A multi-target compiler for DEVS

M. Cristiá[a,*], D.A. Hollmann[b,*], C.S. Frydman[c,*]

[a] *CIFASIS and UNR, Rosario, Argentina*
[b] *CIFASIS-CONICET, Rosario, Argentina*
[c] *Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296,13397, France*

Abstract

Discrete Event System Specification (DEVS) is a modular and hierarchical formalism for system modeling and simulation. Modeling should be performed by using an abstract language; simulation is performed by tools called *concrete simulators*. Each concrete simulator has its own input language which, in the case of atomic DEVS models, it is, essentially, a general-purpose programming language (such as Java or C++). Hence, once engineers have written an abstract model, they need to manually translate them 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, an abstract DEVS modeling language. This multi-target compiler is able to compile a CML-DEVS model to the input language of the PowerDEVS and DEVS-Suite concrete simulators. In this way, the CML-DEVS compiler frees engineers from the manual translation of their abstract models. Besides, the same abstract model can be simulated on many concrete simulators by simply re-compiling the model.

Keywords: DEVS, compiler, CML-DEVS

1. Introduction

Discrete Event System Specification (DEVS) is perhaps the most general and used modeling and simulation (M&S) formalism [1]. In 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 two languages to its users: *a*) a language to compose atomic or coupled models into coupled models, that can be called *structuring language*; and *b*) a programming language to program atomic models, which in general is the same programming language of the concrete simulator.

Structuring languages are easy to use as they frequently rest on some sophisticated graphical user interface (GUI) that allows engineers to graphically compose their atomic and coupled models. Indeed, these languages let engineers not in the habit of programmig, to compose their models as they learned in textbooks. They also learned that DEVS atomic models should be described in the standard language of mathematics by using equations, functions, sets, etc. However, when they want to simulate these atomic models they need to program them in the input language of a concrete simulator, which means to write code in Java or C++ or other general-purpose programming language. Or else, they need to ask to a programmer to do that. Furthermore, if they want to try out different concrete simulators they need to re-implement their models for each of them. The process of translating the abstract model (i.e. written in mathematics) to the input language of a concrete simulator, may induce errors that would render the simulation activity not as accurate as it should be.

*Corresponding author

For these reasons, we developed CML-DEVS [9], a DEVS specification language based on standard mathematics and inspired in formal notations such as Z [12], B [13] and TLA+ [14], 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 with the structuring languages provided by concrete simulators. 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 compiler can be compiled as indicated by the concrete simulator in order to simulate it. Therefore, the combination of CML-DEVS plus its multi-target compiler relief engineers from the error-prone, annoying task of translating their abstract models into concrete models. CML-DEVS plus its multi-target compiler let engineers to think in terms of mathematics and to use several different concrete simulators for simulating the same model.

In this first version the compiler produces PowerDEVS [5] and DEVS-Suite [10, 7] code, that is, essentially, C++ and Java code, respectively. However, we show how it can be easily 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, 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 intended to be a proof-of-concept tool, 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 to the DEVS community an alternative, complementary technology for modeling atomic DEVS models. In spite of this, we encourage the DEVS community to try the current version of the CML-DEVS compiler as it is a fully functional program providing a new way of writing DEVS atominc models.

The CML-DEVS compiler can be freely downloaded from `http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS`.

The paper is structured as follows. In Section 2 we introduce, by means of a class-room example, the CML-DEVS specification language, assuming the reader is familiar with DEVS (otherwise refer to [1]). The CML-DEVS multi-target compiler is described in Section 3 where we comment on key design decisions that guided us towards its implementation. An empirical evaluation of the compiler is presented in Section 4. This evaluation consists in collecting fourteen 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.

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 essentially the same than writing standard mathematics or the same than writing atomic models as in DEVS textbooks or as in the classroom. An 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 process. Therefore, we pick an atomic model written as in DEVS textbooks and present the CML-DEVS code for it. The chosen model is used by Hans Vangheluwe (professor at McGill University's School of Computer Science) in his course "Modelling of Software-Intensive Systems". The model can be found in his class notes [11] and in Figure 1. 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.

We are not going to delve into the behavior of this model as it is quite simple and we assume readers are familiar with the DEVS formalism. Instead, we want to emphasize the fact that Figure 1 is a mathematical,

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

$T = \mathbb{R}$
$X = \{M, A\}$
$\omega : T \to X \cup \{\phi\}$
$S = \{RG, RY, GR, YR, BB\}$
$\delta_{int}(RG) = RY; \delta_{int}(RY) = GR$
$\delta_{int}(GR) = YR; \delta_{int}(YR) = RG$
$ta(RG) = 60s; ta(RY) = 10s$
$ta(GR) = 50s; ta(YR) = 10s$
$ta(BB) = +\infty$

$\delta_{ext}((RG, e), M) = BB$
$\delta_{ext}((RY, e), M) = BB$
$\delta_{ext}((GR, e), M) = BB$
$\delta_{ext}((YR, e), M) = BB$
$\delta_{ext}((BB, e), M) = RY$
$Y = \{GREEN, YELLOW, BLINK\}$
$\lambda(RG) = \lambda(RY) = \lambda(GR) = GREEN$
$\lambda(YR) = YELLOW$
$\lambda(BB) = BLINK$

Figure 1: Professor Vangheluwe's atomic DEVS model of two traffic lights

abstract, simulator-independent description of a DEVS atomic model. In other words, we claim that people with a similar background to Professor Vangheluwe would agree in that Figure 1 represents a typical textbook description of a DEVS atomic model.

In turn, Figure 2 shows a pretty-printing of the CML-DEVS source code shown in Figure 3 corresponding to Vangheluwe's traffic lights atomic model. Note that Figure 2 is, essentially, a mathematical formula much like Vangheluwe's formulation of the same model (i.e. Figure 1). It rests on equations, functions and set theory, with no influence whatsoever from a general-purpose programming language. Information not quite standard included in Figure 1 is not present in Figure 2—i.e., $T = \mathbb{R}$ and $\omega : T \to X \cup \{\phi\}$. In fact the main differences between both figures reside in the way functions are given: Vangheluwe gives them through function application expressions while Figure 2 uses definitions by cases. Actually, definitions by cases are more general than application expressions, which can only be used for discrete and small functions. Observe that, in this particular case, these definitions by cases could be automatically translated into function application expressions. Another difference is that CML-DEVS follows Zeigler's et al. definition of atomic models where inputs (X) and outputs (Y) are associated to ports [1], whereas Vangheluwe's formulation omits this feature. Including port information in atomic models prepares them to be easily coupled with other models. Probably, Professor Vangheluwe did not include this information for pedagogical issues, keeping the formalism as simple as possible. The inclusion of port information makes it possible to couple the model but it also makes it necessary to indicate from (to) what port an event (output) comes (goes) from (to). This is represented by CML-DEVS reserved word value (see δ_{ext} definition). In CML-DEVS each event (output) is an ordered pair of the form $(port, value)$, where *value* is the value received (sent) in port *port*. Hence, when a transition is defined by the arrival of a value, regardless of the port at which it arrives, the second component of the pair is represented by value.

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 engineers write their abstract models. Although it resembles the form of a program it is important to make the following observations:

1. It is based on logic, set theory, equations and function definitions, which is hardly the case of, for instance, Java or C++ programs;
2. There are no side effects as it is a declarative language enjoying *referencial transparency* [15], which makes it to be closer to the language of mathematics and farther from imperative programming languages;
3. This source code can be generated by, for example, a formula editor featuring a rich graphical user interface, thus relieving engineers from learning the syntax of CML-DEVS; and
4. This source code can be automatically pretty-printed as in Figure 2, by developing a simple translation tool producing LaTeX or XML code as used by word processors or engineering graphical tools.

$$TrafficLights = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \text{ where}$$
$$X = \{(in, x) : x \in \{M, A\}\}$$
$$S = \{RG, RY, GR, YR, BB\}$$
$$\delta_{int}(s) = \begin{cases} RY & \text{if } s = RG \\ GR & \text{if } s = RY \\ YR & \text{if } s = GR \\ RG & \text{if } s = YR \end{cases}$$
$$ta(s) = \begin{cases} 60 & \text{if } s = RG \\ 10 & \text{if } s \in \{RY, YR\} \\ 50 & \text{if } s = GR \\ +\infty & \text{if } s = BB \end{cases}$$
$$Y = \{(out, y) : y \in \{GREEN, YELLOW, BLINK\}\}$$
$$\delta_{ext}(s, \mathsf{value}) = \begin{cases} BB & \text{if } s \in \{RG, RY, GR, YR\} \wedge \mathsf{value} = M \\ RY & \text{if } s = BB \wedge \mathsf{value} = A \end{cases}$$
$$\lambda(s) = \begin{cases} (out, GREEN) & \text{if } s \in \{RG, RY, GR\} \\ (out, YELLOW) & \text{if } s = YR \\ (out, BLINK) & \text{if } s = BB \end{cases}$$

Figure 2: Pretty-printing of the CML-DEVS source code for Vangheluwe's traffic lights atomic model

3. 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.

The CML-DEVS compiler is a Java program based on a standard one-pass compiler design and on the ANTLR parser generator [16]. 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 PowerDEVS [5] and DEVS-Suite [10, 7] 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 including comments (15 KLOC of pure Java code).

The CML-DEVS grammar informed in [9] was written in the grammar language supported by ANTLR. In this way, ANTLR generated the *lexical analyzer* or *scanner* and the *syntax and semantic analyzer* or *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 a concrete syntax tree (CST) of the CML-DEVS model. This CST is a central data structure as it organizes the model being compiled as a tree structure. The CST 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 who, 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

```
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
    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
    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
    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
    Y is
        out : {GREEN, YELLOW, BLINK}
    end Y
    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
end atomic
```

Figure 3: CML-DEVS source code for Vangheluwe's traffic lights atomic model

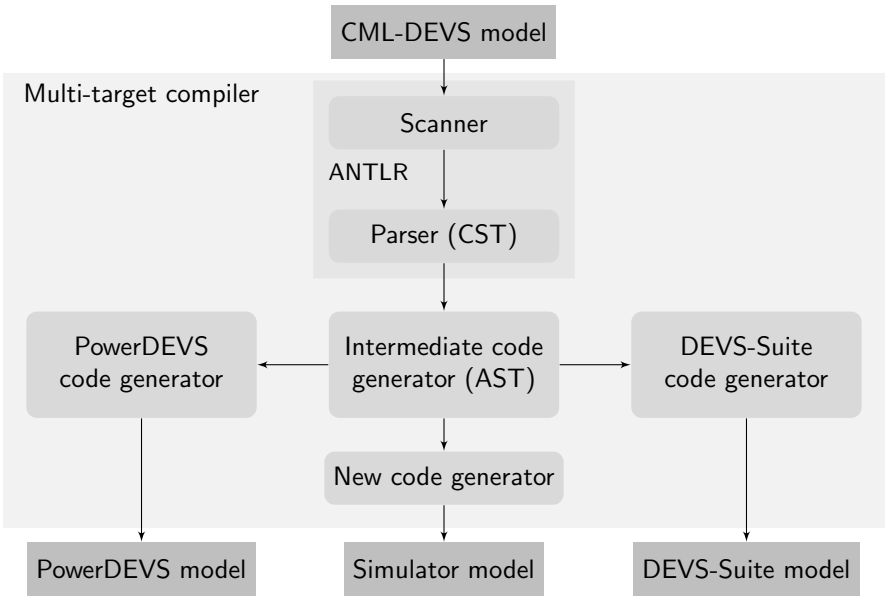

Figure 4: Descriptive block diagram of the CML-DEVS multi-target compiler

classes provides only syntactic information. For instance, in this phase is not possible to know what is the type of each expression. ANTLR organizes these classes according to the Composite design pattern [17, ch. 4–Composite], which allows a uniform access to the structure. In particular, an object structure adhering to a Composite can be analyzed by implementing the Visitor design pattern [17, ch. 5–Visitor]. This combination of design patterns facilitates the implementation of several key functions of the compiler.

One of the key functions of the semantics analysis implemented by a Visitor is type checking and inference. In turn, this is one of the CML-DEVS characteristics that increases its abstraction level moving it away from programming languages and getting it closer to the language of mathematics. There are three important issues in the CML-DEVS type system:

- It works with types not usually found in the input languages of concrete simulators (e.g. sets and functions);
- It avoids mistakes that surface when the model is simulated and are hard to find;
- It provides an easy way of defining sets that mix elements that usually belong to different types (e.g. $\mathbb{R} \cup \{\infty\}$). These sets are frequently used by the DEVS community when describing abstract models.

Although in this first version type checking is not fully implemented, class TypeChecker is a Visitor implementing type inference. Every heir of class Expression is visited by TypeChecker in order to infer the type of the expression. These inferred types are used to find out the type of expressions in the target language, and will later be used to perform type checking.

Besides, ANTLR automatically generates a template[1] Visitor interface (CMLDEVSVisitor) specifically tailored to analyze the CST generated after the parsing phase. Currently, the CML-DEVS compiler implements this interface with a set of classes headed by CMLDEVSBaseVisitor whose function is to generate an abstract syntax tree (AST) containing semantic information about the model. The implementation of CMLDEVSBaseVisitor requires to set the template parameter with the various classes representing elements of the intermediate code. In this way, it can be said that the implementation of CMLDEVSBaseVisitor represents the *intermediate code generator* (cf. Figure 4).

[1]That is, an interface or a class parametrized by a type.

The AST generated by the implementation of CMLDEVSBaseVisitor is organized as a Composite design pattern headed by the CMLDEVSData interface. Each node in the AST contains information such as the semantic role played by each syntax element and the type of expressions. For example, in the AST the *ta* node of the CST 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 information is given by storing it in classes whose types convey this semantic information. Note that at the AST level there are concepts that have no direct implementation in an imperative programming language. For example, δ_{ext}, δ_{int}, etc, as expressed in CML-DEVS are mathematical functions according to its semantics, which needs a non trivial implementation in an imperative programming language. However, this intermediate language makes it really simple to generate code for each concrete simulator.

3.1. Code generation

Carefully designing the code generation phase (cf. 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 and simple and easy to add.

When calling the CML-DEVS compiler users must pass a parameter telling to what simulator language the compilation has to be done. 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 [10, 7] 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. Then, 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 to reserve more words at the CML-DEVS level because this would mean to collect 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 [17, ch. 3–Abstract Factory]. Hence, the CML-DEVS compiler defines TargetLaguageFactory, an interface for instantiating objects that depend on the target language.

Target code generation (i.e. the first responsibility listed above) is organized according to the Visitor design pattern [17, ch. 5–Visitor]. These visitors visit the Composite that structures the AST headed by CMLDEVSData (i.e. the intermediate representation) and *print* 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 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, or any other formatted text output, developed by ANTLR's designer [18].

Therefore, implementing the code generator for PowerDEVS (respect. DEVS-Suite) implies to provide, among others, a heir of TargetLaguageFactory, called PowerDEVSFactory (respect. DEVSSuiteFactory), and an implementation of Printer, called PrinterPowerDEVS (respect. PrinterDEVSSuite). We will focus on PrinterPowerDEVS as PrinterDEVSSuite is very similar, and printers are the most interesting components of code generation. Implementing PrinterPowerDEVS entails to define a StringTemplate template and implement some of its methods by calling StringTemplate. Figure 6 shows code snippets of the implementation of three

```
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

methods of PrinterPowerDEVS, and Figure 7 shows an excerpt of the template. As can be seen, the template consists in the basic structure of the code to be generated with place holders that are replaced each time the template is used. The replacement can be done with a library provided by StringTemplate. The place holders are replaced with the actual data taken from the intermediate representation. 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 a heir of TargetLaguageFactory 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 is one of the simplest forms of code generation, thus allowing a 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 (cf. Figure 4) is marginal with respect to the total effort (recall that currently the CML-DEVS compiler is about 20 KLOC), as it is otherwise expected if proved 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.

4. Examples and empirical evaluation

In this section we show some examples of the application of the CML-DEVS compiler. At the same time we use these examples as the basis for an empirical evaluation of the whole proposal (i.e. the CML-DEVS modeling language and its multi-target compiler).

We have written in CML-DEVS and compiled to PowerDEVS and DEVS-Suite 14 DEVS atomic models, including the example of the traffic lights shown in Section 2. In order to use these models as the basis for an empirical evaluation of our proposal we took them from third-party resources such as books, web sites and courses. In other words, these models were not proposed by us which would have biased the evaluation. What we have done, though, is to translate them from the mathematics or semi-formal descriptions used by their authors into CML-DEVS specifications. In doing so we tried to follow the mathematical structure

```
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

Model	Source	T	Size (in bytes)			Time
			CML-DEVS	PowerDEVS	DEVS-Suite	
ACCtrlUnit	Wainer's sample of DEVS models [19]	D	2,422	4,245	4,946	2s
ACTempProp	Wainer's sample of DEVS models [19]	D	1,311	2,903	3,669	2s
ATMVerif	Wainer's sample of DEVS models [20]	D	961	2,584	2,991	2s
BilliardBall	Zeigler and Sarjoughian [10]	D	735	2,389	2,945	1s
BinaryCounter	Zeigler and Sarjoughian [10]	J	631	2,104	2,557	2s
ConstGen	PowerDEVS model library	C	335	1,415	1,800	1s
CoolUnit	Wainer's sample of DEVS models [19]	D	746	2,178	2,692	2s
ElevatorDoor	Wainer's sample of DEVS models [21]	D	1,440	3,262	3,964	2s
ElevatorEngine	Wainer's sample of DEVS models [21]	D	1,755	3,464	4,323	2s
Generator	Zeigler and Sarjoughian [10]	J	384	1,491	1,861	1s
HInt	Cellier and Kofman [22]	C	1,196	2,771	3,312	1s
TrafficLights	Vangheluwe's class notes [11]	D	1,051	2,713	3,388	1s
Server	Wainer's course material [23]	D	799	2,372	3,046	2s
Switch	Zeigler and Sarjoughian [10]	D	1,045	2,810	3,453	2s

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

suggested by each author. Table 1 provides some elemental data of each model, while all the experimental data and the CML-DEVS compiler are publicly available[2]. The first column simply identifies the model; column Source gives the origin and a reference to the authors of the model; column T indicates whether the CML-DEVS specification was written from a DEVS description (D) or from the source code of an atomic PowerDEVS (C) or DEVS-Suite (J) model; 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; finally, column Time is the approximated compilation time (of both PowerDEVS and DEVS-Suite since the differences are negligible).

As can be seen from Table 1, the examples cover a wide range of applications, origins and authors, thus representing a reasonable sample. 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 of 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 course "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 himself in one of his class presentations. That is, there are models written by experts and students as well. In fact, some of the students' models have to be slightly modified or corrected as they contain errors such as, for instance, modifications of the state inside the output function (λ). The models listed in Table 1 where compiled on the following platform: AMD Athlon(tm) 7850 Dual-Core Processor CPU at 1.40GHz 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.

[2] `http://www.cifasis-conicet.gov.ar/hollmann/projects/CML-DEVS`

The compilation times shown in the table are approximate and rounded. As the table shows, times are reasonable, furthermore considering that the current version of the CML-DEVS compiler is, above all, a proof-of-concept. It is also clear that the sizes of the compiled models are higher than the CML-DEVS specifications, which is an indication of the syntactic complexity of the formers with respect to the latter. Complexity that is hidden to engineers using this approach.

Precisely, as an example of the code generated by the CML-DEVS compiler, Figure 8 lists the result of compiling the traffic lights model (Figure 3) to the PowerDEVS input language (in the Appendix, Figure A.13 lists the result of compiling the same model to DEVS-Suite). As the figure 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 library auxFunc which in turn is made available to the PowerDEVS model. PowerDEVS' users would have to write their own set manipulation functions if they would have implemented the model without the CML-DEVS compiler. Instead, by using the compiler, they can simply write s in $\{RG, RY, GR\}$ and let the compiler to implement it. Last, but not least, compare the simplicity, familiarity and cleanness of the CML-DEVS source code of Figure 3 with respect to the C++ code of Figure 8. For example, in the former there are no things such as casts and pointers (i.e. programming, not modeling, concepts), which are necessary in the latter. We argue that the *model* of Figure 3 can be written by an engineer completely unaware of C++, which is not the case for the *program* of Figure 8.

Finally, we would like to comment on examples BinaryCounter, ConstGen, Generator and HInt. They were written from PowerDEVS (ConstGen and HInt) and DEVS-Suite (BinaryCounter and Generator) source code rather than from pure DEVS models. We did so to compare the code generated by the CML-DEVS compiler with respect to the code written by PowerDEVS and DEVS-Suite expert users. Model HInt is an *hysteretic quantized integrator* which is used in continuous system simulation, as defined by Cellier and Kofman [22]. Model ConstGen is a very simple model that generates a constant. BinaryCounter is a model producing an output for every two inputs that are received. Generator is an example of a proactive system which produces output at a regular pace. 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. Figure 9 lists the PowerDEVS code of HInt as proposed by Cellier and Kofman [22, p. 545]. In turn, the CML-DEVS code is in Figure 11 and the result of compiling it is in Figure 10. As can be seen, both PowerDEVS programs are similar in size, structure and functionality. Furthermore, in Figure 12 we can see the results of using both implementations (i.e. Figures 9 and 10) 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 with respect to the original model.

We believe that 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.

Therefore, we foresee the following simple integration strategy of the CML-DEVS multi-target compiler within these simulators:

1. Write CML-DEVS code with some programming editor. Ideally, a CML-DEVS editor, such as a formula editor, can be developed and integrated into existing simulation environments.
2. Compile each CML-DEVS model into the input language of the simulator you are using. Here the CML-DEVS editor can call the compiler.

```cpp
#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 8: Result of compiling to PowerDEVS input language (C++) the traffic lights model shown in Fig. 3

```
#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 9: 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 10: PowerDEVS (C++) implementation of HInt resulting from compiling the CML-DEVS model of Figure 11

```
atomic HInt is ⟨X, S, Y, dint, dext, lamda, ta⟩ where
    S is
        xS : R
        dX : R
        q : R
        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 11: CML-DEVS source code for Cellier and Kofman's HInt

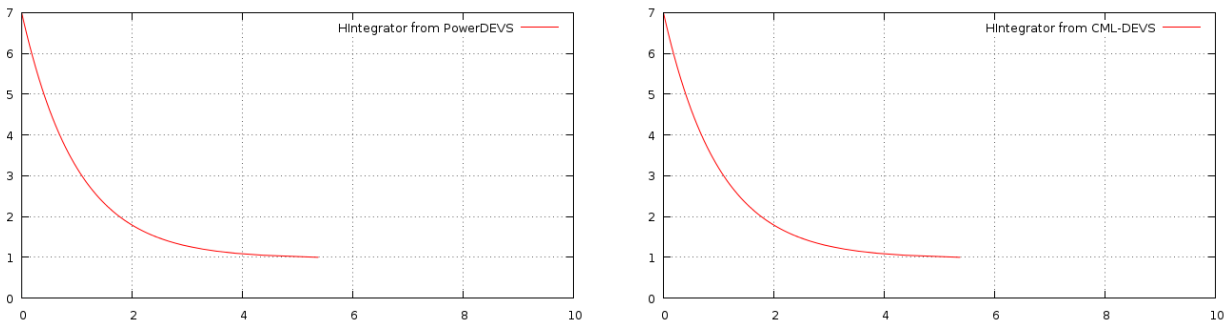

Figure 12: Plot of the curves obtained by simulating the model given in Figure 9 (left) and Figure 10 (right)

3. Save the compiled model as any other atomic model of the simulator. CML-DEVS compiled models are indistinguishable from atomic models compiled by other means.
4. Now users can couple compiled CML-DEVS models with other models as is normally done in this particular simulator (i.e. 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 the 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 regarding the automatic generation of atomic DEVS models. However, there are some works that in a 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 on by the DEVS community [24]. Within the standardization areas identified by this group [25, 26, 27], model representation could be approached with notations such as CML-DEVS. In particular, 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 by Hollmann et al. elsewhere [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 [30, 31, 27, 32]. One of the reasons is that XML is platform independent. We believe that XML bears no relation with the notion of abstract model as we understand it, i.e. in the sense to its 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 the formal notations used in software engineering such as Z [12], B [13] and TLA+ [14]. For example, the semantics of DEVS can be formalized in TLA+ [33] as this notation is based on temporal logic which provides the semantics for reactive systems. Engineering and scientific software tend to have many errors that turn decision-making based on them risky [34, 35, 36]. Researchers and engineers use software that has not been verified [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) attempts 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]. Note that the final goal of MDE applied to DEVS models and the CML-DEVS approach is the same. In this setting, CML-DEVS would be the modeling language used to describe an abstract model and the CML-DEVS compiler would be a model transformation. However, in the MDE community formal specification and compiling techniques are rarely mentioned. On the other hand, the DEVS community has attempted to adopt concepts and techniques from MDE and MDD, in particular there are efforts in defining model transformations [39, 40, 41, 42, 43, 44, 45]. In these approaches, different modeling or metal-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 metal-modeling languages proposed within the DEVS community, instead, are based on general on 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 to write 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 in 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 as 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, what makes 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 design patterns. In fact, plugging-in a new code generator would require no code modification but 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, 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 along a lengthy time period. This 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. Letting errors apart, the 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 rest on a programmer to try out their models. Put it in another way, how much time and effort would need, say, an electric engineer to learn C++ in such a way as to be able to produce the code of Figure 8? And conversely, how much time and effort would (s)he need to learn CML-DEVS, provided (s)he already knows DEVS, in such a way as to be able to produce the code of Figure 3? 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: a) the compiler can be optimized by experts in such a way as to produce the best possible code; and b) once the compiler is proved correct, model translation stops being a source of errors and problems.

Acknowledgments

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

References

[1] B. P. Zeigler, T. G. Kim, H. Praehofer, Theory of Modeling and Simulation, Academic Press, Inc., Orlando, FL, USA, 2000.
[2] H. J. Cho, Y. K. Cho, DEVS-C++ Reference Guide, The University of Arizona (1997).
[3] T. G. Kim, DEVSim++ User's Manual. C++ Based Simulation with Hierarchical Modular DEVS Models., Korea Advance Institute of Science and Technology (1994).
[4] G. A. Wainer, CD++: a toolkit to develop DEVS models, Softw., Pract. Exper. 32 (13) (2002) 1261–1306. doi:10.1002/spe.482.
URL http://dx.doi.org/10.1002/spe.482
[5] F. Bergero, E. Kofman, PowerDEVS: a tool for hybrid system modeling and real-time simulation, Simulation 87 (1-2) (2011) 113–132. doi:10.1177/0037549710368029.
URL http://dx.doi.org/10.1177/0037549710368029
[6] J. Filippi, P. Bisgambiglia, JDEVS: an implementation of a DEVS based formal framework for environmental modelling, Environmental Modelling and Software 19 (3) (2004) 261–274. doi:10.1016/j.envsoft.2003.08.016.
URL http://dx.doi.org/10.1016/j.envsoft.2003.08.016
[7] S. Kim, H. S. Sarjoughian, V. Elamvazhuthi, DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring, in: Wainer et al. [46].
URL http://dl.acm.org/citation.cfm?id=1639809.1655390
[8] 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, 2007, pp. 55–60.
[9] D. A. Hollmann, M. Cristiá, C. Frydman, CML-DEVS: A specification language for DEVS conceptual models, Simulation Modelling Practice and Theory 57 (2015) 100 – 117. doi:http://dx.doi.org/10.1016/j.simpat.2015.06.007.
URL http://www.sciencedirect.com/science/article/pii/S1569190X15001021
[10] B. P. Zeigler, H. S. Sarjoughian, Introduction to DEVS modeling and simulation with Java: Developing component-based simulation models, Tech. rep. (2003).
[11] H. Vangheluwe, The Discrete EVent System specification (DEVS) formalism.
URL http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/notes.DEVS.pdf
[12] J. M. Spivey, The Z notation: a reference manual, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.
[13] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, NY, USA, 1996.
[14] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[15] C. Strachey, Fundamental concepts in programming languages, Higher-Order and Symbolic Computation 13 (1/2) (2000) 11–49. doi:10.1023/A:1010000313106.
URL http://dx.doi.org/10.1023/A:1010000313106
[16] T. Parr, The Definitive ANTLR 4 Reference, Oreilly and Associate Series, Pragmatic Programmers, LLC, 2013.
URL http://books.google.com.ar/books?id=SBXuLwEACAAJ
[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[18] T. J. Parr, Enforcing strict model-view separation in template engines, in: S. I. Feldman, M. Uretsky, M. Najork, C. E. Wills (Eds.), Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, ACM, 2004, pp. 224–233. doi:10.1145/988672.988703.
URL http://doi.acm.org/10.1145/988672.988703
[19] L. Fal, G. Vasconcelos, Simulation of discrete event systems – course assignment 1.
URL http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/airconditionPARALLEL.zip
[20] H. Saadawi, Sysc-5807 - methodological aspects of modeling and simulation – course assignment 1.
URL http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/airconditionPARALLEL.zip
[21] G. Herrero, Simulation of discrete event systems – course assignment 1.
URL http://www.sce.carleton.ca/faculty/wainer/wbgraf/samples/airconditionPARALLEL.zip
[22] F. E. Cellier, E. Kofman, Continuous System Simulation, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[23] G. Wainer, Sysc-5104 – methodologies for discrete-event modelling and simulation.
URL http://www.sce.carleton.ca/courses/sysc-5104/doku.php
[24] DEVS Standardization Group, http://cell-devs.sce.carleton.ca/devsgroup/ (Accessed: 2016-01-18).
[25] G. A. Wainer, K. Al-Zoubi, D. R. Hill, S. Mittal, J. L. R. Martín, H. Sarjoughian, L. Touraille, M. K. Traoré, B. P. Zeigler, Discrete-event modeling and simulation: Theory and applications, Taylor & Francis, 2010, Ch. An Introduction to DEVS Standardization, pp. 393–425.
[26] G. A. Wainer, K. Al-Zoubi, D. R. Hill, S. Mittal, J. L. R. Martín, H. Sarjoughian, M. K. T. Luc Touraille, B. P. Zeigler, Discrete-event modeling and simulation: Theory and applications, Taylor & Francis, 2010, Ch. Standardizing DEVS Model

Representation, pp. 427–458.
URL http://books.google.com.ar/books?id=WQvzk7ZnwHkC

[27] L. Touraille, M. K. Traoré, D. R. C. Hill, A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models, in: Wainer et al. [46].
URL http://dl.acm.org/citation.cfm?id=1639809.1655392

[28] K. J. Hong, T. G. Kim, Devspecl: DEVS specification language for modeling, simulation and analysis of discrete event systems, Information & Software Technology 48 (4) (2006) 221–234. doi:10.1016/j.infsof.2005.04.008.
URL http://dx.doi.org/10.1016/j.infsof.2005.04.008

[29] S. Mittal, S. A. Douglass, DEVSML 2.0: the language and the stack, in: G. A. Wainer, P. J. Mosterman (Eds.), 2012 Spring Simulation Multiconference, SpringSim '12, Orlando, FL, USA, March 26-29, 2012, Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, SCS/ACM, 2012, p. 17.
URL http://dl.acm.org/citation.cfm?id=2346633

[30] P. A. Fishwick, XML-based modeling and simulation: using XML for simulation modeling, in: J. L. Snowdon, J. M. Charnes (Eds.), Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers, San Diego, California, USA, December 8-11, 2002, WSC, 2002, pp. 616–622. doi:10.1109/WSC.2002.1172938.
URL http://dx.doi.org/10.1109/WSC.2002.1172938

[31] M. Röhl, A. M. Uhrmacher, Flexible integration of XML into modeling and simulation systems, in: Proceedings of the 37th Winter Simulation Conference, Orlando, FL, USA, December 4-7, 2005, WSC, 2005, pp. 1813–1820. doi:10.1109/WSC.2005.1574456.
URL http://dx.doi.org/10.1109/WSC.2005.1574456

[32] H. S. Sarjoughian, Y. Chen, Standardizing DEVS models: an endogenous standpoint, in: G. A. Wainer, M. K. Traoré, R. Heckel, J. Himmelspach (Eds.), 2011 Spring Simulation Multi-conference, SpringSim '11, Boston, MA, USA, April 03-07, 2011. Volume 4: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS)., SCS/ACM, 2011, pp. 266–273.
URL http://dl.acm.org/citation.cfm?id=2048511

[33] M. Cristiá, Formalizing the semantics of modular DEVS models with temporal logic, in: 7ème Conférence on Modélisation, Optimisation et Simulation des Systèmes MOSIM 08, 2008.

[34] L. Hatton, A. Roberts, How accurate is scientific software?, IEEE Trans. Software Eng. 20 (10) (1994) 785–797. doi:10.1109/32.328993.
URL http://dx.doi.org/10.1109/32.328993

[35] L. Hatton, The chimera of software quality, IEEE Computer 40 (8) (2007) 104, 102–103. doi:10.1109/MC.2007.292.
URL http://dx.doi.org/10.1109/MC.2007.292

[36] D. E. Post, L. G. Votta, Computational science demands a new paradigm, Physics Today 58 (1) (2005) 35–41.

[37] L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'Hara, D. Gavaghan, S. Emmott, Troubling trends in scientific software use, Science 340 (6134) (2013) 814–815. arXiv:http://science.sciencemag.org/content/340/6134/814.full.pdf, doi:10.1126/science.1231535.
URL http://science.sciencemag.org/content/340/6134/814

[38] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012. doi:10.2200/S00441ED1V01Y201208SWE001.
URL http://dx.doi.org/10.2200/S00441ED1V01Y201208SWE001

[39] H. Vangheluwe, Foundations of modelling and simulation of complex systems, ECEASST 10.
URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/162

[40] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer, Systematic transformation development, ECEASST 21.
URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/290

[41] D. Çetinkaya, A. Verbraeck, M. D. Seck, Model continuity in discrete event simulation: A framework for model-driven development of simulation models, ACM Trans. Model. Comput. Simul. 25 (3) (2015) 17. doi:10.1145/2699714.
URL http://doi.acm.org/10.1145/2699714

[42] D. Cetinkaya, A. Verbraeck, M. D. Seck, Applying a model driven approach to component based modeling and simulation, in: Proceedings of the 2010 Winter Simulation Conference, WSC 2010, Baltimore, Maryland, USA, 5-8 December 2010, WSC, 2010, pp. 546–553. doi:10.1109/WSC.2010.5679131.
URL http://dx.doi.org/10.1109/WSC.2010.5679131

[43] D. Cetinkaya, A. Verbraeck, M. D. Seck, A metamodel and a DEVS implementation for component based hierarchical simulation modeling, in: R. M. McGraw, E. S. Imsand, M. J. Chinni (Eds.), Proceedings of the 2010 Spring Simulation Multiconference, SpringSim 2010, Orlando, Florida, USA, April 11-15, 2010, SCS/ACM, 2010, p. 170.
URL http://dl.acm.org/citation.cfm?id=1878537.1878714

[44] D. Cetinkaya, A. Verbraeck, Metamodeling and model transformations in modeling and simulation, in: S. Jain, R. R. C. Jr., J. Himmelspach, K. P. White, M. C. Fu (Eds.), Winter Simulation Conference 2011, WSC'11, Phoenix, AZ, USA, December 11-14, 2011, WSC, 2011, pp. 3048–3058. doi:10.1109/WSC.2011.6148005.
URL http://dx.doi.org/10.1109/WSC.2011.6148005

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

[46] G. A. Wainer, C. A. Shaffer, R. M. McGraw, M. J. Chinni (Eds.), Proceedings of the 2009 Spring Simulation Multiconference, SpringSim 2009, San Diego, California, USA, March 22-27, 2009, SCS/ACM, 2009.

Appendix A. DEVS-Suite Code of the CML-DEVS Traffic Lights Model

Fig. A.13 lists the Java code resulting from compiling the traffic light model of Fig. 3 with the CML-DEVS compiler choosing Java as the target language. This Java represents an atomic DEVS model of the DEVS-Suite simulator. The code has been edited to make it fit into a single page.

```
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 A.13: Result of compiling to DEVS-Suite input language the traffic lights model shown in Fig. 3