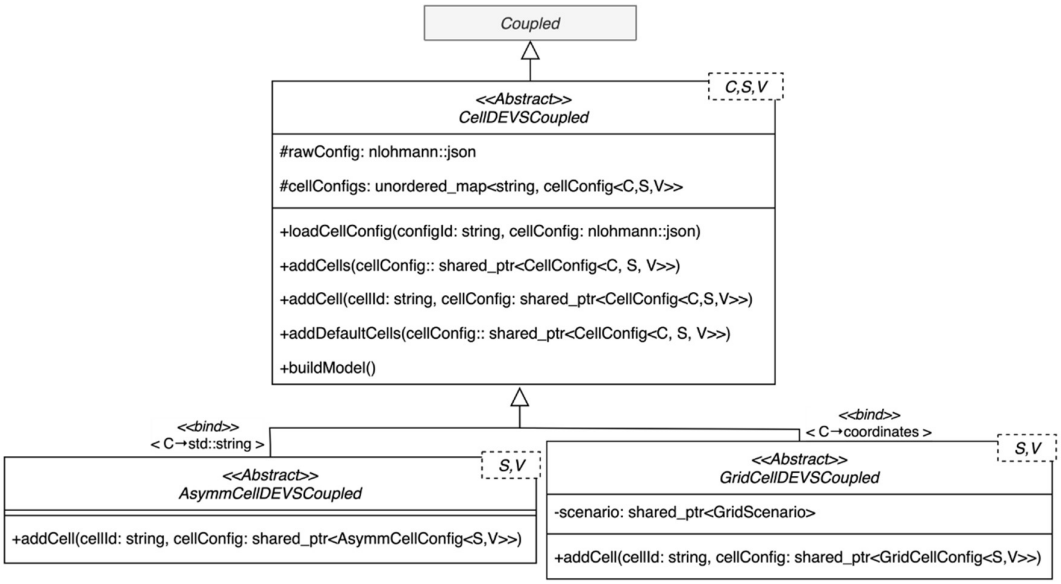


Figure 11. UML class diagram of coupled models in the Cadmium Cell-DEVS library.

`CellDEVSCoupled<C,S,V>` is an abstract implementation of coupled Cell-DEVS models. When a new object of the class `CellDEVSCoupled<C,S,V>` is created, it reads the JSON configuration file and stores its content in the `rawConfig` attribute. The `addCell` function of the `CellDEVSCoupled<C,S,V>` class is a virtual method and must be overwritten. This method is executed for creating every cell in the scenario. `AsymmCellDEVSCoupled<S,V>` and `GridCellDEVSCoupled<S,V>` are class specializations of the `CellDEVSCoupled<C,S,V>` class for asymmetric and symmetric Cell-DEVS coupled models, respectively. These classes maintain the `addCell` function as a virtual method. However, they constraint the

input cell configuration to the `AsymmCellConfig<S,V>` and `GridCellConfig<S,V>` classes, respectively. If required, modelers can add additional input and output ports to the coupled Cell-DEVS model to integrate other DEVS models.

4.4. Configuring Different Scenarios with the JSON Configuration File

Once the main program is compiled, we can run multiple simulations for different scenarios by modifying the input configuration file, without re-compiling the entire project. The configuration of the scenario to be simulated is provided via a JSON file. The structure of this file depends on whether it defines an asymmetric or classic Cell-DEVS scenario. Code 1 shows an example for defining asymmetric scenarios.

Code 1. Example of a JSON configuration file for an asymmetric Cell-DEVS scenario.

```
1 {
2   "cells": {
3     "default": <default_cell_configuration>,
4     "cell_0": <cell_0_configuration_modification>,
5     "cell_1": <cell_1_configuration_ modification>,
6   }
7 }
```

The JSON object named `cells` is used to configure all the cells in the scenario. The `default` object defines a default configuration for all the cells. Inside `default`, modelers select the configuration of all the cells (e.g., delay type function or initial state). Each key inside the `cells` object (except from `default`) corresponds to an asymmetric cell in the scenario, and its value is a modification to the default configuration. In this example, there are only two cells modified: `cell_0` and `cell_1`. Table 1 presents all the fields that cell configuration objects may contain in the JSON file for asymmetric Cell-DEVS scenarios.

Table 1. Cell configuration fields in the JSON file for asymmetric Cell-DEVS models.

Field name	Description
`delay`	String indicating the cell delay type function. It only can be set to `"inertial"`, `"transport"`, or `"hybrid"`.
`model`	Cell model identifier. It determines the logic implemented by the cell. The accepted values depend on the use case.
`state`	Initial cell state. Modelers must provide a function to parse the JSON object to a data structure of the corresponding type in C++.
`neighborhood`	Cell neighborhood. Cell neighborhoods are defined in a key-value fashion. The key identifies the neighboring cell ID, and the value defines the vicinity factor of the neighboring cell over the cell. Modelers must provide a function to parse the JSON object to a data structure of the corresponding type in C++. This field is optional.
`config`	JSON object with any additional configuration parameters for the cell model. This field is optional, and modelers are responsible for parsing its content in the cell model constructor method.

eic	JSON array that describes EICs. It contains tuples of strings that specify these couplings. If **eic** is **[["coupled_p1","cell_p1"]]**, Cadmium will generate an EIC from the **coupled_p1** input port of the Cell-DEVS coupled model to the **cell_p1** input port of the cells affected by this configuration. This field is optional.
eoc	JSON array of strings that describes EOCs. These connect the cell state output port to the output port of the Cell-DEVS coupled model specified in **eoc**. For instance, if **eoc** is **["coupled_p2"]**, the Cadmium library will generate an EOC from the cell output port of the cells affected by this configuration to the **coupled_p2** port of the Cell-DEVS coupled model. This field is optional.

For symmetric Cell-DEVS models, the JSON file also describes the spatial properties of the scenario. Code 2 shows an example for symmetric Cell-DEVS models.

Code 2. Example of a JSON configuration file for a symmetric Cell-DEVS scenario.

```
{
  "scenario":{
    "shape": [25, 25],
    "origin": [-12, -12],
  },
  "cells": {
    "default": <default_cell_configuration>,
    "custom": {
      <custom_cell_configuration>,
      "cell_map": [[-12,-12], [-12,12], [12,-12], [12,12]]
    },
  }
}
```

The **scenario** object defines the shape of the scenario, as well as the origin cell. It is also possible to define whether a scenario is wrapped or not. Table 2 presents all the fields that scenario configuration objects may present for symmetric Cell-DEVS scenarios.

Table 2. Scenario configuration fields in the JSON file for classic Cell-DEVS models.

Field name	Description
shape	JSON array of integers representing the shape of the scenario. All the numbers in **shape** must be greater than zero. In the example shown in Code 2, the scenario if a lattice of 25 by 25 cells.
origin	JSON array of integers representing the origin cell of the Cell-DEVS scenario. It must contain the same number of integers as the **shape** array. In the example shown in Code 2, the origin cell is located at (-12, -12). If not specified, all the coordinates of the origin cell are set to zero.**Error! Reference source not found.**
wrapped	Boolean that indicates whether the cellular model is wrapped or not. If not specified, it is set to **false**.

Again, the **default** JSON object inside **cells** defines the default configuration for all the cells in the scenario. If no configuration modifications are provided, all the cells in the

lattice will be configured with the default values. The rest of the keys inside the `cells` JSON object are also modifications to the default configuration. However, they do not correspond directly to any cell. Instead, the `cell_map` key determines which cells in the scenario implement the alternative configuration. In the example shown in Code 2, only cells (-12, -12), (-12, 12), (12, -12), and (12, 12) (i.e., the cells at the corners of the scenario) are affected by the `custom` configuration modification. All the remaining cells implement the default configuration.

In symmetric Cell-DEVS scenarios, neighborhoods are described as a list of JSON objects. Cadmium provides additional tools to describe different types of neighborhoods. Table 3 describes all these neighborhood types, as well as all the configuration parameters that modelers can use to describe the neighborhoods. Neighborhoods are parsed in the same order as the neighborhood list. Thus, if a cell is part of more than one neighborhood in this list, Cadmium will only use the latest vicinity and ignore the previous configuration for that neighboring cell.

Table 3. Neighborhood fields in the JSON file for classic Cell-DEVS models.

Field name	Description
`type`	String that determines the neighborhood type under description. Other fields of the neighborhood configuration depend on the neighborhood type. Supported neighborhood types are described below.
`vicinity`	Vicinity factor of neighboring cells over the cell under configuration. Modelers must provide a function to parse the JSON object to a data structure of the corresponding vicinity type in C++.
Only for neighborhoods with `"relative"` or `"absolute"` type	
`neighbors`	JSON array with coordinates of all the neighboring cells. In `relative` neighborhood types, these coordinates are relative to the cell under configuration. On the other hand, if the neighborhood type is `absolute`, these coordinates refer to the exact location of neighboring cells in the scenario. Let us assume that `neighbors` is set to `[[0,0]]`. In `relative` neighborhood types, the cell being configured will be neighbor of itself. In contrast, in `absolute` neighborhood types, the cell located at the coordinates (0, 0) will be a neighbor of the cell under configuration, regardless of the location of the later.
Only for neighborhoods with `"von_neumann"` or `"moore"` type	
`range`	Integer indicating the range of the neighborhood. See Figure 3 for more details about von Neumann and Moore neighborhoods.
Only for neighborhoods with `"minkowski"` type	
`p`	Integer indicating which Minkowski distance is used when defining the neighborhood. For example, if `p` is set to 2, the neighborhood uses the Euclidean distance to identify neighboring cells.
`range`	Double-precision floating-point number indicating the maximum distance between one cell to another to consider the prior a neighbor of the later. If `p` is set to 2 and `range` is set to 2.5, all the cells of the scenario

which Euclidean distance to the cell under configuration is less than or equal to 2.5 will be considered neighbors.

5. Using Cadmium to Execute Cell-DEVS Models

This section illustrates how the Cadmium simulator can be used to define asymmetric and symmetric Cell-DEVS models. In addition, discuss how to combine Cell-DEVS models with other tools oriented to the visualization of the results or the automatic generation of scenarios.

5.1. Modeling a communicable disease with Cadmium

Here we show how modelers can follow the workflow shown in Figure 8 to develop a simple asymmetric Cell-DEVS model of communicable disease. We adapt the SIR compartmental model presented in Figure 1 and represent it using asymmetric Cell-DEVS formalism to include spatial considerations in the dynamics of the disease. Then, we translate the resulting conceptual model to an equivalent computational model implemented with the new version of Cadmium. Then, we define different JSON configuration files to explore different pandemic scenarios. The implementation of the presented use cases are included in GitHub (Cárdenas and Trabes, 2022).

Conceptual SIR Model Using the Asymmetric Cell-DEVS Formalism

The state for each cell C_i is represented by the following 4-tuple:

$$s_i = \langle P_i, S_i, I_i, R_i \rangle, \tag{19}$$

where P_i represents the population inhabiting the cell and S_i, I_i, and R_i correspond to the ratio (from zero to one) of susceptible, infected, and recovered people, respectively. The sum of all the compartments must be equal to 1:

$$S_i + I_i + R_i = 1. \tag{20}$$

Every time the local computation function of a cell is triggered, it transitions to a new state $s'_i = \tau_i(s_i, N_i, X_i) = \langle P_i, S'_i, I'_i, R'_i \rangle$. The population of the cell remains the same, but people transition from one compartment to another as described in Equation (21):

$$\begin{aligned} S'_i &= S_i \cdot (1 - i_i), \\ I'_i &= S_i \cdot i_i + I_i \cdot (1 - \gamma), \\ R'_i &= R_i + I_i \cdot \gamma, \end{aligned} \tag{21}$$

where γ corresponds to the recovery factor (i.e., the probability of an infected person

recovering from the disease) and i_i represents the ratio of susceptible people that got infected since the last time the local computing function was triggered:

$$i_i = \sigma \cdot \min\left\{1, \sum_{\langle C_j, V_i^j \rangle \in N_i} \left(V_i^j \cdot \frac{P_j}{P_i} \cdot \upsilon \cdot I_j\right)\right\}, \quad (22)$$

where σ is the susceptibility factor (i.e., the probability of a susceptible individual getting infected if exposed) and υ is the virulence factor (i.e., the probability of an infected person exposing others to the illness) In this model, the vicinity factor V_i^j corresponds to a real number between 0 and 1 that weights how the state of cell C_j influences over the state of cell C_i The way we compute the correlation factor is a configuration parameter of the model. In this example, we compute V_i^j as the length of the shared border divided by the perimeter of the region represented by cell C_i. Finally, we set the output delay function to one day (i.e., cells wait for one day before sending their new state to influenced cells).

Implementing the Corresponding Computational Model with Cadmium

After describing the conceptual model under study, we develop a computational model with the help of the new version of Cadmium. First, we define the data structure used to represent a cell state. Code 3 shows how to implement a data structure for the SIR model.

Code 3. Definition of cells state data structure.

```
 1 #include <iostream>
 2 #include <nlohmann/json.hpp>
 3
 4 using namespace std;
 5 using namespace nlohmann;
 6
 7 struct SIR {
 8   int p;     // population
 9   double s;  // susceptible ratio
10   double i;  // infected ratio
11   double r;  // recovered ratio
12   SIR(): p(0), s(1), i(0), r(0) {}
13 }
14
15 bool operator != (const SIR& x, const SIR& y) {
16   return x.p != y.p || x.s != y.s || x.i != y.i || x.r != y.r;
17 }
18
19 ostream &operator << (ostream& os, const SIR& x) {
20   os << "<" << x.p << "," << x.s << "," << x.i << "," << x.r << ">";
21   return os;
22 }
23
24 void from_json(json& j, SIR& s) {
25   j.at("p").get_to(s.p);
26   j.at("s").get_to(s.s);
27   j.at("i").get_to(s.i);
28   j.at("r").get_to(s.r);
29 }
```

The structure is defined from lines 7 to 13. It contains all the fields of the 4-tuple shown in Equation (19). We need to define a default constructor (see line 12). We also must

implement the inequality operator (`!=`) for two cell states (see lines 15 to 17). Cadmium uses this inequality operator to check whether the local computation function returned a different state. We define the insertion operator (`<<`) in lines 19 to 22 to configure how Cadmium outputs the cell states in the simulation results. Finally, we implement the `from_json` function for our structure (lines 24 to 29). In this way, Cadmium will be able to parse the JSON configuration file to create simulation scenarios.

If we used a complex data structure to represent the vicinity factor between cells, we would have to do a similar process as with the cell state structure. However, in this example the vicinity factor is represented by a double-precision floating-point number. C++ already provides all the required functionalities for this data type.

Now, we define the cell logic. Code 4 shows how to implement the model with Cadmium. As this is an asymmetric Cell-DEVS scenario, our cell model will inherit from the `AsymmCell` class (see Figure 10). In line 6, we select `SIR` and `double` data structures as the cell state and vicinity factor data types, respectively. In line 7 to 9, we add the attributes `rec`, `susc`, and `vir` for the recovery, susceptibility, and virulence factors. These are not part of the cell state. However, we want them to be configurable from the JSON file. To do so, we parse the `config` JSON parameter (see Table 1) in the constructor function (lines 13 to 15).

Code 4. Logic of an asymmetric Cell-DEVS cell for a pandemic spread model.

```
 1 #include <cadmium/celldevs/asymm/cell.hpp>
 2 #include <cmath>
 3
 4 using namespace cadmium::celldevs;
 5
 6 class SIRCell: public AsymmCell<SIR, double> {
 7   double rec;   // recovery factor
 8   double susc;  // susceptibility factor
 9   double vir;   // virulence factor
10  public:
11   SIRCell (string& id, shared_ptr<AsymmCellConfig<SIR, double>>& config):
12   AsymmCell<SIR, double>(id, config) {
13     config->rawCellConfig.at("rec").get_to(rec);
14     config->rawCellConfig.at("susc").get_to(susc);
15     config->rawCellConfig.at("vir").get_to(vir);
16   }
17
18   SIR localComputation(SIR state,
19      const unordered_map<string, NeighborData<SIR, double>>& neighborhood,
20      const PortSet& x) const override {
21     auto aux = 0
22     for (const auto& [nId, nData]]: neighborhood) {
23        SIR s = nData.state;
24        double v = nData.vicinity;
25        aux += s->i * s->p * v;
26     }
27     auto newI = state.s * susc * min(1, vir * aux / state.p);
28     auto newR = state.i * recovery;
29
30     state.r = std::round((state.r + newR) * 1000) / 1000;
31     state.i = std::round((state.i + newI - newR) * 1000) / 1000;
32     state.s = 1 - state.i - state.r;
33     return state;
34   }
35
36   double outputDelay(const SIR& state) const override {
37     return 1;
```

```
38   }
39 }
```

Modelers only need to implement the local computation and output delay functions. The local computation function corresponds to lines 18 to 34. In lines 21 to 27, we compute the ratio of new infections in the cell as defined in Equation (22). We iterate over all the neighboring cells as shown in line 22. The neighboring cell ID is `nId`, and `nData` contains its latest known state and the vicinity factor. The ratio of new recoveries is computed in line 28. Lines 30 to 32 updates the cell state. Note that we discretize the compartments to three decimal numbers. Finally, the output delay function is set to 1 day regardless of the input cell state (lines 36 to 38). In this use case scenario, all the cells implement the same logic. If we want to describe a special behavior for certain cells, we just need to repeat the cell implementation process.

Now, modelers must implement the coupled Cell-DEVS model. Code 5 shows how to implement the model under study using Cadmium. As this is an asymmetric Cell-DEVS scenario, our Coupled Cell-DEVS model will inherit from the `AsymmCellDEVSCoupled` class (see Figure 11).

Code 5. Coupled Cell-DEVS model for pandemic spread scenarios.

```
 1 #include <cadmium/celldevs/asymm/coupled.hpp>
 2
 3 class SIRCoupled: public AsymmCellDEVSCoupled<SIR, double> {
 4  public:
 5   SIRCoupled(const string& id, const string& configFilePath):
 6   AsymmCellDEVSCoupled <SIR, double>(id, configFilePath) {}
 7
 8   void addCell(const string& cellId,
 9        const shared_ptr<AsymmCellConfig<SIR,double>>& conf)const override{
10     string cellModel = conf->cellModel;
11     if (cellModel == "SIR") {
12       addComponent(SIRCell (cellId, conf));
13     } else {
14      throw bad_typeid();
15     }
16   }
17 }
```

In line 3, we select `SIR` and `double` data structures as the cell state and vicinity factor data types, respectively. These must be the same for the coupled Cell-DEVS model and for all the cell models. Coupled Cell-DEVS models only must map a given cell model ID to its corresponding cell model. In our use case, there is only one cell model (the SIR cell class). Thus, we only expect the cell model ID to be `SIR` (see line 11). If so, we add a cell of the `SIRCell` class to the scenario, as shown in line 12. Otherwise, the coupled Cell-DEVS model will throw an exception (line 14).

After implementing all the cell models and the coupled Cell-DEVS model for a given scenario, modelers must implement the main function for executing simulations. Code 6 shows an example of a main function. When creating a coupled Cell-DEVS model, we need to provide a file path to the JSON file with the scenario configuration. In this example, this file is in `./scenario_config.json` (see line 5). After building the model in lines 8 and 9, a Cadmium coordinator is created in line 10 to run the simulations. Additionally, it is possible to set a logger to the coordinator to store simulation results in

a CSV file, as shown in lines 11 and 12. To simulate a scenario, we must execute the **`start`**, **`simulate`**, and **`stop`** methods of the coordinator (see lines 13 to 15). It is possible to set a maximum simulation time executed by the coordinator. In line 4, we set **`simTime`** to 1000. When calling the simulate method of the coordinator, we provide this value to limit the simulated time to 1000 days.

Code 6. Main program for simulating pandemic spread models in Cadmium.

```
1  #include <cadmium/core/simulation/coordinator.hpp>
2  #include <cadmium/core/logger/csv.hpp>
3
4  double simTime = 1000;  // maximum simulation time
5  string configFilePath = "./scenario_config.json";  // path to config file
6
7  int main() {
8    auto model = SIRCoupled("sir_coupled", configFilePath);
9    model.buildModel();
10   auto coordinator = cadmium::Coordinator(model);
11   auto logger = std::make_shared<cadmium::CSVLogger>("log.csv");
12   coordinator.setLogger(logger);
13   coordinator.start();
14   coordinator.simulate(simTime);
15   coordinator.stop();
16 }
```

Exploring Different Scenarios with JSON Configuration Files

Once the main program is compiled, we can run multiple simulations for different scenarios by modifying the input configuration file, without re-compiling the entire project. Let us assume that we want to simulate an asymmetric scenario with three regions as shown in Figure 12. All the cells have shared borders with each other. In this example, we compute V_i^j as the length of the shared border divided by the perimeter of the region represented by cell C_i. Thus, all the vicinity factor of every cell over themselves is 1. The vicinity factor of C_2 and C_3 over C_1 is 0.25. The vicinity factor of C_1 and C_3 over C_2 is 0.125. Finally, the vicinity factor of C_1 and C_2 over C_3 is 0.125. The population of cell C_1 is 100 individuals, while the other two cells have 200 inhabitants. By default, all the population is susceptible to the disease. However, 10% of the people in cell C_3 is infected.

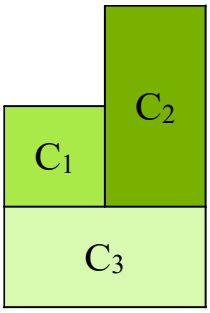


Figure 12. Use case scenario for the asymmetric Cell-DEVS pandemic model.

Code 7 shows the JSON configuration file for simulating this scenario. There are four JSON objects inside the `cells` JSON object. The `default` object (lines 3 to 8) determines the default configuration of all the cells in the scenario. We use the inertial delay type function (line 4). As we have only implemented one cell logic, the cell model type ID of all the cells is set to `"SIR"` (line 5). The coupled Cell-DEVS model will analyze this value when creating every cell in the scenario. As it is set to `"SIR"`, the coupled Cell-DEVS model will generate a cell of the class `SIRCell` (see lines11 to 13 of Code 5). By default, the initial state of all the cells is set to the value shown in line 6. Cells contain 200 inhabitants susceptible to the disease. In the `config` JSON object (line 7), we set other model-related configuration parameters. Specifically, we set the recovery factor to 0.2, the susceptibility factor to 0.8, and the virulence factor to 0.4. Cells have access to these additional configuration parameters in their constructor function. Lines 13 to 15 in Code 4 parse this JSON object to obtain the values.

Code 7. Example of a JSON configuration file for an asymmetric Cell-DEVS scenario.

```
 1 {
 2   "cells": {
 3     "default": {
 4       "delay": "inertial",
 5       "model": "SIR",
 6       "state": {"p": 200, "s": 1, "i": 0, "r": 0},
 7       "config": {"rec": 0.2, "susc": 0.8, "vir": 0.4}
 8     },
 9     "c1": {
10       "state": {"p": 100},
11       "neighborhood": {"c1": 1, "c2": 0.25, "c3": 0.25}
12     },
13     "c2": {
14       "neighborhood": {"c1": 0.125, "c2": 1, "c3": 0.125}
15     },
16     "c3": {
17       "state": {"s": 0.9, "i": 0.1},
18       "neighborhood": {"c1": 0.125, "c2": 0.125, "c3": 1}
19     }
20   }
21 }
```

The `c1`, `c2`, and `c3` JSON objects determine that there are three cells. We modify the default configuration parameters depending on each cell. For example, in line 14, we determine that C_2 has three neighboring cells: C_1, C_2, and C_3, and their vicinity factor over C_2 is 0.125, 1, and 0.125, respectively. Line 10 sets the population of cell C_1 to 100 people. Finally, in line 17, we set the initial ratio of susceptible and infected people in cell C_3 to 0.9 and 0.1, respectively.

Now, we can run the scenario. Simulation results are stored in the `log.csv` file (see line 11 in Code 6). Figure 13 displays the simulation results for first 50 days of the presented use case. At the beginning of the simulation, the ratio of susceptible people in cells C_1 and C_2 is 1, while the 10% of the population in cell C_3 is infected. The disease spreads faster in cell C_1, as is has less population than the other cells and the vicinity factors of neighboring cells over it is greater than in the rest of cells. These aspects negatively affect the ratio of new infections, as shown in Equation (22). The ratio of

infected people reaches its peak in days 12, 18, and 11 for cells C_1, C_2, and C_3, respectively. By the end of the simulation, the ratio of susceptible people for cells C_1, C_2, and C_3 drops to 0.05, 0.21, and 0.19, while the ratio of recovered people is 0.95, 0.79, and 0.81, correspondingly. To run a different scenario (e.g., a lower susceptibility factor due to the use of facial mask to stop the spread of the virus), we just must modify the JSON configuration file and run the simulation again.

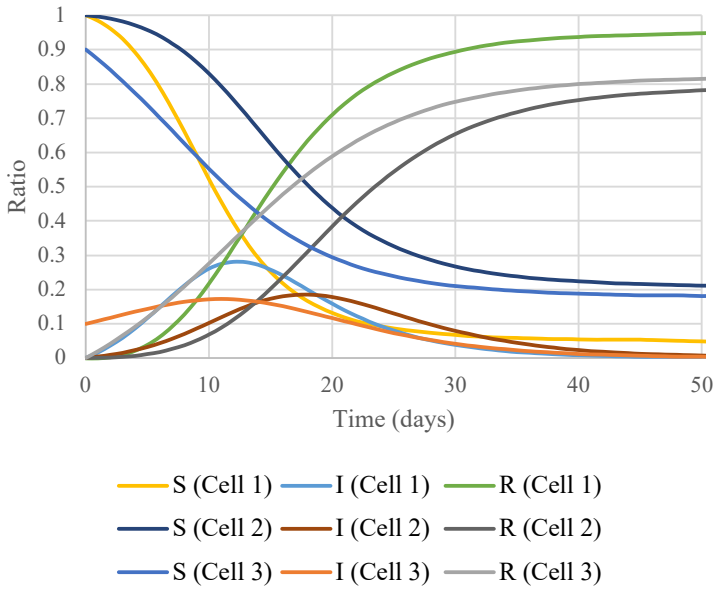


Figure 13. Simulation results of the asymmetric Cell-DEVS use case scenario.

Classic Cell-DEVS Scenarios with the Cadmium Cell-DEVS Library

Let us assume that we now want to explore the scenario shown in Figure 14. In this scenario, every cell has 100 individuals susceptible to the disease. However, 10% of the population of cell $C_{1,1}$ (outlined in red) is infected at the beginning of the simulation.

$C_{-1,-1}$	$C_{-1,0}$	$C_{-1,1}$
$C_{0,-1}$	$C_{0,0}$	$C_{0,1}$
$C_{1,-1}$	$C_{1,0}$	$C_{1,1}$

Figure 14. Use case scenario for the classic Cell-DEVS pandemic model.

In this case, it is more suitable to use the classic Cell-DEVS formalism. The conceptual model remains intact, and the implementation in Cadmium only requires minor changes. First, the `SIRCell` class (see Code 4) inherits from `GridCell<SIR, double>` instead of `AsymmCell<SIR, double>`. On the other hand, the `SIRCoupled` class inherits from `GridCellDEVSCoupled<SIR, double>` instead of `AsymmCellDEVSCoupled<SIR, double>`. The

rest of the code remains intact.

Code 8. Example of a JSON configuration file for a classic Cell-DEVS scenario.

```
 1 {
 2   "scenario": {
 3     "shape": [3, 3], "origin": [-1, -1], "wrapped": false
 4   },
 5   "cells": {
 6     "default": {
 7       "delay": "inertial",
 8       "cell_type": "SIR",
 9       "state": {"p": 100, "s": 1, "i": 0, "r": 0},
10       "config": {"rec": 0.2, "susc": 0.8, "vir": 0.4},
11       "neighborhood": [
12         {"type": "von_neumann", "vicinity": 0.25, "range": 1},
13         {"type": "relative", "vicinity": 1, "neighbors": [[0, 0]]}
14       ]
15     },
16     "infected": {
17       "state": {"s": 0.9, "i":  0.1},
18       "cell_map": [[1, 1]]
19     }
20   }
21 }
```

The `scenario` JSON object (lines 2 to 4) specifies that we want a 2-dimensional scenario of 3 by 3 cells. The coordinates of the origin cells are (-1, -1). The default cell configuration (lines 6 to 15) is like the previous example. However, we define a default neighborhood. In line 12, we define a von Neuman neighborhood of range 1 (see Figure 3a). The vicinity factor of all the cells in this neighborhood is set to 0.25. This vicinity factor matches with the shared border relationship for all the cells in this neighborhood except the cell itself. In line 13, we overwrite the von Neumann vicinity factor of one cell over itself to keep using the same definition of the vicinity factor.

The `infected` JSON object (lines 16 to 20) defines an alternative cell configuration that sets the initial ratio of infected people to 0.1. The `cell_map` array indicates that this alternative configuration only affects cell (1, 1), while the rest of the scenario uses the default configuration.

The proposed asymmetric Cell-DEVS formalism integrates concepts of CA and metapopulation models to define cellular models with more complex topologies and dynamics than other state-of-the-art approaches. The new version of Cadmium eases the development and simulation of computational models based on the asymmetric Cell-DEVS formalism. Modelers do not have to worry about most of the simulation details, and they can focus on implementing the logic of their conceptual models only. The repository of the new Cadmium library includes template projects with instructions to implement classic and asymmetric Cell-DEVS models. Thus, modelers can successfully implement asymmetric Cell-DEVS models with basic programming knowledge.

While the previous models show great flexibility to model different scenarios, a grid-like structure makes it difficult to capture geographical characteristics of a specific region. Cárdenas et al. (2021b) presented a model that adapted the classic Cell-DEVS formalism

for pandemic scenarios with irregular topologies. The model is like the SIIRDS just discussed but using a neighborhood with a vicinity factor that measures the influence of a neighboring cell over the cell. We compute the vicinity factor as a parameter of the model; in this case we computed the vicinity factor as the length of the shared border divided by the perimeter of the region represented by the influenced cell.

From this scenario, we explored the integration of GIS models with Cadmium for the simulation of asymmetric Cell-DEVS based on real geographies. GIS is a spatial system that integrates physical location data (i.e., where things are) with any descriptive information (e.g., the population or weather conditions). With GIS, we can analyze patterns and relationships in a geographical context. For example, we can monitor and forecast changes with GIS to understand trends motivated by geographic characteristics. Figure 15 shows the workflow used to conduct these experiments.

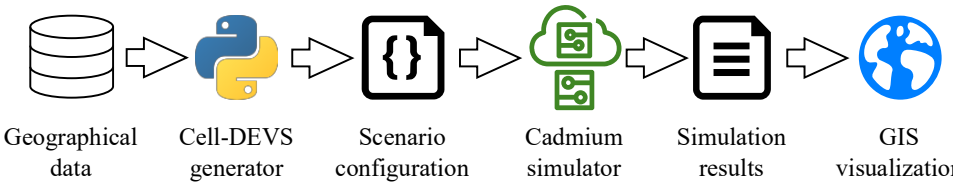


Figure 15. GIS integration for asymmetric Cell-DEVS models

First, a Cell-DEVS scenario generator reads geographical information from a GIS database. The generator adds a cell in the Cell-DEVS scenario for every physical location. Furthermore, it manipulates the descriptive characteristics of the physical location to define their initial state and the vicinity factor of neighboring cells. For the proposed SIIRS model, the generator uses population data to generate the initial state of the cells. Additionally, it computes relationships between physical locations to determine the cell neighborhoods and the vicinity factors to be applied. The resulting asymmetric Cell-DEVS scenario is stored in a JSON scenario configuration file. Cadmium executes the simulation of the scenario and outputs the simulation results. Then, simulation results are displayed on a map corresponding to the area under study using GIS tools and methodologies.

Davidson & Wainer (2021) presented an extended compartmental model that followed the proposed workflow for integrating GIS models with asymmetric Cell-DEVS scenarios. They adapted a GeoJSON file with geographical information about the Ontario province to create an asymmetric Cell-DEVS scenario in which each cell corresponds to a district. For every pair of regions, a vicinity factor was computed as the shared border length between regions divided by the total border length of each region. The neighborhood of the cell is comprised of all the cells which vicinity factor is greater than zero. Simulation results were then adapted to a format compatible with the DEVS Web Viewer tool (St-Aubin et al., 2021). This software represents simulation results of asymmetric Cell-DEVS scenarios on a map, making it easier to understand and analyze simulation results. Figure 16 shows the simulation results of a pandemic scenario over the city of Ottawa. For each district, the ratio of infected people is displayed in red tones.

As simulation advances, all the regions turn darker. Darker cells correspond to districts with more cases of contagion.

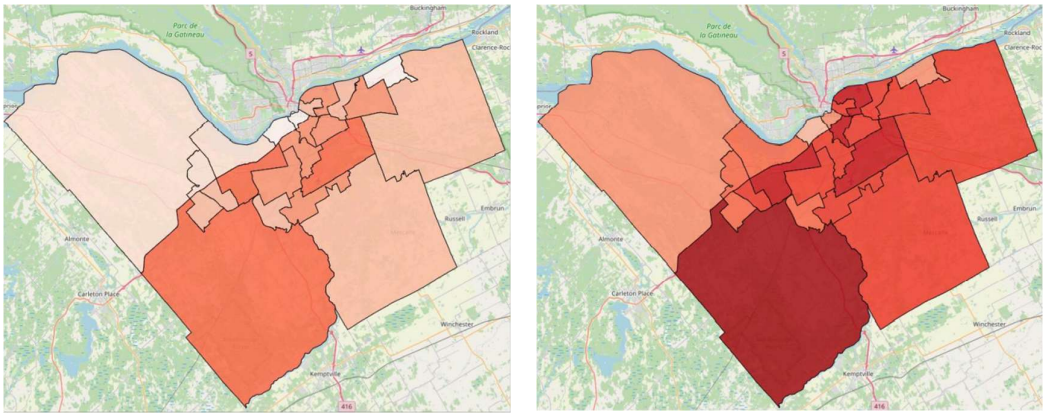


Figure 16. GIS visualization of simulation results for asymmetric Cell-DEVS models.

Additionally, the authors validated the model and calibrated all the configuration parameters (e.g., virulence, recovery, or fatality rates) to reproduce the effect of the COVID-19 disease in Ontario. COMENTAR MAS SOBRE EL MODELO (REVIEWER 4)

6. Conclusions

Cellular models provide a simple means of describing the behavior, dynamics, and structure of natural systems with spatial features. While the classic Cell-DEVS formalism presents advantages over other approaches (e.g., discrete-event nature, performance, or ease of integration with other simulation, visualization, and analysis tools), it is not able to effectively model scenarios with irregular topologies. This issue makes modeling scenarios based on real geographies a complex and inefficient task. Furthermore, if relations other than spatial are needed to be considered (i.e., closeness, friendship, political opinions, or other social aspects) classic Cell-DEVS modelers need to interpret the results of the cellular model accordingly, which is difficult to perceive.

We dealt with these issues by introducing the asymmetric Cell-DEVS formalism, which extends the classic approach by defining the relationships between cells using a graph-based approach. Neighborhoods do not depend only on spatial associations but on an abstract concept called the vicinity factor. The vicinity factor of one cell over another can represent any relationship (e.g., spatial relationship, cultural kinship, or any other spatial relation) and thus allows modelers to define more complex and realistic cellular scenarios. Asymmetric Cell-DEVS was implemented on Cadmium, a simulation tool based on the DEVS formalism. We illustrated the use of Cadmium in simulating both classic and asymmetric Cell-DEVS scenarios with this new version of Cadmium showed a workflow for integrating asymmetric Cell-DEVS models with GIS tools, which allows to generate realistic Cell-DEVS scenarios based on actual geography data automatically. This improves the analysis of simulation results by providing advanced visualizations.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. TensorFlow: A System for Large-Scale Machine Learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA, pp. 265–283.

Adiga, A., Dubhashi, D., Lewis, B., Marathe, M., Venkatramanan, S., Vullikanti, A., 2020. Mathematical Models for COVID-19 Pandemic: A Comparative Analysis. J. Indian Inst. Sci. 100, 793–807. https://doi.org/10.1007/s41745-020-00200-6

Ashlock, D., Kreitzer, M., 2020. Evolving Diverse Cellular Automata Based Level Maps, in: Ciancarini, P., Mazzara, M., Messina, A., Sillitti, A., Succi, G. (Eds.), Proceedings of 6th International Conference in Software Engineering for Defence Applications, Advances in Intelligent Systems and Computing. Springer International Publishing, Cham, pp. 10–23. https://doi.org/10.1007/978-3-030-14687-0_2

Baetens, J.M., De Baets, B., 2012. Cellular automata on irregular tessellations. Dyn. Syst. 27, 411–430. https://doi.org/10.1080/14689367.2012.711300

Belloli, L., Vicino, D., Ruiz-Martin, C., Wainer, G., 2019. Building DEVS Models with the Cadmium Tool, in: 2019 Winter Simulation Conference (WSC). Presented at the 2019 Winter Simulation Conference (WSC), IEEE, National Harbor, MD, USA, pp. 45–59. https://doi.org/10.1109/WSC40007.2019.9004917

Bin, S., Sun, G., Chen, C.-C., 2019. Spread of Infectious Disease Modeling and Analysis of Different Factors on Spread of Infectious Disease Based on Cellular Automata. Int. J. Environ. Res. Public. Health 16, 4683. https://doi.org/10.3390/ijerph16234683

Bray, T., 2017. The JavaScript Object Notation (JSON) Data Interchange Format (RFC No. 8259). RFC Editor.

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, S., Schaub, T., 2016. The GeoJSON Format (RFC No. 7946). RFC Editor.

Cárdenas, R., 2022. Cellular models for pandemic scenarios using the Cell-DEVS formalism. [Online; Accessed on May 23, 2022]. Available at

https://github.com/SimulationEverywhere-Models/CiSE-Pandemic/tree/paper-annsim.

Cárdenas, R., Henares, K., Ruiz-Martín, C., Wainer, G., 2020. Cell-DEVS Models for the Spread of COVID-19, in: Gwizdałła, T.M., Manzoni, L., Sirakoulis, G.Ch., Bandini, S., Podlaski, K. (Eds.), Cellular Automata. ACRI 2020. Lecture Notes in Computer Science. Presented at the Cellular Automata for Research and Industry, Springer International Publishing, Łódź, Poland, pp. 239–249. https://doi.org/10.1007/978-3-030-69480-7_24

Cárdenas, R., Inostrosa-Psijas, A., Wainer, G., 2021a. A Modeling and Simulation Platform for Space-Based Compartmental Modeling of Pandemic Spread, in: 2021 Annual Modeling and Simulation Conference (ANNSIM). Presented at the 2021 Annual Modeling and Simulation Conference (ANNSIM), IEEE, Fairfax, VA, USA, pp. 1–12. https://doi.org/10.23919/ANNSIM52504.2021.9552046

Cárdenas, R., Martin, C.R., Wainer, G., Dobias, P., Rempel, M., 2021b. Studying the Spread of Diseases Using Geographical Data and Irregular Topologies with Cell-DEVS, in: 2021 Annual Modeling and Simulation Conference (ANNSIM). Presented at the 2021 Annual Modeling and Simulation Conference (ANNSIM), IEEE, Fairfax, VA, USA, pp. 1–12. https://doi.org/10.23919/ANNSIM52504.2021.9552115

Cárdenas, R., Trabes, G., 2022. Cadmium 2: An object-oriented C++ M&S platform for the PDEVS formalism. [Online; Accessed on: June, 3, 2022] Available at https://github.com/SimulationEverywhere/cadmium_v2.

Chang, K.-T., 2019. Geographic Information System, in: International Encyclopedia of Geography. American Cancer Society, pp. 1–10. https://doi.org/10.1002/9781118786352.wbieg0152.pub2

Davidson, G., Wainer, G., 2021. Studying COVID-19 Spread Using a Geography Based Cellular Model, in: 2021 Winter Simulation Conference (WSC). Presented at the 2021 Winter Simulation Conference (WSC), IEEE, Phoenix, AZ, USA, pp. 1–12. https://doi.org/10.1109/WSC52266.2021.9715520

Dennunzio, A., Formenti, E., Manzoni, L., Mauri, G., Porreca, A.E., 2017. Computational complexity of finite asynchronous cellular automata. Theor. Comput. Sci. 664, 131–143. https://doi.org/10.1016/j.tcs.2015.12.003

Fatès, N., 2013. A Guided Tour of Asynchronous Cellular Automata, in: Kari, J., Kutrib, M., Malcher, A. (Eds.), Cellular Automata and Discrete Complex

Systems, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 15–30. https://doi.org/10.1007/978-3-642-40867-0_2

Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.-P., 1993. A technique for drawing directed graphs. IEEE Trans. Softw. Eng. 19, 214–230. https://doi.org/10.1109/32.221135

Gerhardt, M., Schuster, H., 1989. A cellular automaton describing the formation of spatially ordered structures in chemical systems. Phys. Nonlinear Phenom. 36, 209–221. https://doi.org/10.1016/0167-2789(89)90081-X

Hanski, I., 1994. A Practical Model of Metapopulation Dynamics. J. Anim. Ecol. 63, 151. https://doi.org/10.2307/5591

Hanski, I., Simberloff, D., 1997. The Metapopulation Approach, Its History, Conceptual Domain, and Application to Conservation, in: Metapopulation Biology. Elsevier, pp. 5–26. https://doi.org/10.1016/B978-012323445-2/50003-1

Hatzikirou, H., Basanta, D., Simon, M., Schaller, K., Deutsch, A., 2012. "Go or Grow": the key to the emergence of invasion in tumour progression? Math. Med. Biol. 29, 49–65. https://doi.org/10.1093/imammb/dqq011

Heide-Jørgensen, M.P., Richard, P.R., Dietz, R., Laidre, K.L., 2013. A metapopulation model for Canadian and West Greenland narwhals: Narwhal metapopulation. Anim. Conserv. 16, 331–343. https://doi.org/10.1111/acv.12000

Huang, J.-L., Koh, C.-K., Cauley, S.F., 2009. Logic and circuit simulation, in: Electronic Design Automation. Elsevier, pp. 449–512. https://doi.org/10.1016/B978-0-12-374364-0.50015-1

Ingerson, T.E., Buvel, R.L., 1984. Structure in asynchronous cellular automata. Phys. Nonlinear Phenom. 10, 59–68. https://doi.org/10.1016/0167-2789(84)90249-5

Janelle, D.G., 2005. Time–Space Modeling, in: Encyclopedia of Social Measurement. Elsevier, pp. 851–856. https://doi.org/10.1016/B0-12-369398-5/00347-9

Kermack, W.O., McKendrick, A.G., 1927. A contribution to the mathematical theory of epidemics. Proc. R. Soc. Lond. Ser. Contain. Pap. Math. Phys. Character 115, 700–721. https://doi.org/10.1098/rspa.1927.0118

Khalil, H., Wainer, G., Dunnigan, Z., 2020. Cell-DEVS Models for CO_2 Sensors Locations in Closed Spaces, in: 2020 Winter Simulation Conference (WSC). Presented at the 2020 Winter Simulation Conference (WSC), IEEE, Orlando, FL, USA, pp. 692–703. https://doi.org/10.1109/WSC48552.2020.9383937

Muneepeerakul, R., Weitz, J.S., Levin, S.A., Rinaldo, A., Rodriguez-Iturbe, I., 2007. A neutral metapopulation model of biodiversity in river networks. J. Theor. Biol. 245, 351–363. https://doi.org/10.1016/j.jtbi.2006.10.005

Murray, A.B., Paola, C., 1994. A cellular model of braided rivers. Nature 371, 54–57. https://doi.org/10.1038/371054a0

Muzy, A., Innocenti, E., Aiello, A., Santucci, J.F., Wainer, G., 2005. Specification of discrete event models for fire spreading. Simulation 81, 103–117. https://doi.org/10.1177/0037549705052230

Sonnenschein, M., Vogel, U., 2001. Asymmetric cellular automata for the modelling of ecological systems, in: Sustainability in the Information Society. Presented at the EnviroInfo.

St-Aubin, B., Menard, J., Wainer, G., 2021. A Web Based Modeling and Simulation Environment to Support the DEVS Simulation Lifecycle, in: 2021 Annual Modeling and Simulation Conference (ANNSIM). Presented at the 2021 Annual Modeling and Simulation Conference (ANNSIM), IEEE, Fairfax, VA, USA, pp. 1–12. https://doi.org/10.23919/ANNSIM52504.2021.9552123

Stroustrup, B., 2013. The C++ programming language, Fourth edition. ed. Addison-Wesley, Upper Saddle River, NJ.

Tang, L., Zhou, Y., Wang, L., Purkayastha, S., Zhang, L., He, J., Wang, F., Song, P.X. -K., 2020. A Review of Multi-Compartment Infectious Disease Models. Int. Stat. Rev. 88, 462–513. https://doi.org/10.1111/insr.12402

Tariq, J., Kumaravel, A., 2016. Construction of cellular automata over hexagonal and triangular tessellations for path planning of multi-robots, in: 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). Presented at the 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), IEEE, Chennai, pp. 1–6. https://doi.org/10.1109/ICCIC.2016.7919686

Toffoli, T., Margolus, N., 1987. Cellular Automata Machines: A New Environment for Modeling, MIT Press series in scientific computation. MIT Press, Cambridge, Mass.

Vangheluwe, H., 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling, in: Proceedings of the IEEE International Symposium on Computer-Aided Control System Design. Presented at the IEEE International

Symposium on Computer-Aided Control Systems Design, IEEE, Anchorage, AK, USA, pp. 129–134. https://doi.org/10.1109/CACSD.2000.900199

Wainer, G., 2009. Discrete-event modeling and simulation: a practitioner's approach, Computational analysis, synthesis, and design of dynamic models series. CRC Press, Boca Raton.

Wainer, G., 2006. Applying Cell-DEVS Methodology for Modeling the Environment. SIMULATION 82, 635–660. https://doi.org/10.1177/0037549706073698

Wainer, G., 2002. CD++: a toolkit to develop DEVS models. Softw. Pract. Exp. 32, 1261–1306. https://doi.org/10.1002/spe.482

Wainer, G., Giambiasi, N., 2001. Timed Cell-DEVS: Modelling and Simulation of Cell Spaces, in: Discrete Event Modeling and Simulation Technologies. pp. 187–214.

Wang, S., Van Schyndel, M., Wainer, G., Rajus, V.S., Woodbury, R., 2012. DEVS-based Building Information Modeling and simulation for emergency evacuation, in: Proceedings - Winter Simulation Conference. IEEE. https://doi.org/10.1109/WSC.2012.6465087

Ward, D.P., Murray, A.T., Phinn, S.R., 2000. A stochastically constrained cellular model of urban growth. Comput. Environ. Urban Syst. 24, 539–558. https://doi.org/10.1016/S0198-9715(00)00008-9

Watts, D.J., Muhamad, R., Medina, D.C., Dodds, P.S., 2005. Multiscale, resurgent epidemics in a hierarchical metapopulation model. Proc. Natl. Acad. Sci. 102, 11157–11162. https://doi.org/10.1073/pnas.0501226102

Wolfram, S., 2002. A new kind of science. Wolfram Media, Champaign, IL.

Wolfram, S., 1984. Cellular automata as models of complexity. Nature 311, 419–424. https://doi.org/10.1038/311419a0

Zeigler, B.P., Praehofer, H., Kim, T.G., 2000. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems, 2nd ed. ed. Academic Press, San Diego.

Zhong, S., Huang, Q., Song, D., 2009. Simulation of the spread of infectious diseases in a geographical environment. Sci. China Ser. Earth Sci. 52, 550–561. https://doi.org/10.1007/s11430-009-0044-9

Appendix A. Cadmium Simulation Library for DEVS Models

This appendix presents the API provided by Cadmium for developing computational models based on the DEVS formalism. The new version of Cadmium uses this API to support the classic and asymmetric Cell-DEVS formalism. Figure 17 shows a simplified UML class diagram of the library provided by Cadmium to implement DEVS models.

Cadmium implements the DEVS formal specifications presented in Section 2. To build atomic models, a new class needs to be defined, which must inherit from the `Atomic<S>` class. The template argument `S` specifies the data type used by the model to represent its state. The state transition, output, and time advance functions correspond to pure virtual methods that must be overwritten. These methods are marked as constant so that the compiler prevents them from modifying model attributes directly. Instead, state transition functions provide a mutable reference to the model's state, and the output function offers a reference to the model's output set for creating new output events. With this approach, modelers must comply the DEVS specifications (e.g., it is impossible to modify the model's state from an output function). The `Atomic<S>` class inherits from the `AtomicInterface` class. The latter is an interface for atomic models that hides their state data type and allows us to treat atomic models equally regardless of their state data type.

The `Coupled` class provides a set of attributes and methods to build DEVS coupled models (i.e., adding subcomponents and couplings between them). The subcomponents of a given coupled model may be either atomic or coupled. The `AtomicInterface` and `Coupled` classes inherit from a common parent: the `Component` class, which implements those aspects shared by atomic and coupled models (namely, the input and output sets). In Cadmium, the input and output sets are implemented by the `PortSet` class. An object of the `PortSet` class is a vector of elements of the `Port<T>` class, which are vectors of messages of type `T`. Couplings between models are defined from one port to another. When an atomic model executes its lambda function and injects a message into one of its output ports, this message is propagated to all ports coupled to the port. Ports of class `Port<T>` can only store messages of their template type `T`. Therefore, only ports of the same type can be coupled. Therefore, Cadmium will detect invalid couplings before starting a simulation. The `PortInterface` class allows storing together ports that contain messages of different data types.

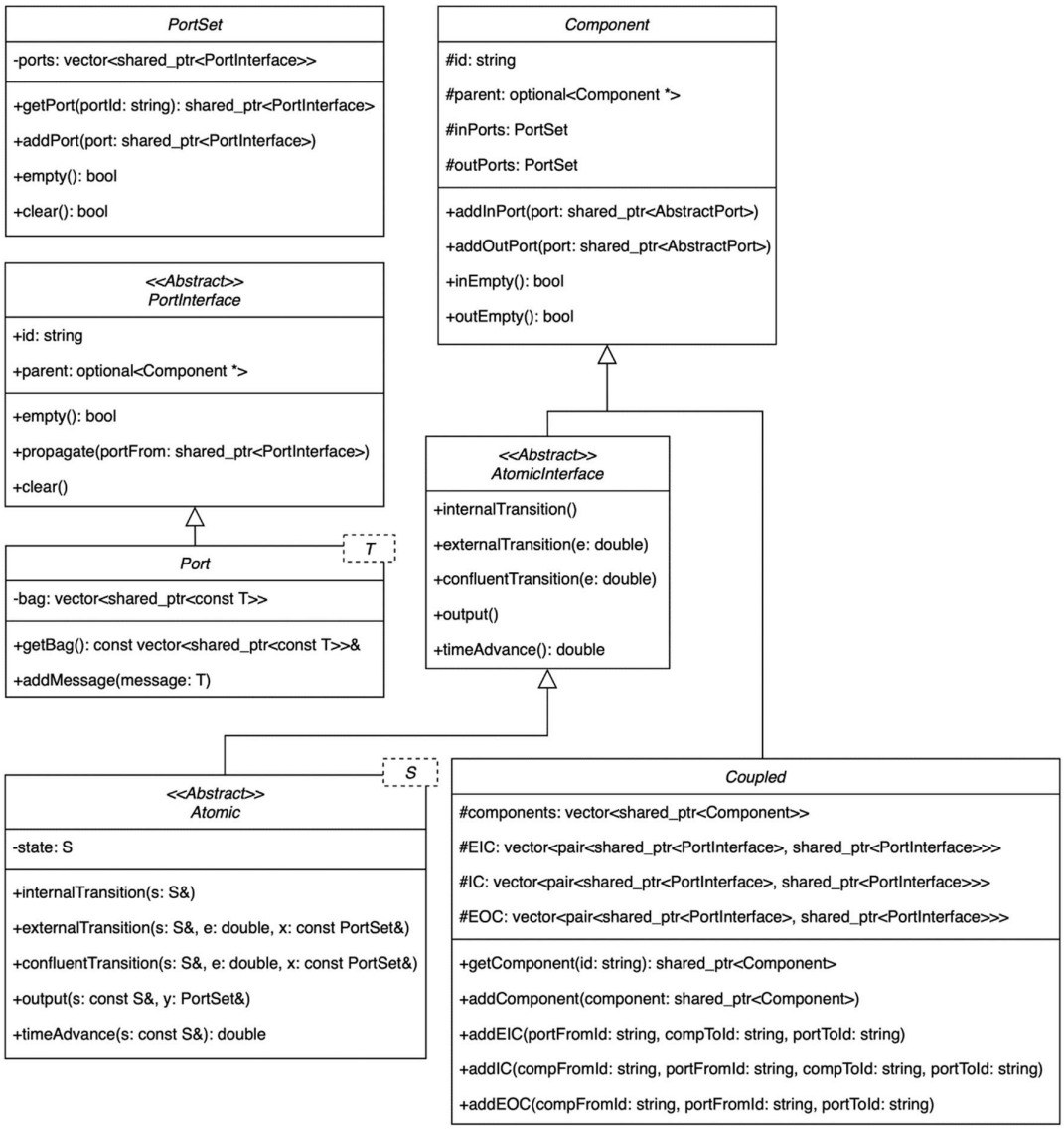


Figure 17. UML class diagram of DEVS models in Cadmium.