TOWARDS A UNIVERSAL REPRESENTATION OF DEVS: A METAMODEL-BASED DEFINITION OF DEVS FORMAL SPECIFICATION

María Julia Blas
Silvio Gonnet

Instituto de Desarrollo y Diseño INGAR
CONICET
Universidad Tecnológica Nacional
Santa Fe, ARGENTINA
{mariajuliablas,sgonnet}@santafe-conicet.gov.ar

Bernard P. Zeigler

University of Arizona
Tucson, AZ
RTSync Corp.
Chandler AZ, USA
zeigler@rtsync.com

ABSTRACT

The Discrete Event System Specification (DEVS) is one of the key formalisms to define Discrete Event Simulation (DES) models and underlying concepts. In this paper, we propose a conceptualization of DEVS designed as a metamodel that supports the formalization task. Such a conceptualization is part of a multi-level layered structure that defines a universal representation of DEVS as a combination of both theory and practice points of view. For the metamodel specification, we employ UML. With the existing modeling technology, our metamodel can be implemented to provide a framework that supports: *(a)* consistency validation between discrete event system descriptions and formal models using a DES metamodel, *(b)* interoperability between formalization and most used implementations using a platform-independent metamodel, and *(c)* consistency verification between formal models and their implementation. All these benefits are derived from the capability of defining model-to-model transformations over the modeling levels proposed in our conceptualization.

Keywords: abstraction to implementation, conceptual modeling, discrete event simulation, metamodeling.

1 INTRODUCTION

The Discrete-Event System Specification (DEVS) formalism is a modeling formalism based on systems theory that provides a general methodology for hierarchical construction of reusable models in a modular way (Zeigler, Muzy, and Kofman 2018). Over the years, DEVS has found an increasing acceptance in model-based simulation research because of the ease of model definition, model composition, reuse, and hierarchical coupling. However, in the Modeling and Simulation (M&S) field, there is no commonly agreed-upon precise definition of Discrete Event Simulation (DES) and its underlying concepts (Guizzardi and Wagner 2010). Moreover, the entire field of computer simulation is suffering from a plethora of different concepts, formalisms, and technologies along with a lack of common modeling languages and standards.

According to (Guizzardi and Wagner 2012), DES is concerned with the simulation of real-world systems that are conceived as discrete event systems. The interplay of abstraction and concreteness between such a simulation and DEVS simulation models can be analyzed following the progression of abstraction to implementation detailed in (Zeigler 2019). Such a progression is defined as a sequence of three steps: *abstraction*, *formalization*, and *implementation*. *Abstraction* focuses on an aspect of reality (phenomenon) and greatly reduces the complexity of the reality being considered. This is the case of DES over discrete

event systems. Then, a *formalization* makes it easier to work out the implications of the *abstraction* and implement it in reality. This is the role of DEVS formalism as the widespread M&S language for the specification of DES. Finally, an *implementation* can be considered as providing a concrete realization of the *abstraction* through *formalization*. For DEVS, this is the case of DEVS executable models developed using M&S software tools. Hence, DEVS can be seen as a vehicle that allows formalizing the DES abstraction by becoming the foundation for a family of simulation systems. Since by nature simulation is a technical field (Robinson 2020), the implementation of such formal models is the practitioner's activity frequently performed during M&S activities.

In this paper, we propose a conceptualization of DEVS formalism designed as a metamodel that supports the *formalization* task. Such a conceptualization is part of a multi-level layered structure for studying the implicit and explicit knowledge related to DEVS formalism as a combination of both theory and practice points of view. The main goal of our research is to improve the vision of the DEVS-based community using a conceptual modeling perspective through the definition of a set of metamodels that act as a common modeling language for the development of a *universal representation of DEVS*. This representation uses the steps of the progression of abstraction to implementation as different (but related) Modeling Levels (ML) that support the progress of knowledge in DEVS-based M&S maintaining the traceability among concepts. At the core of the DEVS universal representation, a metamodel for defining the formal specification of DEVS formalism is required. Such a metamodel is presented in this paper as a Unified Modeling Language (UML) diagram restricted by Object Constraint Language (OCL) constraints.

We analyze conceptual models of DEVS developed by researchers in terms of the three ML proposed in our representation (i.e., *abstraction*, *formalization*, and *implementation*) to show how the DEVS community has treated the DEVS representation problem over the years. Over this literature review, we set the guidelines for our universal representation using metamodels as the core of it. The main contribution is the UML metamodel designed for the instantiation of DEVS models following their formal specification in *Classic DEVS with Ports* (Zeigler, Muzy, and Kofman 2018). Due to the extension of the metamodel, we include a general package description and the main concepts used to define DEVS models. Our aim is *i)* to offer a conceptual modeling structure for the definition of DEVS simulation models using a universal representation of DEVS, and *ii)* to be able to integrate all DEVS views (i.e., the ML) into a traceable network of metamodels in a way that they can be used in a compliance fashion way.

The remainder of this paper is structured as follows. Section 2 presents a literature review that summarizes existent representations of DEVS based on metamodels and ontologies. It also describes the need of building a universal representation of DEVS and the structure of our proposal. This structure sets the foundations of our research, introducing the guidelines that motivate the development of a metamodel for DEVS formalization. Section 3 introduces the core of the DEVS universal representation: the "DEVS Formal Specification" metamodel. In this section, we review the formal definition of *Classic DEVS with Ports* and present our conceptualization of this domain using UML diagrams. It also presents an instance of the metamodel as proof of concepts of our proposal. In this case, we use as an example the "Switch" model. Section 4 is dedicated to the discussion of our results and their relation with the universal representation of DEVS. Finally, Section 5 is devoted to conclusions and future work.

2 RELATED WORK

Over the years, researchers have designed several conceptual models to study the DEVS formalism. Most papers use ontologies and metamodels to analyze an individual view of DEVS formalism. These views can be grouped in three ML following the M&S dimensions (mentioned in Section 1) as *abstraction*, *formalization*, and *implementation*.

At the *abstraction* ML, the conceptualizations deal with the domain of discrete event systems. In Guizzardi and Wagner (2010), the authors present DESO, a foundational ontology for discrete event system modeling derived from the foundational ontology UFO. The main purpose of such an ontology is to provide a basis for evaluating discrete event simulation languages. Authors claim that "a discrete event

system model may be expressed at different levels of abstraction". Hence, the paper presents two ontologies: *(a)* the Design-Time Ontology that describes a discrete event system by defining the entity types and *(b)* the Run-Time Ontology that deals with individuals of different types from the simulator perspective. Since DESO represents a general conceptualization of discrete systems, it can serve as a reference ontology for evaluating the simulation languages of DES frameworks (e.g., DEVS formalism).

A DEVS conceptualization based on the *formalization* ML should be designed focusing on the entities that describe the simulation models. Several researchers have proposed models to address this ML. Hu, Zhao, and Rong (2013) propose an ontology-based model representation named DEVSMO (DEVS math ontology). Such a model is composed of three ontologies: *(a)* the DEVS model ontology that describes the classification of simulation models according to DEVS formalism, *(b)* the model structure ontology that defines both atomic and coupled model structures, and *(c)* the model behavior ontology that specifies the behavior of a DEVS model. Most concepts related to the *formalization* ML are defined in *(a)* and *(b)*. In (Touraille 2012), the author presents a metamodel of DEVS that provides a pivot format for building simulation models compatible with available software tools. Such a metamodel conceptualizes the specification of DEVS simulation models using two distinct parts: *(a)* a structural part that deals with state variables, ports, components, and connections, and *(b)* a behavioral part that describes the temporal evolution of models in terms of the DEVS functions. In the same direction, a DEVS behavioral metamodel is proposed in (Sarjoughian, Alshareef, and Lei 2015). Authors use metamodeling to generate concrete models from domain-specific metamodels. Therefore, the metamodel identifies a set of abstractions for state transitions in the external, internal, and confluent transition functions. Similarly, appropriate abstractions are designed for output and time advance functions.

On the other hand, Hollmann, Cristiá, and Frydman (2015) propose a formal modeling language called CML-DEVS (Conceptual Modeling Language for DEVS) that provides an abstract description of DEVS models in terms of logical and mathematical expressions. In this case, CML-DEVS can be seen as an improved version of the specification language for DEVS models named DEVSpecL that has been previously developed by Hong and Kim (2006). By describing DEVS models in their most abstract form, independent of any particular implementation, this modeling language has two main advantages: *(a)* it allows the interoperability between practitioners and/or researchers; and *(b)* it facilitates both maintenance and modification of the simulation models. Moreover, the main benefit of this language is that the modeler can define a model without having programming skills.

Finally, at the *implementation* ML, most conceptualizations deal with programming code generation processes. Since there is no common DEVS format used by all tools, modelers are tied to their M&S tool (Van Tendeloo and Vangheluwe 2017). Hence, each conceptualization used for implementing DEVS models is unique. Garredu et al. (2012a) propose a platform-independent metamodel for DEVS that is enriched with OCL constraints in (Garredu et al. 2012b). In this case, the authors state that such a metamodel is "accurate enough to specify several DEVS models". From a different point of view, Kapos et al. (2014) discuss the adoption of Model-Driven Architecture concepts to transform SysML (Systems Modeling Language) models into executable DEVS models. In this paper, a MOF metamodel for the implementation of DEVS is presented to provide a standard representation for the simulation-specific domain. A similar approach is presented in (Cetinkaya, Verbraeck, and Seck 2010) where authors introduce a metamodel for component-based hierarchical simulation based on hierarchical DEVS. This research initiates a study of defining a formal component-based conceptual modeling technique to overcome the problems in hierarchical simulation. The authors also present a prototype of a simulation model design environment with a Java interpreter which transforms the visual simulation models into DEVS models. Hence, the conceptualization at *implementation* ML is attached to the Java platform.

2.1 The Need for a Universal Representation of DEVS

According to (Guizzardi and Wagner 2012), DES is concerned with the simulation of real-world systems that are conceived as discrete event systems. The behavior models of this type of systems can be described at different levels of abstraction. This implies that each abstraction level may employ different

formalisms. The particular formalism and level of abstraction depends on the background and goals of the modeler as much as on the system modeled (Vangheluwe 2008).

There is widespread agreement that the paradigm of DES forms a core discipline of simulation (Guizzardi and Wagner 2010). That is because reality is often represented employing discrete-event models. In these models, time evolves continuously, but the state of the system only changes at a finite number of points in time in a bounded time-interval (called event-times). Therefore, for DES, DEVS has found increasing acceptance in the model-based simulation research community as a suitable M&S formal specification.

DEVS is formalized using set theory and systems theory. The formalism includes two types of DES models: *atomic* and *coupled*. Both models are described using equations, functions, sets, etc. Hence, DEVS is an abstract formalism for the specification of simulation models that is independent of any particular implementation. However, when engineers want to simulate these models they need to program them in the input language of a concrete simulator, which means writing code in Java or C++ or another general-purpose programming language (Cristiá, Hollmann, and Frydman 2019). Such implementation is often called "reduction to concrete form" (Zeigler 2019).

Nowadays, there are multiple software tools and simulators for DEVS models (Van Tendeloo and Vangheluwe 2017). Even when conceptualizations of *implementations* exist, in most cases modelers must represent the DEVS model as programming code using predefined libraries offered by each simulator (Nikolaidou et al. 2008). Each simulator has its own input language. Generally, these input languages are different, hindering the interoperability between simulation software tools (Hollmann, Cristiá, and Frydman 2015).

Even when over the years several conceptualizations of DEVS have been developed, most approaches are isolated from each other due to the conceptual view used during their design. By nature, simulation is a technical field (Robinson 2020). Hence, most conceptualizations focus their attention on the more "scientific" elements of the simulation project life-cycle, that is, model development (led from computer science) and analysis (led from mathematics and statistics). Moreover, even when it is a fact that DEVS models can be mathematically described (at the *formalization* ML), its simulation is performed by concrete DEVS simulation systems (at the *implementation* ML). When concrete DEVS models are developed using programming languages, it is difficult to ensure they conform to their formal model (Sarjoughian, Alshareef, and Lei 2015). Moreover, the mathematical properties and constraints defined in DEVS models should be guaranteed in any implementation of it. However, the conceptualizations developed at the *formalization* ML are rarely a "real" representation of DEVS formal models.

Since most formalization metamodels are used to get DEVS executable models, commonly their definition involves some programming features. For example, in (Touraille 2012), the author uses programming concepts such as "EClass" and "EDataType" to define state variables and ports. In (Sarjoughian, Alshareef, and Lei 2015), the DEVS functions are directly defined as extensions of "EOperation" to allow including content that can be transformed into concrete (programming) code. The formal modeling language proposed in (Hollmann, Cristiá, and Frydman 2015) is the most complete formal specification of DEVS simulation models. In this case, the authors complement the proposal with a multi-target compiler that allows getting DEVS executable models for two distinct M&S tools (DEVS-Suite and PowerDEVS). Most executable representations are tool-specific, though efforts are underway to define a common standard (Wainer et al. 2010). Even when the *formalization* ML can be used as support of the *implementation* ML, the variety of executable representations difficult the alignment task.

In this context, a universal representation of DEVS can help to understand (and study) the following ML maintaining the traceability among them: *(a)* DES approaches, *(b)* DEVS formal models, *(c)* the implementation of DEVS models in a platform-independent model, and *(d)* the implementation of DEVS models in platform-specific software tools. Each ML can be interpreted as a view of DEVS. All the views together provide a full understanding of DEVS simulation models in a unique conceptualization that allows outline interactions among distinct views.

From the traditional point of view, a good conceptual model lays a strong foundation for successful simulation modeling and analysis (Robinson et al. 2010). Over such interpretation, conceptual modeling serves as a bridge between problem owner and simulation modeler. It is important to denote that the universal representation of DEVS does not include the conceptual model of the DEVS solution. This conceptualization is attached to the problem-solution modeling task and is outside the scope of the DEVS universal representation. In such a case, the systems concept underlying DEVS should be considered to get a conceptual description at the general system level.

2.2 Our Approach: A Multi-Level Layered Conceptualization

Figure 1 shows the architecture of a multi-level conceptualization of DEVS defined as a set of five layers. Each layer is an independent ML (i.e., *phenomenon*, *abstraction*, *formalization*, *platform-independent implementation*, and *platform-specific implementation*) that supports the definition of a DEVS view through the use of metamodels. All views together give a structure for the DEVS universal representation.

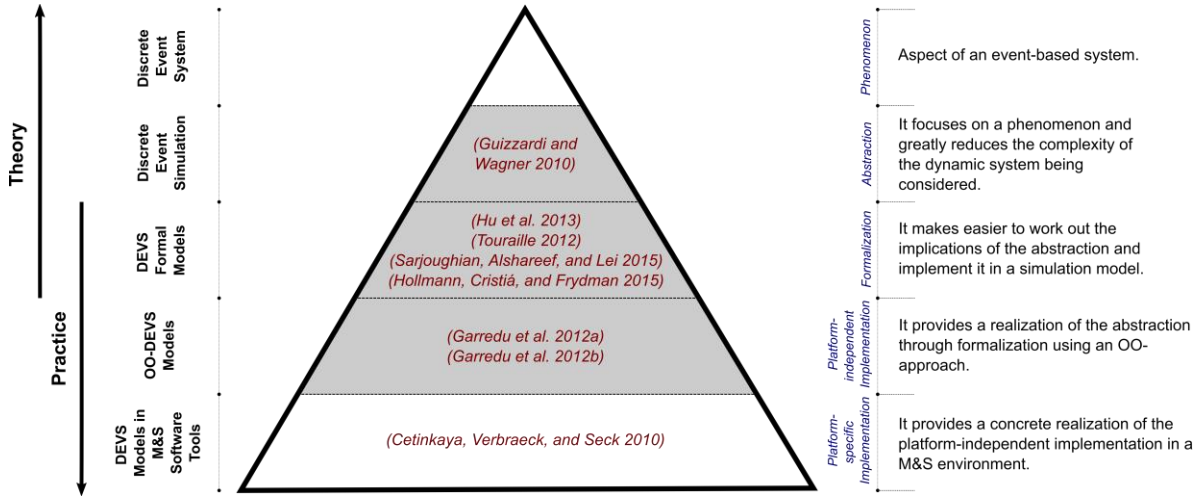

Figure 1: Multi-Level Layered Conceptualization of DEVS.

The *phenomenon* ML models an aspect of the dynamic system of interest. The *abstraction* ML represents the model of the phenomenon in terms of the DES. It focuses on applying the simulation perspective over the main components and relationships identified at the top level. Over the DES perspective, the *formalization* ML offers a M&S language (i.e., the DEVS formalism) to define the simulation model of the event-based system under consideration. That is, it details how the abstraction of discrete events, states, and components can be formalized in DEVS simulation models. Even when the abstraction is achieved by the formal statement of the modeling domain, such a formalization can be described using distinct formal statements than the ones used for the formalization ML. In Figure 1, the abstracted knowledge will be represented with generic-DES concepts. Meanwhile, the formalized model will be represented with DEVS concepts. Hence, the phenomenon definition becomes the colloquial description that cannot be operable without having any DES specificity. Finally, the *implementation* ML is divided into two levels: *platform-independent implementation* and *platform-specific implementation*. Such a separation of concerns at the *implementation* ML is designed to allow including the conceptual models of available DEVS software tools as part of our proposal. Considering that *i)* DEVS is most naturally implemented in a computational form in an Object-Oriented (OO) framework (Zeigler, Muzy, and Kofman 2018), and *ii)* a formalization model can potentially be used to produce diverse OO implementations; the model defined at the *platform-independent implementation* ML condense an OO representation of DEVS executable models. Hence, this ML is focused on the common-concepts included in most OO implementations of DEVS software tools. Then, the *platform-specific implementation* ML provides a concrete realization of the *platform-independent implementation* that can be deployed in a target M&S environment based on an OO programming code.

At the top level, the first three MLs refer to the theoretical field of DEVS. At the bottom, the last three MLs refer to the DEVS practical field. Due to the general system basis of DEVS, the *formalization* ML is shared by both theory and practice fields. In this context, the main benefit of our architecture is the interplay of DEVS theory and practice in a holistic approach that establishes the foundations of the core concepts of a DEVS conceptualization as a clarification of its real-world semantics through the use of multiple layers. The goal of building a universal representation of DEVS is to explicitly define a set of common views of DEVS structured as metamodels that can be used to instantiate a group of related discrete-event specifications. Such views enable a broad M&S advance for DEVS conceptualizations. The use of metamodels to support the conceptual architecture provides a common basis for defining instances of such models and handling transformation between MLs (i.e., model-to-model transformations).

Our current research is centered on the three middle levels of Figure 1 (i.e., levels highlighted in gray): *abstraction*, *formalization*, and *platform-independent implementation*. The *phenomenon* and *platform-specific model* are not included at this stage but they will be added in future works. Figure 2 outlines how metamodels of different MLs will provide support to the DEVS model definition. Here, a full DEVS model definition is given by the set of models instantiated as *{abstraction model, formalization model, platform-independent implementation model}*. Each model represents a view of the final DEVS simulation model based on a metamodel.

In Figure 2, three metamodels are defined: *(a)* Discrete Event Simulation (DES), *(b)* DEVS Formal Specification (DFS), and *(c)* DEVS OO-Implementation (DOI). Each metamodel is sketched as a set of concepts and relationships. All metamodels have a different number of concepts and distinct relationships between these concepts with aims to show their domain independence. As an interoperability example, all metamodels include the *State* concept. However, even when all *States* refer to the same DEVS model, each metamodel uses the concept to define the state of the model from a specific point of view. Then, the metamodel structure allows defining traceability relationships (i.e., the *trace* association) between concepts. Such traceability can be extended to relationships. This traceability allows aligning concepts from different metamodels with aims to get new knowledge regarding the model definition.

Hence, the structure of our DEVS conceptualization provides a basis for defining transformation and traceability between metamodels. Moreover, each ML can be used to support interoperability with other metamodels of the same level. For example, the DES metamodel can be used to interoperate with the abstraction level of other simulation approaches. At the core of the universal representation of DEVS, the grand challenge is the development of the DFS metamodel. Such a metamodel is presented in Section 3.

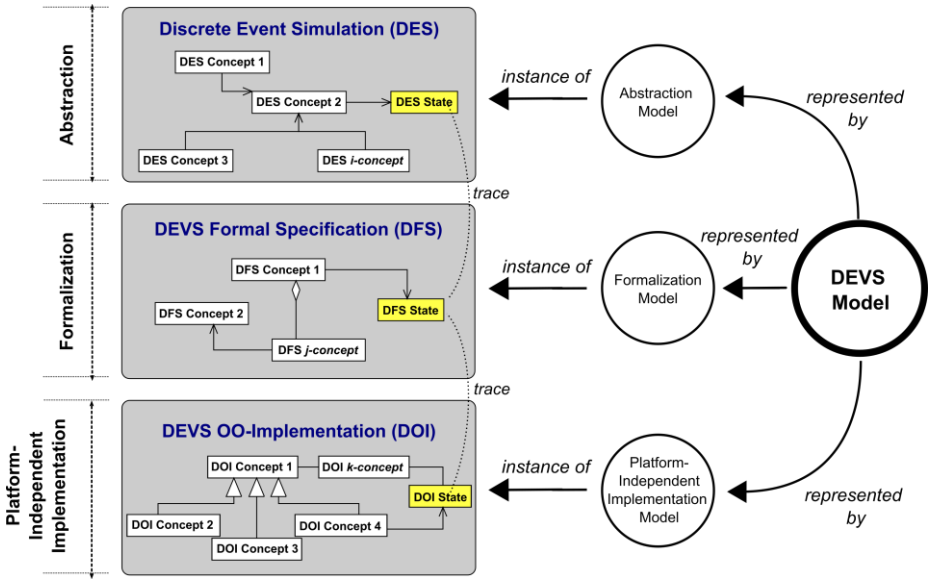

Figure 2: Views of DEVS structured as metamodels.

3 THE "DEVS FORMAL SPECIFICATION" METAMODEL

3.1 Classic DEVS with Ports

Since modeling is made easier with the introduction of input and output ports (Zeigler, Muzy, and Kofman 2018), we employ the *Classic DEVS with Ports* specification to develop our metamodel. In this formalism, a *DEVS atomic model* is defined as:

$$\text{DEVS} = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta) \tag{1}$$

where

$X = \{(p,v) \mid p \in \text{InPorts}, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in \text{OutPorts}, v \in Y_p\}$ is the set of output ports and values;

S is the set of sequential states;

δ_{ext}: $Q \times X \rightarrow S$ is the external state transition function, with $Q := (s,e) \mid s \in S, 0 \leq e \leq ta(s)$;

δ_{int}: $S \rightarrow S$ is the internal state transition function;

λ: $S \rightarrow Y$ is the output function;

ta: $S \rightarrow R_0^+ \cup \infty$ is the time advance function.

On the other hand, a *DEVS coupled model* is defined as:

$$N = (X, Y, D, M_d \mid d \in D, EIC, EOC, IC, Select) \tag{2}$$

where

$X = \{(p, v) \mid p \in \text{IPorts}, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p, v) \mid p \in \text{OPorts}, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names;

For each $d \in D$, $M_d = (X_d, Y_d, S, \delta_{ext}, \delta_{int}, \lambda, ta)$ is a DEVS, with $X_d = \{(p,v) \mid p \in \text{IPorts}_d, v \in X_p\}$ and $Y_d = \{(p,v) \mid p \in \text{OPorts}_d, v \in Y_p\}$

$EIC \subseteq \{((N,ip_N), (d,ip_d)) \mid ip_N \in \text{IPorts}, d \in D, ip_d \in \text{IPorts}_d\}$ is the set of external input couplings that connect external inputs to component inputs;

$EOC \subseteq \{((d,op_d),(N,op_N)) \mid op_N \in \text{OPorts}, d \in D, op_d \in \text{OPorts}_d\}$ is the set of external output couplings that connect component outputs to external outputs;

$IC \subseteq \{((a,op_a), (b,ip_b)) \mid \{a,b\} \in D, op_a \in \text{OPorts}_a, ip_b \in \text{IPorts}_b\}$ is the set of internal couplings that connect component outputs to component inputs;

Select: $2^D \rightarrow D$, the tie-breaking function.

3.2 UML Metamodel

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. We have divided the concepts and relationships in a UML package diagram (Figure 3). Each package groups concepts and relationships according to their scope. Due to length constraints, for each package, we introduce its purpose along with a link to our repository where the full description is provided as a UML class diagram.

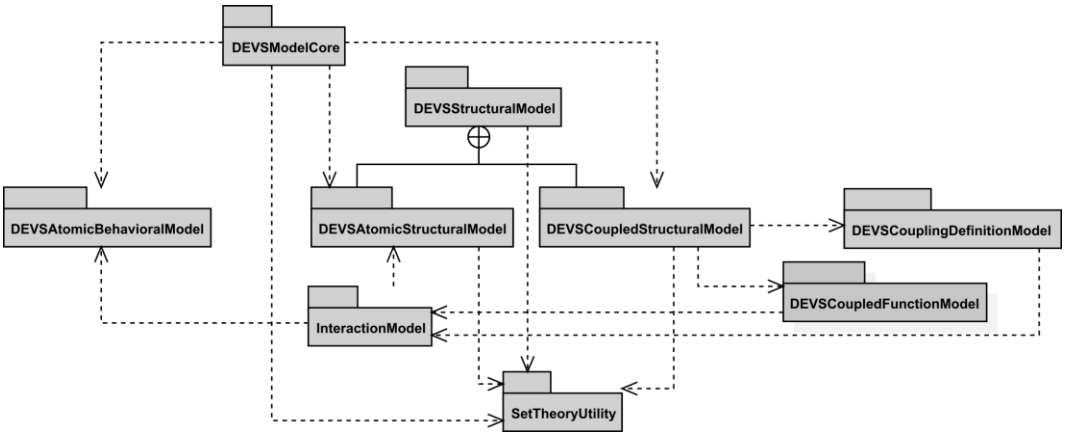

Figure 3: UML package diagram of the metamodel.

The package named *DEVSModelCore* defines the concepts *AtomicModel* and *CoupledModel*. These concepts are used to instantiate a *DEVSModel*. Our conceptualization of *DEVSModel* adds two elements to the traditional definitions. These elements are used to arrange the sets used to build the model. The *EmptySet* (from *SetTheoryUtility*) provides a conceptualization of a mathematical empty set that can be further employed in the specification of new sets. The *Definition* (from *SetTheoryUtility*) is used as an optional feature of the model that groups all sets and structures used during the model definition.

An *AtomicModel* is conceptualized as a container of two components: *(a)* the *AtomicStructuralPart* (from *DEVSAtomicStructuralModel*) that defines the inputs and outputs sets and the state composition, and *(b)* the *AtomicBehavioralPart* (from *DEVSAtomicBehavioralModel*) that specifies required functions. On the other hand, a *CoupledModel* is conceptualized using a *CoupledStructuralPart* (from *DEVSCoupledStructuralModel*) that defines the inputs and outputs sets, the couplings, and the select function. The structure of the couplings is defined in the *DEVSCouplingDefinitionModel* package. This package includes the concepts used to define the specific set of couplings required as part of the *CoupledModel* definition such as *EIC*, *EOC*, and *IC*. For each set, the package defines how a specific coupling should be structured. For example, the *IC* is defined as a container of a list of *InternalCoupling*. Each *InternalCoupling* refers to two *Components* included in the *CoupledStructuralPart*.

The *DEVSStructuralModel* package defines the *StructuralPart* of a *DEVSModel* as a container of *i)* a list of *InputPort*, *ii)* a list of *OutputPort*, *iii)* an input set *X*, *iv)* a list of *Parameter*, and *v)* an output set *Y*. Using this definition, the *DEVSAtomicStructuralModel* package defines the *AtomicStructuralPart* by adding the definition of *State* as part of the components included in the container. The *DEVSCoupledStructuralModel* package also extends the *StructuralPart* definition to conceptualize the *CoupledStructuralPart*. The package adds *i)* the set *D*, *ii)* the set *EIC*, *iii)* the set *EOC*, *iv)* the set *IC*, and *v)* the *Select* function (from *DEVSCoupledFunctionModel*). As opposite, the package named *DEVSAtomicBehavioralModel* defines the *AtomicBehavioralPart* as a container of *i)* a *ExternalTransitionFunction*, *ii)* a *InternalTransitionFunction*, *iii)* an *OutputFunction*, and *iv)* a *TimeAdvanceFunction*. For *ExternalTransitionFunction* and *InternalTransitionFunction*, the model uses a list of *NextStateSpecification*. A *NextStateSpecification* is related to a collection of *VariableSpecification* that plays the role of *stateVariable*. The *VariableSpecification* (from *InteractionModel*) is also related to a *StateVariable* (from *DEVSAtomicStructuralModel*) to define how a *StateVariable* (defined in the *AtomicStructuralPart*) evolves during the state transition function. A similar approach was used to define the *OutputFunction* and *TimeAdvanceFunction* using *OutputSpecification* and *TimeAdvanceSpecification*, respectively. The *SelectFunction* was also defined following this conceptualization guideline as part of the *DEVSCoupledFunctionModel* package.

The *InteractionModel* package specifies the intermediate elements to be used for linking: *(a)* the *AtomicStructuralPart* and the *AtomicBehavioralPart* compositions for *AtomicModel*, and *(b)* the

Coupling definition among *Component* for *CoupledModel*. In the first case, the concepts abstract the interactions between both parts by modeling the components of the behavioral level as elements that depend on the structural level. This allows instantiating the specification of the functions using independent elements that have a dependency on the structure of the model. In the second case, depending on the type of coupling, the concepts abstract the specification of input/output ports of the *CoupledModel* with the ports of its *Component*.

Finally, the *SetTheoryUtility* package includes all the concepts required to define the set theory basis used when DEVS models are formally specified. These concepts are used according to the formalism definition to detail sets for input and output ports, state variables, parameters, and so on. The package includes concepts as *BooleanAlgebraSet*, *NaturalNumberSet*, and *RealNumberSet* among others. Moreover, it allows the modeler to define new sets using the *Element* hierarchy. Then, for example, the modeler can create a *NewSet* by defining specific values to a set of *StringElement*.

Over the UML model, we restrict the instantiation with OCL constraints to ensure consistency between the formal definition and the metamodel. OCL is a declarative language describing rules applying to UML models. Hence, we use OCL to ensure consistency related to the correctness of DEVS formalism. For example, these constraints ensure *i)* no direct feedback loops in *CoupledModel*, *ii)* correctness of the *Coupling* specification in terms of the sets defined in the *Port* of each *Component*, and *iii)* the set of *Elements* used in an *Expression* needs to be compatible with the expected value of the *StateVariable*.

3.3 Proof of Concepts

We use as an example the “switch” model. A switch is modeled as an atomic model with pairs of input and output ports. When the switch is in the standard position, the elements arriving on port “in” are sent out on port “out”, and similarly for ports “in1” and “out1”. When the switch is in its other setting, the input-to-output links are reversed. The formal specification of this model is detailed in (Zeigler, Muzy, and Kofman 2018). Over such a definition, we consider *V = {0,1}* and *processing_time = 15.85*.

Table 1 summarizes how each part of the formal specification is defined as a model derived from our metamodel. Some of the extra instances defined to complete the model instantiation are *i)* the *Parameter* with *name="processing_time"* and definition as a *RealNumber* with *value=15.85*, and *ii)* the sets used to define the type attached to ports and state variables. In this last case, for example, the definition of *V* is instantiated with a *NewSet* that includes two instances of *IntegerNumber*. The first instance is set with *value=0*, while the other one is set with *value=1*.

For space reasons, the proof of concepts only includes an *AtomicModel* instantiation. However, following the same reasoning, an instantiation of *DEVSCoupled* can be easily defined.

4 DISCUSSION

Following the proof of concepts, Figure 4 shows the instantiation of the state variable named *phase* as part of the state definition. The object diagram (i.e., the objects depicted on the left side of the figure) uses the concepts of *State*, *StateVariable*, *NewSet*, and *StringElement* as instances of the metamodel with aims to detail how the state variable “phase” is defined at the *formalization* ML.

On the other hand, the right side of Figure 4 (i.e., the classes with a light gray background) shows how the formalization definition can be interpreted at the *implementation* ML. In both MLs, the concept of *State* exists. At the *implementation* ML, each *StateVariable* defined formally in the model is an attribute of the *State* class. In the example, the *StateVariable* with *name = phase* is translated to the *phase* attribute. Since the *StateVariable* is attached to a *NewSet* defined as two instances of *StringElement*, the type of the *phase* attribute is defined using an *Enumeration* (i.e., *PhaseValue*). In such an *Enumeration*, the literals are defined using the *value* property of *StringElement*. Hence, when the proposed MLs are defined using metamodels, suitable traceability relationships can be specified across concepts and relationships.

Table 1: The "Switch" model as instance of our metamodel.

Model	Instances
SWITCH	An `AtomicModel` with `name="SWITCH"`. The instance of `AtomicModel` contains an `AtomicStructuralPart` and an `AtomicBehavioralPart`.
X	A `X` containing two instances of `InputDefinition`. The first one relates the `InputPort` with `name="in"` and the `InputVariable` with `name="x_in"`. The second instance links the `InputPort` with `name="in1"` and the `InputVariable` with `name="x_in1"`.
Y	A `Y` containing two instances of `OutputDefinition`. The first one links the `OutputPort` with `name="out"` and the `OutputVariable` with `name="y_out"`. The other one links the `OutputPort` with `name="out1"` and the `OutputVariable` with `name="y_out1"`.
S	A `State` containing five instances of `StateVariable` identified with *i)* `name="phase"`, *ii)* `name="sigma"`, *iii)* `name="inport"`, *iv)* `name="store"`, and *v)* `name="Sw"`.
δ_{ext}	An `ExternalTransitionFunction` that contains two `NextStateSpecification`. E.g., the first specification is defined as an ordered set that includes: *i)* a `ValueFromSet` with `value="active"`, *ii)* a `ParameterName` with `name="processing_time"`, *iii)* an `InputPortIdentifier`, *iv)* an `InputPortValue`, and *v)* a `PreviousStateVariable` with `name="Sw"`. `Condition` instances are defined appropriately for each specification.
δ_{int}	An `InternalTransitionFunction` that contains a `NextStateSpecification` that includes: *i)* a `ValueFromSet` with `value="passive"`, *ii)* a `InfinityParameter`, *iii)* a `PreviousStateVariable` with `name="inport"`, *iv)* a `PreviousStateVariable` with `name="store"`, and *v)* a `PreviousStateVariable` with `name="Sw"`.
λ	An `OutputFunction` that contains four instances of `OutputSpecification`. As an example, the first `OutputSpecification` takes as value the `PreviousStateVariable` with `name="store"` and as port the `OutputPortName` with `name="out"`. `Condition` instances are defined appropriately for each specification.
ta	A `TimeAdvanceFunction` that contains a `TimeAdvanceSpecification` defined using the `PreviousStateVariable` with `name="sigma"`.

Figure 4 is an example of the traceability relationship depicted in Figure 2 for the *State* concept. As part of the universal representation of DEVS, our metamodel can help to *(a)* validate the consistency between discrete event system descriptions and DEVS formal models through the use of a DES metamodel, *(b)* study the relation between DEVS formalization and most commonly used implementations when combined with a platform-independent metamodel, and *(c)* verify the consistency between DEVS formal models and their implementation at a platform-independent level to ensure the that the expected executable model is the one formalized. All these benefits are derived from the capability of defining model-to-model transformations over the distinct ML (i.e., defining traceability relationships between concepts defined at different levels). With the existing modeling technology (such as, Ecore), all these metamodels can be implemented to provide a software framework that supports all these features.

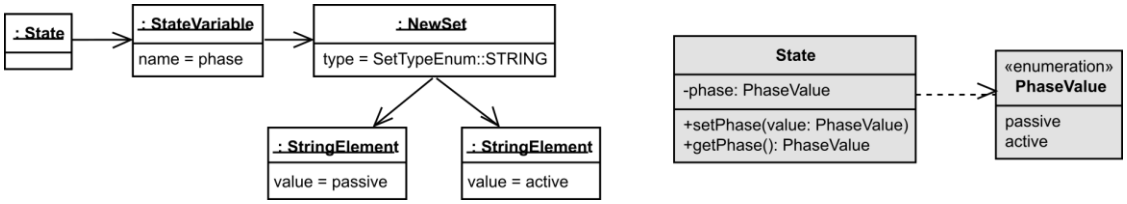

Figure 4: Example of the "Switch" model state formalization vs. implementation.

As a remark, it is interesting to briefly discuss how our proposal is related to the Model-Driven Architecture (MDA) approach. MDA involves the specification of models at three levels: models, metamodel, and meta-meta models. The capabilities of our metamodel can be improved with a meta-meta model that acts as a standard definition of formalisms. Hence, such a meta-meta model will allow bridging the gap between DEVS formal models and other alternative modeling formalisms. Moreover, our metamodel can be also used to provide a foundation for building multi-formalism models based on DEVS and its extensions. An appropriate alignment of the models' composition can be defined to study the impact of multi-formalism model definitions at a conceptual level.

5 CONCLUSIONS AND FUTURE WORK

DEVS is a popular modular and hierarchical formalism for modeling complex dynamic systems using discrete-event abstraction. In this paper, we have presented a metamodel that defines a conceptualization of DEVS formal models. Regarding the metamodel specification, we employ UML to define all the concepts and relationships required to instantiate the *Classic DEVS with Ports* specification. Moreover, we restrict the instantiation of such a model with a set of OCL constraints to ensure consistency between the formal definition and the metamodel.

This paper is the starting point of a work-in-progress aimed to define a universal representation of DEVS using a multi-level layered architecture of metamodels. Such a representation extends the DEVS conceptualization trying to capture the relationships among distinct ML designed as DEVS views. We are working on a software tool that allows instantiating a formal specification of the *AtomicModel* concept using a guided-design process. The metamodel implementation is based on Ecore aiming to provide a further integration to Java as an OO platform. The future work includes the design of the metamodels attached to other views. For the *phenomenon* ML, the systems concept underlying DEVS (i.e., the I/O system which is specifiable at multiple levels of structure and behavior with their associated morphisms) will be included to define a conceptual description at the general system level. In such a case, the uniqueness and well-definition of the DEVS abstract simulator will be considered.

REFERENCES

Cetinkaya, D., A. Verbraeck, and M. D. Seck. 2010. "A Metamodel and a DEVS Implementation for Component based Hierarchical Simulation Modeling". In *Proceedings of the 2010 Spring Simulation Multiconference*, pp. 130-138. Society for Computer Simulation International.

Cristiá, M., D. A. Hollmann, and C. Frydman. 2019. "A Multi-target Compiler for CML-DEVS". *Simulation*, vol. 95(1), pp. 11-29.

Garredu, S., E. Vittori, J. Santucci, and P. Bisgambiglia. 2012a. "A Meta-Model for DEVS-Designed following Model Driven Engineering Specifications". In *Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Techniques and Applications*, pp. 152-157.

Garredu, S., E. Vittori, J. Santucci, and D. Urbani. 2012b. "Enriching a DEVS Meta-model with OCL constraints". In *Proceedings of the 24th European Modeling and Simulation Symposium*, pp. 216-225.

Guizzardi, G., and G. Wagner, G. 2010. "Towards an Ontological Foundation of Discrete Event Simulation". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, pp. 652-664. Piscataway, New Jersey, Institute of Electrical and Electronics Engineers, Inc.

Guizzardi, G., and G. Wagner, G. 2012. "Conceptual Simulation Modeling with Onto-UML". In *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A.M. Uhrmacher, pp. 52-66. Piscataway, New Jersey, Institute of Electrical and Electronics Engineers, Inc.

Hollmann, D. A., M. Cristiá, C. Frydman. 2015. "CML-DEVS: A Specification Language for DEVS Conceptual Models". *Simulation Modelling Practice and Theory*, vol. 57, pp. 100-117.

Hong, K., and T. Kim. 2006. "DEVSpecL: DEVS Specification Language for Modeling, Simulation and Analysis of Discrete Event Systems", *Information and Software Technology*, vol. 48, pp. 221–234.

Hu, Y., J. Xiao, H. Zhao, and G. Rong. 2013. "DEVSMO: An Ontology of DEVS Model Representation for Model Reuse". In *Proceedings of the 2013 Winter Simulation Conference*, edited by R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, pp. 4002-4003. Piscataway, New Jersey, Institute of Electrical and Electronics Engineers, Inc.

Kapos, G. D., V. Dalakas, A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos. 2014. "Model-based System Engineering using SysML: Deriving Executable Simulation Models with QVT". In *Proceedings of the 2014 IEEE International Systems Conference*, pp. 531-538.

Nikolaidou, M., V. Dalakas, L. Mitsi, G. D. Kapos, D. Anagnostopoulos. 2008. "A SysML Profile for Classical DEVS Simulators". In *Proceedings of the 3rd International Conference on Software Engineering Advances*, pp. 445-450.

Robinson, S. 2020. "Conceptual Modelling for Simulation: Progress and Grand Challenges". *Journal of Simulation*, vol. 14(1), pp. 1-20.

Robinson, S., R. Brooks, K. Kotiadis, and D. J. Van Der Zee. 2010. *Conceptual modeling for discrete-event simulation*. Boca Raton, USA: CRC Press.

Sarjoughian, H. S., A. Alshareef, Y. Lei. 2015. "Behavioral DEVS Metamodeling". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, W K V. Chan, I Moon, T. M K Roeder, C Macal, and M D Rossetti, pp. 2788-2799. Piscataway, New Jersey, Institute of Electrical and Electronics Engineers, Inc.

Touraille, L. 2012. *Application of Model-driven Engineering and Metaprogramming to DEVS Modeling & Simulation*. Doctoral dissertation, Universite´ d'Auvergne, France.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "An Evaluation of DEVS Simulation Tools". *Simulation*, vol. 93(2), pp. 103-121.

Vangheluwe, H. 2008. Foundations of Modelling and Simulation of Complex Systems. In *Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques*, pp. 1-12.

Wainer, G., K. Al-Zoubi, S. Mittal, J.L. Risco Martin, H. Sarjoughian, and B. P. Zeigler. 2010. "An Introduction to DEVS Standardization". In *Discrete-Event Modeling and Simulation: Theory and Applications*, edited by G. Wainer and P. Mosterman, pp. 393-425. Boca Raton, USA: CRC Press.

Zeigler, B. P. 2019. "How Abstraction, Formalization and Implementation Drive the Next Stage in Modeling and Simulation". In *Summer of Simulation*, edited by J. Sokolowski, U. Durak, N. Mustafee, A. Tolk, pp. 25-37. Switzerland: Springer Nature.

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.

AUTHOR BIOGRAPHIES

MARIA JULIA BLAS is a Postdoctoral Fellow at INGAR and an Assistant Professor at UTN. She received her Ph.D. degree in Engineering from UTN in 2019. Her research interests include discrete-event M&S. Her email address is mariajuliablas@santafe-conicet.gov.ar.

SILVIO GONNET received his Ph.D. degree in Engineering from UNL in 2003. He currently holds a Researcher position at CONICET. His research interests are models to support design processes and conceptual modeling. His email address is sgonnet@santafe-conicet.gov.ar.

BERNARD P. ZEIGLER is Professor Emeritus of Electrical and Computer Engineering at the University of Arizona (USA) and Chief Scientist of RTSync Corp. (USA). Dr. Zeigler is a Fellow of IEEE and SCS and received the INFORMS Lifetime Achievement Award. He is a co-director of the Arizona Center of Integrative Modeling and Simulation. His e-mail address is zeigler@rtsync.com.