Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo and M. Loper, eds.

COMPLEXITY ANALYSIS ON FLATTENED PDEVS SIMULATIONS

Guillermo G. Trabes

Department of Systems and Computer Engineering
Carleton University
and Universidad Nacional de San Luis
Ottawa, ON K1S 5B6, CANADA

Veronica Gil-Costa

Universidad Nacional de San Luis
and CCT CONICET San Luis
Ejército de Los Andes 950
San Luis, D5700, ARGENTINA

Gabriel A. Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By
Ottawa, ON 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 exist in the literature to execute simulations. DEVS was enhanced to handle better simultaneous events in the PDEVS formalism. To execute PDEVS simulations, a well-know and widely accepted algorithm was introduced: the PDEVS simulation protocol. However, since its creation, the protocol has evolved, and several versions have been proposed and implemented. In this work we propose and analytical approach to fully define and analyze this protocol. We divide the protocol into steps and sub-steps and for each of them we present a computer complexity analysis based on two key factors of the protocol's execution: the messages the components interchange and the computations the components execute.

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.

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

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.

Besides the efforts to define the formalisms, there have been efforts to develop algorithms to execute simulations. The most famous and widely accepted algorithm to execute PDEVS simulations is the PDEVS simulation protocol. However, since its creation, this protocol has evolved, several versions have been proposed and this created new problems we need to address. The first problem is that many details in the execution of the algorithm have not been completely defined, for example how to handle the event-list on the simulation. This has been implemented on several ways on different simulators. Second, the communication between components and what information they must interchange was implemented in different ways. In third and last place, we need more efforts to fully understand the advantages and limitations of using this protocol. Its message passing mechanism is one of the greatest overheads and finding ways to minimize the messages interchanged will provide more efficient executions. In addition, any efforts to reduce the computations needed to execute simulations will be beneficial to performance.

In this work we propose an analytical approach to solve this problem. We present a computational complexity analysis on the execution of the PDEVS simulation protocol that addresses this issue. To this end, we analyze the complexity two important factors: the number of messages transmitted between the components and the number of computations the algorithm must perform to execute the simulation.

The rest of the paper is organized as follows. In section 2, we summarize DEVS and PDEVS. We also explain the ideas behind the PDEVS simulation algorithm and the flattening algorithm for DEVS. In section 3, we develop a complexity analysis for the PDEVS simulation protocol by dividing the protocol in steps and sub-steps and analyzing them individually. Finally, in section 4, 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.

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 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 = <*X, Y, D, {M_i}, {I_i}, {Z_{ij}}, SELECT*>, 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) 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}: $Q \times X^b \rightarrow S$.

2.3 PDEVS Simulation Protocol

In addition to the development of the formalisms, 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.

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. In Figure 1, an example of a PDEVS model and its corresponding PDEVS abstract simulation structure.

Once the abstract simulator was defined, there is a mechanism to execute PDEVS simulations, called PDEVS simulation protocol. It was initially proposed in (Chow et al. 1994). However, since then several versions have been proposed and implemented. We base our work in the one defined in (Nutaro, 2019).

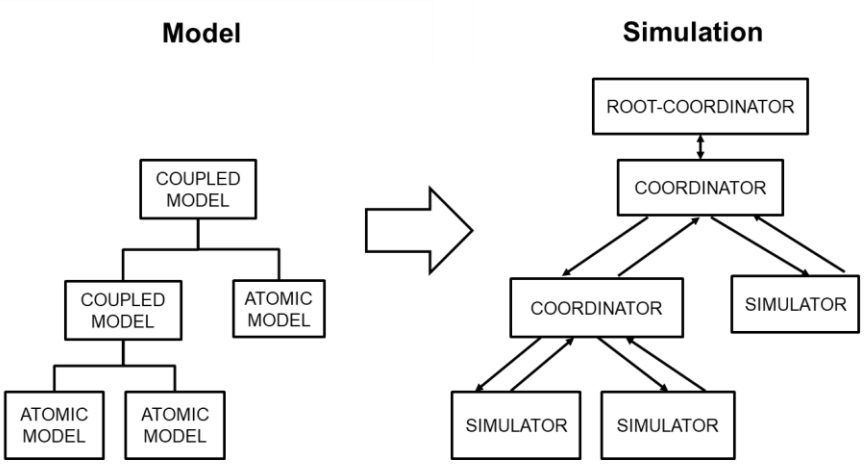

Figure 1: Example of a PDEVS model and its corresponding PDEVS abstract simulator structure.

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, coordinators enable 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 * messages in each simulation cycle. The complete details for this algorithm can be found in (Nutaro 2019).

2.4 Flattening Algorithm for the DEVS Abstract Simulator

As mentioned in the previous section, the simulation execution is message-driven; it is based on message exchange among components. The message passing overhead is significant if the model structure is too complex or extremely large (Glinsky and Wainer 2002; Kim et al. 2000, Wainer et. al 2011). This overhead can be minimized if the simulator's hierarchy is flattened. This way, the number of exchanged messages is reduced, and better performance can be obtained with the flattened simulation approach. In addition, it was proven than any hierarchical DEVS simulation can be transformed into an equivalent flattened that can give the same simulation results. In Figure 2, we can see an example of how a hierarchical abstract simulator can be transformed into a flat one.

The flattening algorithm works by eliminating every intermediate level coordinator by directly connect all simulators with the top-most coordinator. Therefore, in a flat abstract simulator the structure is composed by one root-coordinator, one coordinator and one or more simulator subcomponents.

To conclude this section, it is worth to mention that even though this approach was originally proposed for Classic DEVS, it can be used in the same way for PDEVS abstract simulators.

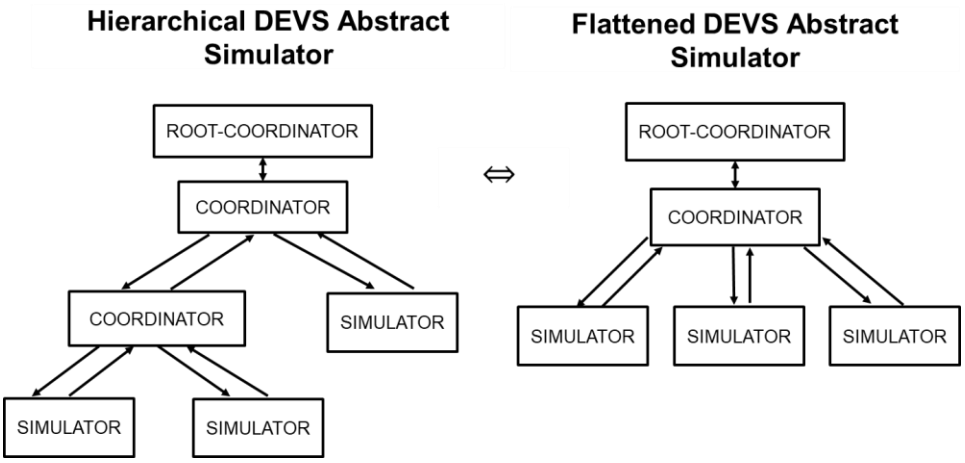

Figure 2: Example of a hierarchical PDEVS simulation structure and an equivalent flattened structure.

3 PDEVS SIMULATION PROTOCOL COMPLEXITY ANALYSIS

In this section we present our computational complexity analysis on the PDEVS simulation protocol. We use a basis for our analysis a generic flattened PDEVS abstract simulator. Since any hierarchical PDEVS abstract simulator structure can be transformed into an equivalent flat PDEVS Abstract Simulator we can generalize our analysis for any PDEVS simulation execution. We can see a graphical representation for this structure in Figure 3.

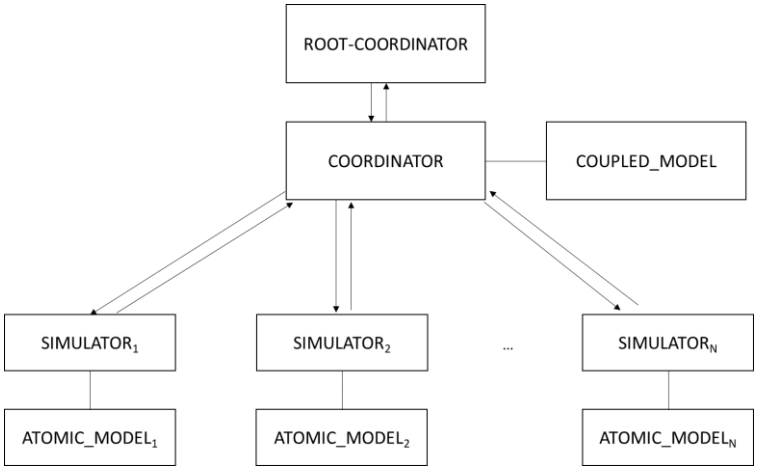

Figure 3: Flattened PDEVS abstract simulator structure.

There are several reasons to use a flattened PDEVS simulator approach. In first place, the algorithm is simpler and therefore easier to understand, analyze and implement. Second, the number of messages transmitted between components is reduced by eliminating all intermediate levels in the tree hierarchy. Also, some of the most complex parts of the coordinator algorithm, like analyzing which outputs received must be send to the parent component are simplified. Therefore, the complexity of executing the algorithm, in the way we analyze it in this work, is reduced. In third place there is empirical evidence showing the benefits of applying this method and how this can accelerate simulation's executions in practical implementations (Wainer et. al 2011).

Before starting our analysis some details on the PDEVS simulation protocol must be clearly defined. In first place, as the execution takes place on different components while they interchange messages it

should be clearly stated how the components interchange message and the information that must be transferred between them. It is not clear in (Nutaro 2019) how the information for next event's time is transmitted from children to parent components. To define this properly, we use an idea from the original PDEVS Simulation protocol in (Chow et al. 1994). This idea is to include a (done, *time*) message. This way, communication between components is clearly defined. In addition, a waiting for done message mechanism is included when a component needs to hold its execution while waiting from other components messages.

In second place, another important topic is how to determine the next events taking place on the simulation. On (Nutaro, 2019) an *event-list* in coordinator is mentioned, but there are no details on how this list should be implemented.

```
1: PDEVS-ROOT-COORDINATOR
2: variables:
3:  t
4:  sim_time
5: t = t_0
6: send (i,t) to child
7: wait until (done, t_next) is received
8: t = t_next
9: while (t<sim_time)
10:  send(*,t) message to child
11:  wait until (done, t_next) is received
12:  t = t_next
13: end-while
14: end-PDEVS-ROOT-COORDINATOR
```

Figure 4: PDEVS Root-Coordinator pseudocode.

```
1: FLATTENED-PDEVS-COORDINATOR
2: variables:
3:  DEVN = (X,Y,D,{M_d},{I_d},{Z_i,d})
4:  parent
5:  t_L
6:  t_n
7:  event-list
8:  IMM
9:  receivers
10:  y_1,..,y_D //output bags for children
11:  x_1,..,x_D //input bags for children
12: when receive i-message (i,t)
13:  for d ∈ D do
14:   send (i,t) to child d
15:  end-for
16:  wait until (done,t_nd) is received
     from every child
17:  for d ∈ D do
18:   insert d in event-list according to
      t_n
19:  end-for
20:  t_n = first element's time in event-
     list
21:  t_L = last element's time in event-list
22:  send (done, t_n) to parent
23: end-when
24: when receive *-message (*,t)
25:  if t ≠ tn then
26:   error: bad synchronization
27:  end-if
28:  IMM = first element in event-list
29:  remove first element from event-list
30:  for d ∈ IMM do
31:   send *-messages(*,t) to d
32:  end-for
33:  wait until (y_d) is received
     from every child in IMM
34:  for d ∈ IMM do
35:   for r such that r ∈ I_d do
36:    add y_d to x_r
37:   end-for
38:  end-for
39:  receivers = {r|r ∈ children ∧ x_r≠∅}
40:  for a ∈ (IMM U receivers) do
41:   send x-messages (x_r,t) to a
42:  end-for
43:  wait until (done,t_next) is received
     from every child in IMM U receivers
44:  for r ∈ receivers do
45:   remove r in event-list
46:  end-for
47:  for a ∈ (IMM U receivers) do
48:   insert a in event-list according
      to t_n
49:  end-for
50:  t_L = t
51:  t_n = first element's time in
     event-list
52:  send (done, t_n) to parent
53:  end-when
54: end-FLATTENED-PDEVS-COORDINATOR
```

Figure 5: Flattened PDEVS Coordinator pseudocode.

```
 1: PDEVS-SIMULATOR
 2: variables:
 3:  parent
 4:  t_l
 5:  t_n
 6:  DEVS
 7:  y
 8: when receive i-message (i,t)
 9:  t_l = t – DEVS.e
10:  t_n = t_l + DEVS.ta(DEVS.s)
11:  send (done,t_n) to parent
11: end-when
12: when receive *-message (*,t)
13:  if t = t_n then
14:   y = DEVS.λ(s)
15:   send y-message(y,t) to parent
16:  end-if
17: end-when
18: when receive x-message (x,t)
19:  if x = ∅ ∧ t = t_n then
20:   DEVS.s = DEVS.δ_int(DEVS.s)
21:  else if x ≠ ∅ ∧ t = t_n
22:   DEVS.s = DEVS.δ_conf(DEVS.s,x)
23:  else if x ≠ ∅ ∧ (t_l ≤ t < t_n)
24:   DEVS.e = t - t_l
25:   DEVS.s = DEVS.δ_ext(DEVS.s,DEVS.e,x)
26:  end-if
27:  t_l = t
28:  t_n = t_l + DEVS.ta(DEVS.s)
29:  send (done,t_n) to parent
30: end-when
31: end-PDEVS-SIMULATOR
```

Figure 6: PDEVS Simulator pseudocode.

The event-list or future event list (*FEL*) representation is a core problem in Discrete Event Simulation (DES). The aim is to being able to store and access efficiently events scheduled to occur in the future on the simulations. Several data structures have been proposed, such as queues (Himmelspach and Uhrmacher 2007), unordered and ordered lists, heaps (Franceschini et al. 2015) and calendar queues (Brown 1988). In general DES simulators, only two operations are required: insertion of events (called enqueue in the literature) and remove the elements with the minimum time (named dequeue in the literature). For DEVS and PDEVS one additional operation is required, this is since some components with events already on the structure may have to update their time for the next event in the case they receive an external transition. Therefore, the elements with outdates time for the next event must be updated or removed and inserted again. To do this, the simplest solution is to remove the elements and use the insertion operation. Therefore, for our analysis, we need to define the complexity for three operations on the operations on the event-list: enqueue, dequeue and deletion.

From all data structures proposed for the event-list, one stands over the others: The Ladder Queue (Tang et al. 2015). This structure is a special implementation of a calendar queue. This structure is a multi-level list with three levels, and it has the advantage of delaying the ordering of events scheduled far in the future until the ordering is needed. The complexity for this structure is O(1) in enqueue, dequeue and deletion on the average case (Furfaro and Sacco 2018). Furthermore, there is empirical evidence of the use of Ladder Queue in PDEVS is faster than other proposed structures (Franceschini et al. 2015). To the best of our knowledge this is the best structure proposed up to date and for that reason we use the complexity of its operations for our analysis.

With these details defined we can start with our analysis. As mentioned, we propose to analyze the complexity cost for executing the PDEVS simulation protocol on a flattened PDEVS abstract simulator. The complete pseudocode for the algorithm used in our approach is show in Figures 4,5 and 6.

For our analysis we will consider the complexity on two important factors that summary the cost of executing this algorithm.:

- The number of messages transmitted between components.
- The number of computations performed by the components.

Our approach involves analyzing the interaction and execution of the ROOT-COORDINATOR, the COORDINATOR, and the SIMULATORs.

3.1 Step 0: Initialization

The first step in the simulation is an initialization step. The objective of this step, which we call step 0, is to obtain the time for the first events occurring in the simulation in ROOT-COORDINATOR. In addition, the time for the next event on every SIMULATOR will be stored on the event list on COORDINATOR.

Let us analyze how this step works analyzing the interaction between the components in the abstract PDEVS simulator. A graphical representation on how this step executes can be seen on Figure 7.

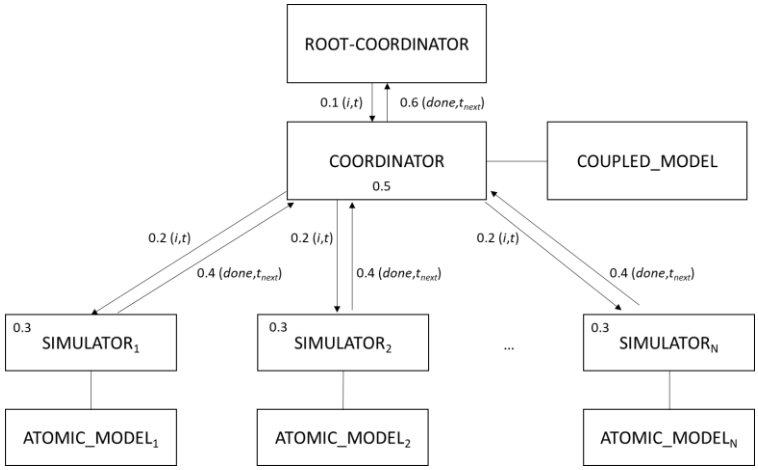

Figure 7: Step 0 - Initialization.

This step is composed by the following sub-steps:

- Sub-step 0.1: ROOT-COORDINATOR sends a (*i*,*t*) message to COORDINATOR, this can be seen on line 6 in Figure 4. The complexity of this sub-step is O(1) messages.
- Sub-step 0.2: COORDINATOR sends a (*i*,*t*) message to every SIMULATOR on the structure, this can be seen on lines 13-15 in Figure 5. After this, COORDINATOR waits until all done message are receiver from SIMULATORs. The complexity of this step is O(*N*) messages.
- Sub-step 0.3: Each SIMULATOR computes its time advance function to determine its time for the next event. We can see this sub-step in lines 8-10 In Figure 6. The complexity for this step is *N*, the number of simulators, multiplied by the maximum cost for the time advance function executed. Therefore, the complexity of this sub-step is O(*N***max*(*ta*)) computations.
- Sub-step 0.4: Each SIMULATOR sends a (*done*, *t*) message to COORDINATOR. This sub-step can be seen in line 11 in Figure 6. The complexity is O(*N*) messages.
- Sub-step 0.5: When COORDINATOR receives all done messages, it inserts the time and id of each simulator on the event-list. This way, all SIMULATOR that must execute next can be found on the first element of event-list. Using a Ladder Queue, the complexity or insert each element is O(1) on the average case. After this COORDINATOR obtains the time for the last and next event, both operations have a complexity of O(1). This can be seen in lines 17-21 in Figure 5. Therefore, the complexity of this step is O(*N*) computations.
- Sub-step 0.6: To finish this step, COORDINATOR sends a (*done*, *t*) message to ROOT-COORDINATOR. This can be seen in line 22 in Figure 5. This step has a complexity of O(1) messages.

3.2 Step 1: Output Collection

After the initialization step is completed a simulation cycle begins. The simulation will execute until the time limit for the simulation is reached. To simplify our analysis, we divided the simulation cycle in 2 steps:

output collection named step 1 and state transition executions named step 2. In this section we analyze step 1. The objective on this step is to collect all outputs from imminent models in COORDINATOR. In Figure 8 we can see a graphical representation for this step.

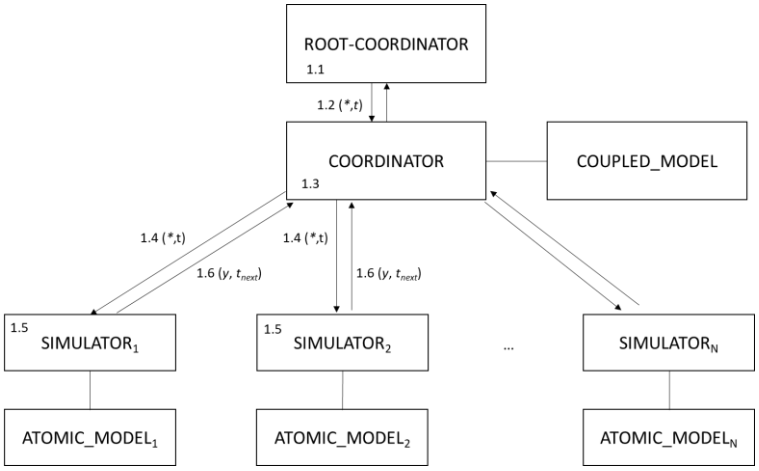

Figure 8: Step 1 - Output collection.

Let us analyze define and analyze the sub-steps required to execute this step:

- Sub-step 1.1: On the first sub-step, ROOT-COORDINATOR checks if the simulation is over comparing the simulation time with the time for the next event. This step is show in line 9 in Figure 4. The complexity of this step is O(1) computations.
- Sub-step 1.2: ROOT-COORDINATOR sends a (*,*t*) message to COORDINATOR. This step is show in line 10 in Figure 4. This step has a complexity of O(1) messages.
- Sub-step 1.3: COORDINATOR selects the first element in *event-list* and store it into the *IMM* set and removes it from the list. This way, *IMM* set contains the id of all imminent subcomponents. This is show in lines 28-29 in Figure 5. This step has a complexity of O(1) computations.
- Sub-step 1.4: Next, COORDINATOR sends a (*,*t*) to every SIMULATOR child in *IMM*. Then COORDINATOR waits until it receives a reply from all of them. This is show in lines 30-33 in Figure 5. The complexity of this step is O(|*IMM*|) messages.
- Sub-step 1.5: In this sub-step, each simulator calculates its output function. This can be seen in line 14 in Figure 6. The complexity for this sub-step is the number of imminent models multiplied by the maximum complexity of the output functions executed. Therefore, the complexity of this sub-step is O(|*IMM*|**max(λ)*) computations.
- Sub-step 1.6: Finally, every imminent child reply with a *y*-message with the output generated on the previous step and the messages are received by COORDINATOR. This is show in line 15 in Figure 6 and in line 33 in Figure 5. This sub-step has a complexity of O(|*IMM*|) messages.

Following these steps, COORDINATOR receives all outputs from imminent SIMULATORS, and the step is concluded.

3.3 Step 2: State Transition Execution

The objective of this step is twofold, in first place to execute the SIMULATOR's state transitions and second, obtain the time for the next simulation cycle. Not every SIMULATOR must execute on this step.

Only those who are imminent and those who receive an input from an imminent component in the previous step. In Figure 9 we can see a graphical representation for this step.

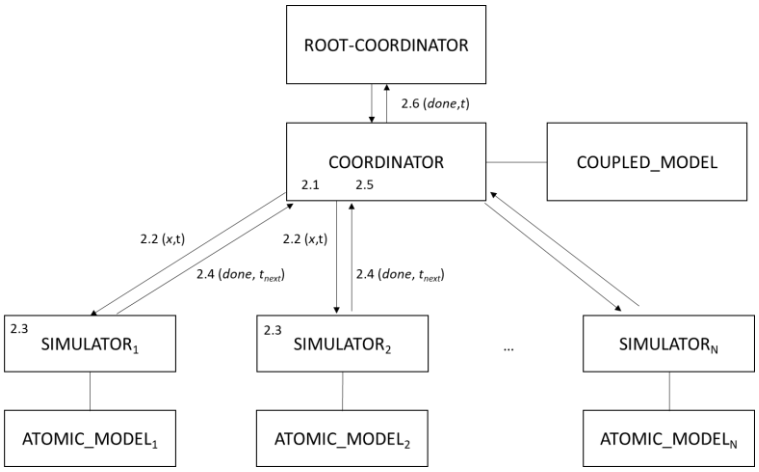

Figure 9: Step 2 – State Transition Execution.

Let us analyze define and analyze the sub-steps required to execute this step:

- Sub-step 2.1: The goal for this sub-step is to fill the bags that must be send to the receivers. This is achieved by iterating over every imminent model. For each imminent SIMULATOR, the output from that simulator must be inserted on each influencee component bag. This sub-step can be seen in lines 34-38 in Figure 5. The complexity of this step is the number of imminent components multiplied by the maximum number of influencees imminent components have. Therefore, the complexity is O((|*IMM*|*max(|I_i|)) computations.
- Sub-step 2.2: Next, COORDINATOR sends a x-message to every imminent and receiver component. This is show is lines 40-42 in Figure 5. The complexity of this step is O(|*IMM U RECEIVERS*|) messages.
- Sub-step 2.3: In this step, each imminent and receiver simulator calculates its state transition function and the time for its next event. This step can be seen in lines 18-28 in Figure 6. The complexity for this sub-step is the number of imminent models multiplied by the maximum complexity of the state transition functions plus time advance executed. This sub-step has a complexity of O(|*IMM U RECEIVERS*|**max*(δ+*ta*)) computations.
- Sub-step 2.4: After computing the internal transition function and time advance, every SIMULATOR in imminent or receivers, sends a (done, t) message to its parent COORDINATOR This is show in lines 29 in Figure 6. The complexity of this sub-step is O(|*IMM U RECEIVERS*|) messages.
- Sub-step 2.5: In this sub-step, the event list should be updated, and the minimum time should be determined. To do this, three tasks should be executed. First, elements in *RECEIVERS* should be deleted from *event-list*. Second, all imminent and receivers' components should be added into *event-list*. Third and last, the time for the next event should be selected from the first element in *event-list*. Delete each element in the Ladder Queue has a complexity of O(1). Therefore, the complexity of deleting all receiver components is O(|*RECEIVERS*|) computations. Similarly, insert an element in Ladder Queue has a complexity of O(1) computations. Therefore, insert all imminent and receiver components has a complexity of O(|*IMM U RECEIVERS*|) computations. Last, obtain the time for the last and the next event has a complexity of O(1). As the insertions have the higher complexity order, the complexity for this sub-step is O(|*IMM U RECEIVERS*|) computations. This step can be seen in lines 44-51 in Figure 5.

- Sub-step 2.6: Finally, COORDINATOR sends a (done, t_n) sends to ROOT-COORDINATOR. This sub-step is show in line 52 in Figure 5 and has a complexity is O(1) messages.

3.4 Complexity Cost to Execute a PDEVS Simulation

With the analysis made in previous sections, we can determine the complete cost to execute a DEVS simulation under this protocol. The initialization step, which we call step 0, will be performed only once at the beginning of the execution, and therefore the complexity of executing this step is given by the maximum sub-step. As we can see from the analysis the cost on this step depends on the number of simulators and on the maximum time advance function defined by the user. On Table 1 we can see a summary of the complexity of this step.

As mentioned in previous sections, steps 1 and 2 occur on a simulation cycle. A summary for the complexity of this steps is presented in Table 2. For both steps 1 and 2, the complexity cost is given by the maximum sub-step on each of them. As it can be seen from the analysis, the cost of step 1 depends on: the number of imminent simulators on the step and by the cost of the maximum output function on the imminent models on the step. Similarly, for step 2, the complexity cost depends on: the number of imminent models, the number of receiver models, the number of influences each imminent simulator has and on the maximum cost of the state transition function and time advance on the imminent and receivers' simulators on the step.

We can see that the complexity depends on many factors, some static such as the number of elements, but also some factors that depend on the dynamics of the simulation, for example, the number of imminent and receivers on a particular simulation step. In addition, it must be considered that any PDEVS simulation will end after the execution of a finite number of simulation cycles, *M*. This is another factor to consider when defining a cost for the simulation's execution. The cost for each simulation cycle is given by the maximum complexity cost of executing steps 1 and 2 in that cycle. Then, the cost for this simulation loop is given by the cycle with the maximum cost, multiplied by *M*, the number of cycles. In summary, the complete complexity of executing the simulation is: O(*max*(simulation cycle's cost)**M)*.

Table 1: Summary on the complexity of step 0.

Sub-step	Complexity
0.1	O(1) messages
0.2	O(*N*) messages
0.3	O(*N***max*(*ta*)) computations
0.4	O(*N*) messages
0.5	O(*N*) computations
0.6	O(1) messages

Table 2: Summary on the complexity on the simulation cycle, steps 1 and 2.

Sub-step	Complexity	Sub-step	Complexity
1.1	O(1) computations	2.1	O((|*IMM*|**max*(|I_i|)) computations
1.2	O(1) messages	2.2	O(|*IMM U RECEIVERS*|) messages
1.3	O(1) computations	2.3	O(|*IMM U RECEIVERS*|**max*(δ+*ta*)) computations
1.4	O(|*IMM*|) messages	2.4	O(|*IMM U RECEIVERS*|) messages
1.5	O(|*IMM*|**max*(λ)) computations	2.5	O(|*IMM U RECEIVERS*|) computations
1.6	O(|*IMM*|) messages	2.6	O(1) messages

4 CONCLUSIONS AND FUTURE WORK

In this work we presented a complexity analysis on the execution of flattened PDEVS simulations using the PDEVS simulation protocol. The objective of any PDEVS simulation algorithm is to minimize the number of messages interchange between the components and the number of operations require to execute

the simulations. Our aim with this work is to present a base algorithm divided in steps and sub-steps where each of them can be clearly identify and its complexity stated. This way, improvements can be made on specific parts in the algorithm and different versions can be compared.

As future work, we propose to continue this analysis by analyzing and categorizing the factors that contribute to the cost of executing simulations and how this can help to enhance the development of practical applications. In addition, we plan to implement the algorithm describe as an improvement for the Cadmium simulator (Belloli et al. 2019). Finally, we plan to use this analysis to identify which parts can be executed in parallel. This way, a parallel algorithm can be proposed, and the execution cost for that algorithm can be determined.

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, 45–59. Piscataway, New Jersey, USA: Institute of Electrical and Electronics Engineers, Inc.

Brown, R. 1988. Calendar Queues: a Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of ACM* 31(10):1220–1227.

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, M. Manivannan, D. A. Sadowski, and A. F. Seila, 716–722. 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 P. A. Fishwick. 157-163. Gainesville, FL, USA: Institute of Electrical and Electronics Engineers, Inc.

Furfaro, A., and L. Sacco. 2018. "Adaptive Ladder Queue: Achieving O (1) Amortized Access Time in Practice.". In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, edited by F. Quaglia, A. Pellegrini, and G. K. Theodoropoulos. 101-104. Ney York, NY. USA: Association for Computing Machinery.

Franceschini, R., P. A. Bisgambiglia, and P. Bisgambiglia. 2015. "A Comparative Study of Pending Event Set Implementations for PDEVS Simulation" In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, edited by F. Barros, M. H. Wang, H. Prähofer, and X. Hu. 77-84. San Diego, CA: USA: Society for Computer Simulation International.

Glinsky, E. and G. Wainer. 2002. Definition of Real-Time simulation in the CD++ toolkit. In *Proceedings of 2002 Summer Computer Simulation Conference*. San Diego, CA: USA: Society for Computer Simulation International.

Himmelspach,, J. and A. M. Uhrmacher. 2007. "The Event Queue Problem and PDEVS." In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, edited by M. J. Ades. 257-264. San Diego, CA: USA: Society for Computer Simulation International.

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. 339 – 372. San Diego, CA, USA: Academic Press.

Tang W. T., Goh R. S. M., and Thng I. L. 2005. Ladder Queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 15(3):175–204.

Wainer G., E. Glinsky, and M. Gutierrez-Alcaraz. 2011. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION* 87(7):555-580.

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

AUTHOR BIOGRAPHIES

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.

VERONICA GIL COSTA is a former researcher at Yahoo! Labs Santiago hosted by the University of Chile. She is currently an associate professor at the University of San Luis and researcher at the National Research Council (CONICET) of Argentina. Her email address is gvcosta@unsl.edu.ar.

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.