The 9th International Conference on Modelling, Identification and Control
(ICMIC 2017), Kunming, China, July 10-12, 2017

ACD++: a Domain Specific Language for Cell-DEVS Modelling

Chong Jiao and Baohong Liu

***Abstract*— This paper introduces a library ACD++ for modeling and simulation of cellular models based on Cell-DEVS formalism. The goal is to allow the modeling of cellular models more flexible and adaptive. ACD++ is implemented in Ruby programming language, providing an internal Domain Specific Language (DSL) to simplify the construction of cellular models. Ruby's meta-programming characteristics and plentiful syntactic sugar enables the easy expression of complex logics behind the models. The Cell-DEVS formalism proved consistent with the DEVS hierarchy, improving the description of complex systems. Another strength lies in the extensibility of the DSL, allowing the modelers to introduce their domain specific vocabulary to facilitate the definition of specific models. The use of this library has allowed the development more flexible and adaptive, significantly reducing development time.**

***Index Terms*— DSL; Cell-DEVS; cellular model; modeling and simulation**

I. INTRODUCTION

In recent years, the flexibility and adaptability of modeling and simulation has become the main concern especially in the area of decision-making. For instance, in the simulation of complex systems for decision making, the decision makers often expect the simulation to be adaptive to the dynamic changing decision environment [1]. Many technologies have been proposed to enable the flexibility of modeling from the perspective of design pattern, aspect-oriented programming, or modeling language. Our work should be placed at the latter one, which aims to provide an internal Domain Specific Language (DSL) to improve the description of complex systems which can be represented by cell spaces. Cellular Automata formalism (CA) has been widely used to describe those systems which can be represented as cell spaces [2]-[3]. CA is defined as n-dimensional lattices composed of cells with discrete and finite states. It can be viewed as a dynamic system, which evolves at separate points in time. However, the discrete time paradigm poses constraints on the precision and efficiency of the simulated models [4]. In addition, most cells would not update their states in each iteration, which further reduce efficiency. To solve it, the discrete event cellular models are proposed in paper [5] and are specified as Cell-DEVS, which is an extension of Discrete EVent System specification (DEVS). DEVS is proposed by professor Zeigler in 1976 and used to specify formally discrete event systems using a modular description. The discrete event paradigm and formal specification of Cell-DEVS formalism provide great advantage of being efficient, accurate and easy to verify. Many simulation tools have provided support for the Cell-DEVS formalism, such as James-II, CD++, and ADEVS [6]. However, these tools are either limited to the implemented programming language, or constrained by the modeling methodology. Thus they cannot provide enough flexibility and adaptability. To overcome it, we expect the simulation tools can provide mechanism for domain experts to create and modify the models according to domain knowledge. DSL is competent for it. A Domain-Specific Language (DSL) is a programming or description language tied to a specific application domain [7]. Compared with general programming language, DSL is more expressive, easily verified, and provides a good way to enable the flexibility of modeling. Ruby is a programming language invented in 1993, whose meta-programming characteristics and plentiful syntactic sugar provide strong support to grow a DSL. In this paper, we propose a modeling and simulation library, ACD++, based on Cell-DEVS specification. It is implemented in Ruby programming language and is devote to providing an internal DSL to improve the description of complex systems which can be represented by cell spaces.

The research work is supported by National Natural Science Foundation of China under Grant No. 61374185.

Chong Jiao and Baohong Liu are with College of Information System and Management, National University of Defense Technology, Changsha, 410073, P. R. China (jch_email@qq.com; liubh@nudt.edu.cn)

II. BACKGROUND

A. DEVS and Cell-DEVS formalism

DEVS formalism originates from system theory, providing a framework for the construction of hierarchical models in a modular manner [8]. In DEVS, basic models are specified as black boxes with input and output ports. Several models can be integrated together to form a hierarchical model. The integrated model is either atomic (behavioral) model or coupled (structural) model. The former models autonomous behavior and is specified as:

$$AM = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Where
X is the set of inputs
Y is the set of outputs
S is the set of states
$\delta_{int} : S \to S$ is the internal transition function
$\delta_{ext} : Q \times X \to S$ is the external transition function, where $Q = \{(s, e) | s \in S, 0 \leq ta(s)\}$ is the total state set and e is the elapsed time since last transition
$\lambda : S \to Y$ is the output function
$ta : S \to R_0^+$ is the time advance function

On reception of external event, $\delta_{ext}(s, e, x)$ is invoked using input value *x*, elapsed time *e*, and current state *s*. In the absence of external events, an atomic model will remain in state *s* until *ta(s)* expires. Then the output function $\lambda(s)$ is called and the model will transform into the new state

$\delta_{int}(s)$. The coupled model is composed of child models, each of them being atomic or coupled. Formally, the classical coupled models are specified as:

$$N =< X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, SELECT >$$

where
X is the input set of the coupled model
Y is the output set of the coupled model
D is the set of component references
$\{M_d | d \in D\}$ is the set of child models
EIC is the external input coupling set which connect inputs to component inputs
EOC is the external output coupling set which connect component outputs to outputs of *N*.
IC is the internal coupling set which connect component outputs to component inputs
$Select : 2^Z - \phi \rightarrow Z$ is the tie-breaking function
Therefore, a coupled model defines the components and their interactions. Detailed description can be found in [8].

Cell-DEVS is an extension of DEVS formalism, which defines each cell as an atomic model, and the cell space as a coupled model. Each cell holds state variables and some rules which are used to update its state using its present state and neighborhoods. The formal specification of Cell-DEVS atomic model can be found in [9]-[10]. It can be specified as:

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

Where
X, *Y*, and *S* represent input set, output set and state set respectively. *N* represents the number of inputs of a cell. The delay function *d* is associated with each cell, after which, the new state value is sent out. τ represents the local rules which are responsible to compute the future state using its current state and neighbourhoods' state. A complete cell space model can be defined as:

$$GCC =< X_{list}, Y_{list}, X, Y, n, \{t_1, t_2, \cdots, t_n\}, N, C, B, Z >$$

Where
X_{list} and Y_{list} represent input coupling list and output coupling list, which are used to define the coupling relation among external DEVS models. *n* defines the dimension of the cell space.$\{t_1, t_2, \cdots, t_n\}$ is the number of cells in each dimension. *N* defines the number of inputs of each cell, which is the same as the definition of *TDC*. *C* is an array of atomic cells. *B* represents the border cells, which may have different behaviours from the rest cells. *Z* represents the transition rules.

Both of DEVS and Cell-DEVS provide a formal specification that can reduce the cost of development. Thus they provide a good starting point to discuss flexibility of modelling and simulation.

B. Domain Specified Language

A Domain-Specific Language (DSL) is a programming or description language tied to a specific application domain. Unlike general-purpose language, it is designed for a particular kind of problem.

A DSL can either be external or internal. The former exists independently from other languages. The latter rely on the hosting language, which can be viewed as an enhancement of the hosting language [11]. In this paper, we provide an internal DSL implemented in Ruby. Ruby's dynamic binding mechanism and closures (a function of block can capture its referencing environment) makes you can execute the closure with the context of arbitrary objects, which provides strong support for the internal DSL.

Many simulation tools have provided support for the Cell-DEVS formalism, such as ADEVS [12], CD++ [13]-[14] and James-II [15]. Except for ADEVS that provides API for modelers to create Cell-DEVS models, CD++ and James-II both provide a DSL. However, both of them are based on an external DSL, which poses constraints on the extensibility and expressive power. Our work proposes an internal DSL implemented in Ruby for the modeling of Cell-DEVS models. Modellers can benefit from the convenience brought by DSL, while defining complex logics using Ruby. Another strength lies in the extensibility, allowing it to be extended to the specified domain.

III. DESIGN AND IMPLEMENTATION

ACD++ has been implemented using Ruby. The ACD++ library itself focuses on the improvement of modelling flexibility for cellular models. Our work is an extension of DEVS-Ruby, which is a DEVS modelling and simulation library implemented in Ruby proposed by paper [11].

In this section, we will first propose our design of DSL. Then, we discuss our modelling architecture and implementation.

A. Design of DSL

To improve the flexibility of modelling, we propose some requirements that an ideal DSL should possesses. (1) Be consistent with the Cell-DEVS specification; (2) Support the Cell-DEVS vocabulary, and be more accessible to the non-experts; (3) The created models can be easily integrated into the DEVS architecture; (4) Offer a way to extend the DSL.

As ACD++ implements Cell-DEVS theory, our first concern is to remain consistent with the specifications. The vocabulary of our DSL includes all the elements in the specification, such as neighbourhood, state, dimension.

Table I shows the specification of '*Game of Life*' [16]. From the specification, we can learn that the dimension of the cell space is 20×20. Each cell owns a state variable whose domain is [DEAD, ALIVE]. The state of cells whose coordinator is beyond the dimension is regarded as DEAD.

Moreover, the behaviour of cells is defined as a set of rules with the form $\{$*ACTION, DELAY, CONDITION*$\}$. These indicate that when the *CONDITION* is satisfied, the cell would take the *ACTION* after the *DELAYed* time expires. This is similar to the definition in CD++, which is described

TABLE I
THE SPECIFICATION OF GAME OF LIFE

```
size 20,20
states :state=>[:DEAD,:ALIVE]
border :constant, :state=>:DEAD
neighbor_type :moore, 1
init_with_value state: :ALIVE
init_with_maps [0,0]=>{:state => :DEAD},
               [0,1]=>{:state=> :DEAD}
```

TABLE II
THE RULES IN GAME OF LIFE

```
rule action{state(:DEAD)}, delay{1},
            condition{state == :ALIVE and
         count_range(2..3, :state => :DEAD)}
rule action{state(:ALIVE)}, delay{1},
            condition{state == :DEAD and
            count(:state=> :ALIVE) == 3}
```

in paper [13]-[14]. However, unlike CD++ which limits that each cell can only hold one state variable, ACD++ allows a cell having multi-variables. Thus, the *ACTION* can update several state variables at the same time. The rules in '*Game of Life*' are specified in Table II. A cell will transform into dead if its 2 or 3 neighbourhoods are dead. If the cell is dead, it will transform into alive when 3 of its neighbourhoods are alive.

In fact, the rules specified in Table II are purely Ruby code. The vocabularies such as *rule*, *action* and *condition* are just functions in Ruby. Those in the braces are block parameter of Ruby. This means modellers can define any logic conformed to Ruby's syntax, which provides convenience for models with complex logics. Each cell space can be coupled with other DEVS models. 1 presents the default structure of cell space informally. The default cell space model has an input port *in* and output port *out*. On reception of external event from input port *in*, the received messages will be broadcast to each cell. In this way, we can communicate with the cells on the fly, providing a flexible way to influence the behaviour of cells. Besides, each cell will send its current state when its state changes. Fig.

Table III gives an example in which a cell space model is coupled with a DEVS model, which receives messages from the port out and then prints the states of the cell space. First, we add the cell space model and the observer model into the DEVS hierarchy. Then, we couple the output *out* of the cell

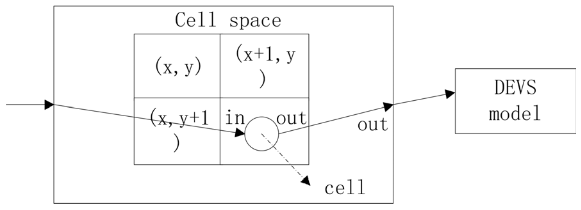

Fig. 1. The default structure of cell space

TABLE III
COUPLE THE CELL SPACE MODEL WITH OTHER DEVS MODELS

```
DEVS.simulate do
 duration 300
 add_cellspace(name: :cellspace) do
 size 20, 20
# other specification of cell space
 end
# another DEVS model
 add_model type: CellObserver, name::observer,
           with_args: [[20,20]]
  # coupled the two mode
 plug cellspace@out, with: observer@input
end
```

space with the input port *in* of the observer by calling the function *plug* and finish our specification.

In addition, to further improve the flexibility and adaptation, ACD++ provides several hooks to customize the behaviours of cell space. These hooks are listed is Table IV. Modellers can customize the behaviours of cell space like any other DEVS models.

Despite that we simplify the definition of Cell-DEVS models, we encourage the modeler to extend our DSL to introduce their own specific vocabulary. In this way, the end user can focus on the specified domains. To extend our DSL,

TABLE IV
PROVIDED HOOKS TO CUSTOMIZE THE BEHAVIOURS OF CELLS

hooks	description
cell_control(&block)	To customize the behavior of each cell. The block will be executed when a cell is instantiated. You can customize the cells behavior in the block the same as other DEVS models.
execute_before_cells-_be_established-(&block)	To customize the structure of cell space model. The block will be executed before all cells established. You can add sub-models, ports or coupling relations in the block for the space model.
execute_after_cells-_be_established-(&block)	To customize the structure of the cell space model. The block will be executed after all cells established.

the modellers should define the specified vocabulary based on our provided vocabulary, and then ACD++ provides a way to integrate them into our modelling architecture.

B. Implementation

Fig. 2 shows the modelling architecture of ACD++. The *CellModel* class represents the class of cells, and it is an atomic model. The *CellSpace* class represents the class of cell space and it is a coupled model.

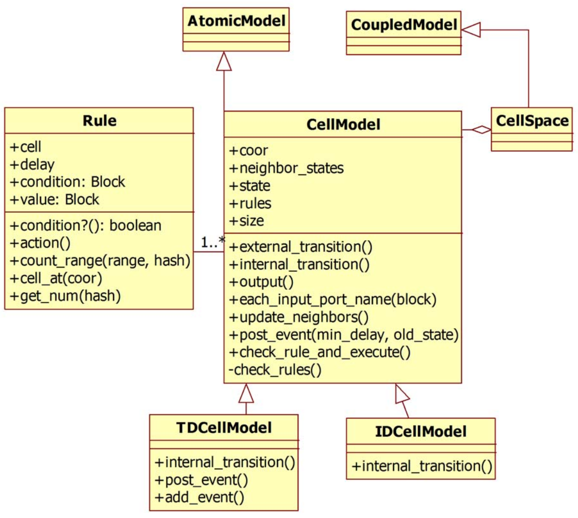

Fig. 2. Modelling architecture

Each cell is associated with a delay. According to the type of delay, the cell model can be divided into two kinds: transport and inertial, which is represented by *TDCellModel* and *IDCellmodel* in Fig. 2. Fig. 3 presents the informal specification of a cell with transport delay and inertial delay. When received an external event, the local rules are activated. The result of this computation will be delayed during *delay* time units. To do so, an internal event is scheduled. For a cell with transport delay in Fig 3 (a), a queue is used to preserve the result of computation. During the time of delay, new external event can arrive. But for a cell with inertial delay shown in Fig 3 (b), the result of computation is saved in *f*. External event during the delay may change the value of *f*.

Paper [9] proved these two kinds of models are both consistent with DEVS specification.

To build a simulation, we propose an internal DSL aiming to make the procedure of modelling and simulation more convenient and flexible. Our implementation is on the basis of DEVS-Ruby, which provides a DSL for DEVS modelling

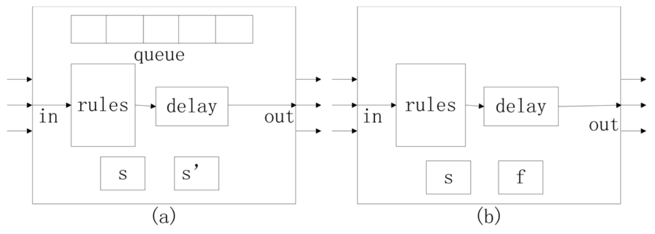

Fig. 3. Informal specification of a cell with transport delay and inertial delay. (a) transport delay (b) inertial delay

and simulation, whereas our work mainly focus on the modelling of Cell-DEVS models.

As seen in Table III, the function *simulate*, defined under the DEVS namespace, serves as the entry point into our DSL. It will do two things. First, it instantiates a *SimulationBuilder*, which is responsible to instantiate the root coordinator and root model. Then, it will execute the given block within the context of the builder.

The entry point of our DSL proposed for Cell-DEVS models is the function *add_cellspace*. Similarly, this will do three things. First, it will instantiate a *CellBuilder*, which is responsible to instantiate the cell space model and its associated processor. Then it will execute the given block within the context of the builder to complete the specification. Finally, it instantiates the cells and their associated processor according to the specification.

In addition, to improve the flexibility and adaptability of modelling, we adopt several hooks which has been listed in Table IV. The detailed procedures when establishing a Cell-DEVS model are depicted in Table V.

The relationship among builders is shown in Fig. 4 . The *AtomicBuilder* and *CoupledBuilder* are responsible for the building of atomic model and coupled model. The *SimulationBuilder* is inherited from *CoupledBuilder*, and it defines some vocabulary relevant to simulation and is responsible for the building of root model. The *CellBuilder* is inherited from *CoupledBuilder*, which is responsible for the building of Cell-DEVS models. Through the Mix-In mechanism of Ruby, the function *add_cellspace* can be mixed into *CoupledBuilder*, thus each coupled model can add Cell-DEVS model as its child model.

One of our strengths of our proposed DSL lies in its extensibility. We encourage the modellers to extend our DSL to introduce their own specific vocabulary. To extend our DSL, the modeller should define a builder inherited from our *CellBuilder*, which is responsible to instantiate the model and its associated processor. Then, the modeller defines the

TABLE V

PROCEDURES OF ESTABLISHING A CELL-DEVS MODEL

```
1. the function add_cellspace is called with
   a block parameter;
2. Instantiate the CellBuilder, and invoke
   its constructor function using the given
   block parameter;

   2.1 Instantiate a coupled model as the
       cell space as well as its processor;
   2.2 Execute the block within the context
       of CellBuider and complete the spec-
       ification;
   2.3 Invoke the before_establish hook to
       customize the structure of cell space
       model, which is described in hook
       execute_before_cells_be_established;
   2.4 Instantiate the cell model as the
       child model of cell space according
       to specification;
   2.5 Establish the coupling relation among
       the child models;
   2.6 Invoke the after_established hook to
       customize the structure of cell space
       model.
3. Finish the specification.
```

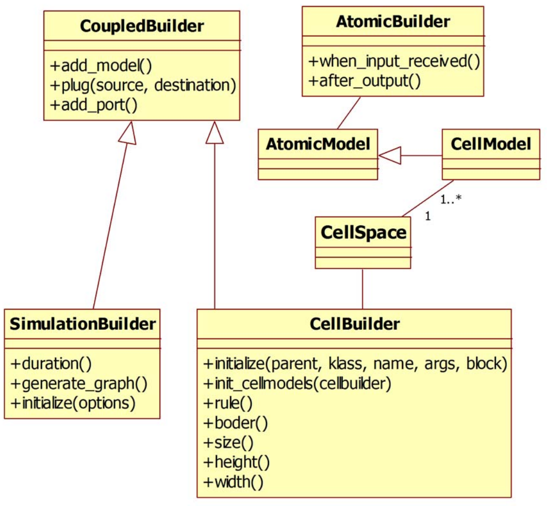

Fig. 4. The relationship among builders

TABLE VI

EXTEND DSL TO DESCRIBE THE TERRAIN MODEL

```
1. Declare the TerrainBuilder inherited from
   CellBuilder, and introduce the specified
   vocabulary as the instance methods, for
   example, side_length.
2. Define the entry point of DSL;
    module Terrain
      def add_terrain_model(name, &block)
        TerrainBuilder.new(name: name, &block)
      end
    end
3. Mix the function add_terrain_model into the
   CoupledBuilder.

   DEVS::CoupledBuilder.send :include Terrain
```

specified vocabulary as the instance methods of this builder. Third, the modeller should define an entry point similar to the function *add_cellspace*, which is responsible to instantiate the *CellBuilder*. Similar to the provided function *add_cellspace*, the entry point should also be mixed into *CoupledBuilder* so that each coupled model can add the specified model as its child model. An example which is used to extend out DSL to describe the terrain model is illustrated in Table VI.

On benefit of the abstract simulation mechanism of DEVS, we just utilize the simulation algorithms implemented in DEVS-Ruby [11], so that we can focus on the modelling of Cell-DEVS models.

IV. USE CASE STUDY

One of the strength of ACD++ lies in that it provides an internal DSL implemented in Ruby. Modellers can benefit from the convenience brought by DSL, while defining complex logics using Ruby. Another strength lies in the extensibility, allowing it to be extended to the specified domain. This section will illustrate a well-known model for fire propagation of forest fires [13], [17]. This model uses environmental and vegetation conditions to infer the fire spread ratio. To simplify the model, we only take the wind into account here. The state variable of each cell is a continuous variable which represent the time when the cell begin to burn. The fuel model, the speed and direction of the wind, terrain of topology and dimensions of cell are used to get the spread ratio in each direction. For instance, if the current is not burning (*ignite-time*==0), and the north-east neighbourhood (1,-1) has started to burn (whose *ignite-time*>0), the state variable *ignite-time* will be set to the result of the *ignite-time* of (1,-1) plus distance of the two cells

TABLE VII

FIRE SPREADING BUILDER

```
class FireSpreadBuilder < CellBuilder
  #east spreading ratio
  def east_speed(speed=nil)
   @east_speed=speed unless speed.nil?
   @east_speed
  end
  #other directions of spreading ratio are
    omitted out
  def fire_behavior
    #east_spread
   rule(
     action{igniteTime(cell_igniteTime_at(
           [1,0])+ side_length_of_cell/
            east_speed)},
     delay {side_length_of_cell/east_speed},
     condition{ igniteTime==0 and
           cell_igniteTime_at([1,0])>0}
        )
    # north_east_spread
    rule(
     action{igniteTime(cell_igniteTime_at(
            [1,-1])+Math.sqrt(2)*side_length-
            _of_cell/north_east_speed)},
     delay {Math.sqrt(2)*side_length_of_cell/
            north_east_speed},
     condition{ igniteTime==0 and cell_ignite-
            Time_at([1,-1])>0}
         )
    #north, north_west, west, south_west,
     # south, and south_east
  end
end
```

divided by north-east spreading ratio. To demonstrate the extensibility of our DSL, we will extend our DSL for this fire propagation simulation. As mentioned in section III, we first define a builder inherited from *CellBuilder*. As seen in the Table VII, we declare a builder *FireSpreadBuilder* which defines the specified vocabulary as its instance methods. For instance, the function *fire_behavior* defines the rules which are used to update their state. For instance, the first rule defined in the function *fire_behavior* indicates that when the cell is not burning or burned (*ignite-time*==0) and the east neighbourhood (1,0) is burning (ignite-time>0), it will become burning after the delayed time (using distance divided by east spread ratio). Due to the limitation of space, we did not give all the instance methods.

Last, we should define the entry point, function *add_fire_spread_model*, which is responsible to instantiate the *FireSpreadBuilder*, and mix it into the *CoupledBuilder*. Thus, we can add this fire propagation model as the child model of any coupled model. Finally, we finish specification of our simulation using the extended DSL in Table VIII.

TABLE VIII

SPECIFICATION OF THE FIRE PROPAGATION SIMULATION

```
DEVS.simulate do
  duration 30
  add_fire_spread_model(name: :fire_model) do
    size 20,20
    states :igniteTime=>[0,inf]
    neighbor_type :moore, 1
    border :constant, igniteTime: 0
    init_with_value igniteTime: 0

    init_with_maps [9,9]=>{igniteTime: 1}

    north_speed 5.106976
    north_west_speed 17.967136
    west_speed 5.106976
    south_west_speed 1.872060
    south_speed 1.146091
    south_east_speed 0.987474
    east_speed 1.146091
    north_east_speed 1.872060
    side_length_of_cell 15.24

fire_behavior

  end
# add the observer
 add_model type: CellObserver, name::observer
# couple the two models
 plug "fire_model@out", with:"observer@input"
end
```

As is shown is Table VIII , the length of simulation is set to 30. The first cell to start burning is at (9,9) at time 1, and Fig. 5 gives the 4 snapshots during the simulation.

Compared with CD++, one of strengths of our proposed DSL lies in its extensibility, which providing a good way to distinguish the role of modeller from the end-user. Thus the end-user does not have to know a lot about the library but focus on the domain knowledge.

V. CONCLUSION

This paper proposes a modelling and simulation library, ACD++, based on Cell-DEVS specification. The start point of our work is to allow the modelling of cellular models more flexible and adaptive. To achieve this, we propose an internal DSL allowing to easily express the modelling specification of Cell-DEVS. By using the Cell-DEVS specification, the cell model can be easily coupled with other DEVS models, providing a flexible way to influence the behaviour of cells. In addition, the internal DSL implemented in Ruby determines that modeller can easily define complex logics using Ruby.

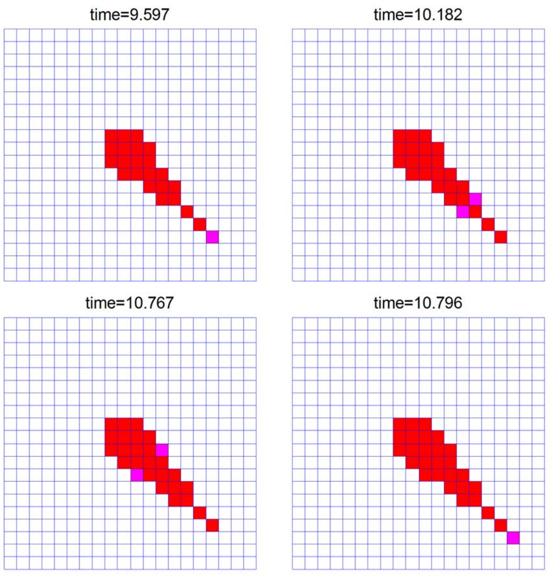

Fig. 5. snapshots during the simulation

References

[1] B. Liu and L. Lin, "The collaboration and coevolution of experts system and simulation system in agile modeling and simulation," in *Third International Conference on Intelligent System Design and Engineering Applications*, 2013, pp. 1404–1407.

[2] M. Sipper, "The emergence of cellular computing," *Computer*, vol. 32, no. 7, pp. 18–26, 1999.

[3] S. Wolfram, *Theory and applications of cellular automata*. World scientific, 1986.

[4] G. A. Wainer and N. Giambiasi, "Application of the cell-devs paradigm for cell spaces modelling and simulation," *Simulation Transactions of the Society for Modeling & Simulation International*, vol. 76, no. 1, pp. 22–39, 2001.

[5] ——, *Timed Cell-DEVS: Modeling and Simulation of Cell Spaces*. New York: Springer, 2001.

[6] R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, and D. Hill, "A survey of modelling and simulation software frameworks using discrete event system specification," in *Imperial College Computing Student Workshop*, 2014, p. 40C49.

[7] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *Acm Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[8] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*. Academic, 2000.

[9] G. A. Wainer and N. Giambiasi, "N-dimensional cell-devs models," *Discrete Event Dynamic Systems*, vol. 12, no. 2, pp. 135–157, 2002.

[10] A. Lpez and G. A. Wainer, "Improved cell-devs model definition in cd++," in *Cellular Automata, International Conference on Cellular Automata for Research and Industry, Acri 2004, Amsterdam, the Netherlands, October 25-28, 2004, Proceedings*, 2004, pp. 803–812.

[11] R. Franceschini, P.-A. Bisgambiglia, P. Bisgambiglia, and D. Hill, "Devs-ruby: a domain specific language for devs modeling and simulation (wip)," in *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative*, Tampa, United States, 2014, p. 15.

[12] J. Nutaro. (1999) Adevs (a discrete event system simulator). [Online]. Available: http://www.ece.arizona.edu/nutaro/index.php

[13] G. A. Wainer, "Cd++: a toolkit to develop devs models," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1261–1306, 2002.

[14] E. Glinsky and G. A. Wainer, "New parallel simulation techniques of devs and cell-devs in cd++," in *Proceedings of the 39th annual Symposium on Simulation*, 2006, pp. 244–251.

[15] S. Zinn, J. Himmelspach, A. M. Uhrmacher, and J. Gampe, "Building mic-core, a specialized m&s software to simulate multi-state demographic micro models, based on james ii, a general m&s framework," *Journal of Artificial Societies and Social Simulation*, vol. 16, no. 3, p. 5, 2013.

[16] M. Gardner, "Mathematical games: The fantastic combinations of john conways new solitaire game life," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.

[17] R. C. Rothermel, "A mathematical model for predicting fire spread in wildland fuels," *Usda Forest Service General Technical Report*, vol. 115, 1972.