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.

USING UML AND OCL AS LANGUAGES TO DEFINE DEVS ATOMIC MODELS

María Julia Blas
Silvio Gonnet

Instituto de Diseño y Desarrollo INGAR (UTN-CONICET)
Avellaneda 3657
Santa Fe, 3000, ARGENTINA

ABSTRACT

This paper presents a work-in-progress intended to define the foundations for building a representation of DEVS using conceptual modeling languages from information system engineering. We use UML and OCL languages to define a metamodel that conceptualizes atomic DEVS models. Such a representation enhances the DEVS modeling activity providing atomic model definitions as instances of the metamodel developed.

1 INTRODUCTION

The Discrete Event System Specification (DEVS) formalism (Zeigler et al. 2018) is a modeling formalism with sound semantics founded on a system theoretic basis. This formalism embodies a set of concepts related to systems theory and modeling to describe discrete event models. DEVS includes two types of models (atomic and coupled) that are formalized using equations, functions, and sets. Therefore, DEVS is an abstract formalism that is independent of any particular implementation. When engineers want to simulate DEVS models they need to program them in the input language of a concrete simulator. Then, DEVS models can be mathematically described but its simulation is performed by a particular simulator. Hence, it would be desirable to use existing modeling languages to build DEVS models from an information system perspective. A "good" conceptualization of DEVS can provide a basis to automatically translate their instances into DEVS simulators. Also, it can be used as a mechanism for documenting the models.

The main goal of our research is to improve the representation of DEVS as abstractions of the system under study using UML and OCL modeling languages. We show how a metamodel can provide a solid basis for describing the behavior of atomic DEVS models employing conceptual modeling from the information system perspective. The representation of coupled DEVS models is out of scope.

2 THE "ATOMIC DEVS MODEL" METAMODEL

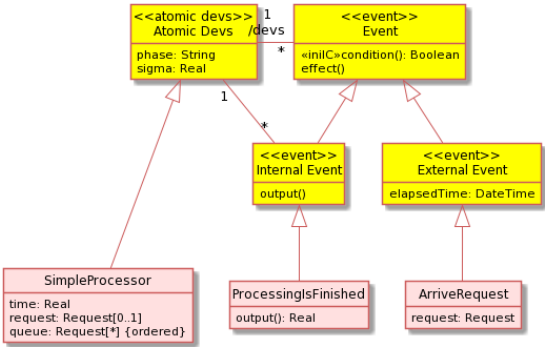
A DEVS model is designed considering the modeler's knowledge about the source system (domain). Mainly, a representation of the domain states needs to be specified in an atomic model. Applying conceptual modeling of information system perspective for modeling the system under study, we can assume that the *state* of a particular domain, at a given time, consists of a set of objects, a set of relationships, and a set of concepts into which these objects and relationships are classified (Olive 2007). The set of entity and relationship types used to observe the state of a domain is the conceptualization of that state (Olive 2007). Such a conceptualization can be specified employing a UML class diagram. Then, if we specify an atomic model as an entity type (class), we could define its structure according to the (yellow) classes of Figure 1. In its simplest form, an atomic DEVS is defined with basic state variables *phase* and *sigma*. Furthermore, specific state variables can be added by extending the model definition (see the *SimpleProcessor* definition).

At any time the model is in some state, *s*. A state change occurs when: *i)* an external event has arrived, or *ii)* no external event occurs and the system was in *s* for time *ta(s)*. In the first case, the current state changes according to the external transition function. When an external event arrives, the model also

receives its elapsed time (*e*). On the other hand, in the second case, the state changes following the internal transition function. The transition is triggered by a timed event. However, before changing the state, the model sends an output event. Once the output event has been dispatched, the internal transition function is executed to bring the model to a new state.

In conceptual modeling of information systems, the behavioral schema specifies the valid changes in the domain state, as well as the actions that the system can perform. Changes in the domain state are domain events. Events are also instances of concepts and they have characteristics, which are relationships with other entities. All events have a relationship with an entity that is a time instant, which corresponds to the time at which the event occurs (Olive 2007). Figure 1 shows events as instances of concepts. An *Event* is defined by the *effect()* that it will produce in the model and the precondition that must occur for that event to happen (*condition()* operation). An *Event* is specialized in *External Event* and *Internal Event* to represent the two kinds of events that can change the domain state of an atomic model. Then, the classes are specialized following the domain events identified in the source system. For *External Event*, the *elapsedTime* attribute is included with aims to represent the information required to determine its transition *effect()*. For the *Internal Event*, the *output()* operation that determines how output events are generated.

Over UML class diagram, OCL can be used to fully define the model behavior (state variable values, initial state definition, and operation description). This language enables a declarative specification of the event effects and, therefore, any implementation of the model that leaves the information base in a state that satisfies the specified model is valid (Olivé 2007). **Then, combining UML and OCL with conceptual modeling of information systems, a full definition of atomic models can be provided.** For example, in Figure 1 the classes in red illustrate the *SimpleProcessor* model. The state definition includes *time*, *request*, and *queue* as new variables. The values for *phase* are *waiting* and *working* (OCL invariant detailed in gray). The initial state (OCL constraints in pink) is defined as *(waiting, infinity, 0,{},{})*. The internal and external events are modeled, respectively, in the classes *ProcessingFinished* and *ArriveRequest*. The preconditions and effects of these events are specified with the OCL constraints detailed in black and blue, respectively. Finally, the output of the *ProcessingFinished* event is specified in the OCL constraint described in green.




```
context SimpleProcessor
inv: phase = "waiting" or phase = "working"

context SimpleProcessor::phase init: "waiting"
context SimpleProcessor::sigma init: infinity
context SimpleProcessor::time init: 0.0
context SimpleProcessor::request init: Set{}
context SimpleProcessor::queue init: OrderedSet{}
```

```
context ArriveRequest::condition():Boolean
body: (devs.sigma = infinity and devs.phase = "waiting") or
(devs.phase = "working")

context ArriveRequest::effect()
post: devs.phase@pre = "waiting" implies (devs.phase ="working"
and devs.sigma = 20 and devs.time = devs.time@pre + elapsedTime
and devs.request = queue and devs.queue= OrderedSet{})
post: devs.phase@pre = "working" implies (devs.phase ="working"
and devs.sigma = devs.sigma@pre - elapsedTime and devs.time =
devs.time@pre + elapsedTime and devs.request = devs.request and
devs.queue = devs.queue@pre->append(queue))

context ProcessingIsFinished::condition()
body: devs.phase = "working"

context ProcessingIsFinished::output():Real
body: Tuple{time= devs.time, queue= devs.request}

context ProcessingIsFinished::effect()
post: devs.request@pre->isEmpty() implies (devs.phase= "waiting"
and devs.sigma = infinity and devs.time = devs.time@pre +
devs.sigma@pre and devs.request = Set{} and devs.queue =
OrderedSet{})
post: devs.queue@pre->notEmpty() implies (devs.phase = "working"
and devs.sigma = 20 and devs.time = devs.time@pre + devs.sigma@pre
and devs.request = queue->first() and devs.queue =
devs.queue@pre->excluding(devs.queue@pre->first()))
```

Figure 1: Simple Processor specification.

REFERENCES

Olivé, A. 2007. *Conceptual Modeling of Information Systems*. 1st ed. New York: Springer Science & Business Media.

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. 3rd ed. London: Academic Press.