Interpreting the PDEVS formalism and algorithms to enhance the state handling mechanism

Clément Foucher

INTERPRETING THE PDEVS FORMALISM AND ALGORITHMS TO ENHANCE THE STATE HANDLING MECHANISM

Clément Foucher
LAAS–CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
Clement.Foucher@laas.fr

KEYWORDS
DEVS, Discrete-Event Modeling, Meta-Models.

ABSTRACT

The DEVS formalism consists in a rigorous frame for model building and simulating. One one hand, the mathematical description of the DEVS meta-model allows for thorough building of models. One the other hand, the proposed algorithms for model simulation allow for quickly building simulating tools for these models. However, in between these two phases, there are some less-detailed steps such as components initial state handling.

This article aims at proposing an interpretation of DEVS formalism slightly different from the one generally observed, and at strengthening the executable model building stage. The article proposes to explicitly split the model description between an user-defined abstract model and an executable model which construction can be automatically done by the simulating tool. Based on this vision, the initial state of executable models can be managed from within the DEVS formalism.

I. INTRODUCTION

Discrete Event System Specification (DEVS) [8] has been introduced decades ago in order to propose a sound mathematical frame for discrete-event system modeling and simulation. The specifications include a meta-model as a mathematical syntax to write models, as well as algorithms defining the simulation behavior for these models. DEVS has since seen many extensions, including the fundamental Parallel DEVS (PDEVS) formalism, which makes it a family of meta-models. Thus, we usually refer to the original DEVS formalism as Classic DEVS (CDEVS) and use the DEVS acronym to refer to the family of formalisms and not specifically to the initial DEVS formalism.

The DEVS formalisms family uses the notion of *components* to build models: a system is represented by a component, which can either be described as a single DEVS model (atomic component) or be composed of multiple other components (coupled component). Atomic components formalism describes models using the notions of internal state and state transitions resulting from time advance or stimuli from outside the model. Coupled components are simple collections of components associated with a connection map describing how events are propagated between components.

However, while the model description and its simulation behavior are strongly defined in DEVS formalisms, there is room for interpretation regarding the simulator building phase, notably its initialization. DEVS algorithms notably assume that at the beginning of the simulation the model is ready and in its initial state when the initial $i - message$ is received. Moreover, we find various interpretations and representations of the initial state notion in the DEVS-related literature.

This article will intend to define a way of handling the simulator building phase and simulator state initialization. It will propose to make explicit how the simulator is built from an existing model, trying to specifically separate abstract model building from simulator building.

The main objective of this work is to strengthen the independence between a model description and its simulation, notably in order to enhance the notion of Platform-Independent Model (PIM) which is the core of the Model-Driven Architecture (MDA) process [5]. This will help advancing on the standardization of the simulation tools behavior. Moreover, this work is a foundation for the author's current developments in relation with components state management from the formalism itself, such as saving or restoring a component state, which is important when dealing with dynamic structure models.

Section II of this article describes the basics of DEVS models formalisms, focusing on PDEVS which we will more specifically consider. Section III will focus on the simulation algorithms and the notion of *simulator* in DEVS. In Section IV, we will look into the variable handling mechanisms at stake in DEVS and propose a refined vision of the variables by separating their definition from their value. In the Section V, we will make a proposal on how the state variable initialization phase could be integrated into the formalism itself rather than being left over to interpretation by the simulation tool. Finally, Section VI will conclude on this topic and review what doors are being opened by this study in the path of future work.

II. DEVS MODELS DEFINITIONS

DEVS family's formalisms define atomic components models to be the foundation by expressing a behavior. Coupled components are also defined which aggregate other components, either atomic or coupled themselves.

The full equivalence between atomic components and coupled components is a fundamental requirement for the formalism. This notion is called *closure under coupling* and has been proven for both CDEVS [8] and PDEVS [2]. This allows for hierarchical construction of components with an unlimited depth and ensures these components will still behave according to the meta-model specifications as, at each level, coupled components are indistinguishable from atomic components made of a single DEVS model.

This section will review the definition of these components in PDEVS in order to identify the relevant notions of state and abstract model.

A. PDEVS Atomic Components Models

The PDEVS formalism defines a model M as a tuple:

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

In this definition, the notions of inputs and outputs are respectively carried by X and Y. This forms the interface of the model indicating how it is allowed to interact with its surrounding environment. The notion of *ports* is usually integrated to the formalism, allowing to distinguish interactions in a finer-grained way than just separating inputs from outputs. This allows for refining the models interfaces by using named ports, as well as specifying their coupling individually in coupled components. Basically, a port is a name associated with a domain of definition, indicating which values are expected or allowed through the port. When seeing them this way, this definition matches the one of a variable: e.g. an output port can be defined as $Q \in \mathbb{N}$.

S is defined as the set of sequential states, i.e. the set of values that can be reached by the model's state. In this sense, S is the domain of definition of the component state itself. Following the vision of [3], we will consider here that S can be expressed as a list of variables associated with the sets of values that they can reach by the model evolution.

The δ functions are responsible for the model state evolution, depending on the source of the event causing the evolution, which can be internal (time triggered) or originate from the environment through an input port. Collisions between various simultaneous events are handled in PDEVS through the δ_{con} function, marking its difference with CDEVS and allowing for a real parallel simulation.

The λ function dictates how the component should emit events on its output ports in reaction to an internal event.

Finally, the ta function associates time to the component's state, in order to allow for internal events.

What can be noted with this model definition is that there are two kinds of mathematical objects involved: variables definitions and functions.

In particular, there is no notion of an initial state at this point in the original definition. Some authors prefer to add it as an s_0 value directly within the component definition to indicate the starting point of an executable model, e.g. [6] or [4]. However, we will here keep it separated from the model definition, and consider this value can be set at simulation time to test different scenarios through different initial states and input vectors. Moreover, this is more coherent with the remark concerning the simulation process in [8]: "typically, we are given a system specification together with the initial state values [...]", explicitly stating the initial state is not part of the system specification.

B. PDEVS Coupled Components Models

There are multiple, equivalent, definitions of a PDEVS coupled component depending if we reason in terms of interface map or influencers. Here, we will adopt the influencers notation by defining a PDEVS coupled component's model N as :

$$N =< X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\} >$$

We find here the same notion of component interface with X and Y as in the atomic components.

D is defined as the set of components contained in the coupled components, while $\{M_d\}$ contains their respective models. This is an explicit separation between a component and its model. We can here treat D as the names (or the references) of the components, their specification being the associated model.

Finally, $\{I_d\}$ and $\{Z_{i,d}\}$ detail the links between the components from D as well as with the interface of N itself. We will not detail this part further as it is not the focus of this article to discuss the components relations.

What is important however is that there is no notion of state at the coupled component level: states are fully handled at atomic component level.

C. Interpretation of the Components Specifications

We can already feel from this brief look at the models definitions that, in Zeigler's view, models of systems are more of a template than an actually runnable object that will evolve over time. It is, indeed, intended: the evolution will be taken care of by the simulator and not by the model itself. We will detail in the following section the concept (actually the concepts) of simulator in DEVS formalisms, but what is important here is to see a model as an unchangeable definition of a system, defining the frame in which it can behave as well as the way in which it will respond to specific stimulus depending on a given state. We also remarked that, in the original definitions, there is no specific initial state.

To follow this approach, we have to consider that the model built at this step does not have to be executable on its own as we are still in the modeling phase. This is what we call an *abstract model*, as opposed to an *executable model*.

Thus, there is no need here for variables, in the sense of a placeholder for a value. What we need is the variables' *definitions*, i.e. their domain. This is valid for an abstract model's state as well as for its ports.

What we can also see is that the various functions defined for an atomic model does not require an internal state. E.g. the δ_{int} function is defined as $S \to S$, i.e. the input state is a parameter of the function and the new state is outputted from the function, thus not requiring it to be internal.

III. PDEVS Simulators

There are three different notions of simulator in the DEVS context: an atomic component simulator, a DEVS component simulator (which is called a simulator for an atomic component and a coordinator for a coupled component), and a simulation tool, usually referred to as a simulator too. To prevent confusion, the lone term *simulator* will here be used to specifically designate an atomic component simulator; we will use the term *DEVS simulator* to encompass atomic components simulators and coupled components coordinators; finally, the term *simulating tool* will refer to a tool implementing DEVS simulators.

In the simulation phase, each atomic component model is thus associated with a simulator, and each coupled component model is associated with a coordinator in a one-to-one relation. There is furthermore a root coordinator handling the general simulation.

This section will aim at quickly describing the structure for simulation variables provided by the simulation algorithms.

A. Atomic Component Simulator

When dealing with an atomic component, there is on one hand the abstract model, defining the states allowed through the simulation as well as the behavior of the component, and on another hand the simulator in charge of the evolution of the model.

When taking a look at the simulator's algorithm, defined as pseudo-code specifying a few functions that are triggered in reaction to message exchanges, we remark that the variables are defined there. The simulator defines a set of variable amongst which the DEVS model itself and time-related variables tl and tn (respectively times of latest and next event). The mention of the variable containing the model (simply called $DEVS$) indicates "with total state (s, e)".

There is a slight incertitude in this definition regarding the current state s (and elapsed time variable e) being part of the model or variables of the simulator. As we chose here to treat the abstract model as an immutable definition, there can't be any variable in it in this vision. So the variables will be part of the simulator itself in our approach, or at least placed outside the abstract model.

We remark that the e variable, while being part of the DEVS definition, is replaced here by tl and tn. However, the value of e is still used once in the algorithms, in the $i - message$ phase, thus the e variable is still present but only reduced to what we could call e_0, the initial value of e. After the initial use of e_0, the variable is no longer used, except as a temporary value to pass as a parameter to the function δ_{ext}. We could also argue that this value *should* be passed to the δ_{con} function too, as this one can use the total state to compute its result, but this is out of the scope of this article. Other variables in the simulator, $parent$ and y, are respectively a constant and a simple temporary record of the output's value(s).

We also notice that, in DEVS definitions, the initial state is supposed to have been defined prior to the simulation launch, as it is not handled in the algorithms. This is why some prefer to treat the initial state as a variable of the model itself by adding s_0 to the M tuple. We however will consider the model description to be independent from its initial state, in order to allow multiple uses of the same model in different use cases.

B. Coupled Component Coordinator

As with coupled components models, we will not detail the coupled components coordinator algorithms too much, as we are not focused here on the model hierarchical structure. We can however remark that the coordinator does indeed have a notion of time through its tl and tn variables, which plays the role of e in the simulation. It means that, while the coupled components models does not have any notion of time, these variables are still required for coordination purpose.

IV. Variables definition and initial state

In the above description of PDEVS models and simulation algorithms, we chose to see a PDEVS model as a static object, that does not need to be executable by itself. This part agrees with PDEVS formalism where a model needs to be associated with a simulator to be run.

However, many DEVS simulating tools choose to directly describe the model as executable, for example declaring the state variables and other objects as being members of a class. This is the case of PythonDEVS [7] for example: the model object is described as a whole, and variables are included within it. The various functions associated with the model description then simply use these variables as object members, not needing to pass them as parameters.

While this way of doing is fully functional, it marks a distinction with the vision that is proposed here: the description done here by the modeler is the *executable model*, there is no abstract model notion within the simulating tool.

What we propose here is to methodically separate the abstract model definition from the executable model. The abstract model notion can then be brought to the simulating tool, and the executable model be automatically built from this abstract model.

In this section, a proposal is made to build an executable model from a DEVS model.

A. Variables of an Executable Model

Regarding the state variables of an executable model, two main variables are to be considered: the state s and elapsed time e, together forming the total state $q =< s, e >$.

As said previously regarding the e variable, this one is broken up into two variables in the DEVS simulation algorithms: tl and tn. With these two variables, e is only used within the simulator for its initial value e_0.

For the s variable, this one can be a collection of multiple variables in order to better reflect the model internal behavior. The initial state s_0 must then be a collection of values for each variable of s.

As said previously, the port definition also resembles variables definitions, as having a name and a domain. Internally, an executable model can thus represent an output or input port as a variable.

Finally, we identified in the DEVS simulators that there are requirements for variables outside the ones defined in the models. E.g. the $parent$ variable is required for communication while, in the meta-model, the only hierarchy is directed from the root to the leaves, coupled models having knowledge of their children but not the other way around. Thus, these variables are internal elements required by the simulator, and can vary depending on the implementation that is made of it. We then consider these variable to not be part of the executable model, and will not discuss them further in.

We treat state variables within s and port variables in a PDEVS abstract model as simple definitions, associating a name with a definition set indicating the allowed values for the variables. In an executable model, implementing such a model would require a specific mechanism for handling definition. It can rely on the notion of *type*, that exist in much programming languages. But this notion alone has limitations toward what we try to accomplish here. As an example, a variable defined as $var1 \in \mathbb{R}$ can be of type $float$ or $double$ in a C++ executable model, limitations due to precision set aside. However, how to treat a variable $var2 \in [\![0 : 1000]\!]$? This requires that the variable not only

be of type int or $unsigned\ int$, but also to have a restriction on its boundaries.

This is why the variable defined in the executable model must remain linked to the variable definition from the abstract model. When building the executable model from the abstract model, the simulating tool has to choose the most relevant type for the variable, but if a restriction is expressed on the domain (i.e. defined as a subset of a generic set), the simulator must remember that checks may have to be made on the value against the variable definition.

B. Initial State Management

One of the main objectives of this discussion is to allow a component's abstract model to be usable in multiple environments, even multiple times in the same simulation, as a class can be instantiated as multiple objects. For that purpose, the initial state has to be provided alongside the model, but not within it. The simulator will be in charge of initializing an executable model instance with a provided initial state.

We also have to remember that the initial state a component requires is q_0, not only s_0. The e variable's initial value is used to initialize tl and tn in the $i - message$ phase. This initial value on e allows for the simulation initial situation to represent a state in which a component has already be waiting for some time. Thus e_0 has to be provided within a component initial state. An initial state for an atomic component M is then defined as follows: $q_{0_M} =< s_{0_M}, e_{0_M} >$.

Providing q_0 is relatively simple for a single atomic component. However, for a multi-component coupled model, including any levels of hierarchy, it requires a bit of attention to make sure each component is correctly assigned its initial state. There must be a way of matching the members of the set of components initial states to the correct component. There are two ways this can be achieved.

The first one is simply to store states in a set and label each one with its associated component name. This only works if all components in the model are given a unique name. We already mentioned that the D set in a coupled component can be seen as the list of inner-components names. However, this list does not have to be text- or number-based: the D set has no restriction over its content in terms of mathematical object nature. Components could be represented as sets or data structures for example.

Moreover, even if components are labeled using an alphanumerical notation, there is no guarantee the names are unique amongst the various levels of hierarchy. This can be a good practice however to ensure from the abstract model that all names are unique within the system's model.

A second way of doing is to represent the initial state as a tree of states matching the model's structure. All states of components residing in a coupled component would be gathered together in a virtual state of the coupled component. Doing so at each level of hierarchy results in a virtual state for the root component in the form of tree. Distributing these states can then be the responsibility of the coupled component in a pre-$i - message$ phase or as a part of it. This way makes the component's name uniqueness only required at a single level of hierarchy.

The difference between these two approaches is thin, as we still require each component to have a label, and that that label must be unique in a certain context. However, the tree approach has the advantage of respecting DEVS' philosophy of coordinators and simulators, and allows it to be integrated within the simulation process itself, not as part of a separate model set-up phase. We will thus propose here to use the tree approach. For a coupled component N, its initial state will then be of the following form: $q_{0_N} = \{q_{0_d} \mid d \in D_N\}$.

V. Executable model building phase

This leads us to the way we propose to interpret executable model building phase. We will use here the PDEVS algorithms provided by Zeigler.

The idea is to input only an abstract model and an initial state to the simulation tool, which will then automatically build the according DEVS simulators. Let a model $PDEVS$ and an arbitrary initial state matching its structure $q_{0_{PDEVS}}$.

The simulation tool will first have to build the PDEVS simulator tree. This part is done as by the original PDEVS behavior, except for the initial state that is not set at this point but only further. The idea is that this part should be integrated within the simulation algorithms themselves.

As defined in the original algorithms, there is a tiny initialization stage in the simulators. This phase intends at setting the time variables of the components. For an atomic component, this allows to run the ta function for the first time and expose it to its parent. For a coupled component, this sets not only the time variables, but also prepares and sort the list of imminent components. This part could have been hidden from the algorithms, supposing that the components already know the initial state of the simulation at the beginning of the simulation, and devolve the responsibility for this to the simulator tool. However, it has been decided to include the phase in the algorithms, probably in order to ensure a careful procedure and make sure there is no divergence of interpretation on this. This has not been the case with the initial state, q_0 being supposed to already be set in the component when the $i - message$ arrives.

This may be because there is a difference in vision in the proposed simulators from the meta-model definition. Indeed, the *DEVS Definition Language* [8] proposed by Zeigler marks a difference with the vision exposed when dealing with mathematically-defined models. In this example of a language allowing to define DEVS components, the initial state is integrated into the model definition, which is a difference from the original statement that a model *and initial state* are provided to the simulator: here, both are a single entity.

In order to allow for the initial state set-up mechanism described earlier, it seems a good approach to integrate that step to the algorithms.

Following this approach, we propose to add a pre-$i - message$ function to the algorithms called the $q0 - message$. This function is depicted in Algorithms 1 and 2.

This phase could also be merged with the $i - message$ phase by simply adding the $q0$ parameter, as described in

Algorithm 1 Initial state function for atomic components

```
when receive q0-message (q0)
    s = q_0.s_0
    e = q_0.e_0
```

Algorithm 2 Initial state function for coupled components

```
when receive q0-message (q_0)
    for-each d in D do
        send q0-message (q_{0_d}) to child d
```

Algorithms 3 and 4. In that case, the persistent e variable becomes unnecessary in the simulator.

Algorithm 3 Modified i-message function for atomic components

```
when receive i-message (i, q_0, t)
    s = q_0.s_0
    tl = t - q_0.e_0
    tn = tl + ta(s)
```

Algorithm 4 Modified i-message function for coupled components

```
when receive i-message (i, q_0, t)
    for-each d in D do
        send i-message (i, q_{0_d}, t) to child d
    sort event-list
    tl = max { tl_d | d ∈ D }
    tn = min { tn_d | d ∈ D }
```

This slight modification allows for explicitly handling the initial states values from the DEVS simulators. This step opens room for further handling of the state: for example, the simulation could be reset to another starting point for the simulation without having to rebuild the DEVS simulator tree, simply by sending a new initial state message.

Moreover, we can also imagine a functionality within the formalism whose purpose would be to extract the state at some point of the simulation to build a q_0 object that can be used to restart the simulation from the current point.

These elements can seem to have few advantages compared to existing state-handling mechanisms already built in many PDEVS simulators. However, by integrating the state management directly within the formalism, this lays the foundation for future work on individual component state handling, and allows for harmonization on how the state is handled between simulator tools.

VI. Conclusion and Future Work

This article tried to propose a clarification on the initial state subject in DEVS models and algorithms, notably in PDEVS. We remarked that the proposed DEVS algorithms allowed to choose an arbitrary simulation starting time, but relied on a preset initial state for the components. We also remarked that despite the initial state not being part of an abstract DEVS model, most implementations, including DEVS Definition Language, still deeply linked the initial state to the model.

On our way to this proposal, we also found that there is a distinction to be made between a variable and its definition, and that while the former was needed for an executable model, only the latter was required for an abstract model.

We then proposed to slightly modify the PDEVS algorithms in order to integrate the initial state management within the PDEVS simulation algorithms. To do so, we proposed two different ways to enhance the DEVS simulator process by adding a initialization phase, either independently from the $i - message$ phase or directly within it.

These considerations allow to advance on a standardization on the component state representation, that we proposed to treat as a tree matching the model structure. Moreover, by directly integrating the component's state handling within the simulation algorithm, we lay the foundations of dynamic components state handling from the simulation algorithm, allowing to integrate mechanisms such as state saving and restoring.

What remains to be done from this point is to further develop these state handling mechanisms beyond the initial value. Indeed, if the interest remains limited in a static structure model such as CDEVS or PDEVS, this allows for much more considerations in dynamic structure models, such as in DS-DEVS [1]. This preliminary work had to be done on static structure models at first in order to allow for an easy formalization of the elements at stake, but will reach their full potential when applied to dynamic structure models.

References

[1] Fernando J Barros. "Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation". In: *Winter Simulation Conference Proceedings, 1995*. IEEE. 1995, pp. 781–785.

[2] A. C. H. Chow and B. P. Zeigler. "Parallel DEVS: a parallel, hierarchical, modular modeling formalism". In: *Proceedings of Winter Simulation Conference*. Dec. 1994, pp. 716–722. DOI: 10.1109/WSC.1994.717419.

[3] Stéphane Garredu. "Meta-modeling approach and model transformations in the context of modeling and discrete event simulation : application DEVS formalism". Theses. Université Pascal Paoli, July 2013.

[4] Hae Young Lee. "Elapsed-time-sensitive DEVS for model checking". In: *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*. Citeseer. 2013, pp. 3998–3999.

[5] MDA OMG. *The Architecture of Choice for a Changing World*. 2016.

[6] Vilian Solcány. "Simulation Algorithms for DEVS Models". In: (2008).

[7] Yentl Van Tendeloo and Hans Vangheluwe. "The modular architecture of the Python (P) DEVS simulation kernel". In: *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation-DEVS*. 2014, pp. 387–392.

[8] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. 2nd. Orlando, FL, USA: Academic press, 2000. ISBN: 0127784551.