Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

A CONTEXT-FREE GRAMMAR FOR GENERATING FULL CLASSIC DEVS MODELS

María Julia Blas
Silvio Gonnet

Instituto de Desarrollo y Diseño INGAR
Universidad Tecnológica Nacional
Avellaneda 3657
Santa Fe, SF 3000, ARGENTINA

Doohwan Kim
Bernard P. Zeigler

RTSync Corp.
6909 W. Ray Rd. STE. 15-107
Chandler, AZ 85226, USA

ABSTRACT

Existing grammars generate Finite Deterministic DEVS models, a restricted subset of DEVS. The proposed context-free grammar generates the unrestricted set of Classic DEVS models. The grammar is implemented in ANTLR, a powerful parser generator for reading, processing, executing, or translating structured text or binary files. ANTLR enables the efficient processing of the specifications needed for generating members of Classic DEVS with ports. Applications include an easier introduction to DEVS for students and easier translation between different DEVS implementations.

1 INTRODUCTION

The Discrete Event System Specification (DEVS) (Zeigler et al. 2018) is a Modeling and Simulation (M&S) formalism that supports a general methodology for describing discrete event systems with the capability to represent both continuous and discrete systems due to its system theoretic basis. Both types of DEVS models (i.e., atomic and coupled) are formalized by combining set theory and systems theory. When practitioners want to simulate DEVS models, they need to program them in the input language of a concrete simulator (Cristiá et al. 2019). That means writing code in Java, C++, or another general-purpose programming language. Such implementation is often called “reduction to concrete form” (Zeigler 2019).

In this regard, a DEVS formal model is defined using DEVS formal specification, and a DEVS concrete model is defined as an implementation of a DEVS formal model. However, is it possible to ensure that a DEVS concrete model developed with a programming language is (in fact) an implementation of a DEVS formal model? According to Sarjoughian et al. (2015), ensuring that an implementation conforms to a formalization is not straightforward. In (Blas and Gonnet 2023), we argue this is because *i)* formalization and implementation are often carried out as two distinct tasks, and *ii)* the principles under which programming languages are designed do not easily conform with the theory used in the DEVS formal specification (i.e., the formal language supporting DEVS model definitions cannot be used directly in programming languages). Thus, building an implementation of a DEVS formal model (i.e., a concrete computer model) in a way that ensures its formal specification (i.e., the mathematical definition) is not easy.

The lack of computer languages supporting the DEVS formalization task (i.e., domain-specific computer languages) is evident. Namely, we are referring to a domain-specific modeling language. This paper addresses the use of grammar and the role of metamodeling in the M&S field devoted to defining DEVS formal models in a computer form through a new modeling language. When building a modeling language, three components are required *i)* abstract syntax, *ii)* concrete syntax, and *iii)* semantics. The main contribution of this paper is the development of a Context-Free Grammar (CFG) named CFG_DEVS that can be used to support textual definitions of DEVS formal specifications as a well-defined concrete syntax. Such a syntax was implemented using ANTLR (ANTLR 2023) and can be combined with the abstract syntax already defined in “the DEVS metamodel” (Blas and Gonnet 2023). The paper points out the technical aspects of the CFG developed. Applications include an easier introduction to DEVS for students since (as we will show with some examples) the grammar allows defining models found in exiting literature as a first step in a DEVS-learning process. Our final goal is to develop a computer language for expressing DEVS formal specifications suitable for being used across different concrete simulators (i.e., M&S software tools based on distinct programming languages). The semantics of the language is out of the scope of this paper.

The remainder of this paper is structured as follows. Section 2 presents the main concepts used across the paper by summarizing the fundamentals of the proposal. Section 3 presents the grammatical model defined as CFG_DEVS. To show both applicability and usability, it includes a set of examples (taken from the DEVS literature) (re)written in the grammar. Section 4 is dedicated to the discussion of our results, including a comparison with other DEVS existing languages and a quality analysis of our development. Finally, Section 5 is devoted to conclusions and future work.

2 COMPUTER LANGUAGES, METAMODELS, AND GRAMMARS

From the taxonomy proposed by Lando et al. (2007), we have that: *i)* programming languages are a subset of general-purpose computer languages designed to express computer programs that can be understood by humans, and *ii)* Domain-Specific Computer Languages (DSCLs) are a subset of computer languages limited to the writing of particular types of expressions. Hence, languages used in computing that have a restricted set of expressions (i.e., a distinct goal than expressing high-level computer programs) are designated as DSCLs. These languages are claimed to increase productivity while reducing the required maintenance and programming expertise (Barisic et al. 2011). Modeling languages are a particular type of DSCLs.

In the software engineering community, a modeling language is a language used to “specify, visualize, construct, and document a software system” (Rumbaugh et al. 2005). Such a definition can be extended more generally into a problem domain, giving a Domain-Specific Modeling Language (DSML). Such a type of computer language enables the specification of (computer) models using concepts and notations of a specific domain (Kelly and Tolvanen 2008). In particular, a DSML for DEVS should enable the specification of DEVS computer models using concepts and notations of the DEVS formal specification domain. In this way, DSMLs enable domain experts to develop, understand, and verify models more easily, without having to use concepts outside of their domain (Van Mierlo et al. 2019).

Modeling languages can be either graphical or textual (Engelen and Van Den Brand 2010). Visual modeling languages provide the means to effectively convey components and interactions allowing developers to focus on parts of a system/domain at any time. However, primarily textual modeling languages achieve these intended uses more concisely without the problems of secondary notation (e.g., layout and typographic cues) (Petre 1995). It is not clear what an appropriate notation (i.e., textual or graphical) is in a given context. There are no rules to apply here: graphical, textual, and hybrid notations are more or less useful depending on specific circumstances. For representing DEVS formal models, we decided to develop a DSML based on a textual notation. That is mainly because *(i)* we want to maintain the mathematical form of notation used in the formal specification, and *(ii)* textual languages are more like programming languages, making it easier for programmers to start using the DSML.

A key question is: *for what do we intend to use a DEVS formal modeling language?* Modeling languages are built to be used by designers (not programmers) (Paige et al. 2000). In this regard, we aim to develop a modeling language friendly to both DEVS designers and programmers, allowing them to build DEVS models following their formal specifications in a simple way. Simulationists (from any field, not necessarily computer sciences) will benefit from such a language. For those who do not have programming skills, the language provides an easy way to develop computational DEVS models starting from the formal specification. That is great for educational purposes since DEVS could be quickly used in practice without requiring teaching basic notions of programming languages. On the other hand, for simulationists that also are great programmers (i.e., those you can assume will have a faster development using general-purpose programming languages), the main benefit is in the verification process. Using the modeling language, the cost of ensuring a concrete model follows its formal definition is very low.

2.1 How to Build a Domain-Specific Modeling Language?

Modeling languages, like programming languages, need to be designed if they are to be practical, usable, accepted, and of lasting value (Paige et al. 2000). Commonly, these languages are defined using three primary blocks (Krahn et al. 2007): abstract syntax, concrete syntax, and semantics. The abstract syntax identifies modeling concepts, while the concrete syntax clarifies how these modeling concepts are rendered by visual and/or textual elements (Baar 2006). Both syntaxes need to be consistently defined to avoid discrepancies and problems while handling the language (Krahn et al. 2007). Frequently, a mapping function is defined to match a concrete syntax with the abstract syntax. At this point, it is worth noting that we can have more than one concrete syntax for the same abstract syntax (depending on the application scenario and modeler profile). Both syntaxes frequently are developed first, and then semantics is designed to define the meaning of the language (Harel and Rumpe 2004). To avoid confusion, the vocabulary used in this paper is defined as follows:

- A concrete modeling language is defined as the abstract syntax, concrete syntax, and semantics that defines the full vocabulary (i.e., it includes the meaning of the models built from such a language).
- For textual concrete syntaxes, a grammatical model is a method for analyzing sentence structures (Crystal 2008). Such a grammatical model includes a methodology of generative grammar designed to produce grammatically correct strings of words (Seuren 2015).

The abstract syntax of a language is typically captured in a metamodel (Kleppe 2008). Metamodeling allows defining such a syntax in a precise, non-ambiguous way. For the DEVS formal specification, we have proposed such a metamodel originally in (Blas et al. 2021). An updated version was introduced in (Blas and Gonnet 2023). The "DEVS metamodel" is defined as a layered set of ten UML packages: *Mathematical Function*, *Set Theory Utility*, *DEVS Model Core*, *DEVS Structural Model*, *DEVS Coupling Definition*, *DEVS Atomic Behavioral Model*, *DEVS Atomic Structural Model*, *DEVS Coupled Structural Model*, *DEVS Atomic Interaction Model*, and *DEVS Coupled Function Definition*. Each package groups concepts and relationships according to their scope. For example, the "DEVS Model Core" package defines basic concepts used to identify a DEVS Model such as *DEVS Model*, *Atomic Model*, *Atomic Structural Part*, *Atomic Behavioral Part*, and *Coupled Structural Part*.

According to (Baar 2006), the abstract syntax definition is the most basic block when defining a modeling language but, at the same time, it is the only block for which a commonly agreed format exists. Other blocks (i.e., concrete syntax and semantics) are given in many cases only informally. This is not our case. In the following sections, we present a grammatical model to address the concrete syntax of "the DEVS metamodel" using a CFG. A CFG is defined as a finite set of symbols (alphabet), a finite set of variables (also called nonterminal character), a finite set of productions (also called rewriting rules), and a start symbol (Hopcroft et al. 2001). All these elements allow for defining a context-free language composed of structured sentences. Indeed, we propose a CFG specifically designed to describe DEVS formal models (atomic and coupled) following the most common mathematical expressions observed in current literature.

As a remark, when building a new language, one of the main aspects to be considered regarding language implementation is the transition from (or to) existing technologies. For example, the designers of UML focused on standardizing the syntax and informal semantics of the modeling language as long as ensuring a degree of syntactic and conceptual compatibility from existing technologies to UML (Paige et al. 2000). Hence, it is not enough to build a DSML for DEVS formal models. It is also important to develop a solution that can be integrated with existing M&S software tools keeping all capabilities already provided by them. As stated earlier, there is a collective agreement that DEVS models are implemented as a tool-depending artifact (Van Tendeloo and Vangheluwe 2017). Porting models between different M&S tools involves rewriting the model from scratch since most M&S software tools are based on general-purpose programming languages (or special libraries developed over such languages). The development of a high-level modeling language with a well-defined meaning can lead to a joint community effort to allow porting models based on the translation of formal model definitions to specific implementations (without requiring rewriting tasks). From this point of view, DEVS models (been implemented as computer programs) will be implemented as computer models based on the DEVS formal specification. Such computer models act as the formalization of previous implementations in a higher level of modeling hierarchy (Blas et al. 2021).

2.2 Related Work: Existing Approaches

Over the years, several notations have been developed to address the definition of DEVS models. Some of them are (in practice) considered modeling languages. Two of the most used (and related to our work) are DEVSML (Mittal and Douglas 2012) and DEVSNL (Zeigler and Sarjoughian 2017). Each one is focused on defining DEVS simulation models from a particular modeling perspective. In both cases, there are software tools supporting their use in practice.

DEVSML (DEVS Modeling Language) provides a platform-independent way to specify DEVS models. The language is based on Finite Deterministic DEVS (FD-DEVS), an extension of DEVS whose sets of events and states are finite among other things (i.e., a restricted subset of DEVS). Like any language, DEVSML uses keywords that allow modelers to build their definitions in a structured manner. The grammar was specified using an Extended Backus-Naur Form (EBNF) notation (i.e., a meta-syntax notation for CFGs). While the atomic model has a notion of ports, the language has a notion of messages specified as entity structures that are eventually transformed into port definitions. Models defined in DEVSML can be transformed into platform-specific language implementations (such as, for example, Java and C++).

On the other hand, DEVSNL (DEVS Natural Language) provides a natural language structure to understand FD-DEVS simulation models. These models can be used to automatically generate DEVS atomic models in Java that have full capability to express messages and states. The modeling perspective is defined as a “constrained natural language specification of DEVS models”. Indeed, as in DEVSML, the textual notation was defined as an EBNF. Due to the EBNF specification, the processing of textual expressions/definitions defined in DEVSNL does not involve any natural language reasoning (i.e., the language does not involve dealing with, for example, natural language ambiguity). In Section 4, we present a DEVS atomic model defined using DEVSNL.

As detailed before, both DEVSNL and DEVSML are based on FD-DEVS. Regarding this perspective, CFG_DEVS (Section 3) can be used to define textual specifications of DEVS formal models in general (i.e., the grammar does not restrict the specification to a particular DEVS extension). Since FD-DEVS is a class of DEVS models, the set of models that can be defined in DEVSNL and DEVSML is smaller than the set of models that can be defined with CFG_DEVS. That is one of the main foundational differences between our paper and existing approaches. Other differences are described with an example in Section 4.

3 THE DISCRETE EVENT SYSTEM SPECIFICATION AS A GRAMMATICAL MODEL

3.1 DEVS Formal Models

Table 1 presents the mathematical specification of DEVS models. Such a mathematical definition with the system basis allows for building hierarchical models using sets and functions. See (Zeigler et al. 2018) for more details regarding the DEVS formal definition.

3.2 The DEVS Context-Free Grammar: CFG_DEVS

The DEVS grammar was developed using ANTLR. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files (ANTLR 2023). By using such a technology, the grammatical model was defined as an ANTLR project composed of five source files (named CFG_DEVS_MODEL.g4, CFG_SET_THEORY.g4, CFG_BOOLEAN_EXPRESSION.g4, CFG_MATH_EXPRESSION.g4, CFG_TOKENS.g4). Each file groups production rules according to its scope. Moreover, each element defined as a concept in “the DEVS metamodel” is represented by a production rule. For space reasons, we describe the main content of each file. In some cases, syntax diagrams are used to denote the structure of nonterminal symbols as a graphical approach to how the grammar is defined.

Table 1: Formal definition of DEVS models.

DEVS Atomic Model	**DEVS Coupled Model**
$DEVS = \{X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\}$ where $X = \{(p,v) \mid p \in InPorts, v \in X_p\}$ is the set of inputs, with InPorts as the set of input ports, p as the pth port of InPorts, X_p as the set of input values for the p, and v as an element of the p-values; $Y = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ is the set of outputs, with OutPorts as the set of output ports, p as the pth port of OutPorts, Y_p as the set of output values for the p, and v as an element of the p-values; S is the set of sequential states; δ_{ext} is the external state transition function; δint is the internal state transition function; λ is the output function; ta is the time advance function.	$N = \{X, Y, D, M_d, EIC, EOC, IC, Select\}$ where X and Y are defined in the same way as in the atomic model, with IPorts as the set of input ports and OPorts as the set of output ports; D is the set of the component names; For each $d \in D$, M_d is a DEVS model, with $X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$, and $Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}$; $EIC = \{((N,ip_N),(d,ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$ are the external input couplings; $EOC = \{((d,op_d),(N,op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$ are the external output couplings; $IC = \{((a,op_a),(b,ip_b)) \mid \{a,b\} \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$ are the internal couplings; Select is the tie-breaking function.

CFG_DEVS_MODEL. The CFG_DEVS_MODEL.g4 file contains the start symbol `devsModel` and imports other source files to use their notation as part of its productions. It defines that a `devsModel` can be either an `atomicModel` or `coupledModel`. For each nonterminal symbol, a production rule is specified. For example, the `atomicModel` symbol is defined as:

```
atomicModel: modelSignature EQUAL atomicModelTuple WHERESYMBOL modelDefinition;

atomicModelTuple: BEGINPARENTHESES setInModel COMMA setInModel COMMA setInModel COMMA
DELTAEXTNAMESYMBOL COMMA DELTAINTNAMESYMBOL COMMA LAMBDANAMESYMBOL COMMA TANAMESYMBOL
ENDPARENTHESES;

modelDefinition: modelSentence (SEMICOLON modelSentence)*;
```

In these rules, elements named as `EQUAL`, `WHERESYMBOL`, `BEGINPARENTHESES`, `COMMA`, `DELTAEXTNAMESYMBOL`, `DELTAINTNAMESYMBOL`, `LAMBDANAMESYMBOL`, `TANAMESYMBOL`, `ENDPARENTHESES` and `SEMICOLON` are tokens defined in the CFG_TOKENS.g4 file. The rule defined for the nonterminal symbol `modelDefinition` implies a model is defined as a sequence of (at least one) `modelSentence` separated by semicolons. Each `modelSentence` can be either a `setExplicitDefinition` or a `function`.

The `setExplicitDefinition` is described in the CFG_SET_THEORY.g4 file. The `function` is a generic nonterminal symbol for defining `delExtFunction`, `delIntFunction`, `lambdaFunction`, `taFunction`, and `selectFunction`. A partial syntax diagram of such a symbol can be seen here. At the basic level of the `function` definition, the nonterminal symbol `mathExpression` is used to describe mathematical expressions. Such a symbol is defined in the CFG_MATH_EXPRESSION.g4 file. Moreover, grammar allows combining conditions (defined in the nonterminal symbols `ifThenCondition` or `ifCondition`) with function statements. These nonterminal symbols are defined in the CFG_BOOLEAN_EXPRESSION.g4 file.

As part of the function definitions, assignments can be defined. To this end, the grammatical model includes a set of productions used to identify such statements as follows:

```
assignIdentifier: variable ASSIGNOPERATOR identifier;
```

```
assignParameter: variable ASSIGNOPERATOR parameter;
```

```
assignValue: variable ASSIGNOPERATOR value;
```

Each nonterminal symbol is used for a specific purpose to ensure a proper definition of model elements. The `ASSIGNOPERATOR` token (from the CFG_TOKENS.g4 file) is used to denote the operation. The `variable`, `identifier`, `parameter`, `variableTuple`, and `tupleValue` are special elements of the grammar defined in a unique form in the CFG_TOKENS.g4 file.

CFG_SET_THEORY. The CFG_SET_THEORY.g4 file contains productions used to denote a set in an implicit (nonterminal symbol `setImplicitDefinition`) or explicit (symbol `setExplicitDefinition`) way. In practice, the `setExplicitDefinition` involves a `setSymbol` and a `setImplicitDefinition`. Hence, we explain here the `setImplicitDefinition` definition defined as follows:

```
setImplicitDefinition: setDefinition | setOperationResult;
```

As the previous rule shows, we consider a set can be defined "statically" in a `setDefiniton` or, instead, be the result of a set operation (`setOperationResult`). For the first case, we consider both `elementCollection` and `tupleCollection`. The `elementCollection` syntax diagram can be seen here.

For operations returning a new set, we provide a set of nonterminal symbols to identify sentence structures for several set operations such as cartesian product, union, intersection, and difference. The precedence of operators in an expression is given following mathematical foundations. Other basic operations involving sets are also allowed in the grammar (e.g., belongs to).

CFG_MATH_EXPRESSION. The file contains production rules used for all nonterminal symbols defining a mathematical expression (`mathExpression`). Available operations to define complex expressions are addition, subtraction, multiplication, division, square root, and power. As in the set theory production rules, the precedence of operators in an expression follows mathematical foundations. A partial syntax diagram of the `mathExpression` can be seen here.

CFG_BOOLEAN_EXPRESSION. As in the previous case, the file contains all the production rules defined for identifying Boolean expressions (nonterminal symbol `booleanExpression`) as follows:

```
booleanExpression: boolExpressionTerm | binaryCondition;
```

```
binaryCondition: boolExpressionTerm (binaryConditionalOperator boolExpressionTerm)*;
```

```
boolExpressionTerm: relationalExpressionTerm | unaryCondition;
```

We consider that a `booleanExpression` can be a term (`boolExpressionTerm`) or the result of a well-defined condition (defined over a `binaryCondition`). A Boolean term is defined as a `relationalExpressionTerm` (based on relational operators such as “equal to”, “greater than”, “less than or equal to”, and so on) or as a `unaryCondition`. On the other hand, a `binaryCondition` is structured using `boolExpressionTerm` combined with (binary) conditional operators. The operators included in the grammar are “and” and “or”. Again, the precedence in an expression follows mathematical foundations.

Finally, the file contains two symbols for identifying conditional structures based on if conditionals:

```
ifCondition: ifSymbol booleanExpression | ifSymbol BEGINPARENTHESES booleanExpression
ENDPARENTHESES | inOtherCaseSymbol;
```

```
ifThenCondition: ifSymbol booleanExpression thenSymbol | ifSymbol BEGINPARENTHESES
booleanExpression ENDPARENTHESES thenSymbol | inOtherCaseSymbol;
```

These structures combined with the `booleanExpression` allow describing complex functions as part of DEVS models.

CFG_TOKENS. The file contains all tokens used in the grammar. These tokens include keywords (either symbols or operators) and special characters (e.g., ‘(’, ‘)’, ‘{’, ‘}’, and so on). Keywords start with ‘\’ and follow with the string naming its purpose. For example, the token `EMPTYSETSYMBOL` is defined as the string “`\emptySet`” while the token `ASSIGNOPERATOR` is defined as “`\assign`”. We decided to use such a naming convention to get a complete identification of sentences (including keywords). If a modeler uses the “real” CFG, he should write models following this naming convention. However, intelligent editors can be developed based on the use of macros (i.e., identifiers associated with token strings) associated with keywords. Once the macros are ready to be used, preprocessing can be performed to substitute the token string for each occurrence of the identifier in the text. In this way, for example, the empty set symbol ∅ (or the LaTex notation \emptyset) will be able to be used in the model definition and (when preprocessing the text) been replaced with the string “`\emptySet`”.

The definition of basic elements is also included in the CFG_TOKENS.g4 file. Each element type is defined using a naming convention as follows:

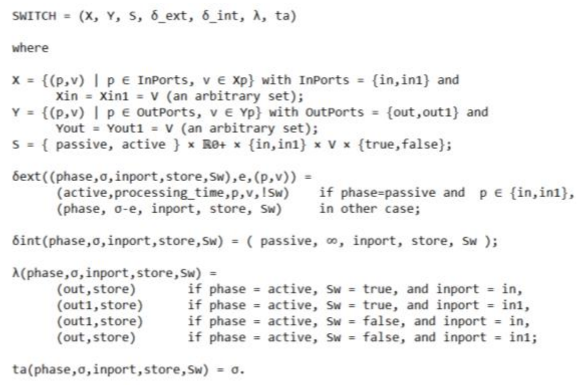
```
modelSymbol: UPPERCASEID | LOWERCASEID | FULLCASEID;

variable: LOWERCASEID APOSTROPHE;

parameter: HASH LOWERCASEID;

identifier: LOWERCASEID;
```

For the `modelSymbol`, any string that starts with a letter and follows with a combination of letters/numbers can be used. For the `variable`, `parameter`, and `identifier`, lowercase letters are mandatory. In these cases, the naming requires an extra symbol to distinguish between them.

3.3 Rewriting Models from the Literature Using CFG_DEVS

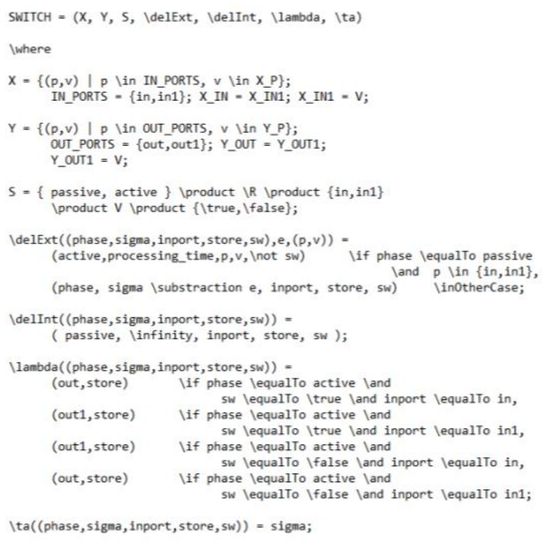
Our first example is “the Switch” proposed in (Zeigler et al. 2018). Figure 1 shows the original definition of the atomic model, while Figure 2 presents the model rewritten using CFG_DEVS. The parse tree obtained when parsing the specification of Figure 2 using the CFG_DEVS can be seen here.

To show how different CFG_DEVS notations work, we use “the Worker model” defined by Goldstein et al. (2013). Instead of defining function results as tuples, such an atomic model details values for state variables using conditional expressions. Figure 3 presents the formal specification. Figure 4 shows one possible way of writing such a model in our grammar. The parse tree of Figure 4 can be consulted here.



```
SWITCH = (X, Y, S, δ_ext, δ_int, λ, ta)

where

X = {(p,v) | p ∈ InPorts, v ∈ Xp} with InPorts = {in,in1} and
      Xin = Xin1 = V (an arbitrary set);
Y = {(p,v) | p ∈ OutPorts, v ∈ Yp} with OutPorts = {out,out1} and
      Yout = Yout1 = V (an arbitrary set);
S = { passive, active } × ℝ0+ × {in,in1} × V × {true,false};

δext((phase,σ,inport,store,Sw),e,(p,v)) =
      (active,processing_time,p,v,!Sw)    if phase=passive and  p ∈ {in,in1},
      (phase, σ-e, inport, store, Sw)     in other case;

δint(phase,σ,inport,store,Sw) = ( passive, ∞, inport, store, Sw );

λ(phase,σ,inport,store,Sw) =
      (out,store)       if phase = active, Sw = true, and inport = in,
      (out1,store)      if phase = active, Sw = true, and inport = in1,
      (out1,store)      if phase = active, Sw = false, and inport = in,
      (out,store)       if phase = active, Sw = false, and inport = in1;

ta(phase,σ,inport,store,Sw) = σ.
```

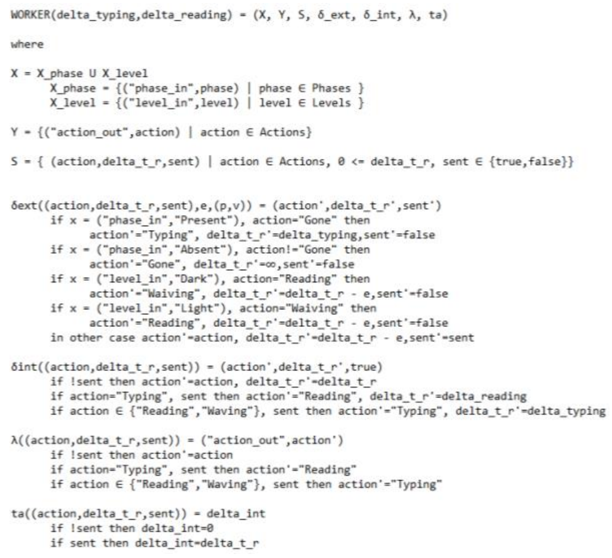
Figure 1:“The Switch ” (Zeigler et al. 2018).



```
SWITCH = (X, Y, S, \delExt, \delInt, \lambda, \ta)

\where

X = {(p,v) | p \in IN_PORTS, v \in X_P};
      IN_PORTS = {in,in1}; X_IN = X_IN1; X_IN1 = V;

Y = {(p,v) | p \in OUT_PORTS, v \in Y_P};
      OUT_PORTS = {out,out1}; Y_OUT = Y_OUT1;
      Y_OUT1 = V;

S = { passive, active } \product \R \product {in,in1}
      \product V \product {\true,\false};

\delExt((phase,sigma,inport,store,sw),e,(p,v)) =
      (active,processing_time,p,v,\not sw)      \if phase \equalTo passive
                                                   \and  p \in {in,in1},
      (phase, sigma \substraction e, inport, store, sw)     \inOtherCase;

\delInt((phase,sigma,inport,store,sw)) =
      ( passive, \infinity, inport, store, sw );

\lambda((phase,sigma,inport,store,sw)) =
      (out,store)       \if phase \equalTo active \and
                              sw \equalTo \true \and inport \equalTo in,
      (out1,store)      \if phase \equalTo active \and
                              sw \equalTo \true \and inport \equalTo in1,
      (out1,store)      \if phase \equalTo active \and
                              sw \equalTo \false \and inport \equalTo in,
      (out,store)       \if phase \equalTo active \and
                              sw \equalTo \false \and inport \equalTo in1;

\ta((phase,sigma,inport,store,sw)) = sigma;
```

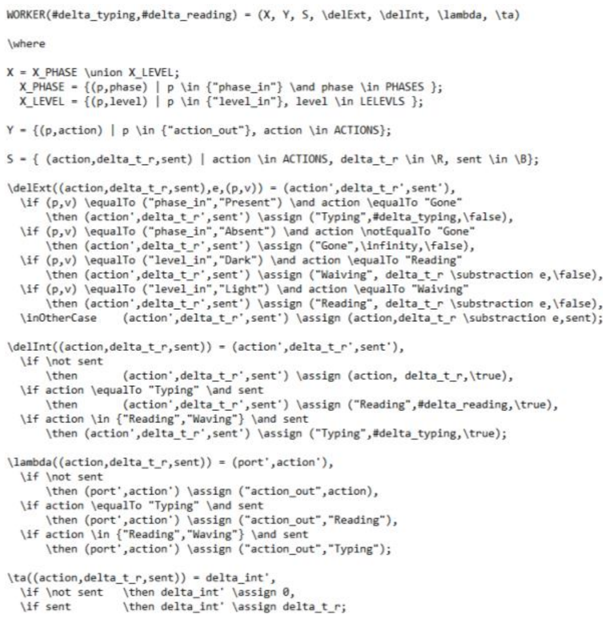
Figure 2: CFG_DEVS specification of Figure 1.

A larger atomic model is “the Hummingbird Feeder” (Zeigler and Sarjoughian 2003). Such a model employs parameters (i.e., time_feed, time_refill, time_query) and special conditions surrounding function statements (for space reasons, see the DEVS formal model here). Figure 5 presents the CFG_DEVS specification of this atomic model. The parse tree can be seen here.



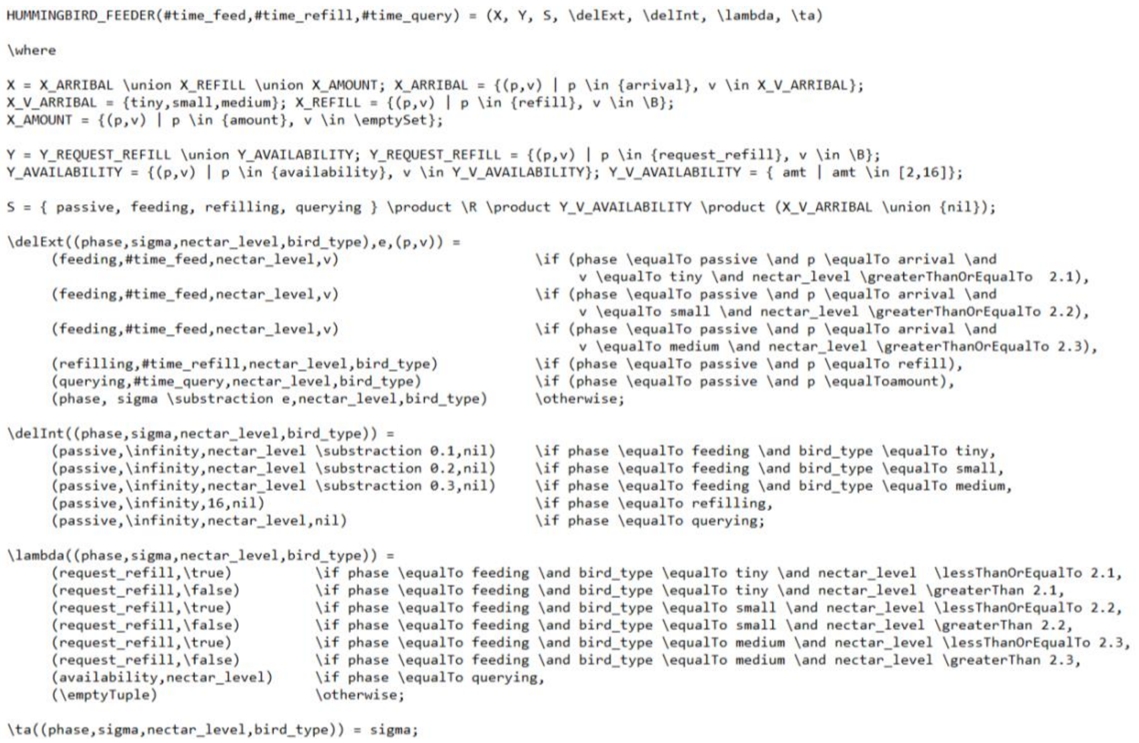
```
WORKER(delta_typing,delta_reading) = (X, Y, S, δ_ext, δ_int, λ, ta)

where

X = X_phase U X_level
      X_phase = {("phase_in",phase) | phase ∈ Phases }
      X_level = {("level_in",level) | level ∈ Levels }

Y = {("action_out",action) | action ∈ Actions}

S = { (action,delta_t_r,sent) | action ∈ Actions, 0 <= delta_t_r, sent ∈ {true,false}}


δext((action,delta_t_r,sent),e,(p,v)) = (action',delta_t_r',sent')
      if x = ("phase_in","Present"), action="Gone" then
            action'="Typing", delta_t_r'=delta_typing,sent'=false
      if x = ("phase_in","Absent"), action!="Gone" then
            action'="Gone", delta_t_r'=∞,sent'=false
      if x = ("level_in","Dark"), action="Reading" then
            action'="Waiving", delta_t_r'=delta_t_r - e,sent'=false
      if x = ("level_in","Light"), action="Waiving" then
            action'="Reading", delta_t_r'=delta_t_r - e,sent'=false
      in other case action'=action, delta_t_r'=delta_t_r - e,sent'=sent

δint((action,delta_t_r,sent)) = (action',delta_t_r',true)
      if !sent then action'=action, delta_t_r'=delta_t_r
      if action="Typing", sent then action'="Reading", delta_t_r'=delta_reading
      if action ∈ {"Reading","Waving"}, sent then action'="Typing", delta_t_r'=delta_typing

λ((action,delta_t_r,sent)) = ("action_out",action')
      if !sent then action'=action
      if action="Typing", sent then action'="Reading"
      if action ∈ {"Reading","Waving"}, sent then action'="Typing"

ta((action,delta_t_r,sent)) = delta_int
      if !sent then delta_int=0
      if sent then delta_int=delta_t_r
```

Figure 3:“The Worker” (Goldstein et al. 2013).



```
WORKER(#delta_typing,#delta_reading) = (X, Y, S, \delExt, \delInt, \lambda, \ta)

\where

X = X_PHASE \union X_LEVEL;
  X_PHASE = {(p,phase) | p \in {"phase_in"} \and phase \in PHASES };
  X_LEVEL = {(p,level) | p \in {"level_in"}, level \in LELEVLS };

Y = {(p,action) | p \in {"action_out"}, action \in ACTIONS};

S = { (action,delta_t_r,sent) | action \in ACTIONS, delta_t_r \in \R, sent \in \B};

\delExt((action,delta_t_r,sent),e,(p,v)) = (action',delta_t_r',sent'),
  \if (p,v) \equalTo ("phase_in","Present") \and action \equalTo "Gone"
      \then (action',delta_t_r',sent') \assign ("Typing",#delta_typing,\false),
  \if (p,v) \equalTo ("phase_in","Absent") \and action \notEqualTo "Gone"
      \then (action',delta_t_r',sent') \assign ("Gone",\infinity,\false),
  \if (p,v) \equalTo ("level_in","Dark") \and action \equalTo "Reading"
      \then (action',delta_t_r',sent') \assign ("Waiving", delta_t_r \substraction e,\false),
  \if (p,v) \equalTo ("level_in","Light") \and action \equalTo "Waiving"
      \then (action',delta_t_r',sent') \assign ("Reading", delta_t_r \substraction e,\false),
  \inOtherCase    (action',delta_t_r',sent') \assign (action,delta_t_r \substraction e,sent);

\delInt((action,delta_t_r,sent)) = (action',delta_t_r',sent'),
  \if \not sent
      \then       (action',delta_t_r',sent') \assign (action, delta_t_r,\true),
  \if action \equalTo "Typing" \and sent
      \then       (action',delta_t_r',sent') \assign ("Reading",#delta_reading,\true),
  \if action \in {"Reading","Waving"} \and sent
      \then (action',delta_t_r',sent') \assign ("Typing",#delta_typing,\true);

\lambda((action,delta_t_r,sent)) = (port',action'),
  \if \not sent
      \then (port',action') \assign ("action_out",action),
  \if action \equalTo "Typing" \and sent
      \then (port',action') \assign ("action_out","Reading"),
  \if action \in {"Reading","Waving"} \and sent
      \then (port',action') \assign ("action_out","Typing");

\ta((action,delta_t_r,sent)) = delta_int',
  \if \not sent   \then delta_int' \assign 0,
  \if sent        \then delta_int' \assign delta_t_r;
```

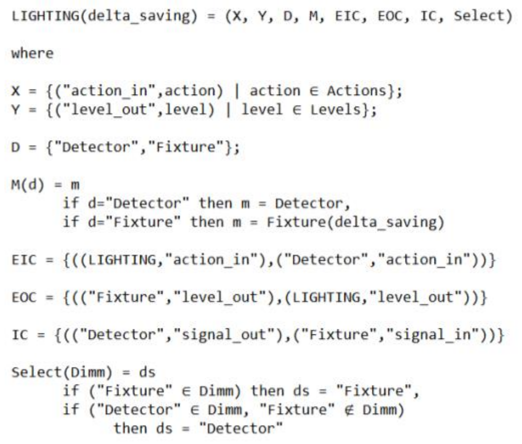
Figure 4: CFG_DEVS specification of Figure 3.



```
HUMMINGBIRD_FEEDER(#time_feed,#time_refill,#time_query) = (X, Y, S, \delExt, \delInt, \lambda, \ta)

\where

X = X_ARRIBAL \union X_REFILL \union X_AMOUNT; X_ARRIBAL = {(p,v) | p \in {arrival}, v \in X_V_ARRIBAL};
X_V_ARRIBAL = {tiny,small,medium}; X_REFILL = {(p,v) | p \in {refill}, v \in \B};
X_AMOUNT = {(p,v) | p \in {amount}, v \in \emptySet};

Y = Y_REQUEST_REFILL \union Y_AVAILABILITY; Y_REQUEST_REFILL = {(p,v) | p \in {request_refill}, v \in \B};
Y_AVAILABILITY = {(p,v) | p \in {availability}, v \in Y_V_AVAILABILITY}; Y_V_AVAILABILITY = { amt | amt \in [2,16]};

S = { passive, feeding, refilling, querying } \product \R \product Y_V_AVAILABILITY \product (X_V_ARRIBAL \union {nil});

\delExt((phase,sigma,nectar_level,bird_type),e,(p,v)) =
     (feeding,#time_feed,nectar_level,v)                  \if (phase \equalTo passive \and p \equalTo arrival \and
                                                               v \equalTo tiny \and nectar_level \greaterThanOrEqualTo  2.1),
     (feeding,#time_feed,nectar_level,v)                  \if (phase \equalTo passive \and p \equalTo arrival \and
                                                               v \equalTo small \and nectar_level \greaterThanOrEqualTo 2.2),
     (feeding,#time_feed,nectar_level,v)                  \if (phase \equalTo passive \and p \equalTo arrival \and
                                                               v \equalTo medium \and nectar_level \greaterThanOrEqualTo 2.3),
     (refilling,#time_refill,nectar_level,bird_type)      \if (phase \equalTo passive \and p \equalTo refill),
     (querying,#time_query,nectar_level,bird_type)        \if (phase \equalTo passive \and p \equalToamount),
     (phase, sigma \substraction e,nectar_level,bird_type)     \otherwise;

\delInt((phase,sigma,nectar_level,bird_type)) =
     (passive,\infinity,nectar_level \substraction 0.1,nil)    \if phase \equalTo feeding \and bird_type \equalTo tiny,
     (passive,\infinity,nectar_level \substraction 0.2,nil)    \if phase \equalTo feeding \and bird_type \equalTo small,
     (passive,\infinity,nectar_level \substraction 0.3,nil)    \if phase \equalTo feeding \and bird_type \equalTo medium,
     (passive,\infinity,16,nil)                                \if phase \equalTo refilling,
     (passive,\infinity,nectar_level,nil)                      \if phase \equalTo querying;

\lambda((phase,sigma,nectar_level,bird_type)) =
     (request_refill,\true)         \if phase \equalTo feeding \and bird_type \equalTo tiny \and nectar_level  \lessThanOrEqualTo 2.1,
     (request_refill,\false)        \if phase \equalTo feeding \and bird_type \equalTo tiny \and nectar_level \greaterThan 2.1,
     (request_refill,\true)         \if phase \equalTo feeding \and bird_type \equalTo small \and nectar_level \lessThanOrEqualTo 2.2,
     (request_refill,\false)        \if phase \equalTo feeding \and bird_type \equalTo small \and nectar_level \greaterThan 2.2,
     (request_refill,\true)         \if phase \equalTo feeding \and bird_type \equalTo medium \and nectar_level \lessThanOrEqualTo 2.3,
     (request_refill,\false)        \if phase \equalTo feeding \and bird_type \equalTo medium \and nectar_level \greaterThan 2.3,
     (availability,nectar_level)    \if phase \equalTo querying,
     (\emptyTuple)                  \otherwise;

\ta((phase,sigma,nectar_level,bird_type)) = sigma;
```

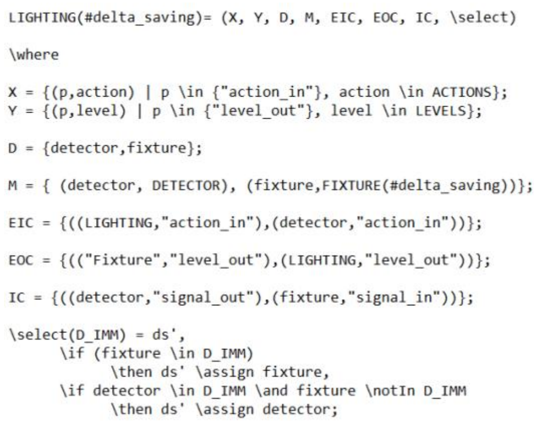
Figure 5: The CFG_DEVS specification for “the Hummingbird Feeder” (Zeigler and Sarjoughian 2003).

As an example of DEVS coupled models, we present "the Lighting model" proposed in (Goldstein et al. 2013). Figure 6 shows the formal specification. Figure 7 presents the model expressed in the CFG_DEVS notation. The parser tree can be consulted here. As the reader can appreciate, in all cases both definitions are quite similar except for the use of keywords instead of symbols.



```
LIGHTING(delta_saving) = (X, Y, D, M, EIC, EOC, IC, Select)

where

X = {("action_in",action) | action ∈ Actions};
Y = {("level_out",level) | level ∈ Levels};

D = {"Detector","Fixture"};

M(d) = m
       if d="Detector" then m = Detector,
       if d="Fixture" then m = Fixture(delta_saving)

EIC = {((LIGHTING,"action_in"),("Detector","action_in"))}

EOC = {(("Fixture","level_out"),(LIGHTING,"level_out"))}

IC = {(("Detector","signal_out"),("Fixture","signal_in"))}

Select(Dimm) = ds
       if ("Fixture" ∈ Dimm) then ds = "Fixture",
       if ("Detector" ∈ Dimm, "Fixture" ∉ Dimm)
              then ds = "Detector"
```

Figure 6: "The Lighting model" (Goldstein et al. 2013).



```
LIGHTING(#delta_saving)= (X, Y, D, M, EIC, EOC, IC, \select)

\where

X = {(p,action) | p \in {"action_in"}, action \in ACTIONS};
Y = {(p,level) | p \in {"level_out"}, level \in LEVELS};

D = {detector,fixture};

M = { (detector, DETECTOR), (fixture,FIXTURE(#delta_saving))};

EIC = {((LIGHTING,"action_in"),(detector,"action_in"))};

EOC = {(("Fixture","level_out"),(LIGHTING,"level_out"))};

IC = {((detector,"signal_out"),(fixture,"signal_in"))};

\select(D_IMM) = ds',
       \if (fixture \in D_IMM)
              \then ds' \assign fixture,
       \if detector \in D_IMM \and fixture \notIn D_IMM
              \then ds' \assign detector;
```

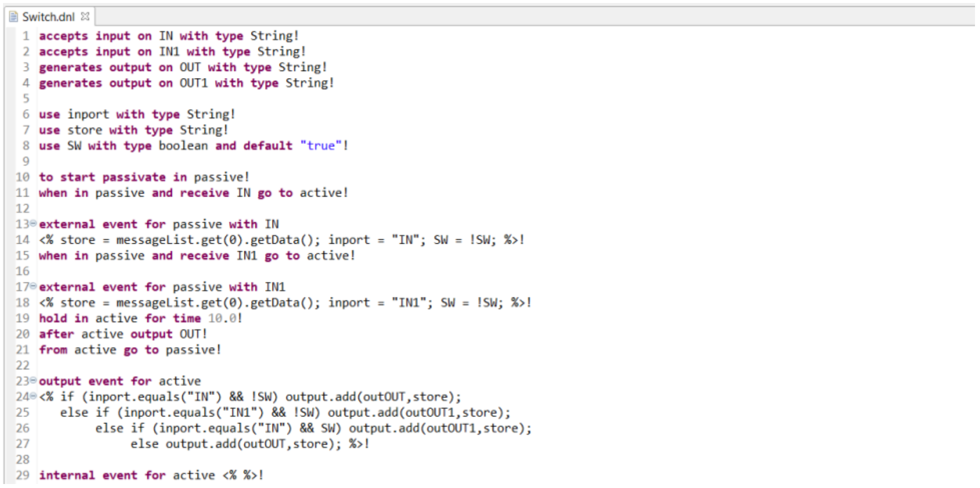
Figure 7: The CFG_DEVS specification for the model detailed in Figure 6.

4 DISCUSSION

Frequently, the DEVS formalization task is left aside due to distinct reasons (e.g., time constraints, good programming skills, and so on) - that is due to the development of DEVS formal models being a time-consuming task that delays the delivery of operational model solutions. However, when introducing DEVS formalism to students (or even in literature), DEVS formal specifications are used. Then, the students (depending on their programming skills) should get the equivalent model implementations that allow model execution. So far, existing grammar implementations are focused on describing DEVS models using other modeling perspectives (such as proving interoperability or using natural language). Moreover, they generate FD-DEVS models (i.e., a restricted subset of DEVS).

Our grammatical model allows defining the unrestricted set of DEVS formal models that can be processed in a computational form to denote well-defined sentences. As examples presented in Section 3.3 show, all definitions are almost identical to the "natural" specification of DEVS (formal) models. Moreover, parse trees show our ANTLR implementation can recognize members of the Classic DEVS with ports structure without any trouble. That enables a further matching among these members and metamodel concepts. Both are advantages of our notation. The latter is about the development of a new modeling language, while the first is regarding our grammar over existing ones.

Take as an example Figure 8 which shows "the Switch" (i.e., a DEVS atomic model) defined in DEVSNL using MS4 Me (MS4 Me 2023). Such a model definition is quite different from the model detailed in Figure 1 (i.e., the model formal specification). But it is also closer to a "natural language" definition (we use quotes because, as stated in Section 2.2, it is not based on natural language processing). That makes sense because the modeling perspective used as a guide in DEVSNL was focused on getting natural language expressions for defining DEVS models. Most likely, natural language expressions are useful for describing DEVS models. Still, for books and teaching processes, the use requires knowing the DEVSNL language structure and how DEVS specifications are translated into such a language. Both alternatives (Figures 2 and 8) are suitable. Still, when using CFG_DEVS, the modeler (students or non-DEVS experts in the context described) only must type the DEVS specification model in an editor (does not need to "translate" the formal modeling perspective to a new form of structured sentences). In this way, CFG_DEVS enables a suitable introduction to DEVS for students.



Switch.dnl

```
accepts input on IN with type String!
accepts input on IN1 with type String!
generates output on OUT with type String!
generates output on OUT1 with type String!

use inport with type String!
use store with type String!
use SW with type boolean and default "true"!

to start passivate in passive!
when in passive and receive IN go to active!

external event for passive with IN
<% store = messageList.get(0).getData(); inport = "IN"; SW = !SW; %>!
when in passive and receive IN1 go to active!

external event for passive with IN1
<% store = messageList.get(0).getData(); inport = "IN1"; SW = !SW; %>!
hold in active for time 10.0!
after active output OUT!
from active go to passive!

output event for active
<% if (inport.equals("IN") && !SW) output.add(outOUT,store);
   else if (inport.equals("IN1") && !SW) output.add(outOUT1,store);
        else if (inport.equals("IN") && SW) output.add(outOUT1,store);
             else output.add(outOUT,store); %>!

internal event for active <% %>!
```

Figure 8: The DEVSNL specification for the model of Figure 1 (“The Switch” (Zeigler et al. 2018)).

Building a software tool as an editor of the CGF_DEVS grammar is crucial. So far, we are parsing model specifications using ANTLR directly. ANTLR implementations can be embedded in other platforms to use parsing capabilities of a defined grammar in distinct environments. Unifying such a parser in an editor with the metamodel (already implemented in Ecore) to instantiate well-defined DEVS formal models is part of the development in progress. Both activities are part of our current joint work with RTSync.

Like the case of DescribeML (Giner 2022), describing models in a structured format based on a textual notation promotes the development of other semi-automated formal scenarios that are not possible in programming language implementations, such as, for example, generating artifacts from a model specification, such as documentation regarding formal definitions and hierarchical traceability. Moreover, using a well-defined modeling language with a semantic meaning enables the translation between different DEVS implementations.

4.1 About the Quality of the CFG_DEVS Notation

Textual notations should follow basic quality principles. Most of these properties are relevant when the concrete syntax is not something brand new (i.e., it uses icons or keywords that users may already associate asemantic with). The analysis performed by Cabot (2022) focuses on how the quality of a good mathematical notation can be applied to evaluate the quality of modeling language. From such an analysis, we can say that the CFG_DEVS achieves the following features: *i) Preservation of quality* (every “natural” concept in the abstract syntax should be easy to express in the notation): Each element defined as a concept in the metamodel is represented by a production; *ii) Error correction/detection* (typos in a well-formed expression should create an expression that can be easily corrected -or at least detected- to recover the original intended meaning -or a small perturbation thereof-): We promote a set of production rules that allow parsing sentence elements detecting pitfalls in expressions; and *iii) Suggestiveness* (the calculus of formal manipulation in the language should resemble the calculus of formal manipulation in other languages that users of the language are already familiar with): The grammar is defined just as the mathematical form of DEVS models.

Other features such as *unambiguity* (every well-formed expression in the concrete notation should correspond to a single instantiation of the abstract syntax), *expressiveness* (every abstract syntax instantiation should be describable in at least one way using the concrete notation), and *transformation* (“natural” transformation of concepts in the abstract syntax should correspond to “natural” manipulation of their symbolic counterparts in the concrete syntax notation) cannot be discussed at this point.

As a remark, we still have not evaluated the “costs” of the notation defined. Such costs are classified into two types (Cabot 2022): “one-time costs” (e.g., the difficulty of learning the notation and avoiding standard pitfalls with that notation) and “recurring costs” (costs incurred with each use of the notation). So far, modelers using the CFG_DEVS notation are primarily related to the development team.

5 CONCLUSIONS AND FUTURE WORK

Formalization and implementation of DEVS models are two activities that are often carried out independently of each other. While formalization deals with the definition of a model over the basis of a formal specification, implementation deals with developing a concrete model (from the formal definition) in a programming language. That is one key difference of DEVS models with other types of formal models, for example, models from the operation research field (where modelers are focused on improving the mathematical definition to obtain better and faster results in the equivalent implementation).

To get a new modeling language to support the definition of DEVS models in a computational form, we have presented a CFG as textual notation for DEVS formal models. Such a grammatical model can be combined with “the DEVS metamodel” to define a new modeling language with computational support. Since the definition of the concrete syntax is a determining factor in the usability of a DSML, we have followed the mathematical expressions used in the DEVS specification. Examples presented in Section 3.3 show how CFG_DEVS grammar can help to learn DEVS allowing students to define DEVS formal models in a computer form without needing to learn programming or other languages. These examples follow the syntactical rules defined in the CGF_DEVS model identifying members of Classic DEVS with ports in a parse tree. In this way, we have shown how the grammatical model allows describing DEVS formal models using mathematical notions. Such a definition allows for building a textual specification of DEVS models that can be verified to ensure syntactical correctness. We are working now on building the syntactical mapping of the concrete syntax (presented in this paper) with the abstract syntax (i.e., the metamodel) in a formal well-defined function.

Regarding semantics, the future work is devoted to completing the language definition by including a formal specification of the semantics. Having semantics explicitly defined makes different interpretations impossible, enabling the possibility of integration among different DEVS implementations.

REFERENCES

ANTLR. 2023. ANother Tool for Language Recognition (ANTLR). https://www.antlr.org/, accessed 8th April.

Barisic, A., V. Amaral, M. Goulao, and B. Barroca. 2011. “Quality in Use of Domain-specific Languages: A Case Study”. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, edited by S. Markstrum, E. Murphy-Hill, and C. Anslow, 65–72. New York, NY: Association for Computing Machinery.

Baar, T. 2006. “Correctly Defined Concrete Syntax for Visual Modeling Languages”. In *Proceeding of the 2006 Model Driven Engineering Languages and Systems Conference*, edited by O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, 111-125. Berlin: Springer Berlin Heidelberg.

Blas, M. J., and S. Gonnet. 2023. “Modeling and Simulation Through the Metamodeling Perspective: The Case of the Discrete Event System Specification”. In *Handbook of Model-Based Systems Engineering*, edited by A.M. Madni, N. Augustine, and M. Sievers, https://doi.org/10.1007/978-3-030-27486-3_86-1. Cham: Springer.

Blas, M., S. Gonnet, and B. Zeigler. 2021. “Towards a Universal Representation of DEVS: A Metamodel-Based Definition of DEVS Formal Specification”. In *Proceedings of the 2021 Annual Modeling and Simulation Conference*, edited by C. Ruiz-Martin, M. Blas, and A. Inostrosa-Psijas, 1-12. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Cabot, J. 2022. *Qualities of a Good Notation – A Mathematician Perspective*. Modeling Languages. https://modeling-languages.com/qualities-notation-dsl-mathematician-perspective/, accessed 6th October 2022.

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

Crystal, D. 2008. *A Dictionary of Linguistics and Phonetics*. 6th ed. Malden: Wiley-Blackwell.

Engelen, L., and M. Van Den Brand. 2010. “Integrating Textual and Graphical Modelling Languages”. *Electronic Notes in Theoretical Computer Science* 253(7):105-120.

Giner, J. 2022. *DescribeML: A Tool for Describing Machine Learning Datasets*. Modeling Languages. https://modeling-languages.com/describeml-machine-learning-datasets/, accessed 12th October 2022.

Goldstein, R., G. Wainer, A. Khan. 2013. “The DEVS Formalism”. In *Formal Languages for Computer Simulation: Transdisciplinary Models and Applications*, edited by P. Fonseca and I. Casas, 62-102. Pennsylvania: IGI Global.

Harel, D., and B. Rumpe. 2004. “Meaningful Modeling: What’s the Semantics of “Semantics”?”. *Computer* 37(10):64–72.

Hopcroft, J., R. Motwani, and J. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Boston: Addison Wesley.

Kelly, S., and J. P. Tolvanen. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. 1st ed. New York: John Wiley & Sons Inc.

Kleppe, A. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. 1st ed. New York: Pearson Education.

Krahn, H., B. Rumpe, and S. Völkel. 2007. "Integrated Definition of Abstract and Concrete Syntax for Textual Languages". In Proceedings of the 2007 Model Driven Engineering Languages and Systems Conference, edited by G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, 286-300. Berlin: Springer Berlin Heidelberg.

Lando, P., A. Lapujade, G. Kassel, and F. Fürst. 2007. "Towards A General Ontology of Computer Programs", In *Proceedings of the Second International Conference on Software and Data Technologies*, edited by J. Filipe, B. Shishkov, and M. Helfert, 163-170. Portugal: Institute for Systems and Technologies of Information, Control and Communication Press.

Mittal, S., and S. A. Douglas. 2012. "DEVSML 2.0: The language and the stack.". In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, edited by H. ElAarag and A. Tolk, 17. New York: Association for Computing Machinery.

MS4 Me. 2023. MS4 Systems. http://www.ms4systems.com/pages/main.php, accessed 18th April.

Paige, R. F., J. S. Ostroff, and P. J. Brooke. 2000. "Principles for Modeling Language Design". *Information and Software Technology* 42(10):665-675.

Petre, M. 1995. "Why Looking isn't Always Seeing: Readership Skills and Graphical Programming". *Communications of the ACM* 38(6):33–44.

Rumbaugh, J., I. Jacobson, and G. Booch. 2005. *The Unified Modeling Language Reference Manual*. 2nd ed. India: Pearson Education.

Sarjoughian, H. S., A. Alshareef, and 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, 2788-2799. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Seuren, P. A. 2015. "Prestructuralist and Structuralist Approaches to Syntax". In *Syntax-Theory and Analysis: An International Handbook*, edited by T. Kiss and A., 134-157. Berlin: Mouton de Gruyter.

Van Mierlo, S., H. Vangheluwe, and J. Denil. 2019. "The Fundamentals of Domain-Specific Simulation Language Engineering". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y. Son, 1482-1494. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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

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, and A. Tolk, 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.

Zeigler, B. P., and H. S. Sarjoughian. 2003. "Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-based Simulation Models". Technical Document, University of Arizona.

Zeigler, B. P., and H. S. Sarjoughian. 2017. "DEVS Natural Language Models and Elaborations". In *Guide to Modeling and Simulation of Systems of Systems*, edited by B. Zeigler and H. Sarjoughian, 39-62. London: Springer.

AUTHOR BIOGRAPHIES

MARIA JULIA BLAS is an Assistant Researcher at INGAR and an Assistant Professor at UTN. She received her Ph.D. degree in Engineering from UTN in 2019. Her research interests include UML modeling and 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.

DOOHWAN KIM is the founder and president of RTSync Corp., which specializes in Predictive Analytics and Model-Based System Engineering based on DEVS M&S technology. Dr. Kim has been involved in the design, development, and delivery of the advanced M&S solutions for highly complex real world information science and engineering problems. He received his Ph.D. degree from the University of Arizona in 1996. His email address is dhkim@rtsync.com.

BERNARD 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 email address is zeigler@rtsync.com.