GENERATION OF DEVS MODELLING & SIMULATION ENVIRONMENTS

Ernesto Posse and Jean-Sébastien Bolduc

Modelling, Simulation and Design Lab

http://msdl.cs.mcgill.ca

School of Computer Science

McGill University

Abstract

DEVS ([7]) is a well known formalism for modelling discrete-event, continuous-time systems, suitable for the definition of efficient simulators of those systems. A tool is presented which allows the graphical definition of DEVS models. The tool is capable of generating a representation suitable for simulation by the PythonDEVS simulator, an implementation of the standard classic DEVS simulation algorithm. The tool itself is automatically generated from a meta-model. The simulator generation is realized by graph-transformation. The paper demonstrates how dedicated modelling and simulation environments can be easily generated from meta-models and graph-transformation environments.

Keywords: DEVS, Graph-transformation, Meta-modelling, simulation.

1 INTRODUCTION

The behaviour of many dynamic systems can be modelled by describing how the system reacts to external stimuli as a function of time. In particular, the behaviour can be described by the internal changes of the system and what output it generates. Complex dynamic systems are conveniently described in a compositional or component-based fashion, that is, a complex system can be described in terms of subsystems. This structure has two central aspects: containment and connectivity.

One formalism that combines these features in systems modelling is the so-called DEVS formalism [7]. Some of the advantages of the DEVS formalism are that it permits the hierarchical description of systems and that there are efficient algorithms for their simulation.

The possibility of defining DEVS models in terms of interconnected submodels makes it natural to describe such models in a graphical fashion. This would naturally lead to a formal description of the operational semantics of the formalism by means of graph-rewriting. However, this is beyond the scope of this paper. Here, we present a tool constructed by means of meta-modelling which uses graph transformation to generate a simulator for specific models. The tool as a whole can be seen as a visual modelling and simulation environment.

The purpose of the paper is to show that graphical representations and graph-transformation techniques are suitable to model and simulate complex systems. The rest of the paper is organised as follows. Section 2 presents a brief review of the DEVS formalism. Section 3 presents the tool itself. Section 4 discusses the code generated by the tool and the PythonDEVS framework. Section 5 reviews some basic concepts about meta-modelling and graph-transformation. Section 6 presents the code generation scheme. Section 7 has some concluding remarks.

2 THE DEVS FORMALISM

In this section we briefly recall the definition of DEVS models [7]. A DEVS model is either *atomic* or *coupled*. An atomic model describes a simple system. A coupled model is the composition of several submodels which can be atomic or coupled. Submodels have *ports*, which are connected by channels. Ports have a type: they are either *input* or *output* ports. Ports and channels allow a model to receive and send signals from and to other models respectively. A channel must go from an output port of some model to an input port of a *different* model, from an input port in a coupled model to an input port of one of its submodels, or from an output port of a submodel to an output port of its parent model.

An atomic model has, in addition to ports, a set of *states*, one of which is the *initial* state, and two types of transitions between states: *internal* and *external*. Associated with each state is a *time-advance* and an *output*.

Definition 1 (Atomic DEVS). [1]
*An **atomic DEVS** is a tuple* $(S, X, Y, \delta^{int}, \delta^{ext}, \lambda, \tau)$ *where* S *is a set of **states**,* X *is a set of **input events**,* Y *is a set of **output events**,* $\delta^{int} : S \to S$ *is the **internal transition function**,* $\delta^{ext} : Q \times X \to S$ *is the **external transition function**,* $\lambda : S \to Y$ *is the **output function** and* $\tau : S \to \mathbb{R}_0^+$ *is the **time-advance** function.*

In this definition, $Q \stackrel{def}{=} \{(s, e) \in S \times \mathbb{R}^+ \mid 0 < e \leq \tau(s)\}$ *is called the **total-state space**, for each* $(s, e) \in Q$, e *is called the **elapsed-time**.*[2]

Informally the operational semantics of an atomic model are as follows: the atomic model starts in its initial state, and it will remain in any given state for as long as its corresponding time-advance specifies or until input is received on some port. If no input is received, when the time of the state expires, the model sends output as specified in the state (before changing the state), and then it jumps to the new state as specified by the internal transition for that state. On the other hand, if input is received before the time for the next internal transition expires, then it is the external transition function which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

The following definition formalises the concept of coupled DEVS models[3].

Definition 2 (Coupled DEVS).
*A **coupled DEVS** named* D *is a tuple* $(X, Y, N, M, I, Z, select)$ *where* X *is a set of **input events**,* Y *is a set of **output events**,* N *is a set of **component names** such that* $D \notin N$, $M = \{M_n \mid n \in N, M_n$ *is a DEVS model (atomic or coupled) with input set* X_n *and output set* $Y_n\}$ *is a set of DEVS **submodels**,* $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ *is a set of **influencer** sets for each component named* n, $Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \to X_n$ *or* $Z_{D,n} : X \to X_n$ *or* $Z_{i,D} : Y_i \to Y\}$ *is a set of **transfer functions** from each component* i *to some component* n, *and* $select : 2^N \to N$ *is the **select** function.*

Connectivity of submodels is expressed by the influencer set of each component. Note that for a given model n, this set includes not only the external models that provide inputs to n, but also its own internal submodels that produce its output (if n is a coupled model.) Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the submodels. This is, each submodel in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* of events whenever there are two submodels that have a transition scheduled to be performed at the same time. Logically, the transitions are assumed to be done in that time instant, but its implementation on a sequential computer is serialized. If the two models are connected by a channel, then the model which is the source of the channel, will undergo the transition before the target model. If they are not interconnected, the coupled model has a *select* function which chooses one of the models to undergo the transition first.

3 THE TOOL

The two types of DEVS models introduced in the previous section can be viewed in terms of graphs. Coupled models have a very evident graphical structure, more precisely that of a higraph. There are different variants of higraphs, but the simplest form consists of two graphs, or in this case, a tree and a graph. The tree represents the containment relation between submodels. The graph, represents the connectivity between the components (given in the previous definition by the influencer sets and the transfer functions.)[4] This is not completely accurate, as we allow the models to have ports. Thus, a coupled model is really an enriched hi-graph, associating to each hyper-edge, not only the sets of input components and output components connected by it, but sets of port-components pairs.

An important subset of atomic DEVS models, namely

[1] For the sake of simplicity, we do not present a formalisation of the concept of "ports".

[2] $\mathbb{R}_0^+$ denotes the positive reals with zero included.

[3] For the sake of simplicity, this "formalisation" leaves out many important details. In particular, as in the atomic case, it does not deal with the subtleties of ports in the formalism, and it leaves out the proof of well-definedness for coupled models, which relies on the fact that DEVS models are closed under coupling, i.e. for each coupled DEVS one can construct a bisimilar atomic DEVS. For the purposes of this paper these definitions suffice.

[4] There are several formal characterizations for higraphs, of which perhaps the most general, as given in [5], is that a higraph is an object in the functor category **Func**$(\cdot \rightrightarrows \cdot, \textbf{Poset})$ where $\cdot \rightrightarrows \cdot$ is the category with only two objects and two morphisms between them and **Poset** is the category of partially-ordered sets and monotonic functions.

finite-state models, can also be described to an extent as a graphical structure. This structure can be seen as that of a finite-state automaton enriched with ports, two kinds of transition edges (internal and external), and states labelled with their corresponding time-advance and output. In this representation, external transition edges are also labelled with a boolean expression which depends on its source state, some input ports, and the special variable *e* which represents the time elapsed since the last transition.

Such graphical structure calls for a graphical modelling environment, so that the modeller can focus on the design of the system rather than its formalisation. The environment and an example of a DEVS model are shown in Figure 1.

In this tool, the user can create coupled or atomic DEVS models by clicking the corresponding button on the left and then clicking in the canvas. The same applies for each element that forms a DEVS model (states, ports, channels, and transitions.) To create a channel between ports, or a state transition, the user clicks on the Connect button, then clicks on the source and finally on the target. If the link is a state transition, the user is asked to select whether it is an internal or an external transition. To specify that a component is part of another (e.g. a submodel, or a port,) the user clicks on the Connect button, then clicks on the "parent", and finally on the "child". Each graphical element, which is not a link, has a label with its name. This can be edited using the Edit button.

Ports are labelled as either input or output ports. Each state has two attributes apart from its name. These are two fields which may contain an arbitrary Python script to specify the time-advance and output for the state. External transitions between states also have an additional attribute which may contain some Python script to specify whether the transition is enabled or not. This script has as parameters the source state, the elapsed time, and the values at the input ports, and it should return true or false. For example, if there is an external transition link between two states s_0 and s_1, labelled with a condition such as $e < 1.0$ and $x_1 = 3$, where e is a variable representing the elapsed-time since the last transition, and x_1 is the name of some input port, then the external transition will take place if the condition is true. All these attributes for ports, states and external transitions can be specified by using the Edit button.

The Generate simulator button is used to produce the Python code for the DEVS model on the canvas. The generated code is a suitable representation of DEVS models that can be used by the PythonDEVS framework, which is briefly described in section 4.

4 PythonDEVS AND THE GENERATED SIMULATORS

The *PythonDEVS Modelling and Simulation Package* provides an implementation of the standard classic DEVS simulation algorithm as introduced in [7]. Python is an interpreted, high-level, object-oriented programming language. The package consists of two files, the first of which (`DEVS.py`) provides a class architecture that allows hierarchical classic-DEVS models to be easily defined by subclassing the `AtomicDEVS` and `CoupledDEVS` classes. The simulation engine (SE) itself is implemented in another file (`Simulator.py`). Based on the DEVS simulator described in [7], it uses the same message-passing mechanism. Both the modelling architecture and the SE are described in detail elsewhere (see [1].)

The package was originally meant as a simple API to design and experiment with DEVS models. As an early prototype it still has important limitations. For instance a model cannot be thoroughly validated using the methods provided. More importantly, the SE offers limited means to terminate a simulation and provides no easy model-reinitialization possibilities yet. Further versions of the package will also support sets declarations and extended type-checking as well as information methods to help debugging.

The code generated by the tool described in the previous section follows the standard approach for the implementation of DEVS models. Each model (atomic and coupled) is compiled into a class, which is a subclass of `AtomicDEVS` or `CoupledDEVS`.

The simulation algorithm of PythonDEVS calls the methods in the generated classes for the models, according to the operational semantics of DEVS. The subclasses of `AtomicDEVS` implement the methods `extTransition`, `intTransition`, `outputFnc` and `timeAdvance` (for $\delta^{ext}, \delta^{int}, \lambda$, and τ respectively.) The subclasses of `CoupledDEVS` specify the model's composition and connectivity, including the ports and submodels.

As an example, consider the model in Figure 2. In the atomic model *A*, there is an external transition labelled `evt` from state `s0` to state `s1`. This transition has as condition the following script:

```
if e < 1.0:
  if i1 == 'a' and i2 == 0 or i1 == 'b'
```

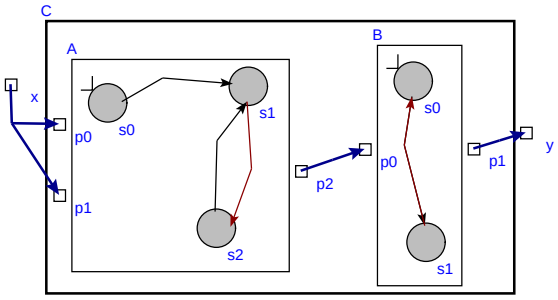

Figure 1: A coupled DEVS model.

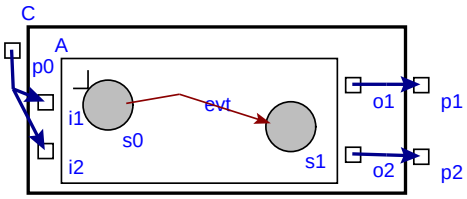

Figure 2: Another coupled DEVS model.

```
      or i2 > 0: return 1
    else: return 0
  elif e < 2.0: return i2 >= 1
  else: return 0
```

where 0 stands for false and 1 for true, following Python's convention. Then, an excerpt of the code generated for *A* looks like

```
class A(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.state    = 's0'
    self.elapsed  = 0.0
    self.i1 = self.addInPort()
    self.i2 = self.addInPort()
    self.o1 = self.addOutPort()
    self.o2 = self.addOutPort()
  def extTransition(self):
    s = self.state
    e = self.elapsed
    i1 = self.peek(self.i1)
    i2 = self.peek(self.i2)
    if s == 's0':
      def guard1_condition(e, i1, i2):
        if e < 1.0:
          if i1 == 'a' and i2 == 0 or i1 == 'b'
            or i2 > 0: return 1
          else: return 0
        elif e < 2.0: return i2 >= 1
        else: return 0
      if guard1_condition(e, i1, i2):
        return 's1'
```

The code generated for the coupled model *C*, is as follows:

```
from A_devs_model.A import *
class C(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    self.p0 = self.addInPort()
    self.p1 = self.addOutPort()
    self.p2 = self.addOutPort()
    self.A = self.addSubModel(A())
    self.connectPorts(self.A.o1, self.p1)
    self.connectPorts(self.p0, self.A.i1)
    self.connectPorts(self.A.o2, self.p2)
    self.connectPorts(self.p0, self.A.i2)
```

5 META-MODELLING AND GRAPH-TRANSFORMATION

Having described the tool, we turn our attention to how meta-modelling is used to create the DEVS graphical environment and graph-transformation is used to generate custom simulators.

5.1 Meta-modelling

Meta-modelling refers to the definition or description of modelling languages or formalisms. A meta-model is a structure that describes a class of models in a formalism or language. For instance, an Entity-Relationship diagram can be used to describe the set of possible DEVS models. This meta-model is pictured in Figure 3.

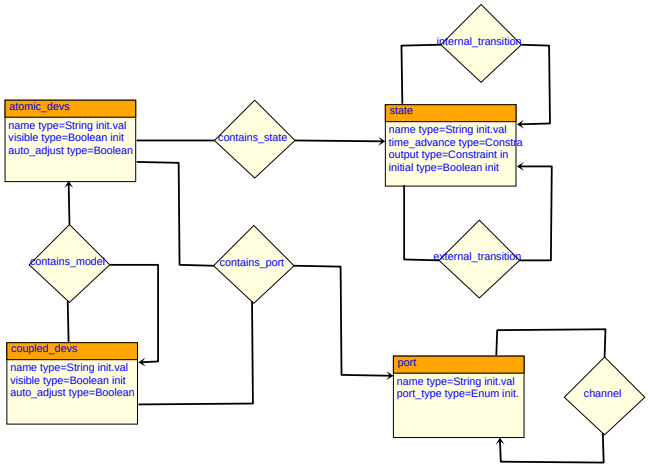

Figure 3: The DEVS meta-model.

The modelling and simulation of complex systems requires the use of possibly many different formalisms to describe them or their components. A meta-modelling

tool allows the user to design such different formalisms by creating meta-models for them and generate, from these meta-models, tools for manipulating models in the corresponding formalisms. The DEVS modelling environment was created using AToM^3 ([4]), a meta-modelling environment.

In AToM^3 the models (and the meta-models) are plain graphs. The description of DEVS models graphically requires higraphs as mentioned in section 3. Since a higraph consists of two graphs over the same set of nodes (a tree representing containment and a graph representing connectivity,) the meta-model from Figure 3 can correctly describe the class of higraphs corresponding to hierarchical DEVS models. There is nonetheless one practical difficulty: AToM^3's visual primitives only provide support for the manipulation of plane graphs, not higraphs. This difficulty is resolved by associating which each node, actions that can be triggered by GUI events[5]. For instance, when a coupled model is connected to a submodel to represent a containment relation, an action is triggered which changes the physical shape of the coupled model to enclose the submodel.

5.2 Graph Transformation

An approach to manipulate graphical structures, such as our representation of DEVS models, is graph transformation. Graph transformation extends the idea of term rewriting to arbitrary graphs. The theory behind graph transformation has been thoroughly studied (see for example [6],) but there are still few practical software tools that support it. AToM^3 is one such tool.

The central notion in graph transformation is that of a *graph-grammar*. A graph-grammar is a collection of *productions* or *rules* specifying how a (sub)graph of a so-called *host graph* can be replaced by another (sub)graph.

Some graph-grammars are enriched by associating with each rule, some additional conditions and actions. These can be used to model side-effects.

Informally, the semantics of a graph-grammar is as follows. We start from a *host* graph and a graph-grammar. A *direct derivation* is the result of matching some subgraph of the host graph to the left-hand side of some rule in the grammar, checking if the additional condition is true, and if so, replacing that subgraph by the corresponding right-hand side of the rule, and perform any additional actions associated with the rule. Some graph-rewriting systems associate priorities to the rules, so that if more than one rule matches the host graph, the priorities act as tie-breakers. An *execution* or *trace* is a sequence of direct derivations. [6]

Graph transformation has been used for many different applications, such as specifying the operational semantics of graphical languages, and specifying formalism translations.

6 CODE GENERATION

In order to generate simulators from DEVS models represented graphically we use graph transformation. Code generation can be understood in terms of formalism transformation where the original representation is the source formalism and the language of the generated code is the target formalism. While it is theoretically possible to provide a pure graphical translation from a formalism such as DEVS into a real programming language such as Python, it is not a very practical approach, since it would require defining a meta-model for the target language. Real programming languages have too many constructs and special cases to make this approach feasible in practice. However, we can still have a graph-transformation approach since rules in a graph-grammar can have associated actions encoding side-effects. In our approach we use the graphical nature of the source formalism to traverse and annotate the model which is being translated, while the rule actions generate the associated code.

To apply a graph-grammar in order to generate code we must introduce some extensions to the meta-model. In particular we need some "pointers" or "markers" to traverse the DEVS model and mark which submodels have been already processed. There are two equivalent approaches to this: 1) use a graphical pointer, or 2) use an attribute in the nodes to represent the fact that a node has already been visited. Our graph-grammar uses the second approach.

Another issue in the code generation scheme is that for a given model node we might require access to several of its neighbour nodes to generate its code. For instance, when generating code for any model we need to know which are the node's ports, or when generating code for a coupled model we need to know which are its submodels. None of these situations can be handled by a single rewriting rule, since the left-hand side of a rule always has a fixed number of nodes, but we need to apply the

[5] This is a feature supported by AToM^3.

[6] This informal definition, as implemented in AToM^3, is most closely related to the so-called SPO approach to graph-transformation ([6], [2]).

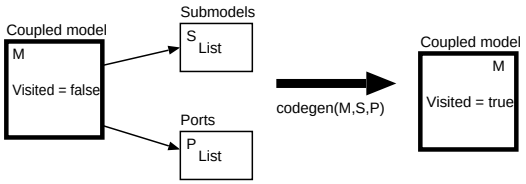

Figure 4: A typical code generation rule.

rule of interest for an arbitrary number of neighbours. One possible solution is to create a special "collecting" node, and have a rule that adds the neighbours to a list in this collecting node. This rule, when applied, marks each neighbour as visited so that it is not added twice. The rule also should have a priority higher than that of the actual code generation of the model of interest, since code generation should happen only after all the relevant neighbour's information has been collected.

The code generation rules themselves do not perform any important rewriting aside from getting rid of annotations such as the collecting nodes mentioned above. The code generation is performed by the actions, which can access the annotations.

An example of the code generation rule of a coupled model, showing the collecting nodes is depicted in Figure 4. The collecting nodes (S and P) each contain a list of the names of the submodel nodes and port nodes respectively. The rule simply deletes the annotations (the collecting nodes,) and its action is to call an external function passing it the model and the relevant annotations. The action is executed before the graph rewriting takes place. The rule also marks the model's node as visited so that it will not be applied again to that node.

7 CONCLUSIONS

The DEVS formalism allows the rigorous description of complex dynamic systems. Its main advantages are the definition of component-based models and the efficient simulation algorithms for these models. The graphical structure of such component-based models naturally leads to graph-based representations. This in turn, motivates the construction of a visual modelling and simulation environment. In this paper we have introduced such an environment, which generates well-structured dedicated simulators for the models. We also emphasise meta-modelling and graph-transformation as suitable frameworks for the construction of such modelling tools.

Praehofer and Pree [3] proposed a similar modelling and simulation tool. The main difference from our approach is that the modelling component and the simulation component of our tool are explicitly decoupled. The simulator generator is independent of the modelling environment. This generator, specified by a graph-grammar, can be easily replaced in order to target other DEVS simulation tools such as ADEVS, CD++, DEVS/C++, JDEVS, etc. [7]

Future work will address the extension of the tool to other variants of the DEVS formalism, in particular *parallel* DEVS. Another line of work that needs attention is the use of graph-transformation to formally describe the operational semantics of DEVS.

References

[1] Jean-Sébastien Bolduc and Hans Vangheluwe. A modelling and simulation package for classic hierarchical DEVS. Technical report, McGill University, School of Computer Science, 2002.

[2] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). 73:1–69, 1979.

[3] Praehofer H. and Pree D. Visual modelling of DEVS-based multiformalism systems based on higraphs. In *Proceedings of the 1993 Winter Simulation Conference*, pages 595–603. SCS, 1993. Los Angeles, CA.

[4] Juan De Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, volume 2306 of *LNCS*. Springer-Verlag, 2002.

[5] John Power and Konstantinos Tourlas. An algebraic foundation for graph-based diagrams in computing. In Stephen Brooks and Michael Mislove, editors, *Electronic Notes in Theoretical Computer Science*, volume 45 of *ENTCS*. Elsevier Science Publishers, 2001.

[6] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1999.

[7] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2000.

[7] Information about all these tools can be found at http://www.sce.carleton.ca/faculty/wainer/standard/.