DEVS-Ruby: a discrete event modeling system

J. Paul Daigle
Georgia State University

General Terms: Modeling

Additional Key Words and Phrases: DEVS, Ruby

1. MOTIVATION

The DEVS formalism provides a consistent and practical means of defining discrete event simulations. However, in application, use of DEVS has been limited to the DEVSJAVA system, use of which requires an understanding not only of the Java language, but of a large and complex object system.

In recent years, a number of other object-oriented languages have been gaining in popularity, such as Scala, Ruby, Groovy, and C♯. Many of these languages have features that Java does not that make them easier to use, such as type inference (Scala, C♯, Ruby, Groovy), duck typing (Ruby, Groovy, Python), and monkey patching (Ruby), as well as functional structures such as Lambda functions, object enhancements such as Scala and Ruby mixins, and in the case of Ruby, portability across both operating systems and virtual machines.

Our goal with DEVS-Ruby is to simplify the DEVS object system and the semantics of generating DEVS models without compromising the reliability of simulations. In addition, we wish to make it easier to install and use a DEVS modeling system.

2. CORE DESIGN

DEVS-Ruby is currently quite primitive. There are only three classes in the core library, a the **DEVSAtomic** class, **EventQueue** class, and an **Event** class. These three classes do suffice for building satisfactory primitive models, but do not constitute a complete and reliable DEVS system.

2.1 EventQueue

The class EventQueue has two roles. First, as the name implies, it maintains the discrete events and the global clock. This is done using an internal class with methods for adding events and removing future, obsolete events from the queue if neccessary[1]. In addition to this, the EventQueue maintains a data structure for tracking couplings between objects.

The **Coupling** class is an example of simplifying the use of DEVS. In DEVS-JAVA, each coupling must be defined separately. If object A is to have a coupling to both objects B and C, two couplings must be created. In DEVS-Ruby, a the

[1]The accompanying ruby documentation contains complete descriptions of each class, including attributes and methods

constructor for the coupling object takes an arbitrary number of arguments. The first argument is the sender and all subsequent arguments are the recievers. This simplifies both reading and writing of code.

2.2 Event

The Event class again uses Ruby's optional constructor arguments. The constructor technically takes four arguments, but all arguments are optional. This disposes of the need to have multiple constructors for different numbers of argument sets, again simplifying the code. An Event is marked with the sender, the time stamp, the message, and whether it represents and internal or external event. These attributes are used by the EventQueue in processing.

2.3 DevsAtomic

The DevsAtomic class is extended when describing new models. Unlike the prior two classes, which should never need to be altered by the modeler, the DevsAtomic class is meant to be inherited. This is necessary in order for the modeler to create new models.

2.3.1 *Base Class.* There are two default transitions in the DevsAtomic class: `active_state` and `passive_state`. Both of these are called via the `base_transition` method. This basic transition method handles the default behaviors common to all DevsAtomic objects and then passes control to either the `internal_transition` or `external_transition` method, depending on what kind of message it has recieved.

These methods are empty in the base class, and are meant to be defined in the child class built by the modeler.

Internal and External messages are not strictly part of the DEVS formalism. The purpose of these two classifications is to simplify the design of the EventQueue. An internal "event" should simply schedule the internal transition function.

External events are incorrectly implemented in this alpha version of the software. Currently, the modeler is able to schedule output messages using the time advance function. Because of the design of the system, however, this is an easy patch. The modeler creates a new event by calling the DevsAtomic method `new_event`, which takes as its only argument the message to be sent. The time stamp of the Event object that will be placed on the queue is set within the `new_event` method, so a single change to this method can take scheduling out of the hands of the modeler.

2.3.2 *Derived Class.* The simple test model that we implemented consists of a student and an alarm clock. The student sets the alarm clock and goes into a passive state, the alarm clock sends a "ring" message to the student (which wakes the student) and then goes into a passive state.

The entire **Student** class is implemented in less than 40 lines of code. It requires no constructor definition, only definitions for the `external_transition` and `internal_transition` methods. Two additional methods are defined, but these are conveniences that exist only to improve the readability of the two required methods. Example 1 shows the **Student** class.

The major flaw in the current implementation is shown in Line 26. Here we see the modeler using the sigma function to schedule a future event. Currently, this

Example 1 Class Definition of Student

```
require 'devs_atomic'

class Student < DevsAtomic
  def external_transition message
    case @state
    when :passive
      case message
      when :ring
        wake_up
      end
    end
  end

  def internal_transition message
    case @state
    when :active
      case message
      when :sleep
        set_alarm
      end
    end
  end

  def set_alarm
    passive_state
    @sigma = 0
    new_event(:set)
  end

  def wake_up
    @state = :active
    @sigma = 16
    new_internal(:sleep)
  end
end
```

will have an effect on when other models receive this method. As mentioned, this is a simple patch.

3. BUILDING A SIMULATION

To build our simulation of the student-clock system, three classes were required, a class to represent students, a class to represent the clock, and a class to represent the simulation itself. Example 2 shows the code for the simulation.

The method `kick_start` sets the intial state of the system. After this, the method `EventQueue.next_event` will update the system.

Example 2 Simulation Code

```
1 require 'devs_atomic'
2 require 'event_queue'
3
4 class Coupling
5   attr_accessor :event_queue
6   attr_reader :clock, :student
7   def initialize
8     @event_queue = EventQueue.new
9     @student = Student.new(@event_queue)
10    @clock = Clock.new(@event_queue)
11    @event_queue.add_coupling @student, @clock
12    @event_queue.add_coupling @clock, @student
13  end
14
15  def kick_start
16    @event_queue.add_event(Event.new :sender => @student, :
          stamp => 0, :message=>:set)
17  end
18 end
```

4. FUTURE WORK

At this time DEVS-Ruby only supports very simple DEVS modeling. There is an implied confluent function in the EventQueue (it handles internal events first), but no means for the modeler to specify more advanced methods. Extended DEVS models include input and output ports, which are not supported by DEVS-Ruby.

Ruby's metaprogramming system should be useful for implementing ports in particular. At this time, there are two "ports", the external and internal. The future plan is to move these from an implied to an explicit state by creating a port list and a method that writes transition methods using that list for both input and output ports.

This will allow the modeler to specify ports in the model and use the syntax shown in Lines 1.27 and 1.33 to specify ports by creating a `new_[portname]` message, and a `[portname_transition]` method for handling cases where the input port is crucial to model behavior.

The current system does not contain any visualization tools, which would be a strong addition, and does not contain a test runtime to allow the modeler to run the system step by step and investigate the state. These are very useful tools that would make a great addition to DEVS-Ruby.

Finally, while the DEVS-Ruby system is built to be simple to use and to use the vocabulary from DEVS as much as possible, it does not go far enough. Many modern langugages, such as Scala, Ruby, and Io, have strong support for building Domain Specific Languages. A DSL for DEVS modeling that could run be interpreted by multiple platforms, allowing formal specifications of DEVS models without ambiguity, would be a benefit to the modeling community as a whole.