DEVS-Ruby: a Domain Specific Language for DEVS Modeling and Simulation

Romain Franceschini and Paul-Antoine Bisgambiglia and Paul Bisgambiglia and *David Hill

University of Corsica
UMR SPE 6134 CNRS, UMS Stella Mare 3460
TIC team
Campus Grimaldi, 20250 Corti

***Blaise Pascal University**
BP 10448, F-63000 Clermont-Ferrand, France
CNRS, UMR 6158, ISIMA LIMOS
F-63173 Aubiere, France

r.franceschini@univ-corse.fr and bisgambiglia@univ-corse.fr and bisgambi@univ-corse.fr and david.hill@univ-bpclermont.fr

Keywords: Discrete event simulation, DEVS, Domain specific language, DSL

Abstract

This paper introduces a new Discrete EVent system Specification (DEVS) modeling and simulation library implemented in Ruby: a dynamic, reflective, object-oriented programming language freely available for all major platforms. Its syntactic sugar and features such as monkey patching, lexical closures, custom dispatch behavior and native plug-in API provides strong support to grow a Domain Specific Language (DSL). The library, by providing an internal DSL, allows formal specifications of DEVS models. The greatest strength of DEVS-Ruby lies in the extensibility of the DSL, allowing to meet each modeler's domain specific vocabulary and thus, to evolve from a general modeling and simulation formalism to a specialized tool. We also optimize simulations by implementing bottlenecks in native language with the plug-in API. As Ruby gained popularity with web frameworks, it is also particularly suited to expose models as web services, especially since we can feel a momentum towards system interoperability in the DEVS community.

1. INTRODUCTION

In the field of modeling and simulation we find recurring themes as simulation optimization, standardization, model and simulator interoperability, model reusability, modeling languages [1]. Our work should be placed in the latter item as [2]. In this paper, we present a modeling and simulation library based on the Discrete EVent system Specification (DEVS) formalism. It relies on a Domain Specific Language (DSL) to provide a high-level language to help modelers to code and model their systems with an intuitive and domain-specific vocabulary.

Domain-specific languages (DSLs) are tailored to application domains and have definite advantages over general-purpose languages [2–6]. Among the benefits of using a DSL we can name expressiveness and, therefore, increased productivity, as well as ease-of-use, easier model specification verification and optimization [4, 6–10]. Domain experts can directly use the DSL to model their systems. Another important factor is that a DSL requires less effort from an end-user to write valid programs. In this paper, we propose a domain-specific language to describe DEVS models and simulate them.

The discrete event systems specification (DEVS) formalism is based on system engineering principles and is used for modeling and simulation purposes in many application domains. Our main concerns in writing this framework were: (1) fulfill the DEVS formalism specifications; (2) provide scalability through parallel or distributed simulations; and (3) encourage model reusability and simulation configuration with our DSL. In this paper we present a new implementation of the DEVS framework based on the Ruby programming language which focuses on providing a domain-specific language and proposes to choose between several simulation algorithms (classic or parallel). This paper consists of three main components: (1) the background which gives an overview of the state-of-the-art; (2) an overview of the design and implementation of the framework, and (3) an example of how we build a simulation with DEVS-Ruby.

2. BACKGROUND

2.1. Domain-Specific Languages

A Domain-Specific Language (DSL) is a programming or description language tied to a specific application domain. It is designed for a particular kind of problem, and contrasts with a general purpose language which is aimed at any kind of software. DSLs are very common and widely used in computing and includes HTML for document markup, CSS for stylesheets, SQL for database queries or QML and Tcl/Tk for GUI scripting and plenty of graphical languages. In this study, we focus on textual DSLs, compared to equivalent programs written in general purpose languages, programs using DSLs can be much easier to program with. If well-designed they can be used by domain experts who aren't expert programmers because they can use their specific vocabulary to describe their models. DSLs are useful tools - they allow to reduce the gap between computer scientists and domain experts.

A textual DSL can either be external or internal. The former describe a language with its own syntax, which exists independently from any other language. It needs a specific parser and interpreter. CSS or SQL for example, are two good examples of external DSLs. The latter form, also called embedded DSL, rely on the hosting language, it lives inside another programming language, as an enhancement. Both forms have their advantages and drawbacks. Since an internal DSL is written inside a host language, it can be a little less readable than an external DSL as it must conform to a valid syntax within the host language. Conversely, an external DSL is completely independent and can have any syntax, this implies the cost of time and efforts to build a parser and grammar. In practice, a DSL isn't inevitably used in isolation from a general programming language and embedded DSLs already inherits from all features of the host language [4]. Users can take full advantage of the host language, share a common core language and use all available libraries and tools of the host language. In our case, the modeler benefits from a DSL, nevertheless he has to define the logic behind its models. That's why we chose to embed our DSL in Ruby, which is a very convenient language for writing internal DSLs.

Listing 1. Ruby internal DSL used to edit crontabs

```ruby
every 3.hours do
  command "echo 'hello world'"
end
every :monday, :at => '12pm' do
  command "echo 'Have a nice week'"
end
```

The listing 1 is an example of an internal DSL used to edit crontabs. This chunk of code is fully conforming to the Ruby syntax. However, what looks like keywords (”every” and ”command”) are not reserved keywords of the language, instead they are defined as being part of the library's internal DSL. In the next section we will see how we can achieve this in Ruby.

2.2. DEVS Formalism

DEVS, which stands for Discrete EVent system Specification, was originally defined in the seventies as an abstract formalism for discrete-event modeling and simulation [11]. Several discrete-event formalisms were formerly defined such as finite state automata or Petri nets. However, they are limited to systems with a finite number of states. Conversely, DEVS allows representing any system whose input/output behavior can be described with a sequence of events. For a finite time interval, a finite number of events occur. These events may be the source of a state change of the system. DEVS is a modular formalism which encourage model reusability through hierarchical modeling and separation between modeling and simulation.

2.2.1. Models

The DEVS formalism allows defining hierarchical modular models with two distinct types: atomic (behavioral) and coupled (structural) models. The first describes the autonomous behavior of a discrete-event system; the last one is composed of submodels, each of them being an atomic or a coupled model. Both can receive and send messages but these are interpreted only by atomic models which is the behavioral part of a complex system represented by a coupled model. An atomic model can be *passive*, *autonomous* or both (in a superposition of these two behaviors). It is passive if it reacts only when it receives a message and it is autonomous when messages are scheduled to be sent independently. For a given time, a model is in a given state and the behavior of the model depends on transitions between these states. There are two different kind of transitions: external or internal. External transitions occur when an external message is received; internal transitions occur when an internal variable value needs to be changed. After a state change, if the model needs to transmit some message, an output function is activated. Formally, an atomic model is described by the following 7 tuple structure:

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

where
- $X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is a set of input values and ports
- $Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is a set of output values and ports
- S the set of state variables
- $\delta_{ext} : Q \times X \to S$ the external transition function
- $\delta_{int} : S \to S$ the internal transition function
- $\lambda : S \to Y$ the output function
- $ta : S \to \mathbb{R}^{+}_{0,\infty}$ the time advance function
- $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of all states with e the elapsed time since the last transition

A complex system can be designed as a coupling of several simpler systems. The DEVS formalism allows such conception through coupled models, formally defined by the following 8 tuple structure:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle$$

where X the set of input values and ports; Y the set of output values and ports; D the set of components; EIC the set of external input couplings, which links an input port with a component; EOC the set of external output couplings, which links a component with an output port; IC the set of internal couplings, which links two components with each other; the $Select$ function is used to prioritize components supposed to be activated at the same time.

2.2.2. Simulation

To define the simulation semantics of DEVS models, Zeigler introduced abstract simulators. The advantage with such an approach lies in the separation between the models and the simulators. DEVS provides the algorithms used to simulate a model hierarchy. There is a *simulator* for each atomic model, and each coupled model is associated with its *coordinator*. Both adheres to a communication protocol allowing to coordinates the simulation execution between them. The allocation of each component (simulator or coordinator) follows the same hierarchical structure of the models until the root of the tree is found.

We use the DEVS formalism because of its openness and extensibility. It offers both a formal framework to define models and a flexible implementation in object-oriented programming. The next section briefly introduces a selection of several implementations that we compare to ours.

2.3. DEVS Frameworks

There are many environments based on the DEVS formalism. Here we selected a set of environments for which we give a brief description with a highlight of their respective goals.

2.3.1. MS4 Me

MS4 Me [12] is a recent framework developed by Zeigler's team. It allows to design, engineer, visualize and test in a single environment without compromising rigor, quality or performance. It is based on the major part of Zeigler's work.

2.3.2. DEVSimpy

Our team also works on DEVSimpy. DEVSimpy [13] is a framework developed at the University of Corsica which provides a simple graphical user interface based on WxPython to create and use DEVS models. The SPE (Sciences Pour l'Environnement) laboratory of the University of Corsica is specialized in the field of environmental systems modeling based on the DEVS formalism. The main idea of DEVSimpy is to provide a GUI atop of PyDEVS [14], an API that allows hierarchical DEVS models to be defined and simulated in Python.

2.3.3. AToM^3

AToM^3 [15] is a multi-paradigm modeling tool. The concerns in developing this tool were twofold: (1) meta-modeling which refers to the modeling of different kinds of different kinds of formalisms; and (2) model-transforming which refers to the process of converting, translating, or modifying a model in a given formalism, into another model that might not be in the same formalism. This tool relies on model driven engineering concepts and uses the PyDEVS API [14].

2.3.4. PowerDEVS

PowerDEVS [16] is a general purpose software tool for DEVS modeling and simulation oriented to the simulation of hybrid systems written in C++. It allows defining atomic DEVS models which can be graphically coupled. Coupled models are automatically translated into code before executing the simulation.

2.3.5. CD++

CD++ [17] is an platform-independent environment which implements DEVS and Cell-DEVS theory. Different simulation algorithms have been implemented: standalone (single CPU), server mode (the simulator is installed as a server accessible through TCP/IP sockets), real time (the simulator is tied to the real-time clock), embedded (E-CD++; the simulator uses the real-time clock and can be embedded in single-board computers), parallel (over a linux cluster or windows-based PC clusters), distributed (over Web Services).

2.3.6. VLE

The Virtual Laboratory Environment (VLE) is a software and an API which supports multi-modeling, simulation and analysis [18]. VLE is based on the DEVS formalism and provides a set of C++ libraries, the VFL (VLE Foundation Libraries) and several programs such as a simulator, a GUI to model and develop models and tools to analyze and visualize simulation outputs. This project is supported by the French National Institute for Agricultural Research (INRA) and is now included in the RECORD platform. It supports different model specification formalisms such as discrete event specifications, differential equations, petri nets, finite state automata among others.

The frameworks we seen offers tools based on Model-Driven Engineering (MDE) concepts to assist the modeling phase. In this paper, we took another approach by proposing a DSL.

3. DESIGN AND IMPLEMENTATION

DEVS-Ruby has been implemented using Ruby; the library itself is packed using RubyGems (Ruby's package manager), and can easily be installed via the command line like this: `gem install devs`. We should specify that the library is currently still in development and so we haven't met all our goals yet. As DEVS-Ruby implements the DEVS theory, our main concern is to remain consistent with the formalism specifications. Ultimately, we want to implement several simulation algorithms that can be found in the literature [19–25] to provide a scalable simulation platform. From all DEVS implementations we seen in section 2.3., our work is closest to PyDEVS [14] because it provides only an API to build DEVS models and simulate them. Although the beginnings of our

implementation provided an API, we added so much support for describing DEVS models easily atop Ruby that it started to feel like a specialized tool. We wanted to develop this aspect and thus, grow an internal DSL. First, we present our modeling class architecture. Then, we discuss our simulation class architecture flexible enough to support several simulation algorithms.

3.1. Modeling architecture

The Figure 1 represents our modeling class architecture. The hierarchical modeling capability of DEVS is expressed through a composite pattern, which enables to represent models into a tree structure. *Model* is the component class, which is the abstraction for both atomic and coupled models and provides common capabilities for the latters, like input and output interfaces. The *AtomicModel* class is meant to be extended to define the behavior of a model. It provides a default passive behavior (the model remain in its initial state forever) along with several attributes: *time* which represent the last simulation time at which the model was activated; *elapsed* which represent the time elapsed since the last activation; *sigma* which is a variable introduced to simplify modeling phase and represent the next activation time. The methods *external_transition*, *internal_transition*, *confluent_transition*, *output* and *time_advance* should be overridden to implement a custom behavior. By default, *time_advance* returns the *sigma* attribute, which is set to *DEVS::INFINITY* constant. We also provide a convenient method (*post*) meant to be used in the method body of *output* to drop off a value onto a given output interface. The other method *fetch_output* must not be used nor overridden by the modeler, it is used internally to retrieve messages from all interfaces.

Listing 2. Atomic model definition example

```
class TrafficLight < AtomicModel
  def initialize
    super()
    add_output_port :out
    @color = :red
    @sigma = 0
  end

  def output
    post @color, :out
  end

  def internal_transition
    @color, @sigma = case @state
    when :red
      [:green, 5]
    when :green
      [:orange, 20]
    when :orange
      [:red, 2]
    end
  end
end
```

Listing 2 is a very concise example definition of the classical traffic light model with an autonomous behavior. It has one output interface and sends at time-interval depending on its current state the appropriate new color. As we wanted to provide as most flexibility as we could, note once the modeler name a given port, all methods expecting a *Port* can receive indifferently either its name or the actual object. We should insist on the expressiveness and readability of the syntax of Ruby compared to other languages.

Most of the time, a modeler will define atomic models. However, as the classic hierarchical DEVS specify, a coupled model must respond to a *select* method that returns a model from a given list of components with an imminent activation. By default, this method returns the first element in that list, but if the modeled system needs another behavior, the modeler must override this method. Otherwise, the *CoupledModel* class can be directly instantiated. Then, the modeler can add its input and output ports, add sub-models through *add_child* method, and also define the couplings between the coupled model interfaces and interfaces of its components with the methods *add_internal_coupling*, *add_external_input_coupling* and *add_external_output_coupling*.

Listing 3. Coupled model definition example

```
class MyCoupledModel < CoupledModel
  def initialize
    super()
    # add input & output interfaces
    add_output_port :out_1
    # add components
    m = TrafficLight.new
    m.name = :traffic_light
    add_child(m)
    # add couplings
    add_external_output_coupling(m, :out_1, :out)
  end

  def select(imminent_children)
    imminent_children.sample
  end
end
```

Listing 3 shows an example of how to build a coupled model. Such modeling architecture offers nothing original. Moreover, the effort needed to build the modeling tree could be reduced using Ruby features for a more readable and more concise code. Our attempt to do this can be read in section 4., but before that we describe our simulation architecture.

3.2. Simulation architecture

The simulation architecture enables the evolution of a modeled system over time whose behavior has been defined as in section 3.1.. Several simulation algorithms for the DEVS for-

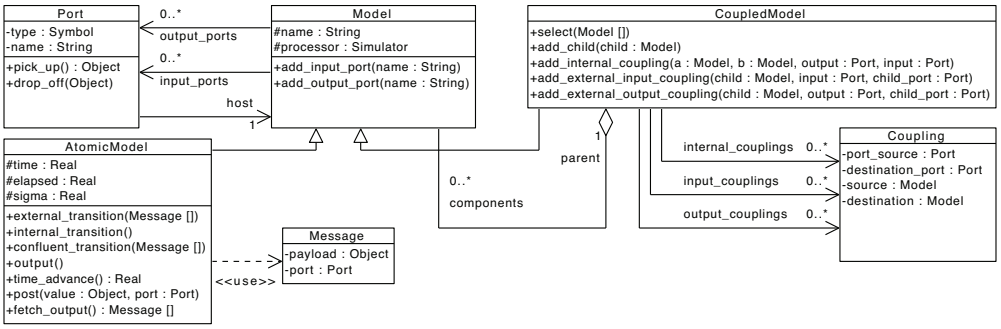

Figure 1. Modeling class diagram

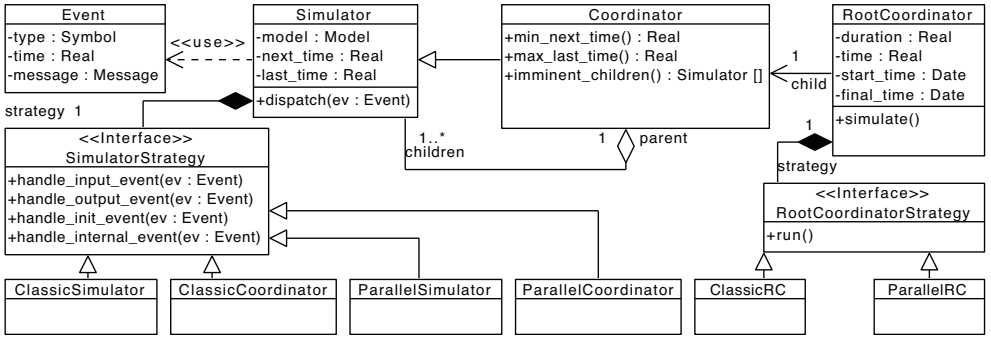

Figure 2. Simulation class diagram

malism have been developed. For now, we support the original classic algorithm along with PDEVS, both defined in [11]. We designed our architecture described in Figure 2 so that multiple simulation algorithms can be easily added over time. As defined in the DEVS formalism and explained in section 2.2.2., we provide a *Simulator* for each atomic model and a *Coordinator* for each coupled model, along with a *RootCoordinator* that coordinates the whole simulation. All these *processors* communicate through message-passing. We differentiate a message (*Message* class) drop off onto a model output interface or onto a model input port and the messages sent between processors (*Event* class). The latter are emitted at a given simulation *time* with a particular *type* and wraps optionally a *Message*. Again, we used a composite pattern to organize the processors as a modeling tree except that we drop the common interface (the component) between the leaf (*Simulator*) and the composite (*Coordinator*). The simulator wraps its associated *Model*, and define *next_time* and *last_time* attributes, that represent respectively the next simulation time at which the associated model will be activated and the last simulation time at which the model was activated. The *dispatch* method handles incoming events from other processors. We moved the core logic of each algorithm in other classes through a strategy pattern. In the *dispatch* method, depending on the type of the event, the appropriate method is called on the strategy object following this pattern: *handle_TYPE_event*. In Ruby, we implemented the strategy pattern through mixins. A mixin is a facility allowing the inclusion of a module within a class definition and a module is a structure allowing grouping a set of functions. When a module is mixed in a class, the functions that were available in the module become available as methods within the class. We use this feature along with monkey-patching, which allows extending or modifying the code at runtime. Thereby, it is fairly easy to implement a new algorithm: we define a module responding to all types of events that are likely to be emitted. The events (i,t), $(*,t)$, (y,t) and (x,t) defined in the classical DEVS algorithm will be respectively implemented as *handle_init_event*, *handle_internal_event*, *handle_output_event* and *handle_input_event*. The same logic is applied for the coordinators. For the root coordinator, it is a module responding to the *run* method.

There is a lot of benefits in using Ruby to implement the DEVS formalism, but performance in terms of execution speed isn't part of them. For that reason, we used the Ruby C API to implement the most demanding methods in C. To identify them, we used a sampling profiler. Unsurprisingly, the results showed that during a simulation all methods relative to processors are critical. With the same models to be simulated, a mean execution time on 100 simulations drop from 3.11 seconds in pure Ruby to 0.67 seconds in Ruby and C (nearly a 5X speedup).

In this section we presented both our base modeling and simulation architectures. The next section explains how we built a DSL based on this work using Ruby dynamic features.

4. BUILDING A SIMULATION

To build a simulation, we propose an internal DSL more convenient that a traditional API. As we said earlier, Ruby is well-suited for such task. To implement it, we rely heavily on *closures* and Ruby's reflection API (specifically *instance_eval*). A closure is a function that captures its referencing environment. Unlike a function, it allows to access non-local variables even when invoked outside its immediate lexical scope. In Ruby, a closure is defined by a block of code between braces or between *do-end* keywords. The *instance_eval* method is accessible to all objects, allowing to evaluates a closure within the context of the receiver. While the closure is executed, the variable *self* is set to the receiver, giving the code access to the methods and instance variables of the receivers.

In this section, we present our DSL syntax proposal for DEVS modeling and simulation built with these features. Then, we explain how to extend our DSL in order to specialize it with the domain of the modeled system.

4.1. DSL Proposal for DEVS

To propose a DSL we found helpful to start thinking to what would be the ideal syntax with the following requirements:

- parameterize and start the simulation
- express hierarchical modeling by adding atomic and coupled models
- specify couplings between all components
- support the DEVS vocabulary along with a simplified one, more accessible to non-experts of the formalism
- offer a way to add a model already defined (class derived from *AtomicModel*) or to define an atomic model on-the-fly

In listing 4 you will find a proposal of syntax taking into account the requirements we listed above. It is a working example of a system which generates a random number within a range from 2 to 10. Then this number is transmitted to another model whose role is to compute each input by a power of two. Finally, the result goes to a coupled model collecting results into a CSV file and a plot is generated.

Listing 4. DSL proposal for DEVS

```
1  require 'devs'
2  require 'devs/models'
3
4  DEVS.simulate do
5    duration 10
6
7    add_model RandomGenerator, with_params: [2,
         10], :name => :random
8
9    add_model do
10     name 'x^2'
11     init { add_output_port :out_1 }
12
13     when_input_received do |*messages|
14       messages.each do |message|
15         value = message.payload
16         @result = value ** 2
17       end
18       self.next_activation = 0
19     end
20
21     output { post(@result, :out_1) }
22     after_output { self.next_activation = DEVS::
           INFINITY }
23     time_advance { self.next_activation }
24   end
25
26   add_coupled_model do
27     name :collector
28     add_model PlotCollector, :name => :plot
29     add_model CSVCollector, :name => :csv
30
31     plug_input_port :a, :with_child => :csv, :
           and_child_port => 'x'
32     plug_input_port :a, :with_child => :plot, :
           and_child_port => 'x'
33     plug_input_port :b, :with_child => :csv, :
           and_child_port => 'x^2'
34     plug_input_port :b, :with_child => :plot, :
           and_child_port => 'x^2'
35   end
36
37   plug :random, :with => 'x^2', :from => :out_1,
         :to => :in_1
38   plug :random, :with => :collector, :from => :
         out_1, :to => :a
39   plug 'x^2', :with => :collector, :from => :
         out_1, :to => :b
40 end
```

The function *simulate*, defined under the *DEVS* namespace, serves as the entry point into our DSL. It expects a namespace (module) along with a closure and does two things. Firstly it instantiates a *SimulationBuilder* (see Figure 3). The constructors of builders have the responsibility to instantiate a processor and its associated atomic or coupled model. Then, it is going to execute the given closure within the context of the builder. Thus, a *SimulationBuilder* instantiate the root coupled model, a *Coordinator*, and the *RootCoordinator*; a *CoupledBuilder* instantiate a coupled model and a *Coordinator*; an *AtomicBuilder* instantiate an *AtomicModel* and a *Simulator*. The *simulate* function also pass a namespace (module), which implements a simulation algorithm as we seen in section 3.2.: *Parallel* by default, or *Classic* if the classic im-

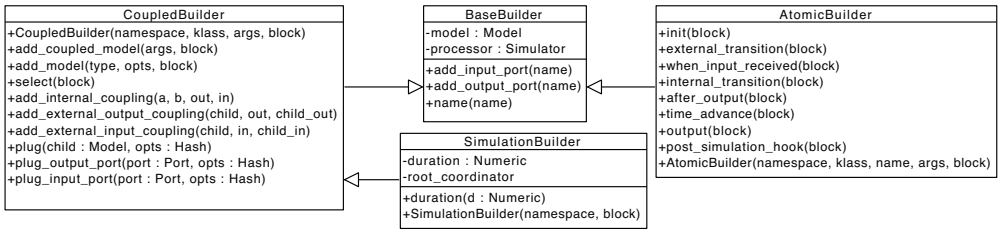

Figure 3. DSL class diagram for builders

plementation is imported. It is used to dynamically mix the given module in each processor. The possibility to define an atomic model on-the-fly is given by another Ruby's reflection method: *define_singleton_method* which allows to define a method dynamically on a given instance.

4.2. Extending the DSL

Despite that we simplify the definition of DEVS models with our DSL, we encourage modelers who have a good understanding of DEVS to package their complex systems as libraries. Defining an atomic model on-the-fly is convenient to prototype a model especially using Ruby's interpreter but this should be avoided for other uses. For more readability and maintainability, we advise to define each model in its own file within a namespace. Ultimately, we encourage the modeler to extend our DSL to introduce its own specific vocabulary. With such practice, end-users could simulate these models easily.

To extend our DSL, the modeler can use the same approach we described in section 4.1., with three steps: (1) define the models, (2) define an entry point, (3) define a builder to initialize the modeling tree. The next section gives an example of a resulting DSL following these precepts to simulate a multi-agent system.

4.3. DSL for Multi-agent systems

An architecture for coupling Multi-agent systems (MAS) and the DEVS formalism is proposed in [26]. We implemented this architecture with our framework as a library and we used ant colonies as an application. The listing 5 shows the resulting DSL. Our entry point is the function *build* defined under the *DEVS::MAS* namespace. The only common verb with the original DSL is *duration*. We can see that new verbs (*environment*, *population_dynamics*, *affiliation*, ...) are related to MAS semantics. Also, it refers to several models that are defined beforehand like *ExponentialGrowth*, *Worker*, *Soldier* or *Nest*.

Listing 5. DSL for MAS

```
require 'devs'
require 'devs/mas'

DEVS::MAS.build do
  duration 100

  # setup the environment
  environment :cellular, dimension: 2, size:
      [100, 100]
  population_dynamics ExponentialGrowth

  # setup affiliations
  affiliation :ant_colony_a do
    base Nest, at: Point.new(8, 15)
    instantiate 30, Worker, with_params: [...]
    instantiate 10, Soldier, with_params: [...]
  end
  affiliation :ant_colony_b do
    base Nest, at: Point.new(67, 53), with_params:
        [453]
    instantiate 54, Worker, with_params: [...]
    instantiate 23, Soldier, with_params: [...]
  end

  # place entities
  place Food, at: Point.new(9, 12), with_params:
      [120]
  place Boulder, at: Point.new(86, 34),
      with_params: [234]
end.simulate
```

As we seen in section 4.1., the technique to introduce new verbs is to define a builder. The listing 6 shows a builder template to extend the DEVS-Ruby DSL.

Listing 6. Builder template used to extend the DSL

```
module DEVS
  module MAS
    class Builder
      def initialize(namespace, &block)
        # do some initialization here
        # and execute the closure within this
            context
        instance_eval(&block)
      end

      def environment(type, opts={}); end

      def population_dynamics(klass); end

      def affiliation(name, &block)
```

```
15        # instantiate a ColonyBuilder
16      end
17
18      def place(resource, opts={}); end
19
20      def simulate
21        DEVS.simulate do
22          # parameterize and start the DEVS
                simulation
23        end
24      end
25    end
26  end
27 end
```

We should emphasize that Listing 5 is devoid from DEVS related vocabulary. Hence, the end-user is not required to be a DEVS expert to read and produce such code, but should be a domain expert, which is precisely what is sought.

With DSL extensions, a biologist or an economist can define its models with his vocabulary. A meaning can be given to transition functions according to each domain. When defining an agent for example, a DEVS transition could represent a neighborhood function. This can facilitate the use and the handling of the formalism.

5. CONCLUSION AND PERSPECTIVES

The work presented in this paper is born from a simple observation: although there are already several DEVS frameworks, most of them require to be familiar with the formalism, even for those providing a GUI to facilitate the modeling phase. We propose another approach to simulate DEVS models by introducing a DSL allowing to express easily the structure hierarchy, atomic models behavior and couplings. By encouraging the extension of our DSL, we can distinguish the modeler from the end-user. The modeler must be familiar with DEVS but the not the end-user, provided that the modeler capture the semantics of the specific domain. Although Ruby is well suited to build DSLs and its expressiveness is a benefit to define models, it is a slow high-level language and thus, not really adapted to simulation. Luckily, it offers a way to extend the language in a low-level language. We took this opportunity to implement the most demanding methods in C to remain as effective as possible.

We already met many of our goals by fulfilling the DEVS formalism specifications, by defining a convenient DSL and by remaining performant. Our perspectives are manifold, at first we want to provide a scalable platform by developing extensions for distributed simulations and parallel execution. Also, we would like to provide other DSL extensions along with a library of models. We also plan to provide a way to save a simulation in order to replay it.

6. REFERENCES

[1] S. J. E. Taylor, R. Fujimoto, E. H. Page, P. A. Fishwick, A. M. Uhrmacher, and G. Wainer, "Panel on grand challenges for modeling and simulation," in *Proceedings of the Winter Simulation Conference*, WSC '12, pp. 232:1–232:15, Winter Simulation Conference, 2012.

[2] S. Mittal and S. A. Douglass, "From domain specific languages to devs components: application to cognitive m&s," in *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, (San Diego, CA, USA), pp. 256–265, Society for Computer Simulation International, 2011.

[3] I. Fister, Jr., I. Fister, M. Mernik, and J. Brest, "Design and implementation of domain-specific language easytime," *Comput. Lang. Syst. Struct.*, vol. 37, pp. 151–167, Oct. 2011.

[4] P. Hudak, "Modular domain specific languages and tools," in *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, (Washington, DC, USA), pp. 134–, IEEE Computer Society, 1998.

[5] T. Kosar, P. E. Martinez L, P. A. Barrientos, and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Inf. Softw. Technol.*, vol. 50, pp. 390–405, Apr. 2008.

[6] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, pp. 316–344, Dec. 2005.

[7] A. van Deursen and P. Klint, "Little languages: little maintenance," *Journal of Software Maintenance*, vol. 10, pp. 75–92, Mar. 1998.

[8] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, pp. 26–36, June 2000.

[9] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering experiment in software component generation," in *Proceedings of the 18th international conference on Software engineering*, ICSE '96, (Washington, DC, USA), pp. 542–552, IEEE Computer Society, 1996.

[10] T. Kosar, N. Oliveira, M. Mernik, M. J. V. Pereira, M. Crepinsek, D. da Cruz, and P. R. Henriques, "Comparing General-Purpose and Domain-Specific Languages: An Empirical Study," *Computer Science and Information Systems*, vol. 7, pp. 247–264, May 2010.

[11] B. P. Zeigler, H. Praehofer, and K. T. Gon, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. San Diego, CA: Academic Press, 2nd ed., 2000.

[12] C. Seo, B. P. Zeigler, R. Coop, and D. Kim, "Devs modeling and simulation methodology with ms4 me software tool," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, DEVS 13, (San Diego, CA, USA), pp. 33:1–33:8, Society for Computer Simulation International, 2013.

[13] L. Capocchi, J. F. Santucci, B. Poggi, and C. Nicolai, "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems," *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, 2011.

[14] J. S. Bolduc and H. Vangheluwe, "A modeling and simulation package for classic hierarchical DEVS," *MSDL, School of Computer McGill University, Tech. Rep*, 2002.

[15] J. de Lara and H. Vangheluwe, "AToM3: A Tool for Multi-formalism and Meta-modelling," in *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, (Berlin, Heidelberg), pp. 174–188, Springer-Verlag, Apr. 2002.

[16] E. Kofman, M. Lapadula, and E. Pagliero, "PowerDEVS: A DEVS-based environment for hybrid system modeling and simulation," *School of Electronic Engineering, Universidad Nacional de Rosario, Tech. Rep. LSD0306*, 2003.

[17] G. A. Wainer, "CD++: a toolkit to define discrete-event models," *Software, Practice and Experience. Wiley*, vol. 32, pp. 1261–1306, November 2002.

[18] G. Quesnel, R. Duboz, and E. Ramat, "The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems," *Simulation Modelling Practice and Theory*, vol. 17, no. 4, pp. 641–653, 2009.

[19] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism," in *WSC '94: Proceedings of the 26th conference on Winter simulation*, Society for Computer Simulation International, Dec. 1994.

[20] A. C. H. Chow, "Parallel DEVS: a parallel, hierarchical, modular modeling formalism and its distributed simulator," *Transactions of the Society for Computer Simulation International*, vol. 13, Dec. 1996.

[21] A. Troccoli and G. Wainer, "Implementing parallel Cell-DEVS," *Simulation Symposium, 2003. 36th Annual*, pp. 273–280, 2003.

[22] E. Glinsky and G. Wainer, "New parallel simulation techniques of DEVS and Cell-DEVS in CD++," *Simulation Symposium, 2006. 39th Annual*, 2006.

[23] Q. Liu, *Distributed Optimistic Simulation of DEVS and CELL-DEVS models with PCD++*. PhD thesis, Aug. 2006.

[24] G. A. Wainer, R. Madhoun, and K. Al-Zoubi, "Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web-Services," *Simulation Modelling Practice and Theory*, vol. 16, pp. 1266–1292, Oct. 2008.

[25] S. Jafer and G. Wainer, "Flattened Conservative Parallel Simulator for DEVS and CELL-DEVS," in *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, (Washington, DC, USA), pp. 443–448, IEEE Computer Society, 2009.

[26] P.-A. Bisgambiglia, P. A. Bisgambiglia, and R. Franceschini, "Agent-oriented approach based on discrete event systems (WIP)," in *DEVS 13: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, Society for Computer Simulation International, Apr. 2013.

Biography

Romain FRANCESCHINI is a PhD student at the University of Corsica. He received a MSc in Computer Science at University of Corsica in 2013. His research interests include multi-agent systems, discrete event systems (DEVS) and simulation.

Paul-Antoine BISGAMBIGLIA is an Associate Professor at the University of Corsica. His research interests include complex systems, fuzzy systems, multi agent systems, discrete systems (DEVS) and simulation. His email and web addresses are bisgambiglia@univ-corse.fr and http://paul-antoine-bisgambiglia.univ-corse.fr/, respectively.

Paul Antoine BISGAMBIGLIA is full Professor at the University of Corsica. His research activities concern the techniques of modeling and simulation of complex systems and the test of systems described at high level of abstraction. Email: bisgambi@univ-corse.fr.

David R.C. HILL was Vice President of Blaise Pascal University (2008-2012). Professor Hill is also past director of a French Inter-University Computing Center (CIRI) (2008-2010). From august 2005 to august 2007, he was deputy director of ISIMA Computer Science & Modeling Institute (French Grande Ecole d'Ingénieur) where he managed various departments before 2005. Professor Hill is now head of the Software Engineering Department at ISIMA and head of the Regional Computing Mesocenter. Since 1990, he has authored or co-authored more than two hundrer papers and he has also published many text books. His Web page is www.isima.fr/ hill.