Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

A NEW SIMULATION ALGORITHM FOR PDEVS MODELS WITH TIME ADVANCE ZERO

Cristina Ruiz Martin
Guillermo G. Trabes
Gabriel A. Wainer

Dept. of Systems and Computer Engineering
1125 Colonel By Dr.
Ottawa, K1S 5B6, CANADA

ABSTRACT

Discrete Event Systems Specification (DEVS) is a well-known formalism to develop models using the discrete event approach. One advantage of DEVS is a clear separation between the modeling and simulation activities. The user only needs to develop models and general algorithms execute the simulations. The PDEVS simulation protocol is a well-know and widely accepted algorithm to execute DEVS simulations. However, when events are scheduled with time advance equal to zero, this algorithm handles them sequentially. Events that occur at the same time are processed one after the other. This may result in unwanted simulation results. In this work, we propose a new algorithm that assures that the output bag of a model is transmitted only when all the outputs corresponding to a given simulation time have been collected.

1 INTRODUCTION

Modeling and Simulation (M&S) has become an essential tool in science and engineering. Its ability to represent problems in several disciplines and perform scientific exploration has increase its popularity. There are many methodologies to develop M&S solutions, and some of them allow defining the models formally, which has a few advantages. In particular, the Discrete Event System Specification (DEVS) (Zeigler et al. 2000) provides a theoretical framework to develop discrete-event M&S and it was used in many applications since its creation.

In DEVS, models are defined using two kinds of components: atomic models and coupled models. Atomic models define the behavior of the elements of the system, whereas coupled models define their structure. The various components of the model interact with each other through well-defined modular interfaces. In some versions of DEVS, such interfaces include the definition of input/output ports.

The formal definition of DEVS provides many advantages. One of them is the capacity to separate model definition, implementation, and experimentation. Models that are valid under a given experimental frame are defined using a formal notation and then simulated using algorithms that have been formally verified. This separation of concerns boosts the reusability of models and ease the verification of the models.

Sometimes, when building a discrete-event model, we need to represent the occurrence of simultaneous events. In classic DEVS, when simultaneous events occur, the simulation algorithm executes the models involved in according to the specifications defined in a tie-break function. This function specifies the order of execution of the model's components when they have simultaneous events to be executed. This way of handling collisions might not be adequate to reflect the actual response of the system to simultaneous events. To deal with this problem, Parallel DEVS (PDEVS) was introduced to deal with simultaneous events more elegantly (Chow and Ziegler, 1994). One of the changes of PDEVS is that enables the modeler to define the behavior of the components when there are collisions of events. To do so, PDEVS adds a new function in atomic components that deals with the collision, removing the need for the tie breaking function. Another major change is that PDEVS models also modifies the way in which inputs and outputs are defined.

PDEVS allows the transmission of bags of events as inputs and outputs, allowing transferring information about multiple input/output events simultaneously.

According to the PDEVS specifications, all outputs for a specific time are stored in an output bag and transmitted simultaneously. However, the PDEVS simulation algorithm in (Chow et al., 1994), which is used in numerous DEVS implementations, does not completely follow the above-mentioned specifications. In some cases, the output bag of a model is transmitted before all the outputs for at a given time are collected. The result is that we transmit multiple output bags at the same simulation time, which make the models more complex to define as the simulation execution does not match the PDEVS specification exactly.

We present a revised version of the PDEVS Abstract Simulator that addresses this issue. The revised version of the algorithm assures that the output bag of a model is transmitted only when all the outputs corresponding to a given simulation time have been collected.

The rest of the paper is organized as follows. In section 2, we summarize DEVS and PDEVS. We also explain the original PDEVS simulation algorithm. In section 3 we explain the issues with the current algorithm through two examples. In section 4, we introduce the proposed modification to the algorithm, and in section 5, we define the execution traces of the new algorithm through two examples. Finally, in section 6, we present the conclusions and future research lines of this work.

2 BACKGROUND AND RELATED WORK

2.1 DEVS

Discrete Event System Specification (DEVS) (Zeigler et al. 2000) is a well-known mathematical formalism that provides a theoretical framework to think about modeling using a hierarchical, modular approach. In DEVS, atomic models provide behavior and coupled models provided structure.

The atomic models are defined as a tuple: $A = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where: S is the set of states, X is the set of input ports and values, Y is the set of output ports and values, δ_{int}: $S \rightarrow S$ is the internal transition function, $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the local state set (where e is the time elapsed since the last transition), δ_{ext}: $Q \times X \rightarrow S$ is the external transition function, $\lambda : S \rightarrow Y$ is the output function, and ta: $S \rightarrow \mathbb{R}^+$ is the time-advance function.

An atomic model, also known as a basic model, is always in a specific state waiting to complete the lifespan delay returned by the *ta* function, unless an input of a new external event occurs. If no external event is received during the lifespan delay, the output function λ is called first, and then the state is changed according to the value returned by the δ_{int} function. If an external event is received, then the state is changed according to the value returned by the δ_{ext} function, but no output is generated.

Coupled models define a network structure in which nodes are atomic or coupled models and directed links represent the routing of events between outputs and inputs or to/from the upper level. Formally, a Coupled Model is represented by the tuple $C = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, SELECT \rangle$, where: X is the set of input events, Y is the set of output events, D is an index for the components, $M_i \mid i \in D$, is a Classical-DEVS models as defined previously, I_i, are the influencees of model i, $\forall j \in I_i$, Z_{ij}: $Y_i \rightarrow X_j$ is the i to j translation function and *SELECT:* $2^D \setminus \emptyset \rightarrow D$ is the tie-breaker function that sets priority in case of simultaneous events.

The formal definitions of DEVS provides many advantages. DEVS has the capacity to separate model definition, implementation, and experimentation. Models that are valid under a given experimental frame are defined using a formal notation and then simulated using algorithms that have been formally verified. This separation of concerns along with its hierarchical and modular approach boosts the reusability of models and ease the verification of the models.

2.2 PDEVS

Even though Classic-DEVS has been used in many applications and tools, it has a limitation when dealing with simultaneous events. Simultaneous events are handled sequentially based on the order specified in the tie-break SELECT function. This collision behavior may not accurately represent the behavior of the actual system.

Parallel DEVS (PDEVS) (Chow and Ziegler, 1994; Chow et al., 1994) was introduced to deal with tie-breaking and better handling of simultaneous events. PDEVS introduces two main characteristics:

- The inputs and outputs for every PDEVS model, *X,* and *Y* respectively, are defined as bags (multisets) instead of sets, as in classical DEVS. In this way, multiple elements can be transmitted at the same time.
- A confluent function is introduced which defines the model's behavior when an internal and external transition are scheduled at the same time.

With these new features PDEVS can handle the occurrence of multiple events at the same time in a simple way, and therefore, tie-break function SELECT, defined in classical DEVS, is no longer needed.

The PDEVS atomic models are defined as a tuple: A = < *S, X, Y,* δ_{int}, δ_{ext}, δ_{conf}, λ, *ta* > where: *S,* δ_{int}, and *ta* are defined as in classical DEVS. As mentioned, *X* and *Y* are defined as bags of elements. The output, external and the additional confluent function are defined respectively, as follows: $\lambda: S \rightarrow Y^b$, δ_{ext}: $Q \times X^b \rightarrow S$ and δ_{conf}: $Q \times X^b \rightarrow S$.

In addition to the development of the model itself, there have been many efforts in the development of a simulator. Following the ideas from classical DEVS, PDEVS makes a clear separation between the model and the simulation. The models are defined by users following the specifications defined by the formalism and, to execute simulations, a general mechanism is provided. This mechanism is known as the PDEVS Abstract Simulator. In this section, we review the Abstract Simulator defined by (Nutaro, 2019).

Given a PDEVS model, the PDEVS Abstract Simulator creates a structure that allows to execute the behavior of the model and to obtain the correct simulation results. The PDEVS Abstract Simulator consists of three types of components: simulators, coordinators, and root-coordinator. Each atomic model is associated with a simulator and each coupled model is associated with a coordinator. One root-coordinator is placed at the root of the structure hierarchy.

This simulation procedure is implemented by exchanging several types of messages between the components. These are messages for initialization (*i*), to compute output (*) to execute a state transition (*x*) and send outputs (*y*). In contrast to classical DEVS where imminent models are sequentially activated, the coordinator enables concurrent execution of state transitions and output calculations for atomic models. The outputs of these models are collected into a bag called the mail. The mail is analyzed to determine the part going outside the scope of the coordinator due to external output coupling and the parts to be distributed internally due to internal coupling. The internal transition functions of the imminent models are not executed immediately since they may also receive input at the same simulation time. Similarly, as with the simulators, the coordinators react to *i*, *, *x* and *y* messages sent by a parent coordinator, and they reply to messages received from a subordinate. At the top of this hierarchy is a root-coordinator whose role is to initiate *i*, *, and *x* messages in each simulation cycle. The algorithms of the root-coordinators, coordinators and simulators are presented in Fig. 1, 2 and 3, respectively.

```
1: Parallel-Devs-root-coordinator
2: variables:
3:  t
4:  child
5: t = t0
6: send initialization message (i,t) to
   subordinate
7: t = tn of its subordinate
8: loop
9:  send(*,t)   message to child
10:  t = tn of its child
11: end loop
12: until end of simulation
13: end parallel-devs-root-coordinator
```

Figure 1: Root-Coordinatorr for PDEVS.

```
Parallel-Devs-coordinator
variables:
    DEVN = (X,Y,D,{M_d},{I_d},{Z_i,d})
    parent
    tl
    tn
    event-list
    IMM
    mail
    y_parent
    {y_d}
when receive i-message (i,t) at time t
 for d ∈ D do
  send i-message to child d
 end for
 sort event-list according to tn_d
 tl = max{tld|d ∈ D}
 tn = min{tnd|d ∈ D}
when receive *-message (*,t)
 if t ≠ tn then
  error: bad synchronization
 end if
 IMM = min(event-list)
 for r ∈ IMM do
  send *-messages(*,t) to r
 end for
when receive x-message(x,t)
 if not(tl ≤ t ≤ tn) then
  error: bad synchronization
 end if
 receivers={r|r ∈ children, N ∈ I_r, Z_N,r(x)≠∅}
for r in receivers do
 send x-messages(Z_N,r(x),t) with input value
  Z_N,r(x) to r
end for
for r ∈ IMM and not in receivers do
 send x-message (Φ, t) to r
end for
sort event-list according to tn_d
tl = t
tn = min tn_d | d ∈ D
when receive y-message (y_d,t) with output y_d
 from d
 if this is not the last d in IMM then
  add(y_d, d) to mail
  mark d as reporting
 else if this is the last d in IMM then
  y_parent = ∅
 end if
 for d ∈ I_N then
  if Z_d,N(y_d) ≠ ∅ then
   add y_d to y_parent
  end if
 end for
 send y-message (y_parent, t) to parent
 for child r, xr = ∅ do
  for d such that d ∈ I_r do
   if Z_d,N(y_d) ≠ ∅ then
    add y_d to y_r
   end if
  end for
 end for
 receivers = {r|r ∈ children, y_r ≠ ∅}
 for r ∈ receivers do
  send x-messages (y_r,t) to r
 end for
 for r ∈ IMM and not in receivers do
  send x-messages (∅,t) to r
 end for
 tl = t
 tn = min tn_d | d ∈ D
 sort event-list according to tn_d
end Parallel-Devs-coordinator
```

Figure 2: Coordinator for PDEVS.

```
Parallel-Devs-simulator
variables:
    parent
    tl
    tn
    DEVS
    y
when receive i-message (i,t) at time t
 tl = t – e
 tn = tl + ta(s)
when receive *-message
 if t = tn then
  y = λ(s)
  send y-message(y,t) to parent coordinator
 end if
when receive x-message (x,t)
 if x = ∅ ∧ t = tn then
  s = δ_int(s)
 else if x ≠ ∅ ∧ t = tn
  s = δ_conf(s)
 else if x ≠ ∅ ∧ (tl ≤ t ≤ tn)
  e = t - tl
  s = δ_ext(s,e,x)
 end if
 tl = t
 tn = tl + ta(s)
end Parallel-Devs-Simulator
```

Figure 3: Simulator for PDEVS.

The PDEVS simulation algorithm was implemented in several simulators over the years such as DEVS-Suite Simulator (Kim et. al, 2009), Adevs (Nutaro 2014), and Cadmium (Belloli et. al, 2019). A detailed list of DEVS simulators can be found in http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm.

3 THE PDEVS SIMULATION ALGORITHM

In this section, we show how the PDEVS simulation protocol works and how this protocol does not follow the definition of input/output bags as in the formal specification. We show how the algorithm handles output bags when there are states with time advance zero.

Let us assume that we want to model and simulate a DEVS couple model whose structure is shown in Figure 4. The model, named COUPLED includes two atomic models: GENERATOR and STORAGE. COUPLED does not have any inputs or outputs; GENERATOR does not have any inputs; and STORAGE does not have any outputs. The output of GENERATOR is connected to the input of STORAGE.

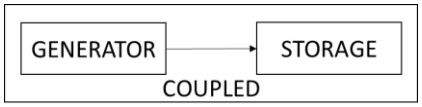

Figure 4: A DEVS model example.

The formal definition of these models is as follows:

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

$S = \{0, 1\}$;
$X = \emptyset$;
$Y = \{0, 1\}$;
$\delta_{int}(0) = 1$; $\delta_{int}(1) = 0$;
δ_{ext}: $\emptyset$;
δ_{conf}: $\emptyset$;
$\lambda\ (0) = \{0\}$; $\lambda\ (1) = \{1\}$;
$ta\ (0) = 1$ s.; $ta\ (1) = 0$ s.

GENERATOR has two states: 0 and 1 ($S = \{0, 1\}$; initial state: 0). It changes its state through internal transitions every second if state is 0 ($ta\ (0) = 1$ s.) and immediately if state is 1 ($ta\ (1) = 0$ s.). The output of GENERATOR is the value of its state, also 0 or 1 ($Y = \{0, 1\}$). When it is on state 0, it outputs a value of 0 ($\lambda\ (0) = \{0\}$), and changes to 1 ($\delta_{int}(0) = 1$); when it is on state 1 ($\lambda\ (1) = \{1\}$), it outputs a 1, and it changes to state 0 ($\delta_{int}(1) = 0$). As this model does not have inputs, the external and confluent transition functions will never occur ($X = \emptyset$; δ_{ext}: $\emptyset$; δ_{conf}: $\emptyset$).

We can notice that when this model is in state 1, it schedules its next state change in 0 seconds. This means that the next event is scheduled to occur at the same time as the previous one.

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

$S = \{0, 1, 0.1\}$;
$X = \{0, 1\}$;
$Y = \emptyset$;
$\delta_{int}(s) = s$;
$\delta_{ext}(s \in \mathcal{S}, e, \{0\}) = 0$; $\delta_{ext}(s \in \mathcal{S}, e, \{1\}) = 1$; $\delta_{ext}(s \in \mathcal{S}, e, \{0, 1\}) = 0.1$;
$\delta_{conf}(s \in \mathcal{S}, e, X^b) = \delta_{ext}(s \in \mathcal{S}, e, X^b)$;
λ: $\emptyset$;
$ta\ (0) = ta\ (1) = ta\ (0.1) = \infty$.

STORAGE has three states: 0, 1 and 0.1 (with initial state: 0). The model changes its state according to the inputs received: when it receives 0, the state is set to 0 (independently from the current state and elapsed time); when it receives 1, the state is set to 1; and when it receives an input bag = {0,1}, the state

is set to 0.1 (representing it received both value). The confluent function defines the same behavior as the external transition. The model is always passivated, the time advance is always set to infinity. Therefore, there are no outputs (λ: $\emptyset$) and the internal transition (which will never execute) does not change the state.

$$\text{COUPLED} = < X^b, Y^b, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >$$

$X^b = Y^b = \emptyset$;
$D = \{1, 2\}$;
$\{M_i\} = \{M_1 = \text{GENERATOR}, M_2 = \text{STORAGE}\}$;
$\{I_i\} = \{ I_{GENERATOR} = STORAGE \}$;
$\{Z_{ij}\} = \{ \text{GENERATOR} \rightarrow \text{STORAGE} \}$.

COUPLED describes the structure of the top model. It does not have any inputs or outputs. The model is composed by two atomic models, GENERATOR and STORAGE. Outputs of GENERATOR are connected to inputs of STORAGE.

To execute this model, an instance of the abstract PDEVS simulator must be created. Figure 5 shows the structure of PDEVS abstract simulator for this specific example. This structure is composed of a root-coordinator, a coordinator, and two simulators. COORDINATOR controls COUPLED; SIMULATOR_1 executes GENERATOR and SIMULATOR_2 executes STORAGE.

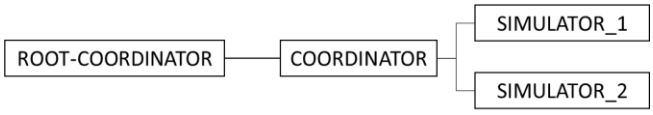

Figure 5: PDEVS abstract simulator for this example.

The simulation execution trace of this model according to the PDEVS simulation algorithm presented in (Chow et al., 1994) is show in Figure 6, where we can see the interactions among the components. We set the simulation start time at 0. The initial state of both models is 0.

First, ROOT-COORDINATOR sets the initial simulation time to 0 and sends an initialization message to COORDINATOR **(1)**, which transmits the message to SIMULATOR_1 and SIMULATOR_2 **(2)**. These two simulators calculate *ta*, *tn* and *tl* and report their time of next event *tn* to COORDINATOR **(3)**, which picks the first in its *event-list* (in this case, *tn=1*, as the SIMULATOR_2 is passive). This value is sent to the ROOT-COORDINATOR, which updates the simulation time to 1 and starts a simulation cycle. To do so, it sends a *collect outputs* message (*, 1) to COORDINATOR **(5)**, which updates its list of *imminent children (IMM)* and **(6)** sends *-messages to all children in *IMM*. Then, SIMULATOR_1 runs the output function, whose value is sent to COORDINATOR **(7)**, which updates the *receivers* set. Then it sends an *execute transition* message (x, 1) to every child in the *receivers* and *IMM* lists **(8)**. It sends ($\emptyset$,1) to SIMULATOR_1 because it has no inputs, and ({0},1) to SIMULATOR_2, because it receives the input bag {0} coming from the output of SIMULATOR_1. Then, SIMULATOR_1 runs the internal transition function and updates its state to 1 ($s = \delta_{int}(0) = 1$). It also calculates the time advance of the new state ($ta(1) = 0$), updates its *tn* and *tl* and report the time of next event *tn* to COORDINATOR **(9)**. At the same time, SIMULATOR_2 runs the external transition function with the input bag {0} and updates its state to 0 ($\delta_{ext}(0, e, \{0\}) = 0$). It also calculates the time advance of the new state ($ta(0) = \infty$.), updates its *tn* and *tl* and report the time of next event *tn* to COORDINATOR **(9),** which picks the first in its *event-list* (in this case, *tn=1*, as the SIMULATOR_2 is passive). This value is sent to the ROOT-COORDINATOR **(10)**, which updates the simulation time to 1 and starts a simulation cycle. Note that a new simulation cycle starts without the simulation time advancing. The simulation time is still 1. To do so, it sends a *collect outputs* message (*, 1) to COORDINATOR **(11)**. Steps **(12) – (17)** are a new simulation cycle like the one presented in steps **(6)-(11).** COORDINATOR updates its list of *imminent children (IMM)* **(12)** and sends *-messages to all

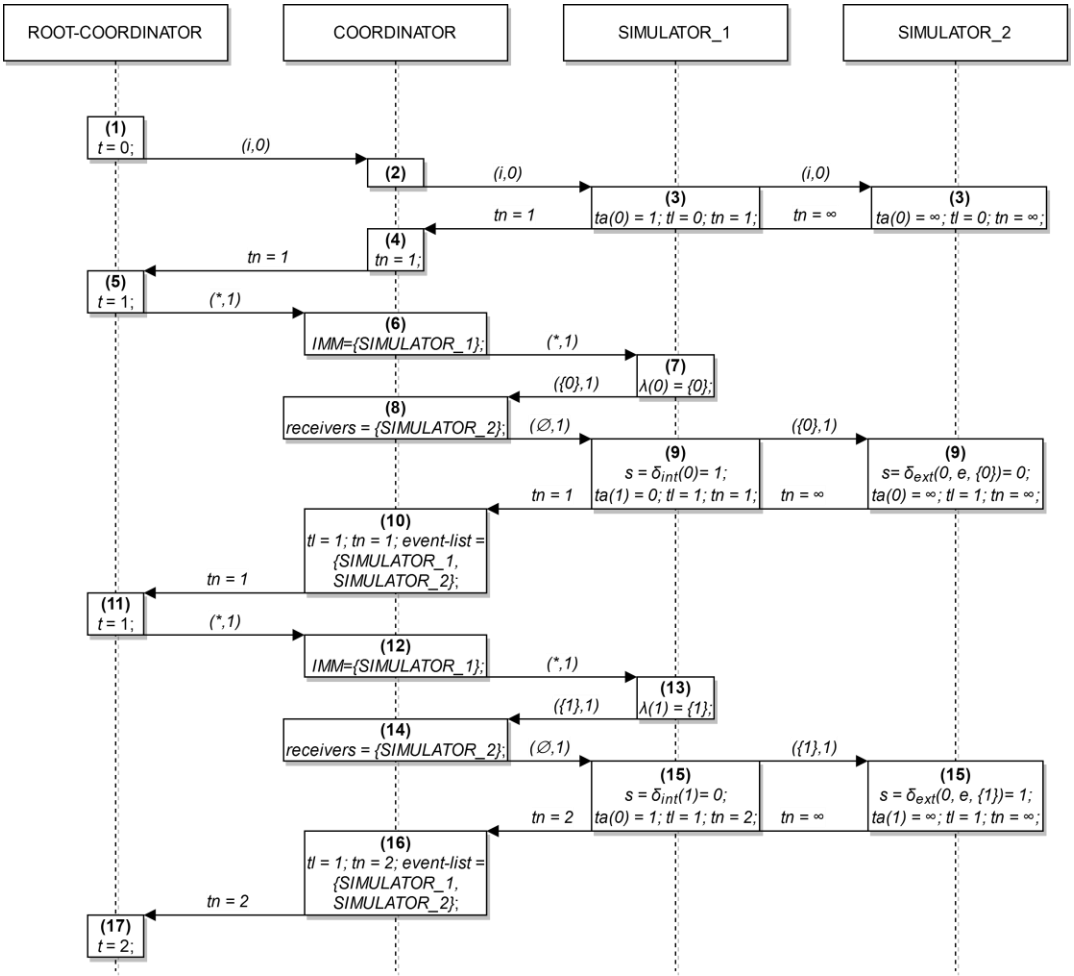

Figure 6: A PDEVS simulation protocol execution trace.

children in *IMM*. Then, SIMULATOR_1 runs the output function, whose value is sent to COORDINATOR **(13)**, which updates the *receivers* set **(14)**. Then it sends an *execute transition* message to every child in the *receivers* and *IMM* lists. It sends (Ø,1) to SIMULATOR_1 because it has no inputs and a ({1},1) to SIMULATOR_2, because it receives an input bag {1} coming from the output of SIMULATOR_1. Then, SIMULATOR_1 runs the internal transition function and updates its state to 1 ($s = \delta_{int}\ (1) = 0$). It also calculates the time advance of the new state (*ta* (0) = 1), updates its *tn* and *tl* and report the time of next event *tn* to COORDINATOR **(15)**. At the same time, SIMULATOR_2 runs the external transition function with the input bag {1} and updates its state to 1 ($\delta_{ext}(0, e, \{1\}) = 1$). It also calculates the time advance of the new state (*ta* (1) = ∞.), updates its *tn* and *tl* and report the time of next event *tn* to COORDINATOR **(15),** which picks the first in its *event-list* (in this case, *tn=2*, as the SIMULATOR_2 is passive). This value is sent to the ROOT-COORDINATOR **(16)**, which updates the simulation time to 2 and starts a simulation cycle (**17**). The simulation continues until the *tn* value of COORDINATOR is infinity or until the maximum simulation time is reached, whichever happens first.

In this execution trace we can see that GENERATOR produces 2 outputs at time 1: {0} (step 7) and {1}, (step 13). These outputs change the state twice on STORAGE. First, it changes to state 0 (step 9), and then, it changes to state 1 (step 15). However, as these outputs occur at the same moment (simulation time 1), a single output bag {0,1} should have been produced by GENERATOR at time 1, which should generate a state change to 0.1 in STORAGE.

The issue in the PDEVS simulation protocol is that it handles time advance zero sequentially: events that occur at the same time are processed one at the time in different simulation cycles. Consequently, the result may not be the one desired, as in this example. This suggests that a modification is needed.

One solution to this problem may be to modify GENERATOR so it can output a single bag with {0,1} value. For example, in this case GENERATOR may be modified to create the bag {0,1} as the output for state 1 (i.e. λ (1) = {0, 1}). However, this changes the semantics for this model, now the output for state 1 is not equal to the value of its state, as originally intended when it was designed. To achieve this, a new state should be included in the model, for example one called "0.1", making it more complex. Therefore, this solution is not ideal: it introduces changes and makes the model more complex without achieving any benefit. It would be desirable to address this issue without modifying the models.

In this work, we propose a modification over the PDEVS simulation algorithm to solve this problem. We modify the algorithm in such a way that we generate complete output bags when there are states with time advance zero instead of handling them one at the time. Models will be able to generate several individual outputs at a specific simulation time, while guaranteeing that they will be transmitted together. In the next section, we detail the proposed algorithm.

4 PROPOSED SIMULATION ALGORITHM

The proposed PDEVS simulation algorithm is like the one presented in (Nutaro, 2019) which follows (Chow et al., 1994). It includes simulators, coordinators, and a root-coordinator. The simulators are associated with atomic models and the coordinators to coupled models. The root-coordinator oversees the overall synchronization.

This simulation procedure is also defined by exchanging several types of messages between the components. These are messages for initialization (i), to compute output (*) to execute a state transition (x) and send outputs (y). Three new messages have been added to the simulation procedure: (1) a message to notify the coordinator that the output bag from the simulator is not yet filled (*nf*), (2) a message to tell the simulator to update its state to be able to continue filling the bag (*su*), and (3) a message to notify the coordinator that the simulator is ready to continue filling the bag (*rcf*).

4.1 Abstract simulator

The new abstract simulator is shown in Figure 7. The modifications to the original simulator are underlined.

```
 1: Parallel-Devs-simulator
 2: variables:
 3:     parent //parent coordinator
 4:     tl     //time of the last event
 5:     tn     //time of the next event
 6:     DEVS   //associated DEVS atomic model with
               //total state (s,e)
 7:     y      //output message bag
 8:     sp     //potential next state
 9:     tp     //potential time advance
10: when receive i-message (i,t) at time t
11:  if t = tn then
12:   y = y + λ(s)
13:   sp = δint(s)
14:   tp = ta(s)
15:  end if
16:  if tp = 0 then
17:   send nf-message (nf,t) to parent coordinator
18:  else
19:   send y-message(y,t) to parent coordinator
20:  end if
21: when receive su-message (su,t) at time t
22:  if tp = 0 then
23:   s = sp
24:   send rcf-message(rcf,t) to parent coordinator
25:  else
26:   error: bad synchronization
27:  end if
28: when receive x-message (x,t)
29:  clear y
30:  if x = ∅ ∧ t = tn then
31:   s = δint(s)
32:  else if x ≠ ∅ ∧ t = tn
33:   s = δconf(s)
34:  else if x ≠ ∅ ∧ (tl ≤ t ≤ tn)
35:   e = t - tl
36:   s = δext(s,e,x)
37:  end if
38:  tl = t
39:  tn = tl + ta(s)
40: end Parallel-Devs-Simulator
```

Figure 7: New simulator for PDEVS.

The changes in the abstract simulator are to ensure that all output messages generated by a simulator at the same simulation time are filled into a single output bag before they are transmitted to the coordinator. This new algorithm works in the following way: when an *-message is received, if the time of the *-message is equal to the simulator's next time, it adds the output (λ(s)) to the message bag (y). It also checks if the bag is ready or if there are more messages to be added. To do so, it calculates the next state of the model through an internal transition (*sp*) and its time advance (*tp*). If *tp* is not equal to zero, then the bag is full. It means, the simulator will not generate more outputs at the same simulation time. Therefore, the simulation continues as in the original version. The simulator sends a y-message to the parent coordinator. And if the simulator receives a su-message, a synchronization error occurred because *tp* is not equal to zero. If *tp* is equal to zero, there are more messages to be added to the output message bag (*y*) before the simulation time advances. In that case, an *nf-message* is sent to the parent coordinator (meaning that the simulator has more messages to add to the bag). In that case, when a *su-message* is received, the simulator updates the state of the model from s to *sp*, and an *rcf-message* is sent to the parent coordinator (meaning that the simulator is ready to continue filling the bag). The rest of the simulator algorithm is the same as in the original version. We clean the output bag before executing a transition and re-use the value of *sp* to avoid calculating the internal transition function again.

```
 1: Parallel-Devs-coordinator
 2: variables:
 3:  DEVN=(X,Y,D,{M_d},{I_d},{Z_i,d})//associated DEVS
coupled model
 4:  parent     //parent coordinator
 5:  tl         //time of last event
 6:  tn         //time of next event
 7:  event-list //element list(d,tn_d)sorted by tn_d
 8:  IMM        //list of imminent children
 9:  mail       //output mail bag
10:  y_parent   //output message bag to parent
                //coordinator
11:  {y_d}      //set of output message bags for
                //each child d
12: when receive i-message (i,t) at time t
13:  for d ∈ D do
14:   send i-message to child d
15:  end for
16:  sort event-list according to tn_d
17:  tl = max{tld|d ∈ D}
18:  tn = min{tnd|d ∈ D}
19: when receive *-message (*,t)
20:  if t ≠ tn then
21:   error: bad synchronization
22:  end if
23:  IMM = min(event-list)
24:  for r ∈ IMM do
25:   send *-messages(*,t) to r
26:  end for
27: when receive nf-message(fn,t) from d ∈ IMM
28:  send su-message(su,t) to d
29: when receive rcf-message (rcf,t) from d ∈ IMM
30:  send *-message(*,t) to d
31: when receive x-message(x,t)
32:  if not(tl ≤ t ≤ tn) then
33:   error: bad synchronization
34:  end if
35:  receivers={r|r ∈ children, N ∈ I_r, Z_N,r(x)≠∅}
36: for r in receivers do
37:  send x-messages(Z_N,r(x),t) with input value
     Z_N,r(x) to r
38: end for
39: for r ∈ IMM and not in receivers do
40:  send x-message (∅, t) to r
41: end for
42: sort event-list according to tn_d
43: tl = t
44: tn = min tn_d | d ∈ D
45: when receive y-message (y_d,t) with output y_d
    from d
46:  if this is not the last d in IMM then
47:   add(y_d, d) to mail
48:   mark d as reporting
49:  else if this is the last d in IMM then
50:   y_parent = ∅
51:  end if
52:  for d ∈ I_N then
53:   if Z_d,N(y_d) ≠ ∅ then
54:    add y_d to y_parent
55:   end if
56:  end for
57:  send y-message (y_parent, t) to parent
58:  for child r, xr = ∅ do
59:   for d such that d ∈ I_r do
60:    if Z_d,N(y_d) ≠ ∅ then
61:     add y_d to y_r
62:    end if
63:   end for
64:  end for
65:  receivers = {r|r ∈ children, y_r ≠ ∅}
66:  for r ∈ receivers do
67:   send x-messages (y_r,t) to r
68:  end for
70:  for r ∈ IMM and not in receivers do
71:   send x-messages (∅,t) to r
72:  end for
73:  tl = t
74:  tn = min tn_d | d ∈ D
75:  sort event-list according to tn_d
76: end Parallel-Devs-coordinator
```

Figure 8: New coordinator for PDEVS.

4.2 Abstract coordinator

The implementation of the new abstract coordinator is presented in Figure 8. The coordinator now handles *nf-messages* and *rcf-messages* from imminent children. When an *nf-message* is received form an imminent child, the coordinator sends a *su-message* to that child to indicate that they can update the state and get ready to be able to continue filling the bag. Instead, when a rcf-message is received, the coordinator sends a *-message to the children, so it can continue filling the bag.

4.3 Root coordinator

The root coordinator stays the same as the one proposed in (Chow et al., 1994), and explained in section 2.

5 EXECUTION TRACE WITH THE PROPOSED ALGORITHM

In this section, we show how our proposed PDEVS simulation protocol (see Section 4) works and what are the differences with the original one explained in section 3 through an example. To exemplify how the new PDEVS Simulation Protocol works when a time advance zero occurs, we use the same model as the one presented in section 3 (see Figure 4).

To execute this model, an instance of the new abstract PDEVS simulator must be created. The structure of the abstract simulator remains the same as in Section 3: one root coordinator, a coordinator and two simulators (see Figure 5).

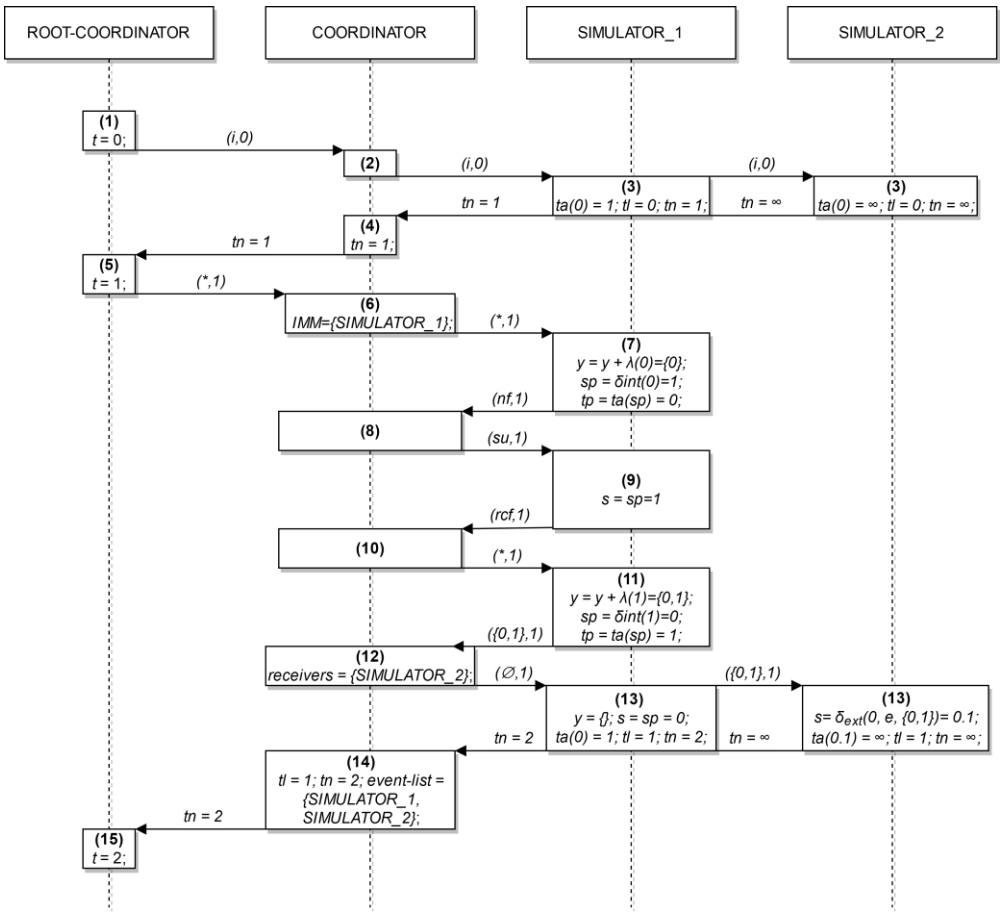

Figure 9: A new PDEVS simulation protocol execution trace.

The simulation execution trace of this model show in Figure 9, where we can see the interactions among the components. We set the simulation start time at 0. The initial state of both models is 0.

The simulation algorithm starts as we presented in section 3 until step **(6),** when COORDINATOR sends *-messages to all children in *IMM*. Then, SIMULATOR_1 runs the output function and instead of sending the y-message to COORDINATOR, SIMULATOR_1 adds the value of the output function to its output bag and checks is the bag is full. To do so, it calculates the potential next state using the internal transition function (sp = δ_{int} *(0)* = 1) and the time advance associate to the potential state (tp = ta(sp) = ta(1) = 0) **(7)**. Because tp is equal to zero, it means that the output bag is not full. Therefore SIMULATOR_1 send a nf-message to COORDINATOR. When COORDINATOR receives the nf-message, it sends a su-message to SIMULATOR_1 to indicate it should update the state **(9)**. Then, SIMULATOR_1 updates its state with the potential state compute previously **(10)** and sends a rcf-message to COORDINATOR to indicate its ready to continue filling the bag. After this, COORDINATOR sends another *collect outputs* message (*,1) to SIMULATOR_1, which computes the output function and adds to its output bag and checks is the bag is full. To do so, it calculates the potential next state using the internal transition function (sp = δ_{int} *(1)* = 0) and the time advance associate to the potential state (tp = ta(sp) = ta(0) = 1) **(11)**. Because tp is not zero, it means that the output bag is full. Therefore, SIMULATOR_1 sends its output bag ({0,1},1) to COORDINATOR, which updates the *receivers* set **(12)**. Then it sends an *execute transition* message (x, 1) to every child in the *receivers* and *IMM* lists. It sends (∅,1) to SIMULATOR_1 because it has no inputs, and ({0,1},1) to SIMULATOR_2, because it receives the input bag {0,1} coming from the output of SIMULATOR_1. Then, SIMULATOR_1 updates its state (s) using the value of the internal transition function already calculated in step (11) and updates its state to 0 (s = sp*).* It also updates the time advance of the new state (*ta* = tp), its *tn* and *tl* and reports the time of next event *tn* to COORDINATOR **(13).** At the same time, SIMULATOR_2 runs the external transition function with the input bag {0,1} and updates its state to 0.1 (δ_{ext}(0, e, {0,1}) = 0.1). It also calculates the time advance of the new state (*ta* (0.1) = ∞.), updates its *tn* and *tl* and reports the time of next event *tn* to COORDINATOR **(13),** which picks the first in its *event-list* (in this case, *tn=2*, as the SIMULATOR_2 is passive). This value is sent to the ROOT-COORDINATOR **(14)**, which updates the simulation time to 2 and starts a simulation cycle (**15**). The simulation continues until the *tn* value of COORDINATOR is infinity or until the maximum simulation time is reached, whichever happens first.

In this execution trace we can see that GENERATOR produces 2 outputs at time 1: {0} (step 7) and {1}, (step 11). However, unlike the execution with the original PDEVS simulation protocol presented in Section 3.2 these outputs produce only one state change in STORAGE. A single output bag {0,1} is produced by GENERATOR at time 1, which generates a state change to 0.1 in STORAGE.

6 CONCLUSIONS AND FUTURE WORK

One of the advantages of DEVS is that it clearly separates modeling and simulation. However, when executing models that have time advances zero, the PDEVS simulation protocol handles the outputs sequentially. This imposes restrictions to the modeler, it should keep in mind how the model will be executed, making the modeling and simulation activities less independent from one another.

In this work, we propose a new algorithm that assures that all outputs produced by a sequence of time advances zero are gather in a single output bag. With this idea, the modeler does not need to worry about how the interaction between components occurs. Simple component models using time advances zero can be defined and coupled together. When the model is executed, the simulation algorithm assures that all outputs produced at the same simulation time are send together. In models without time advance zero, this algorithm works exactly as the PDEVS simulation protocol. As future work we propose to implement this algorithm to make it available in a practical simulation tool.

REFERENCES

Belloli, L., D. Vicino, C. Ruiz-Martin, and G. Wainer. 2019. “Building DEVS Models with the Cadmium Tool”. In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, pp. 45–59. Piscataway, New Jersey, USA, Institute of Electrical and Electronics Engineers, Inc.

Chow A.C. and B. P. Zeigler. 1994. “Parallel DEVS: a parallel, hierarchical, modular, modeling formalism”. In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, pp. 716–722. Piscataway, New Jersey, IEEE Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Chow, A. C., B. P. Zeigler, and D. H. Kim. 1994. "Abstract Simulator for the Parallel DEVS Formalism". In *Proceedings of the Fifth Conference on AI, Simulation, and Planning in High Autonomy Systems*, edited by Paul. A. Fishwick. pp. 157-163. Gainesville, FL, USA.

Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. 2009. “DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring”. In *Proceedings of the 2009 Spring Simulation Multiconference (SpringSim ’09)*. Society for Computer Simulation International, San Diego, CA, USA, Article 161, 1–7.

Nutaro, J. 2014. A Discrete EVent system Simulator. http://web.ornl.gov/~1qn/adevs/adevs-docs/manual.pdf, accessed 22th April 2020.

Nutaro J. 2019. “Chapter 14 - Parallel and Distributed Discrete Event Simulation” in *Theory of Modeling and Simulation. 3rd edition*, edited by B. P. Zeigler, A. Muzy and E. Kofman. pp 339 – 372. San Diego, CA, USA: Academic Press.

Zeigler, B. P., H. Praehofer, and T. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Orlando, FL, USA: Academic Press, Inc.

Zeigler, B. P. and H. Sarjoughian. 2003. *Introduction to DEVS Modeling and Simulation with Java: Developing Component-based Simulation Models*. Tempe, Arizona, USA: Arizona State University.

AUTHOR BIOGRAPHIES

CRISTINA RUIZ MARTIN is a Postdoctoral Fellow at the Department of Systems and Computer Engineering at Carleton University. Her email address is cristinaruizmartin@sce.carleton.ca.

GUILLERMO G. TRABES is a Ph.D. student in Electrical and Computer Engineering (Carleton University) and Computer Science (Universidad Nacional de San Luis). His email address is guillermotrabes@sce.carleton.ca.

GABRIEL A. WAINER is Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of the Society for Modeling and Simulation International (SCS). His email address is gwainer@sce.carleton.ca.