Simulation

Methodology

Simulation: Transactions of the Society for Modeling and Simulation International
1–19

DOI: 10.1177/0037549718765080
journals.sagepub.com/home/sim
SAGE SCS

A multi-target compiler for CML-DEVS

Maximiliano Cristiá[1], Diego A. Hollmann[2] and Claudia Frydman[3]

Abstract

Discrete Event System Specification (DEVS) is a modular and hierarchical formalism for system modeling and simulation. DEVS models can be mathematically described; simulation is performed by tools called *concrete simulators*. Concerning atomic DEVS models, each concrete simulator has its own input language which is, essentially, a general-purpose programming language (such as Java or C++). Hence, once engineers have written the mathematical model, they need to manually translate it into the input language of the concrete simulator of their choice. In this paper we present a multi-target compiler for atomic DEVS models written in CML-DEVS, a mathematics-based DEVS modeling language. This multi-target compiler is able to compile a CML-DEVS model to the input languages of the PowerDEVS and DEVS-Suite concrete simulators. In this way, the CML-DEVS compiler frees engineers from the manual translation of their mathematical models. In fact, the same mathematical model can be simulated on both simulators by simply recompiling the model. The CML-DEVS multi-target compiler can be easily extended to produce code for other concrete simulators.

Keywords

Atomic model, CML-DEVS, compiler, DEVS

1. Introduction

Discrete Event System Specification (DEVS)[1] is perhaps the most general and used modeling and simulation (M&S) formalism. When using DEVS, a system is modeled by giving its structure, through a *coupled DEVS model*, and its behavior, through one or more *atomic DEVS models*, which are composed in intermediate coupled models that at some point form the final coupled model. Simulation of these models is performed by tools called *concrete simulators* (for instance, DEVS-C++,[2] DEVSim++,[3] CD++,[4] PowerDEVS,[5] JDEVS,[6] DEVS-Suite,[7] LSIS-DME[8]). Usually a concrete simulator provides to its users: (1) a way to compose atomic or coupled models into coupled models; and (2) a programming language to program atomic models, which in general is the same programming language as the concrete simulator.

Giving the structure of a coupled DEVS model is rather easy as tools frequently rest on some sophisticated graphical user interface (GUI) that allows engineers to graphically compose their atomic and coupled models. Indeed, these tools allow engineers not in the habit of programming to compose their models as they learned in textbooks. DEVS atomic models should also be described in the standard language of mathematics by using equations, functions, sets, etc. However, when engineers want to simulate these atomic models they need to program them in the input language of a concrete simulator, which means writing code in Java or C++ or another general-purpose programming language. Otherwise, they need to ask a programmer to do this. Furthermore, if they want to experiment with different concrete simulators they need to re-implement their models for each of them. The process of translating the mathematical model to the input language of a concrete simulator may induce errors that would render the simulation activity not as accurate as it should be.

For these reasons, we developed CML-DEVS,[9] a DEVS specification language based on standard mathematics and inspired in formal notations such as Z,[10] B,[11] and TLA+,[12] which are used by the software engineering community. CML-DEVS models may be used to abstractly describe DEVS atomic models, which can later be composed as done by each concrete simulator. In the context of CML-

[1]Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina
[2]CIFASIS-CONICET, Rosario, Argentina
[3]Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296,13397, France

Corresponding author:
Maximiliano Cristitá, Universidad Nacional de Rosario and CIFASIS, Pellegrini 250, (2000) Rosario, Argentina.
Email: cristia@cifasis-conicet.gov.ar

DEVS, *abstract model* and *CML-DEVS specification* denote a model described in the language of mathematics and logic. One of the objectives we had in mind when designing CML-DEVS was that it should be possible to automatically translate any CML-DEVS model into the input languages of the main concrete simulators.

In this paper we present a multi-target compiler for CML-DEVS models. That is, we present a program that reads a CML-DEVS specification and generates a program in the input language of a concrete DEVS simulator. In turn, this program generated by the CML-DEVS compiler can be compiled as indicated by the concrete simulator in order to simulate it. Therefore, the combination of CML-DEVS and its multi-target compiler relieves engineers of the error-prone, difficult task of translating their abstract models into concrete models. CML-DEVS and its multi-target compiler let engineers think in terms of mathematics and use several different concrete simulators to simulate the same model.

In this first version, the compiler produces PowerDEVS[5] and DEVS-Suite[7,13] code – that is, essentially C++ and Java code, respectively. However, we show how it can be extended to produce concrete models for other tools. In effect, by following standard compiler design techniques, our CML-DEVS compiler provides the functionality for parsing, type checking, abstract syntax tree (AST) construction, etc. of CML-DEVS code in such a way that producing object code for different concrete simulators is a rather easy task. The tool presented in this paper is a proof-of-concept, not a production tool. As such, it can be improved in many ways, although it features the basic structure and functionality of more advanced tools. With this tool, we aim at showing the DEVS community an alternative, complementary technology for modeling atomic DEVS models. In spite of this, we encourage the DEVS community to experiment with the current version of the compiler as it provides a new way of writing DEVS atomic models.

The CML-DEVS compiler can be freely downloaded, modified, and extended. It can be found at www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS.

The paper is structured as follows. In Section 2 we introduce, by means of a classroom example, the CML-DEVS specification language, assuming the reader is familiar with DEVS (otherwise refer to the work of Zeigler et al.[1]). The CML-DEVS multi-target compiler is described in Section 3, where we comment on key design decisions that guided us toward its implementation. An empirical evaluation of the compiler is presented in Section 4. This evaluation consists of collecting 14 atomic DEVS models, writing them in CML-DEVS, and compiling them to PowerDEVS and DEVS-Suite input languages with the CML-DEVS compiler. Integration of the CML-DEVS approach with existing DEVS tools is discussed in Section 5. Similar and related works are described in Section 6. Finally, we give our conclusions in Section 7. The Appendix contains further technical information referenced throughout the paper.

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

$X = \{c, k\}$	$\delta_{ext}(s_1, e, c) = s_5$
$S = \{s_1, s_2, s_3, s_4, s_5\}$	$\delta_{ext}(s_2, e, k) = s_2$
$Y = \{g, y, b\}$	$\delta_{ext}(s_3, e, c) = s_5$
$\delta_{int}(s_1) = s_2; \delta_{int}(s_2) = s_3$	$\delta_{ext}(s_4, e, c) = s_5$
$\delta_{int}(s_3) = s_4; \delta_{int}(s_4) = s_1$	$\delta_{ext}(s_5, e, k) = s_2$
$ta(s_1) = 20; ta(s_2) = 80$	$\lambda(s_1) = \lambda(s_3) = g$
$ta(s_3) = 3; ta(s_4) = 5$	$\lambda(s_2) = \lambda(s_4) = y$
$ta(s_5) = +\infty$	$\lambda(s_5) = b$

Figure 1. A typical textbook or classroom atomic DEVS model.

2. Introduction to CML-DEVS

CML-DEVS has been discussed in detail elsewhere.[9] Here we will show its main features by means of an example. We want to focus on the fact that writing CML-DEVS code is equivalent to what software engineers do when writing formal specifications in formal notations such as B.[11] In other words, we claim that CML-DEVS captures the mathematics used to write atomic models as in DEVS textbooks or as in the classroom in such a manner that tools can be built to process this language. Another analogy that might apply is that CML-DEVS is to DEVS what LaTeX is to mathematics. In this sense, mathematicians do not find writing math formulas with LaTeX particularly annoying, although it requires some learning.

Figure 1 shows an atomic DEVS model written as in DEVS textbooks or in DEVS courses,[1,13–20] while Figure 2 shows a pretty-printing of the CML-DEVS source code shown in Figure 3, corresponding to the model in Figure 1. What the model in Figure 1 represents is, at this point, not really important. Instead, we want to emphasize the fact that Figure 1 is a mathematical, abstract, simulator-independent description of a DEVS atomic model. In other words, we claim that people in the DEVS community would agree in that Figure 1 represents a typical textbook or classroom description of a DEVS atomic model.

In turn, note that Figure 2 is, essentially, a mathematical formula much like the one shown in Figure 1. It rests on equations, functions, and set theory, with no influence whatsoever from a general-purpose programming language.

On the other hand, the CML-DEVS source code of Figure 3 is aligned with the way specifications in formal notations such as Z, B, and TLA+ are written. We think the code is self-explanatory and respects the way

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \text{ where}$$
$$X = \{(in, x) : x \in \{c, k\}\}$$
$$S = \{s_1, s_2, s_3, s_4, s_5\}$$
$$Y = \{(out, y) : y \in \{g, y, b\}\}$$
$$\delta_{int}(s) = \begin{cases} s_2 & \text{if } s = s_1 \\ s_3 & \text{if } s = s_2 \\ s_4 & \text{if } s = s_3 \\ s_1 & \text{if } s = s_4 \end{cases}$$
$$ta(s) = \begin{cases} 20 & \text{if } s = s_1 \\ 80 & \text{if } s = s_2 \\ 3 & \text{if } s = s_3 \\ 5 & \text{if } s = s_4 \\ +\infty & \text{if } s = s_5 \end{cases}$$
$$\delta_{ext}(s, e, (p, \mathsf{value})) = \begin{cases} s_5 & \text{if } s \in \{s_1, s_3, s_4\} \\ & \quad \wedge\, \mathsf{value} = c \\ s_2 & \text{if } s \in \{s_2, s_5\} \\ & \quad \wedge\, \mathsf{value} = k \end{cases}$$
$$\lambda(s) = \begin{cases} (out, g) & \text{if } s \in \{s_1, s_3\} \\ (out, y) & \text{if } s \in \{s_2, s_4\} \\ (out, b) & \text{if } s = s_5 \end{cases}$$

Figure 2. Pretty-printing of the CML-DEVS source code shown in Figure 3.

atomic M is ⟨X, S, Y, dint, dext, lamda, ta⟩ where
  X is
    $in : \{c, k\}$
  end X
  S is
    $s : \{s_1, s_2, s_3, s_4, s_5\}$
  end S
  Y is
    $out : \{g, y, b\}$
  end Y
  dint is
    defcases
      case $s = s_2$ if $s = s_1$
      case $s = s_3$ if $s = s_2$
      case $s = s_4$ if $s = S_3$
      case $s = s_1$ if $s = s_4$
    end defcases
  end dint
  ta is
    defcases
      case 20 if $s = s_1$
      case 80 if $s = s_2$
      case 3 if $s = s_3$
      case 5 if $s = s_4$
      case INF if $s = s_5$
    end defcases
  end ta
  dext is
    defcases
      case $s = s_5$ if s in $\{s_1, s_3, s_4\} \wedge$ value $= c$
      case $s = s_2$ if s in $\{s_2, s_5\} \wedge$ value $= k$
    end defcases
  end dext
  lamda is
    defcases
      case (out, g) if s in $\{s_1, s_3\}$
      case (out, y) if s in $\{s_2, s_4\}$
      case (out, b) if $s = s_5$
    end defcases
  end lamda
end atomic

Figure 3. CML-DEVS code of the atomic model pretty-printed in Figure 2.

engineers write their abstract models. Consider the following observations:

1. CML-DEVS is based on logic, set theory, equations, and function definitions.
2. There are no side effects as it is a declarative language enjoying *referential transparency*.[21]
3. This source code can be generated by, for example, a formula editor featuring a rich GUI.
4. Pretty-printing (Figure 2) could be done by a simple translation tool producing LaTeX or XML code.
5. It is independent of any concrete DEVS simulator, relieving users of the task of learning several programming languages.

3. The design of the multi-target CML-DEVS compiler

In this section we describe the main features and design of the CML-DEVS multi-target compiler (or compiler for short). The description is somewhat detailed as we intend it to help the DEVS community to either implement similar tools or improve the one described in this paper. Some of the design decisions we show here were made for quickly providing a working tool for the DEVS community. In Section 3.2 we discuss the pros and cons of the present approach.

The CML-DEVS compiler is a Java program based on a standard one-pass compiler design and on the ANTLR parser generator.[22] Figure 4 shows a descriptive block diagram of the structure of the compiler. It is multi-target as it is conceived to generate code for different concrete simulators from the same CML-DEVS model, as we explain in Section 3.1. In this first version, though, it generates only

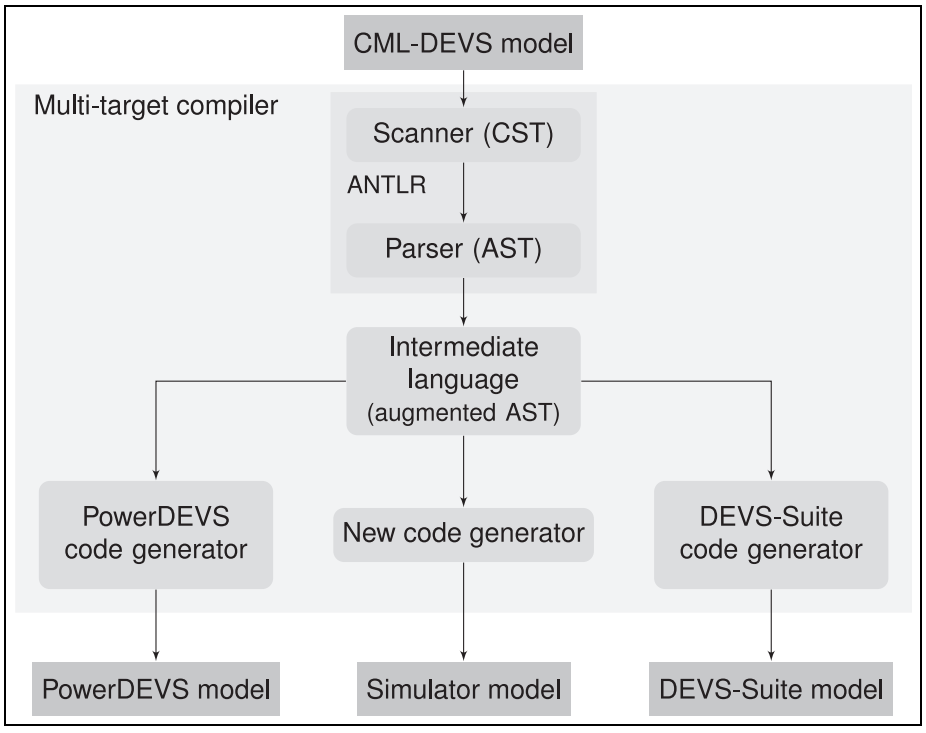

Figure 4. Descriptive block diagram of the CML-DEVS multi-target compiler.

PowerDEVS[5] and DEVS-Suite[7,13] code, which are essentially C++ and Java code, respectively. As we have said, this tool is a proof-of-concept whose main goal is to demonstrate the feasibility of the CML-DEVS approach. Then, we believe that the ''multi-target'' feature is demonstrated by generating code for more than one simulator and by showing that each new code generator can be easily implemented (see Section 3.1). Today, the CML-DEVS compiler is less than 20 KLOC (thousands of lines of code), including comments (15 KLOC of pure Java code).

The CML-DEVS grammar informed by Hollmann et al.[9] was written in the grammar language supported by ANTLR. In this way, ANTLR generated the *lexical analyzer* (*scanner*) and the *syntax and semantic analyzer* (*parser*). These two functional components are implemented by a collection of Java classes automatically generated by ANTLR.

The main function of the parser is to generate an AST of the CML-DEVS model. This AST is a central data structure as it organizes the model being compiled as a tree structure. The AST has a node for each terminal and non-terminal defined in the grammar that is being used in the model, where its children are the tokens that build it. For example, in the CML-DEVS code of Figure 3, *ta* is represented as a node whose only child is the `defcases` structure which, in turn, has four children, one for each `case` sentence. Hence, there is a Java class for each token defined in the grammar. However, these classes provide only syntactic information. ANTLR organizes these classes according to the Composite design pattern,[23] which allows uniform access to the structure. In particular, an object structure adhering to a Composite can be analyzed by implementing the Visitor design pattern.[23] This combination of design patterns facilitates the implementation of several key functions of the compiler.

Attempting to generate target (object) code from this AST is quite complex as the AST does not contain semantic information – for instance, it is not possible to know the type of each expression. For this reason, as shown in Figure 4, we decided to augment the AST with semantic information. In this way code generation (Section 3.1) becomes simpler.

ANTLR automatically generates a template[1] Visitor interface (`CMLDEVSVisitor`) specifically tailored to analyzing the AST generated during the parsing phase (see Figure 15 in the Appendix). Currently, the CML-DEVS compiler implements this interface with a set of classes headed by `CMLDEVSBaseVisitor`, whose function is to generate another AST containing semantic information about the model (i.e., the augmented AST). The heirs of `CMLDEVSBaseVisitor` create the nodes of the augmented AST. In this way, it can be said that the implementation of `CMLDEVSBaseVisitor` represents the *intermediate code generator*.

The AST generated by `CMLDEVSBaseVisitor` is organized as a Composite design pattern headed by the

CMLDEVSData interface (see Figure 16 in the Appendix). Each node in the augmented AST is an heir of CMLDEVSData containing information such as the semantic role played by each syntax element and the type of expressions. For example, in the augmented AST, the *ta* node of the AST mentioned above contains information indicating what is the definition part and the condition part of each case sentence, what is the type of each variable participating in them, etc. This semantic information is stored in the heirs of CMLDEVSData. In this way, it can be said that the augmented AST is an *intermediate language*.

3.1. Code generation

Carefully designing the code-generation phase (see Figure 4) is important in the CML-DEVS compiler as we intend it to be a multi-target compiler. The main design decision is to postpone code generation as much as possible. In this way, code generators do not need to implement other functions as they are provided by previous phases. Then, new code generators are small, simple, and easy to add.

When calling the CML-DEVS compiler, users must pass a parameter telling it what simulator language the compilation should produce. This parameter is used internally to instantiate the proper *code generator*. In the CML-DEVS compiler, each code generator has three main responsibilities:

- produce object code respecting the syntax and conventions of each concrete simulator;
- distribute the final code in files according to the requirements set by each concrete simulator – for example, PowerDEVS requires three files for an atomic model (ModelName.pds, ModelName.h, and ModelName.cpp), while DEVS-Suite[7,13] requires only one (ModelName.java); and
- substitute reserved words of the target language used in the CML-DEVS specification. For example, class is a reserved word in C++, Java, etc., but is not in CML-DEVS. Therefore, engineers may use class in their CML-DEVS specifications as a name for variables, constants, etc., but when the compiler generates code for a concrete simulator whose input language is based on an object-oriented language, this word must be replaced because otherwise the generated model will not compile. We discarded the possibility of reserving more words at the CML-DEVS level because this would mean collecting the reserved words of all possible input languages of concrete simulators.

Each of these responsibilities is assigned to different classes, which have to be carefully created as they are related to each other. Creating families of related objects is the purpose of the Abstract Factory design pattern.[23] Hence, the CML-DEVS compiler defines TargetLanguageFactory, an interface for instantiating objects that depend on the target language (see Figure 17 in the Appendix).

Target code generation (i.e., the first responsibility listed above) is organized according to the Visitor design pattern.[23] This Visitor visits the Composite that structures the augmented AST headed by CMLDEVSData and *prints* the final code. Hence, the CML-DEVS compiler defines the Printer interface such that each of its implementations will *print* object code corresponding to each sentence of the intermediate language. An excerpt of Printer's

```
public interface Printer {
    String print(State s);
    String print(DeltaInt dint);
    String print(TimeAdvance ta);
    String print(Assignment assig);
    String print(Cases cases);
    String print(ListExpression listExpression, CMLDEVSType type);
    String print(NumberSetExpression numberSetExpression, CMLDEVSType type);
    String print(TextValue textValue, CMLDEVSType type);
    String print(NatValue natValue, CMLDEVSType type);
    String print(ComparisonDiff comparisonDiff);
    String print(ComparisonEq comparisonEq);
    String print(OperationPlus operationPlus, CMLDEVSType type);
    String print(OperationMult operationMult, CMLDEVSType type);
    ...
}
```

Figure 5. Part of Printer's interface.

```
public Void print() {
    stHeader.add("modelName", atomic.getName());
    stHeader.add("S", print(atomic.getState()));
    stSource.add("lambda", print(atomic.getLambda()));
    ...
}
public String print(State s) {
    String sString = "";
    sString += decls2string(s.getStateVars());
    return sString;
}
public String print(LambdaCases cases) {
List<String> casesSt = new ArrayList<>();
for (LambdaCase c: cases.getCases())
        casesSt.add("("
                    + c.getCondition().accept(this)
                    + "){\n" + c.getPair().accept(this) + "\n}");
    String otherwise;
if (cases.hasOtherwise())
        otherwise = "\nelse{\n" + cases.getOtherwise().getPair().accept(this) + "\n}";
    else
        otherwise = "\nelse{\n return Event();\n}";
    return "if " + StringUtils.join(casesSt, "\nelse if") + otherwise;
}
```

Figure 6. Snippets of `PrinterPowerDEVS`'s implementation.

```
headerFile(modelName,path,params,S,X,Y,dint,
           dext,functions,funLib) ::= <<
  ...
  class <modelName>: public Simulator {
  public:
    <if(X)> <X> <endif>
  ...
  <modelName>(const char *n): Simulator(n) {};
    void init(double, ...);
    double ta(double t);
    void dint(double);
    void dext(Event , double );
    Event lambda(double);
    void exit();
  };
... <<
```

Figure 7. Excerpt of the StringTemplate template used to generate PowerDEVS code.

interface is shown in Figure 5. Note that there are methods to print each terminal and non-terminal of the intermediate language. In this sense, the classes implementing this interface are known as *pretty-printers* or *printers*. In fact, these printers use StringTemplate technology to produce the final code. StringTemplate is a Java template engine for generating source code developed by ANTLR's designer.[24]

Therefore, implementing the code generator for PowerDEVS (respectively DEVS-Suite) implies providing, among others, an heir of `TargetLanguageFactory`, called `PowerDEVSFactory` (`DEVSSuiteFactory`), and an implementation of Printer, called `Printer PowerDEVS` (`PrinterDEVSSuite`). We will focus on `PrinterPowerDEVS` as `PrinterDEVS Suite` is very similar, and printers are the most interesting components of code generation. Implementing `PrinterPowerDEVS` entails defining a StringTemplate template and implementing some of its methods by calling StringTemplate. Figure 6 shows code snippets of the implementation of three methods of `PrinterPower DEVS`, and Figure 7 shows an excerpt of the template. As can be seen, the template consists of the basic structure of the code to be generated with placeholders that are replaced each time the template is used. The replacement can be done with a library provided by StringTemplate. The placeholders are replaced with the actual data taken from the augmented AST. For example, in the second sentence of print(), in Figure 6, `stHeader` is the instantiation of the template shown in Figure 7. Then, this sentence replaces parameter `S` of `headerFile` with the result of `print(atomic.getState())`, whose implementation can also be seen in Figure 6.

Hence, implementing a new code generator entails repeating the implementation schema followed for the implementation of the PowerDEVS and DEVS-Suite code generators. That is, defining an heir of `Target LanguageFactory` and an implementation of `Printer` and implementing it using StringTemplate. That is, it would be convenient (although not mandatory) to define a new template considering the peculiarities of the input language of the concrete simulator. As a matter of fact, the implementation of the methods shown in

Figure 5 for the DEVS-Suite simulator are almost identical to those of PowerDEVS. This means that the effort of implementing a new code generator is alleviated not only by the general design of the compiler, but also by the fact that existing code generators can be used as the base to implement new ones.

Given that creating object code by printing can be bad in terms of performance, this technique can be changed or improved in the future by tool developers. This technique was chosen because it is one of the simplest forms of code generation, thus allowing rapid prototyping of the compiler.

The code corresponding to the PowerDEVS and DEVS-Suite code generators is about 1 KLOC each. This shows that the effort of implementing new code generators (see Figure 4) is marginal with respect to the total effort (recall that currently the CML-DEVS compiler is about 20 KLOC), as is otherwise expected if proven compiler techniques are followed. In turn, this suggests that the idea of defining a specification language for atomic DEVS models and designing a multi-target compiler for it, was right.

3.2. Discussion

In this section we discuss the advantages and disadvantages of using the CML-DEVS approach (i.e., the CML-DEVS language plus its multi-target compiler). CML-DEVS provides a mathematics-oriented specification language for describing atomic DEVS models. This is aligned with the way DEVS models are presented in courses and textbooks. Instead, using a general-purpose programming language demands engineers not only to be experts in the problem domain, but also to be programmers. The CML-DEVS compiler complements the specification language by generating code for (potentially) many concrete simulators. This allows engineers to write an abstract model once while being able to simulate it on many different simulators. CML-DEVS is expressive enough to specify all DEVS atomic models.[9]

However, the approach is not free of limitations and disadvantages. Engineers need to learn a new language (i.e., CML-DEVS). This can be reduced to a minimum if a formula editor is implemented. Nevertheless, either engineers learn CML-DEVS or they learn to program in the input language of a concrete simulator – in turn this is sometimes not the case because engineers already know how to program. Learning CML-DEVS has the advantage that they can use different concrete simulators easily. The code generated by the CML-DEVS compiler may be inefficient compared to the code programmed by an expert on a particular concrete simulator. Another issue with our approach is that changes in the design of a concrete simulator (e.g., its input language) could require changes in the CML-DEVS compiler. However, the design of the compiler would limit these modifications to specific modules (in general to the code-generation modules).

4. Empirical evaluation

In this section we present the results of an empirical evaluation of the CML-DEVS compiler. The empirical evaluation aims at showing that: (1) mathematically described atomic DEVS models can be written in CML-DEVS simply by adhering to its syntax conventions; (2) the compiler can produce concrete models for PowerDEVS and DEVS-Suite from the *same* CML-DEVS model; (3) the resulting concrete models are syntactically more complex than the CML-DEVS models; and (4) compilation times are reasonable.

The results of this empirical evaluation are summarized in Table 1; Table 2 in the Appendix gives a brief informal description of each atomic model.

In Table 1, column T indicates whether the CML-DEVS specification was written from a mathematical description (D) or from the source code of an atomic PowerDEVS (C) or DEVS-Suite (J) model. Hence, as can be seen from the table, we collected a sample of 10 mathematically described atomic DEVS models plus 4 concrete models (ConstGen, HInt, BinaryCounter, and Generator). All 14 models were taken from third-party resources such as books, websites, and courses, and cover a wide range of applications, origins, and authors, thus representing a reasonable sample – that is, these models were not proposed by us, which would have biased the evaluation. In effect, we have collected models from six different sources and authorships. The sources include Cellier and Kofman's book on continuous system simulation; the PowerDEVS library of atomic models; Professor Vangheluwe's class notes for his course ''Modelling of Software-Intensive Systems'' given at McGill University; Professor Wainer's repository on CD++ models, which includes models written by students who took his courses ''Simulation of Discrete Event Systems'' given at Buenos Aires University and ''Methodological Aspects of Modeling and Simulation'' taught at Carleton University; the technical report from Zeigler and Sarjoughian on M&S describing DEVS-Suite; and a model described by Professor Wainer in one of his class presentations. That is, there are models written by experts and students as well. Next, we have translated the models from the mathematical descriptions used by their authors into CML-DEVS specifications; in the case of the four concrete models we wrote their CML-DEVS specifications from informal descriptions. In doing so, we tried to follow the mathematical structure suggested by each author. We believe this supports claim (1) mentioned above. That is, atomic DEVS models can be easily written in CML-DEVS.

Then, we used the CML-DEVS compiler to compile to PowerDEVS and DEVS-Suite each of the 14 DEVS atomic models. The concrete models produced by the CML-DEVS compiler can be simulated by the corresponding concrete simulator. In particular, models ConstGen,

Table 1. Atomic DEVS models used for the evaluation of the CML-DEVS compiler.

	Model	Source	T	Size (bytes)			Time (s)
				CML-DEVS	PowerDEVS	DEVS-Suite	
1	ACCtrlUnit	Wainer's sample of DEVS models[14]	D	2422	4245	4946	2
2	ACTempProp	Wainer's sample of DEVS models[14]	D	1311	2903	3669	2
3	CoolUnit	Wainer's sample of DEVS models[14]	D	746	2178	2692	2
4	ATMVerif	Wainer's sample of DEVS models[16]	D	961	2584	2991	2
5	BilliardBall	Zeigler and Sarjoughian[13]	D	735	2389	2945	1
6	BinaryCounter	Zeigler and Sarjoughian[13]	J	631	2104	2557	2
7	Constant	PowerDEVS model library	C	335	1415	1800	1
8	ElevatorDoor	Wainer's sample of DEVS models[17]	D	1440	3262	3964	2
9	ElevatorEngine	Wainer's sample of DEVS models[17]	D	1755	3464	4323	2
10	Generator	Zeigler and Sarjoughian[13]	J	384	1491	1861	1
11	HInt	Cellier and Kofman[18]	C	1196	2771	3312	1
12	TrafficLights	Vangheluwe's class notes[15]	D	1051	2713	3388	1
13	Server	Wainer's course material[19]	D	799	2372	3046	2
14	Switch	Zeigler and Sarjoughian[13]	D	1045	2810	3453	2

Column T indicates whether the CML-DEVS specification was written from a mathematical description (D) or from the source code of an atomic PowerDEVS (C) or DEVS-Suite (J) model.

```
#include "HInt.h"
double HInt::ta() {
  return sigma;
}
void HInt::dint(double t) {
  X = X + sigma * dX;
  if (dX > 0) {
    sigma = dq / dX;
    q = q + dq;
  }
  else
    if (dX < 0) {
      sigma = -dq / dX;
      q = q - dq;
    }
    else
      sigma = inf;
}

void HInt::dext(Event x, double t) {
  float xv;
  xv = *(float*)(x.value);
  X = X + dX * e;
  if (xv > 0)
    sigma = (q + dq - X) / xv;
  else
    if (xv < 0)
      sigma = (q - epsilon - X) / xv;
    else
      sigma = inf ;
  dX = xv;
}
Event HInt::lambda(double t) {
  if (dX == 0)
    y = q;
  else
    y = q + dq * dX / fabs(dX);
  return Event(&y,0);
}
```

Figure 8. PowerDEVS (C++) implementation of HInt as given by Cellier and Kofman.

```
#include "HInt.h"
double HInt::ta(double t) {
    return sigma;
}
void HInt::dint(double t) {
    HInt p = *this;
    xS = p.xS + sigma * p.dX;
    if (p.dX > 0) {
        sigma = p.dq / p.dX;
        q = p.q + p.dq;
    }
    else
        if (p.dX < 0) {
            sigma = -p.dq / p.dX;
            q = p.q - p.dq;
        }
        else
            sigma = INFINITY;
}
void HInt::dext(Event x, double t) {
    HInt p = *this;
    double value = *(double*)(x.value);
    xS = p.xS + p.dX * e;
    if (value > 0)
        sigma = (p.q + p.dq - p.xS)/value;
    else
        if ((value < 0))
            sigma=(p.q-p.epsilon-p.xS)/value;
        else
            sigma = INFINITY;
    dX = value;
}
Event HInt::lambda(double t) {
    if (dX == 0) {
        y = q;
    else
        y = q + (dq * dX) / fabs(dX);
    return Event(&y, Y_y);
}
```

Figure 9. PowerDEVS (C++) implementation of HInt resulting from compiling the CML-DEVS model of Figure 10.

```
atomic HInt is ⟨X, S, Y, dint, dext, lamda, ta⟩ where
    S is xS, dX, q, sigma : R end S
    X is xX : R end X
    Y is y : R end Y
    dint is
        xS = xS + sigma * dX
        defcases
            case sigma = dq/dX ∧ q = q + dq if dX > 0
            otherwise defcases
                case sigma = −dq/dX
                        ∧ q = q − dq if dX < 0
                otherwise sigma = INF
            end defcases
        end defcases
    end dint
    dext is
        xS = xS + dX * e
        defcases
            case sigma = (q + dq − xS)/value if value > 0
            otherwise defcases
                case sigma = (q − epsilon − xS)/value
                        if value < 0
                otherwise sigma = INF
            end defcases
        end defcases
        dX = value
    end dext
    lamda is
        defcases
            case (y, q) if dX = 0
            otherwise (y, q + dq * dX/abs(dX))
        end defcases
    end lamda
    ta is sigma end ta
end atomic
```

Figure 10. CML-DEVS source code for Cellier and Kofman's HInt.

HInt, BinaryCounter, and Generator allow us to compare the code generated by the CML-DEVS compiler with respect to the code written by PowerDEVS and DEVS-Suite expert users. In order to keep the presentation concise, we include here the analysis of model HInt, but similar conclusions can be drawn from the other three models. Model HInt is a *hysteretic quantized integrator* which is used in continuous system simulation, as defined by Cellier and Kofman.[18] Figure 8 lists the PowerDEVS code of HInt as proposed by Cellier and Kofman.[18] The result of compiling the CML-DEVS code is shown in Figure 9, while the code itself is shown in Figure 10. As can be seen, both PowerDEVS programs are similar in size, structure, and functionality. Furthermore, in Figure 11 we can see the results of using both implementations (i.e., Figures 8 and 9) as part of a PowerDEVS simulation. It is obvious that both programs yield the same results, which is an indication that the compilation of the CML-DEVS specification behaves the same as the original model.

Given that all 14 CML-DEVS models and the CML-DEVS compiler are publicly available,[2] we believe the above results support claim (2) mentioned at the beginning of this section.

In Table 1, columns CML-DEVS, PowerDEVS, and DEVS-Suite show, respectively, the size in bytes of the CML-DEVS specification and the PowerDEVS and DEVS-Suite source code resulting from compiling the specification with the CML-DEVS compiler. The column Time is the approximate compilation time (of both PowerDEVS and DEVS-Suite as differences are negligible). The compilation times shown in the table are approximate and rounded; they are measured from the command-line shell by simply taking the system time before and after compilation. The platform used for these tests is the following: AMD Athlon™ 7850 Dual-Core Processor CPU at 1.40 GHz with 4 Gb of main memory, running Linux Kubuntu 14.04 (Trusty Tahr) of 64-bit with kernel 3.16.0-67-generic; the CML-DEVS compiler uses Java 1.7, ANTLR 4.5, and StringTemplate 4.0.8.

As Table 1 shows, compilation times are acceptable given that by using the compiler engineers will obtain the concrete models from the mathematical description in a few seconds. Note that programming these models would take much longer. It is also clear that the sizes of the compiled models are higher than the CML-DEVS specifications. This is an indication of how CML-DEVS abstracts away syntactic details that otherwise need to be considered if the input languages of concrete simulators are used. The CML-DEVS compiler fills in these details for the engineer.

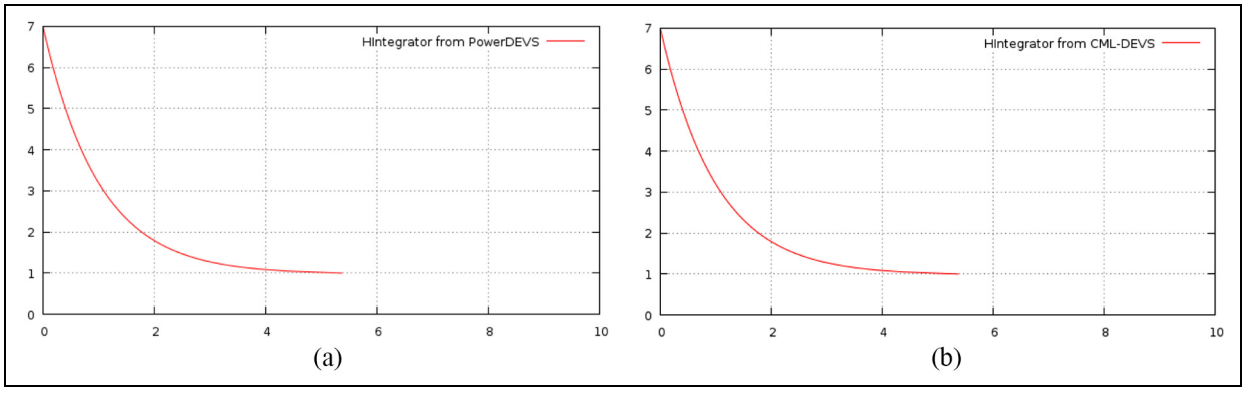

Figure 11. Plot of the curves obtained by simulating the model given in Figure 8(a) and Figure 9(b).

```
atomic TrafficLights is ⟨X, S, Y, dint, dext, lamda, ta⟩ where

    X is in : {M, A} end X
    S is s : {RG, RY, GR, YR, BB} end S
    Y is out : {GREEN, YELLOW, BLINK} end Y

    dint is
        defcases
            case s = RY  if s = RG
            case s = GR  if s = RY
            case s = YR  if s = GR
            case s = RG  if s = YR
        end defcases
    end dint

    dext is
        defcases
            case s = BB  if s in {RG, RY, GR, YR}
                              ∧ value = M
            case s = RY  if s = BB ∧ value = A
        end defcases
    end dext

    lamda is
        defcases
            case (out, GREEN)  if s in {RG, RY, GR}
            case (out, YELLOW)  if s = YR
            case (out, BLINK)  if s = BB
        end defcases
    end lamda

    ta is
        defcases
            case 60  if s = RG
            case 10  if s in {RY, YR}
            case 50  if s = GR
            case INF  if s = BB
        end defcases
    end ta
end atomic
```

Figure 12. CML-DEVS source code for the `TrafficLights` atomic model.

```
#include "TrafficLights.h"

using namespace auxFunc;

void TrafficLights::dint(double t) {
    TrafficLights prev("");
    prev = *this;
    if (prev.s == "RG")       s = "RY";
    else if (prev.s == "RY") s = "GR";
    else if (prev.s == "GR") s = "YR";
    else if (prev.s == "YR") s = "RG";
}

void TrafficLights::dext(Event x, double t) {
    TrafficLights prev("");
    prev = *this;
    std::string value = *(std::string*)(x.value);
    if (findInSet(prev.s, {"RY", "RG", "YR", "GR"})
                    && value == "M")
        s = "BB";
    else if (prev.s == "BB" && value == "A")
        s = "RY";
}

Event TrafficLights::lambda(double t) {
    if (findInSet(s, {"RY", "RG", "GR"})) {
        out = "GREEN";
        return Event(&out, Y_out);
    }
    else if (s == "YR") {
        out = "YELLOW";
        return Event(&out, Y_out);
    }
    else if (s == "BB") {
        out = "BLINK";
        return Event(&out, Y_out);
    }
    else return Event();
}

double TrafficLights::ta(double t) {
    if (s == "RG")        return 60.0;
    else if (s == "RY")   return 10.0;
    else if (s == "GR")   return 50.0;
    else if (s == "YR")   return 10.0;
    else if ((s == "BB")) return INFINITY;
}
```

Figure 13. Result of compiling to PowerDEVS input language (C++) the `TrafficLights` model of Figure 12.

As another example of the code generated by the CML-DEVS compiler, Figure 13 lists the result of compiling the `TrafficLights` model shown in Figure 12 to the PowerDEVS input language (in the Appendix, Figure 14 lists the result of compiling the same model to DEVS-Suite). As Figure 13 shows, the code is clean, well-indented and structured, and strictly follows the conventions set forth by PowerDEVS (e.g., there is a function called `dint` for the internal transition function, another function `dext` for the external transition function, and so on). Note the use of function `findInSet`, which is a function implemented as part of the CML-DEVS framework. Functions such as this are included in the library `auxFunc`, which in turn is made available to the PowerDEVS model. PowerDEVS users would have to write their own set of manipulation functions if they implemented the model without the CML-DEVS compiler. Instead, by using the compiler, they can simply write $s \text{ in} \{RG, RY, GR\}$ and let the compiler implement it. Last, but not least, compare the simplicity, familiarity, and cleanness of the CML-DEVS source code of Figure 12 with respect to the C++ code of Figure 13. For example, in the former there are no elements such as casts and pointers (i.e., programming, not modeling, concepts), which are necessary in the latter. We argue that the *model* of Figure 12 can be written by an engineer completely unaware of C++, which is not the case for the *program* of Figure 13.

In our opinion, compilation times, the sizes of the CML-DEVS models, and the corresponding PowerDEVS and DEVS-Suites concrete models, plus Figures 12 and 13, clearly support claims (3) and (4) mentioned above.

We believe this evaluation shows that the whole approach (i.e., the CML-DEVS specification language and

its multi-target compiler) is feasible and has several advantages over existing technology.

5. Integrating CML-DEVS within existing simulators

Mainstream DEVS simulators usually feature powerful GUIs that allow users to easily compose large models from existing ones. However, as we pointed out, atomic models have to be written in general-purpose programming languages. For this task, DEVS simulators either provide a programming editor or users can use the editor of their choice. Once the new atomic model is written, it can be used as a component of larger models by a simple gesture of the GUI.

The CML-DEVS compiler can be integrated into the PowerDEVS and DEVS-Suite environments. If the appropriate code generators are developed (Section 3.1), the compiler could in principle be integrated into DEVS-based systems such as DEVS-C++,[2] DEVSim++,[3] CD++,[4] JDEVS,[6] and LSIS-DME.[8] Some of these systems are complex, powerful M&S environments. For example, PowerDEVS features a rich GUI interface and a large model library, allowing users to easily compose models. As another example, JDEVS[6] integrates five modules: a simulation kernel, a GUI interface for coupled models, a models library, a connection to a GIS, and a cellular simulation panel. The integration of M&S components into existing systems has a long tradition in the DEVS community.

Therefore, we propose to integrate the CML-DEVS compiler into existing DEVS simulators. In the first place, the corresponding code generator has to be implemented. Once the code generator is available, the compiler can be integrated into the DEVS simulator system as follows:

1. Use the editor provided by your DEVS simulator to write CML-DEVS code for the new atomic models. Ideally, a CML-DEVS editor, such as a formula editor, can also be easily integrated.
2. Compile each CML-DEVS model into the input language of your simulator. Here, the editor can call the CML-DEVS compiler.
3. Save the compiled model as any other atomic model of the simulator. CML-DEVS compiled models are indistinguishable from atomic models developed by other means.
4. Now users can couple compiled CML-DEVS models with other models as is normally done in your simulator (e.g., by using exactly the same GUI gesture).

In this way, simulators' users will build their DEVS models as usual up until the moment they need to write a new atomic model. At this point the simulator environment can call the CML-DEVS editor, allowing users to write more abstract, mathematics-oriented models that will be transparently coupled in larger models. Furthermore, if users want to try out these atomic models on different simulators they can simply take the CML-DEVS sources to the environment of the new simulator (optionally they can compile the CML-DEVS models and export the object code). From this point, coupling these models proceeds as usual in the new simulator.

6. Related work

As far as we know, there is no approach such as the CML-DEVS multi-target compiler for the automatic generation of atomic DEVS models. However, there are some works that in one way or another are related to this approach. We will briefly comment on them in this section.

CML-DEVS has some relation with the standardization effort carried out by the DEVS community.[25] One of the standardization areas identified by this group is model representation.[26–27] Notations such as CML-DEVS could be used for model representation as they are independent of simulators. DEVSpecL, developed by Hong and Kim,[28] which somewhat inspired CML-DEVS, could also be used as an abstract model representation. The relation between CML-DEVS and DEVSpecL was commented on by Hollmann et al.[9] Mittal and Douglass[29] present a domain-specific language based on Finite Deterministic DEVS, which, with some limitations, can also be used to write abstract DEVS models. These last two proposals would allow automatic code generation in order to get executable DEVS code in different DEVS implementations, but apparently they do not face this problem. Several works propose XML as a language to describe DEVS models.[27,30–32] One of the reasons is that XML is platform-independent and thus is sometimes regarded as *abstract*. We believe XML bears no relation to the notion of the abstract model as seen in the CML-DEVS context (i.e., the conceptual distance with respect to the language of mathematics and formal logic). XML could, indeed, be useful to communicate and share models among computers, systems, and tools.

CML-DEVS is inspired by formal notations used in software engineering such as Z,[10] B,[11] and TLA+.[12] For example, the semantics of DEVS can be formalized in TLA+.[33] Engineering and scientific software tend to have many errors that turn decision-making based on them risky.[34–36] Researchers and engineers use software that has not been formally or even extensively verified by experts.[37] Some errors are introduced due to development processes based on informal descriptions. In this sense, the CML-DEVS approach is an attempt to formalize the process of developing a concrete simulation model.

Model-driven engineering (MDE) and model-driven development (MDD) attempt to translate abstract models into more concrete models by means of model transformations. Once the initial model and all the model transformations are given, the final model can be automatically generated.[38] CML-DEVS and its compiler can be seen in terms of MDD: CML-DEVS would be the modeling language used to describe an abstract model and the CML-DEVS compiler would be a model transformation. On the other hand, the DEVS community has attempted to adopt concepts and techniques from MDE and MDD, in particular there are efforts toward defining model transformations.[39–45] In these approaches, different modeling or meta-modeling languages are proposed to describe DEVS models in such a way that they can be automatically transformed by the corresponding model transformations. None of these modeling languages describes atomic DEVS models using only mathematical or logical concepts. The modeling and meta-modeling languages proposed within the DEVS community, instead, are based on general object-oriented technologies and notations, notably UML, XML, OCL, etc. Although some of the model transformations proposed in the works cited above are automatic, some of them still require writing code in some general-purpose programming language. In this way, we think that our work provides a concrete implementation of a modeling language and a model transformation, although not inspired by MDE or MDD concepts.

7. Concluding remarks

We have presented the main features and properties of a multi-target compiler for CML-DEVS specifications. We have shown that CML-DEVS specifications are quite close to the way engineers would use mathematics to write their atomic DEVS models. Then we have shown that these specifications can be compiled into the input language of PowerDEVS and DEVS-Suite, which are mainstream DEVS simulators. We have also provided evidence that the code-generation phase of the CML-DEVS compiler can be easily reimplemented to generate code for other DEVS concrete simulators. Indeed, currently, code generators for PowerDEVS and DEVS-Suite are about 10% of the total compiler code, which makes it evident that code generation is relatively easy. But it is even more important that plugging in a new code generator is favored by the design of the compiler as it is based on well-known compiler designs. In fact, plugging in a new code generator would require no code modification, only new code. A multi-target compiler would enable the possibility of easily simulating the same atomic model on an array of concrete simulators by simply recompiling the CML-DEVS specification.

Having an abstract, mathematics-oriented specification language for DEVS models and a compiler that automatically produces concrete models would make the task of M&S much easier, more productive, and less error-prone. In effect, from the conception of the idea of a DEVS model to its implementation in the input language of major concrete simulators, either the engineer has to learn a programming language or to ask a programmer to implement his or her models. In either case, the initial model is read and interpreted by different persons over a lengthy time period. These multiple readings might introduce errors in the final model with respect to the initial, abstract model. Furthermore, if engineers want to see how the model behaves (in terms of performance, for instance) on different simulators, they need to implement it over and over again, in which case more errors can be introduced. Leaving errors to one side, productivity would be increased if the same CML-DEVS specification can be automatically implemented for different simulators. Moreover, engineers would not need to learn to program nor to rely on a programmer to try out their models. Put in another way, how much time and effort would be needed for, say, an electric engineer to learn C++ in such a way as to be able to produce the code shown in Figure 13? Conversely, how much time and effort would (s)he need to learn CML-DEVS, provided (s)he already knew DEVS, in such a way as to be able to produce the code of Figure 12? What is the core business of an electric engineer – to program or to write mathematical models?

Having a multi-target compiler opens the door to, at least, two important aspects: (1) the compiler can be optimized by experts in such a way as to produce the best possible code; and (2) once the compiler is proved correct, model translation stops being a source of errors and problems.

Appendix 1

Figure 14 lists the Java code resulting from compiling the traffic light model of Figure 12 with the CML-DEVS compiler after choosing Java as the target language. This Java program is an atomic DEVS model of the DEVS-Suite simulator. The code has been edited to make it fit into a single page. Compare the length and complexity of the Java code of Figure 14 with respect to the CML-DEVS code of Figure 12.

Table 2 gives a brief informal description of the models used in the empirical evaluation. These descriptions are taken directly from the authors.

Figures 15–17 depict UML class diagrams of some of the design patterns used to implement the CML-DEVS compiler. Due to space reasons, some elements in these class diagrams are omitted.

```
package TrafficLights;

import GenCol.entity;
import model.modeling.content;
import model.modeling.message;
import view.modeling.ViewableAtomic;
import java.util.*;

public class TrafficLights extends ViewableAtomic implements Cloneable {
    String s = new String();
    String In = new String();

    public class outEnt extends entity {
        String value;
        outEnt(String value) {this.value = value;}
        public String getValue() {return value;}
        public String getName() {return value.toString();}
    }

    public TrafficLights() {
        super("TrafficLights");
        addInport("In");
        addOutport("out");
    }

    public void deltint() {
        TrafficLights prev = null;
        try prev = (TrafficLights)this.clone();
        catch (CloneNotSupportedException ex) System.out.println("Clone not supported");
        if ((prev.s == "RG")) s = "RY";
        else if ((prev.s == "RY")) s = "GR";
        else if ((prev.s == "GR")) s = "YR";
        else if ((prev.s == "YR")) s = "RG";
    }
    public void deltext(double e, message x) {
        TrafficLights prev = null;
        try prev = (TrafficLights)this.clone();
        catch (CloneNotSupportedException ex) System.out.println("Clone not supported");
        String port = x.getPortNames().toArray()[0].toString();
        String value = (String)(x.read(0)).getValue();
        if (((new TreeSet<String>(Arrays.asList("GR", "RG", "RY", "YR"))).contains(prev.s)
             && (value == "M"))) s = "BB";
        else if (((prev.s == "BB") && (value == "A"))) s = "RY";
    }
    public message out() {
        message mess = new message();
        content cont;
        if ((new TreeSet<String>(Arrays.asList("GR", "RG", "RY"))).contains(s))
            cont = makeContent("out", new outEnt("GREY"));
        else if ((s == "YR")) cont = makeContent("out", new outEnt("YELLOW"));
        else if ((s == "BB")) cont = makeContent("out", new outEnt("BLINK"));
        else cont = makeContent("", new entity());
        mess.add(cont);
        return mess;
    }
    public double ta() {
        if ((s == "RG")) return 60.0;
        else if ((s == "RY")) return 10.0;
        else if ((s == "GR")) return 50.0;
        else if ((s == "YR")) return 10.0;
        else if ((s == "BB")) return INFINITY;
        else return INFINITY;
}
```

Figure 14. Result of compiling the traffic lights model to DEVS-Suite input language.

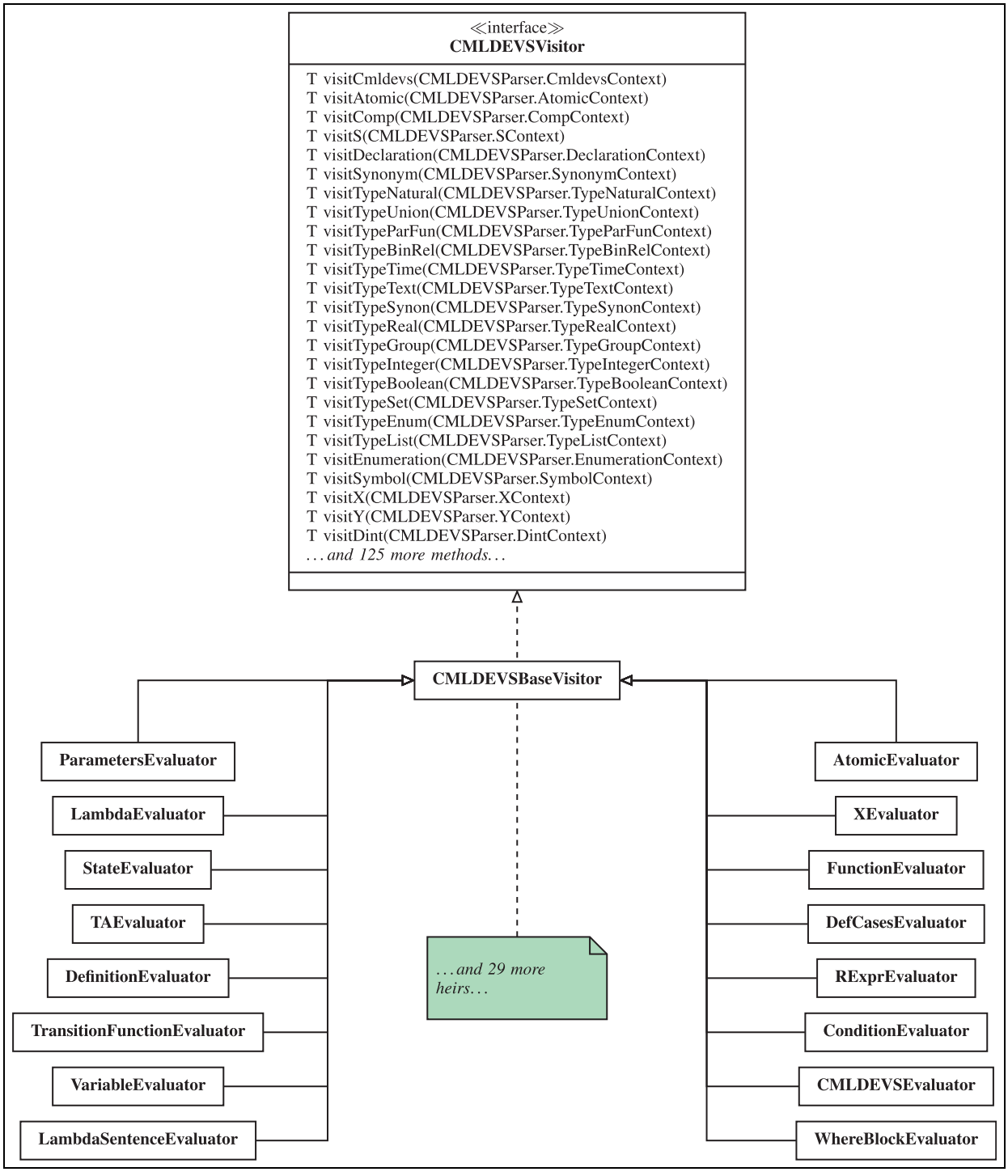

Figure 15. `CMLDEVSVisitor` is the head class of an instance of the Visitor design pattern. These classes visit objects in Figure 16.

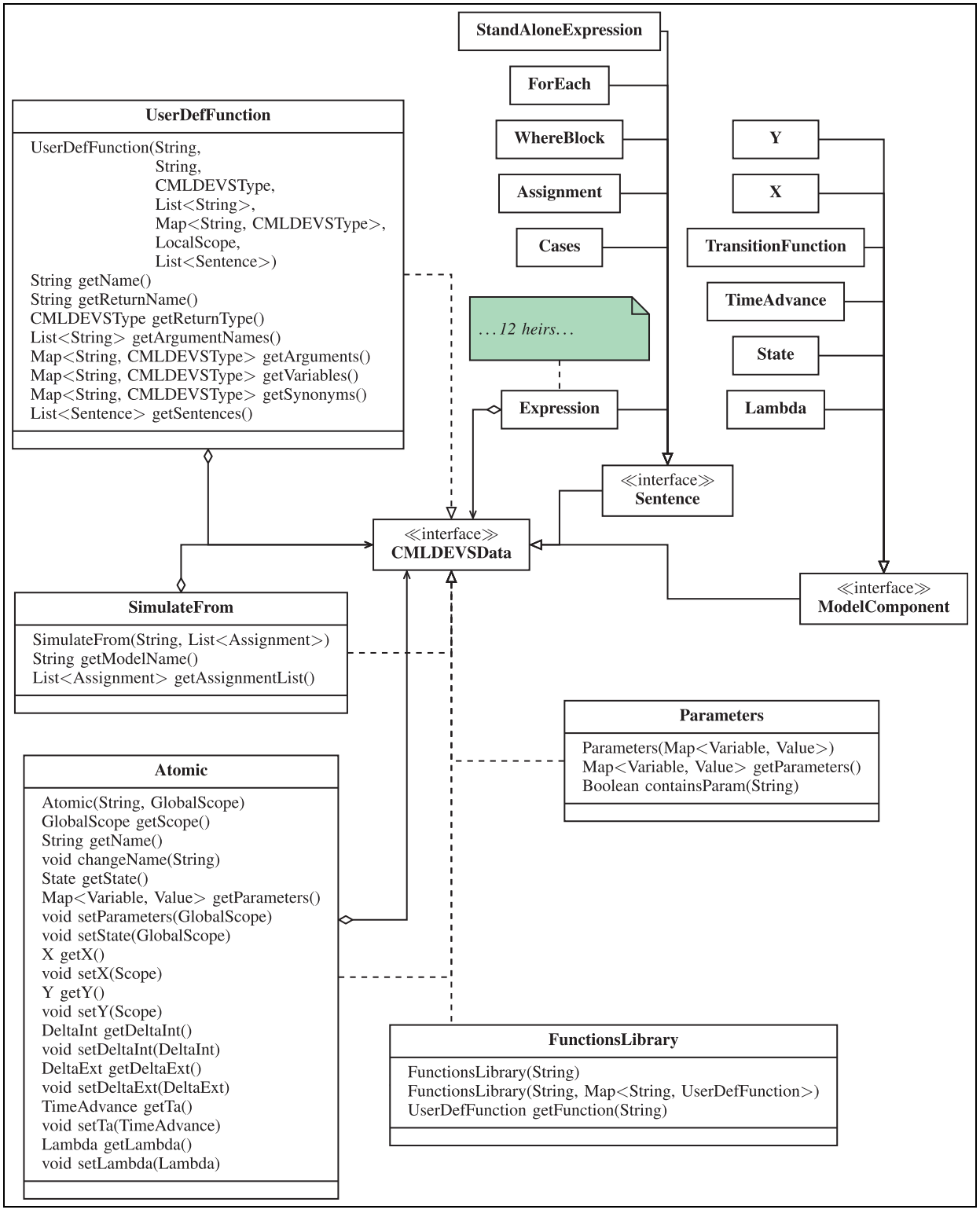

Figure 16. `CMLDEVSData` is the head class of an instance of the Composite design pattern. These classes are visited by classes in Figure 15.

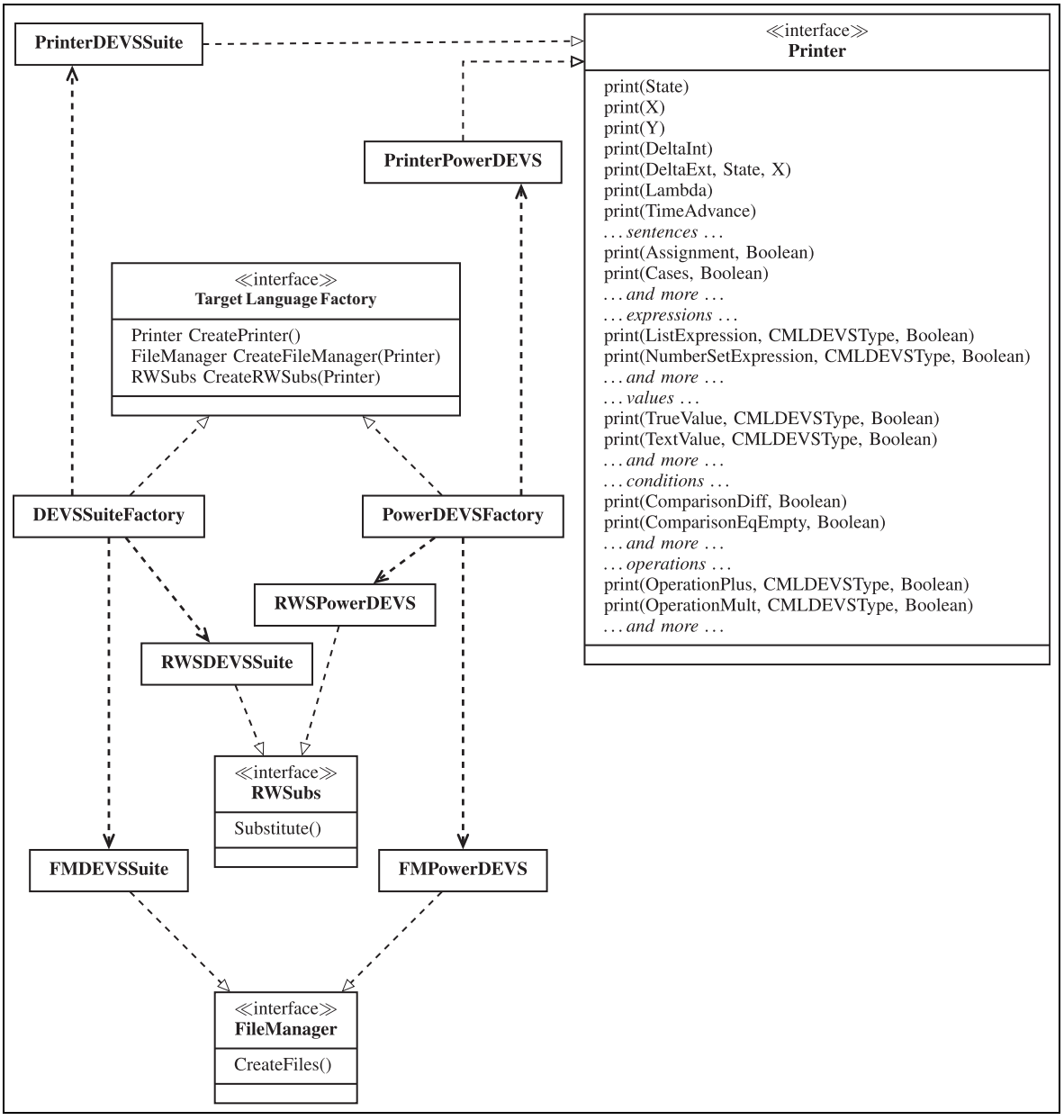

Figure 17. The AbstractFactory design pattern is used to create target-language-dependent components.

Table 2. Brief description of the atomic DEVS models used for the evaluation of the CML-DEVS compiler.

Model	Description
ACCtrlUnit ACTempProp CoolUnit	These are three atomic models of an air-conditioning system with cooling and heating units. The user can set the desired temperature while the system works to maintain this temperature in the room.[14]
ATMVerif	This is one atomic model part of a simple ATM machine. The ATM is only capable of dispensing money to a customer. ATMVerif verifies that the required amount is covered in the balance.[16]
BilliardBall	This models the movement of a billiard ball on a two-dimensional pool table. The ball is struck by a cue (external event), it heads off in a direction at constant speed determined by the impulsive force imparted to it by the strike. Hitting the side of the table is considered as another input that sets the ball going in a well-defined direction.[13]
BinaryCounter	In this model, the system outputs a "one" for every two "one"s that it receives. To do this it maintains a count of the "one"s it has received so far. When it receives a "one" that makes its count even, it goes into a transitory phase, "active," to generate the output.[13]
Constant	This is the simplest of our models since it just outputs once a given constant and then remains idle forever (PowerDEVS model library).
ElevatorDoor ElevatorEngine	These two models are part of a coupled model describing an elevator in a one-elevator building. ElevatorDoor describes the behavior of the elevator's door, and ElevatorEngines describes the behavior of its engine.[17]
Generator	Describes a simple proactive system. It has no inputs but when started in phase "active," it generates outputs with a specific period.[13]
HInt	Models a hysteretic quantized integrator which is used in continuous system simulation.[18]
TrafficLights	This atomic model describes the behavior of two traffic lights in an intersection. These traffic lights have two modes of operation: autonomous, in which the lights behave as expected; and manual, in which the lights blink yellow. There is some external mechanism that switches between modes by sending two events.[15]
Server	This model describes a simple processing server. The server receives jobs to be executed during a user-defined period of time. The server keeps a queue of pending jobs.[19]
Switch	A switch is modeled as a system with pairs of input and output ports. When the switch is in the standard position, jobs arriving on port "in" are sent out on port "out," and similarly for ports "in1" and "out1." When the switch is in its other setting, the input-to-output links are reversed.[13]

Funding

This research was partially funded by CONICET under a post-doctoral grant and by ANPCyT under PICT 2014-2200.

Notes

1. That is, an interface or a class parametrized by a type.
2. www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS

References

1. Zeigler BP, Kim TG and Praehofer H. *Theory of modeling and simulation*. Orlando, FL: Academic Press, 2000.
2. Cho HJ and Cho YK. *DEVS-C++ reference guide*. Tucson, AZ: University of Arizona, 1997.
3. Kim TG. *DEVSim++ user's manual. C++ based simulation with hierarchical modular DEVS Models*. Daejeon: Korean Advanced Institute of Science and Technology 1994.
4. Wainer GA. CD++: a toolkit to develop DEVS models. *Softw Pract Exper* 2002; 32(13): 1261–1306.
5. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87(1–2): 113–132.
6. Filippi J and Bisgambiglia P. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. *Environ Modell Software* 2004; 19(3): 261–274.
7. Kim S, Sarjoughian HS and Elamvazhuthi V. DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring. In: Wainer GA, Shaffer CA, McGraw RM, et al. (eds), *Proceedings of the 2009 spring simulation multiconference, SpringSim 2009*, San Diego, March 22–27, 2009. New York: SCS/ACM, 2009.
8. Hamri MEA and Zacharewicz G. LSIS-DME: an environment for modeling and simulation of DEVS specifications. In: *AIS-CMS international modeling and simulation multiconference*, Buenos Aires, Argentina, 8–12 February 2007. pp.55–60. San Diego, CA: The Society for Modeling & Simulation International.
9. Hollmann DA, Cristiá M and Frydman C. CML-DEVS: a specification language for DEVS conceptual models. *Simul Modell Pract Theory* 2015; 57: 100–117.
10. Spivey JM. *The Z notation: a reference manual*. Hemel Hempstead: Prentice Hall, 1992.
11. Abrial JR. *The B-book: assigning programs to meanings*. New York, NY: Cambridge University Press, 1996.
12. Lamport L. Specifying systems: *The TLA+ language and tools for hardware and software engineers*. Boston, MA: Addison-Wesley Longman Publishing, 2002.

13. Zeigler BP and Sarjoughian HS. *Introduction to DEVS modeling and simulation with Java: developing component-based simulation models*, 2003. Tempe, AZ: Arizona Center of Integrative Modeling and Simulation.
14. Fal L and Vasconcelos G. Simulation of discrete event systems: course assignment 1. wainer/wbgraf/samples/airconditionPARALLEL.zip (2004, accessed 20 March 2018).
15. Vangheluwe H. The Discrete EVent System specification (DEVS) formalism. http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/notes.DEVS.pdf (accessed 20 March 2018).
16. Saadawi H. SYSC-5807: Methodological aspects of modeling and simulation – course assignment 1. www.sce.carleton.ca/faculty/wainer/wbgraf/samples/atm.zip (accessed 20 March 2018).
17. Herrero G. Simulation of discrete event systems: course assignment 1. www.sce.carleton.ca/faculty/wainer/wbgraf/samples/Elevator.zip (accessed 20 March 2018).
18. Cellier FE and Kofman E. *Continuous system simulation*. Secaucus, NJ: Springer, 2006.
19. Wainer G. SYSC-5104: Methodologies for discrete-event modelling and simulation. www.sce.carleton.ca/courses/sysc-5104/materials/private/Lecture5.ppt (accessed 20 March 2018).
20. Gu F. CSC 754: System simulation topics. www.cs.csi.cuny.edu/_gu/teaching/courses/csc754/csc754.html (20 March 2018).
21. Strachey C. Fundamental concepts in programming languages. *Higher-Order and Symbolic Comput* 2000; 13(1/2): 11–49 (accessed 20 March 2018).
22. Parr T. *The definitive ANTLR 4 reference*. Frisco, TX: Pragmatic Programmers, LLC, 2013.
23. Gamma E, Helm R, Johnson R, et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley Longman Publishing, 1995.
24. Parr TJ. Enforcing strict model-view separation in template engines. In: Feldman SI, Uretsky M, Najork M, et al. (eds) *Proceedings of the 13th international conference on World Wide Web, WWW 2004*, New York, USA, May 17–20, 2004, pp.224–233. New York: ACM, 2004.
25. DEVS Standardization Group. http://cell-devs.sce.carleton.ca/devsgroup (accessed 20 March 2018).
26. Wainer GA, Al-Zoubi, K Hill DRC, et al. *Discrete-event modeling and simulation: theory and applications*. London: Taylor & Francis, 2010.
27. Touraille L, Traoré MK and Hill DRC. A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models. In: Wainer GA, Shaffer CA, McGraw RM, et al. (eds) *Proceedings of the 2009 spring simulation multiconference, SpringSim 2009*, San Diego, USA, March 22–27, 2009. New York: SCS/ACM, 2009.
28. Hong KJ and Kim TG. DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems. *Inf Softw Technol* 2006; 48(4): 221–234.
29. Mittal S and Douglass SA. DEVSML 2.0: the language and the stack. In: Wainer GA and Mosterman PJ (eds) *2012 spring simulation multiconference, SpringSim '12*, Orlando, USA, March 26–29, 2012, p.17. New York: SCS/ACM; 2012.
30. Fishwick PA. XML-based modeling and simulation: using XML for simulation modeling. In: Snowdon JL and Charnes JM (eds) *Proceedings of the 34th winter simulation conference: exploring new frontiers*, San Diego, USA, December 8–11, 2002, pp.616–622. Piscataway, NJ: IEEE.
31. Röhl M and Uhrmacher AM. Flexible integration of XML into modeling and simulation systems. In: *Proceedings of the 37th winter simulation conference*, Orlando, USA, December4–7, 2005, pp.1813–1820. Piscataway, NJ: IEEE.
32. Sarjoughian HS and Chen Y. Standardizing DEVS models: an endogenous standpoint. In: Wainer GA, Traoré MK, Heckel R, et al. (eds) *2011 spring simulation multi-conference, SpringSim '11*, Boston, USA, April 3–7, 2011, Volume 4, pp.266–273. New York: SCS/ACM, 2011.
33. Cristiá M. Formalizing the semantics of modular DEVS models with temporal logic. In: *7ème Conférence on Modélisation, Optimisation et Simulation des Systèmes MOSIM 08*, Paris, 31 March–2 April 2008. Paris: Supméca-LISMMA, ENSTIB and CRAN.
34. Hatton L and Roberts A. How accurate is scientific software? *IEEE Trans Software Eng* 1994; 20(10): 785–797.
35. Hatton L. *The chimera of software quality. IEEE Computer* 2007; 40(8): 102–104.
36. Post DE and Votta LG. Computational science demands a new paradigm. *Phys Today* 2005; 58(1): 35–41.
37. Joppa LN, McInerny G, Harper R, et al. Troubling trends in scientific software use. *Science* 2013; 340(6134): 814–815.
38. Brambilla M, Cabot J and Wimmer M. Model-driven software engineering in practice: synthesis lectures on software engineering. Williston, VT: Morgan & Claypool Publishers, 2012.
39. Vangheluwe H. Foundations of modelling and simulation of complex systems. *ECEASST* 2008; 10.
40. Kühne T, Mezei G, Syriani E, et al. Systematic transformation development. *ECEASST* 2009; 21.
41. Çetinkaya D, Verbraeck A and Seck MD. Model continuity in discrete event simulation: a framework for model-driven development of simulation models. *ACM Trans Model Comput Simul* 2015; 25(3): 17.
42. Cetinkaya D, Verbraeck A and Seck MD. Applying a model driven approach to component based modeling and simulation. In: *Proceedings of the 2010 winter simulation conference, WSC 2010*, Baltimore, USA, December 5–8, 2010, pp.546–553. WSC, 2010. Piscataway, NJ: IEEE.
43. Cetinkaya D, Verbraeck A and Seck MD. A metamodel and a DEVS implementation for component based hierarchical simulation modeling. In: McGraw RM, Imsand ES and Chinni MJ (eds) *Proceedings of the 2010 spring simulation multiconference, SpringSim 2010*, Orlando, USA, April 11–15, 2010, p.170. New York: SCS/ACM, 2010.
44. Cetinkaya D and Verbraeck A. Metamodeling and model transformations in modeling and simulation. In: Jain S Jr RRC, Himmelspach J, White KP, et al. (eds) *Winter simulation conference 2011, WSC'11*, Phoenix, USA, December 11–14, 2011, pp. 3048–3058. Piscataway, NJ: IEEE.
45. Touraille L. Application of model-driven engineering and metaprogramming to DEVS modeling & simulation. Doctoral dissertation, Université d'Auvergne, France, 2012.

46. Wainer GA, Shaffer CA, McGraw RM, et al. (eds). *Proceedings of the 2009 spring simulation multiconference, SpringSim 2009*, San Diego, CA, March 22–27, 2009. New York: SCS/ACM, 2009.

Author biographies

Maximiliano Cristiá is a professor of software Engineering at Universidad Nacional de Rosario (Argentina). He is also head of the software Engineering research group at CIFASIS (Argentina) and associated researcher at LIS (Marseilles, France). He received an MSc in mathematics in 1993 from Universidad Nacional de Rosario; an MSc in computer science in 2002 from Universidad de la República (Uruguay); and a PhD from Aix-Marseille Université (France) in 2012. His research interests include: modeling and simulation, formal methods, formal verification, and software design.

Diego Hollmann received his MSc and PhD in computer science in 2009 and 2015, respectively, both from Universidad Nacional de Rosario (Argentina). He currently works as an external contractor for iRobot Corp., a world-class, Massachusetts-based robotics company.

Claudia Frydman is a researcher at Laboratoire d'Informatique et Systèmes and professor of computer science at Aix-Marseille Université (France). She holds an MSc in computing from Universidad de Buenos Aires (Argentina), a PhD from Université de Montpellier, and in 1998 she got the Habilitation á diriger des Recherches. Dr. Frydman has supervised many PhD students and has led a number of projects with high-level industrial partners.