# Translating DEVS conceptual models into simulation models

Diego A. Hollmann[a,*], Maximiliano Cristiá[a,b], Claudia Frydman[a,c]

[a]*CIFASIS*
*Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas*
*Ocampo y Esmeralda, S2000EZP Rosario, Argentina.*
[b]*FCEIA - UNR*
*Rosario, Argentina.*
[c]*Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296,13397*
*Marseille, France.*

**Abstract**

DEVS models are widely used in the research community, in the industry and even in military or defense departments. Therefore, several software tools exist for modeling and simulating these models. However, each of these tools has its specific input language and a DEVS model described within a particular framework cannot be simulated by a different one. Moreover, the practitioners willing to use one of these tools must have non-trivial programming skills or must ask to a programmer to translate their models into the language of the desired tool. In this paper, we present CML-DEVS, a language that allows the conceptual, abstract or mathematical representation of DEVS models, in terms of mathematical and logical expressions without involving programming issues. Models described with CML-DEVS can be automatically translated (i.e. compiled) into the input language of different modeling and simulation tools. We also present a set of rules to translate CML-DEVS models into models that can be simulated with well-known DEVS frameworks such as DEVS-Suite and PowerDEVS. These rules allow the implementation of a multi-target compiler.

*Keywords:* Conceptual Model, Abstract Model, DEVS, Discrete Event Simulation, Abstract Language

## 1. Introduction

Modeling and simulation (M&S) has grown in the last years becoming a discipline by itself. It is used by many different communities, in a wide range of application domains from academia to industry and defense departments.

One of the most popular formalism among all these M&S communities is DEVS (Discrete EVent System Specification) [1], being the most general formalism to describe discrete event

*Corresponding author
*Email addresses:* `hollmann@cifasis-conicet.gov.ar` (Diego A. Hollmann), `cristia@cifasis-conicet.gov.ar` (Maximiliano Cristiá), `claudia.frydman@lsis.org` (Claudia Frydman)

systems (DES). DEVS is an abstract formalism for model specification and simulation that is independent of any particular implementation. It is based on system theory and is expressive enough as to represent all other DES formalisms, i.e. all models representable in those formalisms can be represented in DEVS [2].

Nowadays, many DEVS tools were developed and are currently used with very good results (for instance, DEVS-C++ [3], DEVSim++ [4], CD++ [5], PowerDEVS [6], JDEVS [7], DEVS-Suite [8], LSIS-DME [9]). Each of these tools has its own input language based on some programming language, or use some sentences in a programming language, or is, simply, a programming language like Java, C or C++. Therefore, those practitioners who want to simulate their models using such tools must have, at least, basic programming skills or they need a programmer to implement their models.

Generally, each of these input languages are different, hindering the interoperability between simulation tools. For instance, a simulation model described using PowerDEVS cannot be simulated in JDEVS (or at least not in a straightforward way). On the other hand, the model described in PowerDEVS models essentially the same system that would be modeled in JDEVS. Hence, it would be desirable to be able to describe the abstract model as is seen by the expert and then automatically translate it to one or more input languages.

The contribution of this paper is a a formal modeling language, CML-DEVS (Conceptual Modeling Language for DEVS), that allows the conceptual or abstract description of DEVS models in terms of logical and mathematical expressions without involving programming concepts. That is, engineers can use CML-DEVS to write DEVS models using every-day mathematics. CML-DEVS models, however, can be automatically parsed, analyzed and translated into the input languages of different M&S tools to simulate them with any of these tools. This approach is widely used in the formal method community (cf. Z [10], B [11], Alloy [12]) where formal specifications are written in classic mathematics and logic in such a way that these specifications can be directly used in tools such as type-checkers, theorem provers, code-generators, etc.

Figure 1 depicts an overview of the modeling and simulation process using CML-DEVS. First, the model is expressed in its abstract form by the practitioner using CML-DEVS. Then, using a multi-target compiler, the model is translated into the proper input language of different simulation tools in order to simulate it with the desired M&S framework. Commonly, during the model validation process, for example via simulations [13], the original model may need to be modified. In any case, these modifications must be done over the abstract model, which is recompiled before continuing with the validation process. Anyhow, the practitioner avoids modifying the source code of the translated models.

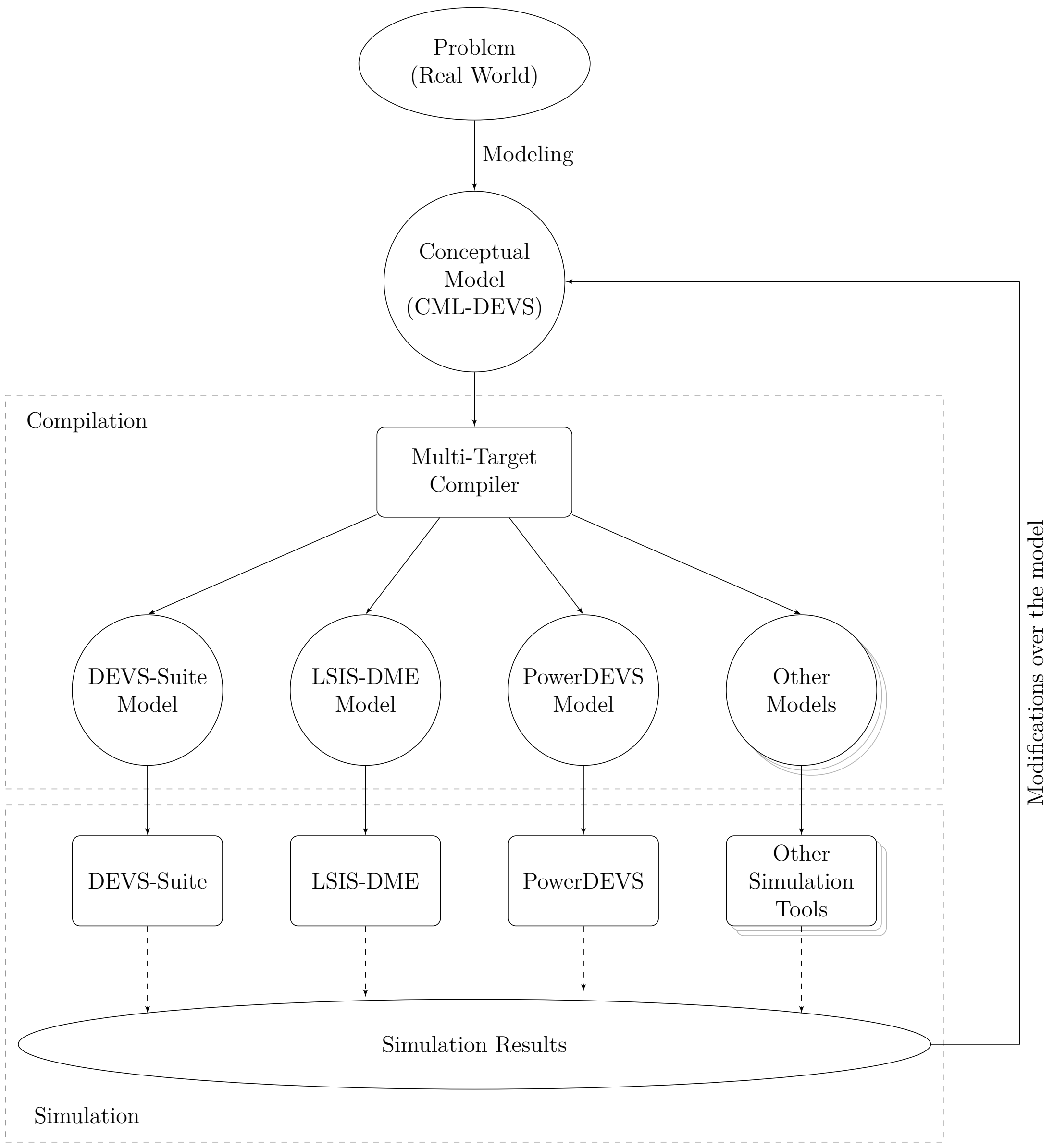

Figure 1: DEVS Modeling & Simulation process

CML-DEVS is described by its syntax and semantics. The former is given by means of examples and its EBNF (Extended Backus Naur Form) while the latter is given in terms of the code that must be generated in the input language of two M&S tools, DEVS-Suite and PowerDEVS. In this way a CML-DEVS multi-target compiler can be built to produce DEVS-Suite and PowerDEVS models from abstract or mathematical DEVS models. To extend CML-DEVS to other target languages, it is only necessary to provide the corresponding set of translation rules. The choice of these particular tools is relatively arbitrary. Since these tools have different input languages (C++ and Java) the intention is to show how a CML-DEVS model can be automatically translated into different simulators languages.

The remainder of this paper is organized as follows. In the following section we describe and comment some other approaches about the description of DEVS models. In Section 3 the DEVS formalism is briefly described. Section 4 presents CML-DEVS showing its purpose with several examples. Section 5 introduces the set of rules to translate a CML-DEVS model into a DEVS-Suite and a PowerDEVS model. In Section 6 we very briefly comment two case studies. Finally, in Section 7 we discuss some conclusions and further consideration.

## 2. Related work and other approaches

DEVS is a powerful and widely used formalism in the M&S community. Several simulation tools were independently developed to describe DEVS models requiring a great effort. Because of this, an international group composed by researchers from several universities around the world, has been created with the goal of defining a standard for modeling and simulating system with DEVS [14].

This group has identified four areas of the DEVS framework that needs to be standardized [15, 16]:

- the DEVS formalism itself, and its variants,
- model representation, which should describe the model structure and dynamics in a platform-independent manner,
- minimum requirements for a simulator to be labeled "DEVS-compliant",
- model libraries, aimed at providing a collection of models usable out of the box.

CML-DEVS attacks the second issue, representing DEVS models in an abstract way and independently of any platform or programming language.

We see CML-DEVS as an extended or improved version of the specification language for DEVS models, DEVSpecL, developed by Hong and Kim [17]. As DEVSpecL, CML-DEVS aims to preserve the abstract concept of the DEVS formalism. However our proposal adds new abstract mathematical notions to DEVSpecL such as a set theory which is essential to describe abstract models. As Hong and Kim mention, DEVSpecL supports only basic features which need user defined APIs to be able to describe general models. The modeler should define these APIs in the programing language or environment in which the model is executed. Furthermore, regarding the *complex types* supported by DEVSpecL, they only include sequences (but actually used as stacks). These considerations limit the abstraction level of DEVSpecL. Moreover, the modeler still has to write programming code to make something complex. CML-DEVS is inspired by the formal notations used in software engineering such as Z, B or Alloy, adding mathematical theories fundamental to model systems.

In addition, we consider that models defined with CML-DEVS has a more "pure DEVS" style compared to DEVSpecL. For instance, with DEVSpecL the modeler should define message types and interfaces to represent input/output events, that may contain general purpose programming language code or structural types. With CML-DEVS we intend to keep input/output events in an abstract way just defining the port values types.

Moreover, the article that presents DEVSpecL lacks sufficient details in some aspects. For instance, the external and internal transition functions consist of sequences of expressions (`expr`), that are not further explained except for a few examples. Also, no reference to the elapsed time, $e$, is made in the external transition function.

Finally, in this paper we present the translation rules that allows the implementation of a multi-target compiler for the input language of two different simulators. This does not seem to have be presented by Hong and Kim.

Mittal and Douglass [18] present a domain specific language, based on Finite Deterministic DEVS (FDDEVS) [19], for Parallel DEVS as a component of the revised DEVSML framework (DEVSML 2.0). It also intends to be used as an abstract representation of DEVS models (among other things). To define DEVS models with some fancy features like code assistance and run-time model validation, they integrate into Eclipse a DEVSML editor using the Xtext framework and the EBNF grammar of their language. However, the DEVSML grammar has some differences and limitations with respect to Parallel DEVS. For instance, input/output values are defined as *message entities*. The elapsed time in the external transition is omitted and replaced by *continue* and *reschedule* sentences. Finally, FDDEVS is a constrained, less expressive subset of DEVS.

Both works described above, would allow automatic code generation, in order to get executable DEVS code, in different DEVS implementations, like DEVSJAVA, DEVSim++, DEVSim-Java.

Several works propose XML as a language to describe DEVS models [20, 21, 16, 22]. According to Touraille et al [16] it seems to be a good choice, since there are many XML tools and it is platform independent. However, XML by itself nor these specific proposals provide an abstract representation of DEVS models neither. Perhaps, XML can be a suitable option as an intermediate representation, i.e. between the abstract description of the model and its representation in some simulation tool.

In a recent work, framed in his PhD thesis, L. Touraille [23] developed a framework aimed to model systems with DEVS. The core of this framework is a meta-model for DEVS. It is virtually separated in two parts, one to specify the "static" part of the model, i.e. states, inputs/outputs; and the other to define the behavior, namely, the internal and external transition functions as well as the time advance and output functions. The latter is based in a semi-generic language defined by him which only supports a restricted set of features. To allow the modeler to use more advanced features, he give the possibility to embed some "un-generic" code, whose actual content will depend on the target platform. Based on such meta-model he developed translations into several DEVS platforms allowing the interoperability between several simulation tools. This transformation is not made from an abstract description of a DEVS model, but from, precisely, this meta-model. This approach is related to model driven engineering, while ours comes from formal methods.

In summary, there are many languages that can be used for describing DEVS models. However, as far as we know, none of them takes as starting point the abstract description (based on mathematics and logic) of a DEVS model. Some of these approaches require at some point the introduction of programming code.

## 3. The DEVS formalism

As mentioned in the introduction, DEVS is a formalism independent of any particular implementation. According to Zeigler et al [1], there are two classes of DEVS models, *Atomic* models and *Coupled* models. Our work concerns only with atomic models. In fact, a coupled model is equivalent to a (complex) atomic model [1]. An atomic DEVS Model is defined by the structure:

$$M = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where:

- $X = \{(p, v) \mid p \in \mathit{InPorts}, v \in X_p\}$ is the set of input ports and values, where *InPorts* represents the set of input ports and $X_p$, the set of values for the input ports;
- $Y = \{(p, v) \mid p \in \mathit{OutPorts}, v \in Y_p\}$ is the set of output ports and values, where *OutPorts* represents the set of output ports and $Y_p$, the set of values for the output ports;
- $S$ is the set of state values;
- $\delta_{int} : S \to S$ is the internal transition function;
- $\delta_{ext} : Q \times X \to S$ is the external transition function, where:
  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set, and
  $e$ is the *time elapsed* since last transition;
- $\lambda : S \to Y$ is the output function; and
- $ta : S \to \mathbb{R}^+_{0,\infty}$ is the time advance function.

$\delta_{int}$, $\delta_{ext}$, $\lambda$ and $ta$ are the functions that define the system dynamics. Each possible state $s \in S$ is associated to a time advance value, $ta(s) \in \mathbb{R}^+_{0,\infty}$, which indicates the time that the system will remain in that state if no input events occur. Once that time has passed, an internal transition is performed, reaching a new state $s'$, $s' = \delta_{int}(s)$. At the same time, an output event, $y$, is generated by the output function, $y = \lambda(s)$. Therefore, $\delta_{int}$, $ta$ and $\lambda$ define the autonomous behavior of the system.

When an input event arrives, an external transition is performed. The new state depends on the input value, the previous state and also the elapsed time since the last transition. If the system is in the state $s$ and the input event $x$ arrives in the instant $e$ (i.e. $e$ time units from the last transition) the new state, $s'$, is calculated as $s' = \delta_{ext}(s, e, x)$. In case of an external transition, no output event is generated.

Let us see briefly how to describe a model using the DEVS formalism with two little examples, a processor and a queue. These examples were taken from the book were B. Zeigler presents the formalism and, therefore, is the original form to describe DEVS models. It can be understood by anyone without programming knowledge or skills and it is independent of any simulator.

**Example 3.1.** Processor

This model represents a processor that receives jobs that take a given time to be processed. The input value received by the model represents the time that takes the processor to process the incoming job. If the processor is busy when a new job arrives, it is ignored. Once the job is processed, the system issues an output event with the time it took to process it.

$$M_{proc} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

- $X = \{(in, x) : x \in \mathbb{R}^+\}$
- $Y = \{(out, y) : y \in \mathbb{R}^+\}$
- $S = \{idle, busy\} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \mathbb{R}^+$
- $\delta_{int}((phase, \sigma, job)) = (idle, \infty, job)$
- $\delta_{ext}((phase, \sigma, job), e, (port, value)) = \begin{cases} (busy, value, value) & \text{if } phase = idle \\ (phase, \sigma - e, job) & \text{otherwise} \end{cases}$
- $\lambda((phase, \sigma, job)) = (out, job)$
- $ta((phase, \sigma, job)) = \sigma$

The set of ports and values for the input events consist of a single port, *in* and of values from $\mathbb{R}^+$, representing the processing time of the incoming job. The output set has the same interpretation, respectively, with the port *out*. The state consists of three variables[1]. The first represents the state of the processor (*idle* or *busy*); the second one, the remaining processing time of the current job ($\infty$ if there is no job being processed); and the third one, the the total processing time of the current job. If a job arrives when the processor is *idle*, the remaining processing time is set according to the input value, *value*, and the phase is changed. Instead, if the phase is *busy*, the job is just ignored and the elapsed processing time is updated. When the processing time finishes, this value is issued as an output. In the corresponding internal transition the phase is changed to *idle* and the remaining processing time is set to to $\infty$; in this case no further transition will occur unless a new job arrives.

**Example 3.2.** Queue

The queue consists on a list that stores jobs IDs (elements from $\mathbb{R}^+$), as they arrive. When a signal is received, instead of a job, the queue transmits the first job of the list (i.e. the first to arrive), if any, and it is removed from the queue.

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

- $X = \{(in, x) : x \in \mathbb{R}^+ \cup \{signal\}\}$
- $Y = \{(out, y) : y \in \mathbb{R}^+ \cup \{\emptyset\}\}$

[1]Or consists of one variable with three components

- $S = (\text{List } \mathbb{R}^+) \times (\mathbb{R}_0^+ \cup \{\infty\})$
- $\delta_{int}((xs, \sigma)) = \begin{cases} (\text{tail } xs, \infty) & \text{if } xs \neq \emptyset \\ (xs, \infty) & \text{otherwise} \end{cases}$
- $\delta_{ext}((xs, \sigma), e, (port, value)) = \begin{cases} (xs \frown \langle value \rangle, \infty) & \text{if } value \in \mathbb{R} \\ (xs, 0) & \text{if } value = signal \end{cases}$
- $\lambda((xs, \sigma)) = \begin{cases} (out, \text{head } xs) & \text{if } xs \neq \emptyset \\ (out, \emptyset) & \text{otherwise} \end{cases}$
- $ta((xs, \sigma)) = \sigma$

We think, that at this point, this example needs no further explanations than saying that List $\mathbb{R}^+$ is a list of real positive numbers; $xs \frown \langle value \rangle$ concatenates the element *value* at the end of *xs*; tail *xs* represents all except the first element of the list *xs* and head *xs*, the first element of *xs*.

## 4. Writing abstract DEVS models with CML-DEVS

As mentioned before, the contribution of this work is a modeling language that allows the description of abstract DEVS models, similarly as the examples shown above, without using programming notions (such as control flow, memory management, data structures, etc).

In Figure 2 we show how the model of the processing queue could be described using CML-DEVS. In this example can be observed, roughly, how is the structure of a CML-DEVS model. It consists of several substructures or components, each one representing a component of an atomic DEVS model. Each component is framed by *markers* or *key words*, like X is ... end X for the input ports. The state variables and input and output ports are defined in the same way, each within the corresponding structure. Variables are declared by giving a name and its type (e.g. $s : \mathbb{R} \cup \{\emptyset\}$). Transition, output and time advance functions, have also similar structures. The body of these functions consist of different type of sentences. Mainly, these sentences are assignments to update the state value or to issue an event on an output port. As can be seen in this example, there are a bit more complex sentences, like the definition by cases (defcases ... end defcases). The definition by cases have a structure very similar to the abstract model of the example, where the result of the function (the new state or the output value, respectively) are bound to a condition.

If a practitioner wants to simulate one of the example models of Section 3, even one so simple, it will involve some programming task. For instance, in Example 3.2 a decision must be made about the representation of $X$, $Y$ and $S$, regarding how to represent the symbol $\emptyset$ and *signal*. Maybe, in this particular case, with a `float` or `double` variable in some language will suffice for $X$ and $Y$ using, say, a negative number to represent *signal* and $\emptyset$ respectively. However, in a slightly different case, for instance $\mathbb{R} \cup \{\emptyset\}$, this representation would not work and another one must be defined. In addition, a data structure is needed

```
atomic $queue$ is $<$ X, Y, S, $\delta$int, $\delta$ext, $\lambda$, ta $>$ where
X is
    $in : \mathbb{R} \cup \{signal\};$
end X
Y is
    $out : \mathbb{R} \cup \{\emptyset\};$
end Y
S is
    $s$ : List $\mathbb{R} \times$ Time;
end S
$\delta$int$((xs, \sigma))$ is
    defcases
    case $s =$ (tail $xs, \infty$); if $(xs \neq \{\})$
    case $s = (xs, \infty)$; if $(xs = \{\})$
    end defcases
end $\delta$int
$\delta$ext$((xs, \sigma), e, (in, x))$ is
    defcases
    case $s = (xs \frown \langle x \rangle, \infty)$; if $(x \in \mathbb{R})$
    case $s = (xs, 0)$; if $(x = signal)$
    end defcases
end $\delta$ext
$\lambda((xs, \sigma))$ is
    defcases
    case $out =$ head $xs$; if $(xs \neq \{\})$
    case $out = \emptyset$; if $(x = \{\})$
    end defcases
end $\lambda$
ta$((xs, \sigma))$ is
    $\sigma;$
end ta
end atomic
```

Figure 2: CML-DEVS model of the queue

in order to manage the elements of the queue. Using CML-DEVS, the practitioner does not need to deal with these issues, he or she can describe the model in its abstract form and later automatically translate it into the language of the desire simulation tool.

Besides the structures that can be observed in Figure 2 CML-DEVS allows the definition of *parameters* of the model, *auxiliary functions* and provides an easy method to set the initial the state for the simulation. These features will be detailed later.

As can be seen, CML-DEVS is intended to be as close as the specialist would describe a model "with pen and paper", like the examples of Section 3. Even more, a CML-DEVS model can be easily formatted to look like those examples. This could be done, for instance, using a language like LaTeX[24] or using a *formula editor*.

As in any language there are *reserved* words. In the queue example they can be distinguished from variables or values because they appear in another font type, like atomic.

In the following sections, we explain in detail the syntax of CML-DEVS. Moreover, the EBNF of the language can be seen in an on-line technical report: http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS/TechReport.pdf

## *4.1. Structure of an atomic model*

As mentioned above, an atomic model in CML-DEVS consists of several substructures related to each component of an atomic DEVS model. The order in which these structures are declared is not relevant. Furthermore, it is not necessary to declare them if they are not used. That is, if a model does not receive any input event, the declaration of $\mathsf{X}$ and $\delta\mathsf{ext}$ can be omitted. If these are not defined, neither should be declared in the vector defining the model: $< \mathsf{X}, \mathsf{Y}, \mathsf{S}, \delta\mathsf{int}, \delta\mathsf{ext}, \lambda, \mathsf{ta} >$. That is, all components declared in that vector, must be defined later.

## *4.2. Definitions and types*

The definition of the state and the input and output ports have the same syntax as shown before in the Example 3.2. In Figure 3 different definitions are presented. These can be either state variables, input ports or output ports, depending on the structure within which they are declared.

$jobs : \mathsf{List}\ \mathbb{R};$
$\sigma : \mathsf{Time};$
$in : \mathbb{R} \cup \{\mathsf{signal}\};$
$out : \mathbb{R} \cup \{\emptyset\};$
$elevator : \{up, down, stopped\};$
$sensor : \mathsf{Boolean};$
$name : \mathsf{Text};$
$years : \mathbb{P}\ \mathbb{N};$
$pair : \mathbb{Z} \times \mathbb{R};$
$sch : \mathbb{Z} \nrightarrow \{idle, exec, finished\};$

Figure 3: CML-DEVS - State, Input Ports and Output Ports

Regarding the input and outputs events, each definition represents a port (input and output, accordingly) and the types or set of the values issued by them. The input and output sets are formed by the Cartesian product of those defined ports and values.

CML-DEVS provides several *basic* types and allows the construction of new *complex* ones like tuples, unions, sets, lists, partial functions and binary relations, as shown above. The basic types are: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathsf{Time}$ (equivalent to $\mathbb{R}_0^+ \cup \{\emptyset\}$), $\mathsf{Text}$, $\mathsf{Boolean}$ and enumerated types expressed within braces, $\{\ \}$.

$\mathbb{P}$, $\mathsf{List}$, $\cup$, $\times$, $\nrightarrow$ and $\leftrightarrow$ allow the construction of complex types from basic or other complex types. $\mathbb{P}$ *Type* represents an unsorted set of elements of type *Type* (without repeated elements). $\mathsf{List}$ *Type*, in turn, is a sorted list of elements of type *Type* (lists may have repeated elements). $\cup$ and $\times$ construct unions and tuples, respectively, of possibly different types (in fact, unions of the same type would make no sense). Finally, $\nrightarrow$ and $\leftrightarrow$ construct partial functions and binary relation, respectively.

CML-DEVS also allows the definition of *synonyms*. They are used to simplify or facilitate the variable declaration, or even the declaration of new synonyms. For example, if some complex type is used several times, the practitioner can name this complex type with some synonym and then use the latter in the declaration of variables. For instance:

$var1 : \mathbb{N} \times \mathbb{R};$
$var2 : (\mathbb{N} \times \mathbb{R}) \times \mathsf{Boolean};$
$var3 : \mathsf{List}\ (\mathbb{N} \times \mathbb{R}) \cup \{\emptyset\};$

Could be replaced by:

$NRPair == \mathbb{N} \times \mathbb{R};$
$var1 : NRPair;$
$var2 : NRPair \times \mathsf{Boolean};$
$var3 : \mathsf{List}\ NRPair \cup \{\emptyset\};$

It is worth mentioning the importance of sets, partial functions and binary relations for the abstract or conceptual modeling. For instance, let us think in the model of a simple scheduler that change the state of a scheduled process according with some signals or values. The state of the scheduler could be:

$sch : \ \mathbb{N} \nrightarrow \{idle, exec, finished\}$

where $\mathbb{N}$ represents the process identifiers and $\{idle, exec, finished\}$, the different process states. Changing the state of a specific process, *proc*, to *idle* could be done easily (using the *overwriting* operator) as follows:

$sch = sch \oplus \{(proc, idle)\};$

However, had the scheduler been modeled with a list, like:

$sch : \mathsf{List}(\mathbb{N} \times \{idle, exec, finished\})$

that simple operation would involve a longer expression:

```
$schTmp : \mathsf{List}(\mathbb{N} \times \{idle, exec, finished\})$;
foreach $p$ in $sch$
    defcases
        if $(p.1 = proc) \Rightarrow schTmp \frown \langle (p.1, idle) \rangle$;
        default $\Rightarrow schTmp \frown \langle p \rangle$;
    end defcases
end foreach
$sch = schTmp$;
```

Hence, in many situations modeling with sets can yield simpler, more readable models than using programming-oriented data structures such as lists. In general this is so because many entities in the "real world" are essentially sets rather than lists.

## *4.3. Transition, output and time advance functions*

The description of the transition functions, and also the time advance and output functions, have the same main structure. An overview of these were presented in the previous section with the example. These function definitions consist of a frame, for example $\delta$int is ...end $\delta$int for the internal transition function, with optional parameters. These parameters, are indeed, the parameters of such functions, i.e. $(s, e, x)$ in $\delta_{ext}$ and $(s)$ in $\delta_{int}$, $\lambda$ and $ta$.

If the parameters are explicitly declared, this must be done in the order the state variables were declared in the *state* section. However, the names used as parameters may not necessarily be the same as those previously defined:

```
S $is$
    $var : \mathbb{R} \times \mathbb{R}$;
end S
$\vdots$
$\delta$int$((var1, var2))$ $is$
    $var1 = \ldots$
    $var2 = \ldots$
end $\delta$int
```

where $var1$ ($var2$) corresponds to the first (second) component of $var$.

This is also the case of the Example 2, where the state is defined by the variable $s$: List $\mathbb{R} \times$ Time, and then, the transition, output and time advance functions, instead of using $s$ as parameter, use the pair $(xs, \sigma)$.

The last two parameters of $\delta_{ext}$, in case they are explicitly defined, can have any arbitrary name. However, the modeler can use the *reserved* CML-DEVS names $\mathsf{e}$, $\mathsf{value}$ and $\mathsf{port}$:

```
$\delta\mathsf{ext}(s, \mathsf{e}, (\mathsf{port}, \mathsf{value}))$ $is$
    $s = \mathsf{value};$
    $\ldots$
$\mathsf{end}\ \delta\mathsf{ext}$
```

If in any function the parameters are not present, the variables declared in the *state* are implicitly used together with the reserved variables $\mathsf{e}$, $\mathsf{port}$ and $\mathsf{value}$:

```
$\mathsf{S}$ $is$
    $var1 : \mathbb{R};$
    $var2 : \mathbb{N};$
$\mathsf{end\ S}$
$\vdots$
$\delta\mathsf{ext}$ $is$
    $var1 = \mathsf{value} + \mathsf{e};$
    $var2 = 0;$
$\mathsf{end}\ \delta\mathsf{ext}$
```

The body of each function is composed by *sentences*. There are four kinds of sentences in CML-DEVS.

*Assignment.* The simplest one is the *assignment* sentence. It is used, mainly, to update the state variables after a transition of the model, like $xs = \mathsf{tail}\ xs$; or to issue a value by an output port, $out = \mathsf{head}\ xs$. The left hand side of an assignment is always a variable name. The right hand side, instead, can be any expression (a value, a variable name, an operation or compound expressions like tuples, sets or lists). Some example of assignments can be observed in Figure 4.

```
$engine = stopped;$
$years = \{1974, 1988, 1990, 1991, 1992, 2004, 2013\};$
$pair = (-14, 3.1416);$
$val = \mathsf{sin}(45);$
$jobs = \langle 2.34, 5.98, 6.83 \rangle;$
$elem = \mathsf{head}\ xs;$
$\sigma = \sigma - \mathsf{e};$
$out = \emptyset;$
$listA = listB;$
```

Figure 4: CML-DEVS - Assignments

CML-DEVS supports a variety of mathematical operations (including list and set operations), mathematical and trigonometric functions and the application of *custom* functions defined using CML-DEVS. The operators and mathematical functions supported are listed in Table 1.

| | | | |
|---|---|---|---|
| $+$ | $-$ | $*$ | $\setminus$ |
| $\cup$ | $\cap$ | $\#$ | $\frown$ |
| sin | cos | tan | arcsin |
| arccos | arctan | log | sign |
| min | max | sqrt | rev |
| head | last | tail | front |
| div | dom | ran | $\lhd$ |
| $\rhd$ | $\oplus$ | | |

Table 1: CML-DEVS - Operators and mathematical functions

*Definition by cases.* The second kind of sentence is a structure that allows the definition of a function by cases. This is widely used and usually a very useful way to define the transition, output or time advance functions. In Example 3.2, $\delta_{int}$, $\delta_{ext}$ and $\lambda$ were defined in such a way.

For instance, an internal transition function defined by cases in DEVS as follows:

$$\delta_{int}(s) = \begin{cases} sentence_1 & \text{if } condition_1 \\ sentence_2 & \text{if } condition_2 \\ \dots & \\ sentence_n & \text{otherwise} \end{cases}$$

can be described with CML-DEVS as:

```
$\delta$int$(s)$ $is$
defcases
    if $condition_1 \Rightarrow sentence_1$;
    if $condition_2 \Rightarrow sentence_2$;
    $\dots$
    default $\Rightarrow sentence_n$;
end defcases
end $\delta$int
```

All cases are framed by defcases ... end defcases. A condition consists of an *relational operator* between two *operands*. The operators supported by CML-DEVS are listed in Table 2. Meanwhile, the operands can be a variable name, a value or an operation. Moreover, conditions can be conjoined, disjointed or negated using the logical operators $\wedge$, $\vee$ and $\neg$ , respectively. Figure 5 shows some examples of conditions. Note that the operator $=$ is overloaded, i.e. it is used in assignments and in comparisons with different meanings.

|  |  |  |  |
|---|---|---|---|
| $<$ | $>$ | $\leq$ | $\geq$ |
| $=$ | $\neq$ | $\in$ | $\notin$ |
| $\subset$ | $\subseteq$ |  |  |

Table 2: CML-DEVS - Relational operators

$\neg\,(engine = stopped)$
$(years \neq \{\})$
$(pair.1 \leq 13)$
$((5.63 \in jobs) \wedge (4.13 \notin jobs))$
$((\{1974, 1990\} \subseteq years) \vee (\#years = 1))$
$(\mathsf{value} = signal)$

Figure 5: CML-DEVS - Condition Examples

The above form to describe cases, is inspired in logical constructions: *if condition then expression* (or *condition* $\Rightarrow$ *expression*). Meanwhile, CML-DEVS provides an alternative way to define such cases, more similar to the mathematical definitions, enunciating the result of the case first, and then the condition (as in the examples of Section 3). In this alternative, each case is defined within the structure case *sentences* if *condition* end case.

The optional command, default, can be used for the case where none of the conditions is satisfied. This is followed only by the *sentences*, without the command if, $\Rightarrow$ nor the *conditions*.

*For each structure.* The third type of sentence aims at modeling expressions such as:

$$\forall\, x \in X \bullet X = (X \setminus \{x\}) \cup \{x * 2\}$$

The above expression can be depicted in CML-DEVS as follows:

```
foreach $x$ in $X$
    $X = (X \setminus \{x\}) \cup \{x * 2\};$
end foreach
```

It is assumed that the type of $x$ is *Type* and that of $X$ is $\mathbb{P}$ *Type*.

These sentences consist of the foreach declaration followed by an identifier, the keyword in, a variable or expression, followed by a set of sentences and finish with end foreach. The type of the expression or variable must be List *Type* or $\mathbb{P}$ *Type*; and the identifier represents an element of that expression, thus its type is *Type*.

*Where structure.* The other usual way to define a function of a model is using auxiliary variables or expressions. For instance:

$\delta_{ext}((xs, \sigma), e, (port, value)) = (ys, \infty)$
where:
$ys = xs \frown \langle value \rangle$

The fourth and last type of CML-DEVS sentence intends to provide such declarations. In Figure 6 is depicted the corresponding CML-DEVS code for the above example. This structure is framed by defwere and end defwhere and consists of sentences and definitions. The definitions and sentences that come after where refer to the auxiliary variables.

```
δext((xs, σ), e, (port, value)) is
defwhere
    xs = ys;
where
    ys : List ℝ;
    ys = xs ⌒ ⟨value⟩;
end defwhere
```

Figure 6: CML-DEVS - Where Example

Recall that the *for each* structure, the definition by cases and the *where* structure can be combined with each other since all three are considered sentences.

A final remark about the transition functions is that if a state variable does not change its value it is not necessary to explicitly declare this situation (for instance $xs = xs;$ ), thus being this statement optional. Variables that do not appear in the left hand side of any assignment are assumed to maintain their values.

The structure of the output function is similar to the transition functions. However, the values of the state variables cannot be modified (because the state can not be modified in the output function). Therefore, the only allowed assignments are those where the the left hand side variable is one of the declared output ports. If necessary, auxiliary variables can be declared (using the defwhere structure). Each assignment over a port represents the production of an output event by that port.

In the time advance function, *ta*, as well as in the output function, the state variables cannot be modified. Again, the structure is similar to the other functions, except for a special feature. The time advance function always returns the value of the built-in variable $\sigma$. Then, there are two options:

(a) the variable $\sigma$ is part of the state of the model or is an argument of *ta*:

```
ta(...) is                  ta(..., σ, ...) is
    σ;                          σ;
end ta                      end ta
```

(b) the variable $\sigma$ has not been used by the modeler as a state variable and, consequently, a value must be assigned inside of *ta*:

```
ta(...) is
    ...
    σ = expresión;
end ta
```

The first alternative was used in the example of the queue because $\sigma$ is not part of the state of the model and is passed as an argument to *ta*.

Note that the built-in variable $\sigma$ can be used, also, as one more model state variable.

## *4.4. Parameters of the model*

Although parameters are not part of the original DEVS formalism, they are used by almost all M&S tools and facilitates the description and maintenance of the models. If the modeler wants to use parameters for a model, he or she must indicate this situation with the sentence (params) after the model name:

```
atomic ModelName(params)  < ··· >  is
```

and later, he or she must define such parameters within the structure:

```
params is
    param1 = val1;
    param2 = val2;
    ...
end params
```

The parameters are not defined as the other variables of the models. Since they are declared with a fixed value (i.e. these values do not change throughout the whole the model) the type is derived from that value. That is, the type is implicit. These parameters can be seen as constant values of the model.

## *4.5. User defined functions*

The definition of auxiliary functions by the modeler is also not part of the formalism originally defined by B. Zeigler neither. However, it is very common and useful to define auxiliary functions to describe the behavior of a part of the model or make some operation. CML-DEVS provides a basic structure to define this type of auxiliary functions. They are all defined within the structure:

```
functions LibraryName is
    ...
end functions
```

The functions defined therein form a *library* and the invocation of such functions is as follows:

$$LibraryName.funcName(\dots);$$

It is worth mentioning that this library can be used by any model which requires any of those functions. That is, the definition of the libraries is independent of the model.

The auxiliary functions are described using almost the same grammar used for the functions of the model. The first line of the body of a function defines its type, i.e. the parameters type and the return value type. This is followed, probably, by the definition of other variables and, later, one or more sentences as described above.

Let us see a how it works with a short example. Consider a kind of filter function that takes a set of integer number, an integer number and returns a set with all the elements of the first parameter that are smaller than the second parameter. Figure 7 shows a possible way to describe this function using CML-DEVS.

```
function $filter$ is
    $S : \mathbb{P}\ \mathbb{N}, n : \mathbb{N} \rightarrow res : \mathbb{P}\ \mathbb{N};$
    foreach $x$ in $S$
        defcases
        if $(x < n) \Rightarrow res = res \cup \{x\};$
        end defcases
    end foreach
end function
```

Figure 7: CML-DEVS - Function example

Note that these functions are defined abstractly using classical mathematical language. Later, they are compiled into a concrete programming language, in the same way that the model that use them.

## *4.6. Initial values for simulations*

CML-DEVS provides, also, an easy way to define initial values for the state variables. The structure is quite simple:

```
simulate $ModelName$ from
    $assignments$
end simulate
```

where the *assignments* have the same form as those already described before and are used, precisely, to assign values to the state variables.

Although this issue does not concern to the modeling phase, but rather to the simulation of the model, we think that it is helpful (and almost mandatory) to provide this alternative in CML-DEVS. As mentioned before, the modeler should not modify the code already translated into the simulation language. Thereby, he or she can set these initial values without dealing with such code.

## 5. Translation of CML-DEVS models into simulation models

The intention of this section is to show that it is possible to automatically translate a CML-DEVS model into different simulation models (in different languages). We think that it is an important contribution for the M&S community to be able to describe abstractly a DEVS model and, then, automatically translate it into the language of the desire simulation tool. As an example, we show how to translate CML-DEVS models into PowerDEVS and DEVS-Suite models.

This translation task is responsibility of the multi-target compiler, shown in Figure 1. The compiler can generate code in different target languages. The abstract syntax tree (AST) generated by the lexical analyzer has all the information about the model in a structured form. Then the AST is processed by a semantic analyzer, responsible for checking the correctness of the model, for example, not attempting to compare a set with text (type checking). Later, once the model is semantically verified, the code for each target language is produced by "specialized" code generators. If one wishes to generate code for a new M&S tool, only needs to add a new code generator. The mult-target compiler can be implemented using basic compiler development techniques. A scheme of a CML-DEVS multi-target compiler is outlined in Figure 8.

### *5.1. Preprocessing*

Before starting with the translation of the CML-DEVS model, must be performed a preprocessing. In this preprocessing the definition of the model is "normalized" according to two criteria. The first one relates to the normalization in the definition of the *Union* types and, the second one, to the definition of complex types in general. A detailed explanation of such preprocessing can be found in the CML-DEVS technical report [25].

### *5.2. Translation rules*

In this section we show and comment, through some examples, how to translate an abstract model written in CML-DEVS into models implemented in DEVS-Suite and PowerDEVS (some translations are shown for DEVS-Suite and others for PowerDEVS). The full description of these rules is in the technical report. As can be noted in that technical report, several rules are defined recursively, or they use other rules, in turn, to carry out the translation.

#### *5.2.1. Model structure*

The first thing to do in order to simulate a model, both in DEVS-Suite and in PowerDEVS is to generate the necessary files, with the structure and code that each tool requires. In the case of DEVS-Suite, a file "ModelName.java" is needed, including in it all the model. This file consists of a Java class representing the model itself. The state variables are declared as

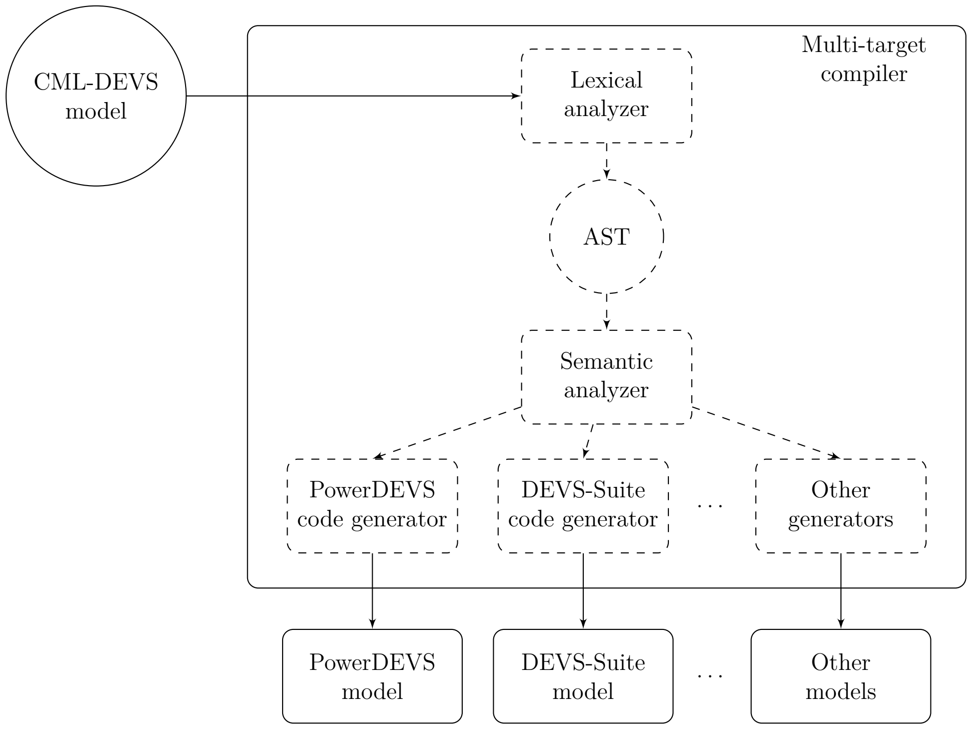

Figure 8: Schema of a CML-DEVS multi-target compiler

class member variables together with the transition, output and time advance functions as class member methods. In the class constructor the variable members are instantiated and the input and output ports are declared. If necessary, DEVS-Suite provides a method for instantiate the state variables.

In the case of PowerDEVS, at least three files are needed. One file with the model structure (path to the model implementation, parameters and port connections), "ModelName.pds". The other two files are the C++ header file, "ModelName.h", and the C++ source code, "ModelName.cpp". In the header file the state variables are defined and the transition, output and time advance functions are declared. Their definitions are in the .cpp file. Furthermore, if the practitioner wants to simulate the model using the GUI of PowerDEVS, one more file is necessary, "ModelName.pdm", including the graphical data (size, colors, position, etc).

### *5.2.2. Definitions*

The *definitions* are used, mainly, to define the state variables. But also, as we mentioned before, CML-DEVS allows the definition of variables in other parts of the model, e.g. in the sentences *where* and in the user defined functions.

The basic types, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, Time, Text and Boolean; are quite simple to translate since for each of these types there is a type or class in Java and in C++ to which it can be translated. Regarding enumerated types, $\{enum_1, \ldots, enum_n\}$, instead of using `enum` of Java and C++ we decided to use `String` and `std::string`, respectively, because this facilitates the inclusion of them in complex types. Sets and Lists are also quite simple since we take advantage of the templates already defined in both languages. Binary relations and partial functions are built from $\mathbb{P}$ and $\times$. Table 3 shows these translations.

| **CML-DEVS** | **DEVS-Suite** | **PowerDEVS** |
|---|---|---|
| *varName*: $\mathbb{N}$; | `Integer varName;` | `unsigned int varName;` |
| *varName*: $\mathbb{Z}$; | `Integer varName;` | `int varName;` |
| *varName*: $\mathbb{R}$; | `Double varName;` | `double varName;` |
| *varName*: Time; | `Double varName;` | `double varName;` |
| *varName*: Boolean; | `Boolean varName;` | `bool varName;` |
| *varName*: Text; | `String varName;` | `std::string varName;` |
| *varName*: $\{\ldots\}$; | `String varName;` | `std:string varName;` |
| *varName*: Set Type; | `Set<Type> varName;` | `std::set<Type> varName;` |
| *varName*: List Type; | `List<Type> varName;` | `std::list<Type> varName;` |

Table 3: Translation of Basic Types, Sets and List

Regarding tuples, unions, partial functions and binary relations, different kinds of classes are defined to represent them. Also, depending on the type, each class needs different methods and constructors that will help in handling them in assignments, comparisons or inclusions. For instance in Figure 9 we show how to translate into Java code (i.e. DEVS-Suite) the following definition of an union:

$$out : \mathbb{R} \cup \{\emptyset\};$$

The method `equals(...)` of Figure 9 is used to compare instances of such type in order to determine whether they are equal or not. In Java, the classes used to translate tuples and unions implement `Comparable<Object>` and this requires to implement the method `compareTo(...)`. Implementing `Comparable<Object>` is necessary to include such unions or tuples in sets or lists.

Concerning the input ports, both in DEVS-Suite and PowerDEVS an extra variable is defined, whose type is the union of the different input port types. This variable is used to handle the input values in the external transition function. Regarding the output ports, an "independent" class is defined for each of them since the values issued by these ports must be accessible from outside of the atomic model.

### *5.2.3. Assignments*

Depending on the variable type or on the right hand side expression, an assignment may involve one or more sentences in the target language. Moreover, some auxiliary functions may be needed. Figure 10 shows the translation of the assignments of Figure 4 into C++ (i.e. PowerDEVS) code.

Besides the *auxiliary functions* `builSet` and `builList`, there are several more that help with the translation. These can be seen in the technical report [25]. Note that these functions are not defined by the modeler, but by the compiler designer.

```
static class T_out extends Entity implements Comparable<Object>{
  public Double v_Number;
  public String v_Enum;
  public String type;
  T_out(){
    v_Number = new Double(0.0);
    v_Enum = new String("");
    type = new String("");
  }
  T_out(T_out other){
    this.v_Number=other.v_Number;
    this.v_Enum=other.v_Enum;
    this.type=other.type;
  }
  T_out(Double v){
    v_Number=v;
    type="Number";
  }
  T_out(String v){
    v_Enum=v;
    type="Enum";
  }
  @Override public boolean equals(Object obj){
    if (obj == this) return true;
    else if (obj==null) return false;
    else if (obj.getClass()!=T_out.class) return false;
    else{
      T_out other = (T_out) obj;
      if (other.type!=this.type) return false;
      else if (this.type=="Number") return (this.v_Number.equals(other.v_Number));
      else return (this.v_Enum.equals(other.v_Enum));
    }
  }
  @Override public int compareTo(Object other){
    return this.equals(other)?0:1;
  }
}
T_out out;
```

Figure 9: Translation of the union *out* : $\mathbb{R} \cup \{\emptyset\}$ from CML-DEVS to DEVS-Suite

```
engine = "stopped";
years = buildSet<int>(7,1974, 1988, 1990, 1991, 1992, 2004, 2013);
pair.v1 = (int)-14;
pair.v2 = (double)3.1416;
val = sin(45);
jobs = buildList<double>(3,2.34, 5.98, 6.83);
elem = (xs).front();
sigma = sigma - e;
out.type = "Enum";
out.v_Enum = "EMPTY";
listA = listB;
```

Figure 10: Translation of CML-DEVS assignments into C++ code

### *5.2.4. "For Each" sentences*

Translating foreach sentences may need some auxiliary variables. Figure 11 shows the translation into Java code of the CML-DEVS foreach sentence of the example in *For each structure* of Section 4.3. Recall that this kind of sentence works only with $\mathbb{P}$ and List.

```
Set<Integer> setTmp = new TreeSet<Integer>(X);
for(Integer x: setTmp){
    X = setDiff(X,buildSet(new Integer(x)));
    X = setUnion(X,buildSet(new Integer(x*2)));
}
```

Figure 11: Translation of a CML-DEVS foreach sentence into Java code

### *5.2.5. "Case" sentences*

The main structure of the case sentences are translated using the classical conditional structures of Java and C++, `if`, `else if` and `else`. Figure 12 shows how to translate into Java the case sentences of the internal transition function of Example 3.2. The key issue is the translation of the conditions. In Figure Figure 13 we show the translation of the example conditions of Figure 5 into C++.

```
if (xs != buildList()){
    s.v1 = xs.subList(1,xs.size()-1);
    s.v2 = INFINITY;
}
else if (xs == buildList()){
    s.v1 = xs;
    s.v2 = INFINITY;
}
```

Figure 12: Translation of CML-DEVS case sentences into Java code

### *5.2.6. "Where" sentences*

The translation of where sentences consists, precisely, in properly rearrange the order of the definitions and sentences in the target language, and perform the corresponding

```
!(engine=="stopped")
year != buildSet<int>()
pair.v1 <= 13
jobs.find((double)5.63)!=jobs.end()
isSubset(buildSet<int>((int)1974), years)
(in.type=="Enum")&&(in.v_Enum=="emitir")
```

Figure 13: Translation of CML-DEVS conditions into C++ code

translations of such definitions and sentences. The translation of the where sentences of Figure 6 into C++ is described in Figure 14.

```
std::list<Double> ys;
ys = listCat(xs,(double)value);
xs = ys;
```

Figure 14: Translation of a CML-DEVS where sentence into C++ code

## *5.3. Post-processing*

Once the model has been translated into either DEVS-Java or PowerDEVS a post-processing must be performed. Each occurrence of a state variable on an assignment or on the right hand side of an assignment, inside the transition function definitions, is replaced by a copy of these variables made at the beginning of the function. This replacement is done because in the internal and external transition functions, these variables may be modified but, their uses in the abstract model refer to their original values.

Suppose that in a model the state has the following representation:

$$S = \mathbb{N} \times \mathbb{R} \times \mathsf{Time}$$

and the internal transition function, $\delta\mathsf{int}$, is as follows:

$$\delta\mathsf{int}((n, r, \sigma)) = (n + 1, r * n, \sigma + n)$$

In this case, in both $r * n$ and in $\sigma + n$ the value of $n$ before performing such transition must be considered, i.e. before $n$ assumes the value $n + 1$. This is performed in two stages:

1. During the main translation step the following code is generated:

   ```
   public void deltint(){
       n = n+1;
       r = r*n;
       sigma = sigma+n;
   }
   ```

2. And during post-processing the code above is modified as follows:

```
public void deltint(){
    modeName prev=new modelName(this);
    n = prev.n+1;
    r = prev.r*prev.n;
    sigma = prev.sigma+prev.n;
}
```

## 6. Case studies

We have used CML-DEVS in two case studies. Both include a DEVS model, its representation with CML-DEVS and its translation into PowerDEVS and DEVS-Suite applying the translation rules presented in this article. Due to of the extension of such case studies (specially the translated models) we do not include them here; they can be found on-line at http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS/models/. One of the models represent the control system of an elevator and the other one a soda can vending machines. These DEVS models make use of all the advanced features of DEVS and some of the operators of set theory. The resulting CML-DEVS descriptions are as long as the original models. In turn, the translated code is considerably larger and complex both in PowerDEVS and DEVS-Suite. In our opinion these case studies show the feasibility of the approach presented in this paper.

## 7. Conclusions and further considerations

We introduced CML-DEVS, a novel modeling language that allows the description of conceptual DEVS models in their abstract form, independent of any platform or implementation. We consider that CML-DEVS is really abstract, specifically on its expressiveness and on its way of defining models. CML-DEVS main purpose is to bring DEVS models specifications closer to formalisms such as Z and B, where set theory is used to specify systems.

The main benefit of this language is that the specialist can define a model without having programming skills or without knowing a particular modeling language of a specific M&S tool. A model described using CML-DEVS looks like a conceptual model, based on mathematics and logic instead of a particular implementation of such model in a M&S tool.

We also present a set of rules to translate CML-DEVS models into PowerDEVS and DEVS-Suite models. This rules can be implemented within a compiler in order to automate these translations. In this way, the practitioner can describe his or her models in an abstract way and, afterwards, simulate them with the desired tool. Although we chose these two particular tools, it can be extended to other M&S tool languages.

The goal of describing DEVS models in their most abstract form, independent of any particular implementation, is twofold: a) it allows the interoperability between practitioners and/or researchers; and b) it facilitates models' maintenance and modifications.

Concerning CML-DEVS syntax and how special symbols like $\mathbb{N}$, $\mathbb{R}$, $\infty$, $\emptyset$ could be handled, there are several alternative solutions. The first one consists in using any word processor, like Write of LibreOffice or Microsoft Word, and use the symbols provided by these tools. Later on, these documents can be easily processed by the compiler. A second option is to write the model in plain text using commands as those of LaTeX or simply define our own commands to refer to those symbols. A third option could be to develop an IDE providing code typing assistance and syntax verification. A graphical environment capable of describing those symbols and showing models in an elegant way can help making user experience more satisfactory and similar to writing models by hand.

As part of our future research, we will develop a CML-DEVS multi-target compiler taking into account all the above considerations. We think that, having already defined the translation rules, it is not a very complex task to create a CML-DEVS IDE.

Future work also involves extending CML-DEVS to characterize variants or extensions of DEVS, like P-DEVS [26], STDEVS [27], Cell-DEVS [28], RT-DEVS [29], VECDEVS [30]; i.e. use CML-DEVS to describe abstractly models in such formalisms. We may consider extending the translation rules of CML-DEVS models into other M&S tool languages.

Although the code generated from automated translations could be different from what expected (concerning complexity and/or optimality), we discourage editing the resulting code. If the practitioners need to make changes to the model, they should modify the CML-DEVS model and re-compile it. Only once the model is properly validated (for instance, using the techniques presented in [13]), the specialist could ask to some developer to make an optimal implementation in the most suitable simulator for such model.

## References

[1] B. P. Zeigler, H. Praehofer, T. G. Kim, Theory of Modeling and Simulation, Second Edition, Academic Press, London, 2000.

[2] B. P. Zeigler, S. Vahie, Devs Formalism And Methodology: Unity Of Conception/Diversity Of Application, in: In Proceedings of the 25th Winter Simulation Conference, ACM Press, 1993, pp. 573–579.

[3] H. J. Cho, Y. K. Cho, DEVS-C++ Reference Guide, The University of Arizona, 1997.

[4] T. G. Kim, DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models., Korea Advance Institute of Science and Technology, 1994.

[5] G. Wainer, CD++: a toolkit to develop DEVS models, Software - Practice and Experience 32 (2002) 1261–1306.

[6] F. Bergero, E. Kofman, PowerDEVS: a tool for hybrid system modeling and real-time simulation, SIMULATION (2010).

[7] J. B. Filippi, M. Delhom, F. Bernardi, The JDEVS Environmental Modeling and Simulation Environment, in: In Proceedings of IEMSS 2002, pp. 283–288.

[8] S. Kim, H. S. Sarjoughian, V. Elamvazhuthi, DEVS-suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring, in: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim '09, Society for Computer Simulation International, San Diego, CA, USA, 2009, pp. 161:1–161:7.

[9] M. E.-A. Hamri, G. Zacharewicz, LSIS-DME: An Environment for Modeling and Simulation of DEVS Specifications, in: AIS-CMS International modeling and simulation multiconference, Buenos Aires, Argentina, pp. 55–60.

[10] J. M. Spivey, The Z notation: a reference manual, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[11] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, NY, USA, 1996.

[12] D. Jackson, Alloy: A logical modelling language, in: ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings, p. 1.

[13] D. A. Hollmann, M. Cristiá, C. Frydman, A family of simulation criteria to guide DEVS models validation rigorously, systematically and semi-automatically, Simulation Modelling Practice and Theory 49 (2014) 1 – 26.

[14] DEVS Standardization Group, http://cell-devs.sce.carleton.ca/devsgroup/, Accessed: 2014-06-18.

[15] H. Vangheluwe, L. Bolduc, E. Posse, DEVS Standardization: some thoughts, in: Winter Simulation Conference.

[16] L. Touraille, M. K. Traoré, D. R. C. Hill, A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models, in: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim '09, Society for Computer Simulation International, San Diego, CA, USA, 2009, pp. 163:1–163:6.

[17] K. J. Hong, T. G. Kim, DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems, Inf. Softw. Technol. 48 (2006) 221–234.

[18] S. Mittal, S. A. Douglass, 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, TMS/DEVS '12, Society for Computer Simulation International, San Diego, CA, USA, 2012, pp. 17:1–17:12.

[19] M. H. Hwang, B. Zeigler, Reachability Graph of Finite and Deterministic DEVS Networks, Automation Science and Engineering, IEEE Transactions on 6 (2009) 468–478.

[20] P. Fishwick, Using XML for simulation modeling, in: Simulation Conference, 2002. Proceedings of the Winter, volume 1, pp. 616–622 vol.1.

[21] M. Rohl, A. Uhrmacher, Flexible integration of XML into modeling and simulation systems, in: Simulation Conference, 2005 Proceedings of the Winter, pp. 8 pp.–.

[22] H. S. Sarjoughian, Y. Chen, Standardizing DEVS Models: An Endogenous Standpoint, in: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, TMS-DEVS '11, Society for Computer Simulation International, San Diego, CA, USA, 2011, pp. 266–273.

[23] L. Touraille, Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation, Ph.D. thesis, Doctoral dissertation, Université d'Auvergne, 2012.

[24] L. Lamport, LaTeX: A Document Preparation System (2nd Edition), Addison-Wesley Professional, 1994.

[25] D. A. Hollmann, CML-DEVS Technical Report. CIFASIS - CONICET, Rosario, Argentina., http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS/TechReport.pdf, 2014.

[26] A. C. Chow, Parallel devs: A parallel, hierarchical, modular modeling formalism and its distributed simulator, TRANSACTIONS of the Society for Computer Simulation 13 (1996) 55–68.

[27] R. Castro, E. Kofman, G. Wainer, A formal framework for stochastic discrete event system specification modeling and simulation, Simulation 86 (2010) 587–611.

[28] G. Wainer, Discrete-events cellular models with explicit delays, Ph.D. thesis, Doctoral dissertation, Université d'Aix-Marseille III, 1998.

[29] J. S. Hong, H.-S. Song, T. G. Kim, K. H. Park, A real-time discrete event system specification formalismfor seamless real-time software development, Discrete Event Dynamic Systems 7 (1997) 355–375.

[30] F. Bergero, E. Kofman, A vectorial devs extension for large scale system modeling and parallel simulation, Simulation 90 (2014) 522–546.