On the Prototype-Based Object Orientation in Systems Modeling and Simulation

Vladimír Janoušek [*]
janousek@fit.vutbr.cz

Abstract: This paper deals with an alternative to traditional class-based approach to the simulation modeling. Benefits of the prototype-based approach are discussed. Our focus is restricted to DEVS, a Disctere Event Systems Specification formalism.

Keywords: DEVS, Smalltalk, Self

1 Introduction

DEVS [1] specifies a system as a hierarchically composable component. A model can be specified as a coupled model comprising interconnected subsystems, or as an atomic model. Atomic model is a state machine described by state and four functions – external transition δ_{ext}, output function λ, internal transition δ_{int}, and time advance ta. The theory behind DEVS comprise also abstract simulators for atomic and coupled models.

Nowadays, the majority of DEVS modeling and simulation tools are implemented in C++ or Java. Implementation of DEVS in class-based object-orented languages obviously leads to modeling by subclassing the existing models. Subclasses can define instance variables for the representation of state, redefine the methods corresponding to the functions δ_{ext}, λ, δ_{int}, ta for atomic models, and specify a component list plus coupling relation for coupled models. In both cases, an initialization method is responsible for creating input and output ports.

SmallDEVS is another DEVS implementation. It is implemented in Squeak, a free and open source Smalltalk. Besides the class-based approach to the modeling, SmallDEVS supports also a more flexible approach—a prototype-based model construction, which is explained in the paper by examples.

2 Prototype-based DEVS implementation

Class-based modeling suffers from flexibility when dealing with evolving and self-modifying models. Especially the models which are built in statically compiled languages such as Java and C++ are very limited in their flexibility because all the code which could be possibly needed has to be known at the compile time. Dynamic modifications of a model during a simulation is limitied to the structural changes only. Dynamic languages are more flexible. Nevertheless, even the dynamic class-based object-oriented languages do not offer enough flexibility. If we have several instances of the same model and we want to change only one of them in a specific

[*] Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 61266 Brno, Czech Republic

way, most likely we have to define a separate class for it. It is not an essential problem, but it is a complication.

Smalltalk offers a possibility of inspecting, creating, and modifying classes during the program execution. Moreover, there also exists an extension (package Prototypes), that allows us to modify the structure and behavior of the individual instances. It makes possible to deal with prototype objects. A prototype object can be created as an instance of the class PrototypeObject, or as a clone of another prototype object. The class PrototypeObject defines a protocol that allows us to edit slots and methods for any particular prototype object without a need to define a new class for it. In this way the prototype objects can behave completely differently depending on their slots and methods which can be edited at run-time. We need such a degree of flexibility in order to allow reflective and evolving systems modeling, and interactive building of model under simulation (we call it *exploratory modeling*).

3 Example

An example of the prototype-based approach to the modeling in SmallDEVS is shown below. The sequence of expressions is a program which builds the Generator-Processor model incrementally without a need to introduce new classes. The program can be executed at once, as a script or, the expressions can be evauated interactively, in a step-by-step manner (expression by expression) in smalltalk workspace. We first have to build a generator of jobs.

```
| system generator processor jobPrototype |

jobPrototype ← PrototypeObject new.
jobPrototype addSlots: { 'n' -> 0.  'size' -> 0.  'name' -> 'aJob'. }.
jobPrototype addMethod: 'setSizeBetween: sl and: sh
                self size: (sl to: sh) atRandom'.

generator ← AtomicDEVSPrototype new.
generator addSlots: {
  'jobPrototype' -> jobPrototype.
  'ia' -> 2. "interval min"   'ib' -> 7. "interval max"
  'sa' -> 5. "job size min"   'sb' -> 10. "job size max"
  'first' -> true.    'n' -> 0.   "number of jobs generated"}.
generator intTransition: 'self first: false'.
generator outputFnc: '
  self n: self n +1.
  self
    poke:
      ((self jobPrototype setSizeBetween: self sa and: self sb) clone
        n: self n;
        yourself)
    to: #out'.
generator timeAdvance: '
  ↑ self first
      ifTrue: [ 0 ]
      ifFalse: [ (self ia to: self ib) atRandom ]'.
generator addOutputPorts: {#out}.
```

The next several expressions build the processor model.

```
processor ← AtomicDEVSPrototype new.
processor addSlots: {
  'queue' -> OrderedCollection new.
  'queueSize' -> 5.
  'processorStatus' -> #idle.
  'currentJob' -> nil.
  'timeSpent' -> 0 }.
processor addInputPorts: {#in}.
processor addOutputPorts: {#out. #discard}.
processor intTransition: '
  self processorStatus caseOf: {
  [ #busy ] -> [
    self queue size > 0
      ifTrue: [
        self currentJob: (self queue removeFirst) ]
      ifFalse: [
        self processorStatus: #idle.
        self currentJob: nil ].
    self timeSpent: 0 ].
  [ #discard ] -> [
    self queue removeFirst.
    self queue size <= self queueSize ifTrue: [
      self processorStatus: #busy ]].
  [ #idle ] -> [ "nothing" ] } '.
processor extTransition: '
  self queue add: (self peekFrom: #in).
    self processorStatus caseOf: {
  [ #idle ] -> [
    self processorStatus: #busy.
    self currentJob: (self queue removeFirst) ].
  [ #busy ] -> [
    self timeSpent: self timeSpent + self elapsed.
    self queue size > self queueSize
      ifTrue: [ self processorStatus: #discard ]].
  [ #discard ] -> [ "nothing" ] } '.
processor outputFnc:  '
  self processorStatus caseOf: {
  [ #busy ] -> [ self poke: self currentJob to: #out ].
  [ #discard ] -> [ self poke: (self queue last) to: #discard ].
  [ #idle ] -> [ "nothing" ] } '.
processor timeAdvance: '
  self processorStatus caseOf: {
  [ #busy ]   -> [ ↑ self currentJob size − self timeSpent ].
  [ #discard ]  -> [ ↑ 0 ].
  [ #idle ]     -> [ ↑ Float infinity ] } '.
```

The final sequence of expressions couples the components together.

```
system ← CoupledDEVSPrototype new.
system addOutputPorts: {
    #out.
    #discard }.
  system addComponents: {
    #generator -> generator.
    #processor -> processor }.
  system addCouplings: {
    #(generator out)  -> #(processor in).
    #(processor out)  -> #(self out).
    #(processor discard) -> #(self discard) }.
```

4 Behavior sharing and reusability

If we need more processors in our example, we can easily make clones of the existing processor. Nevertheless, later modifi cations of a processor behavior will affect only the individual processor. If we need a possibility to modify behavior of all the clones, we need the behavior to be shared. The behavior of all processors have to be specifi ed separately, as a trait (which is nothing but another prototype object), and shared by all processors by means of delegation (which is also called dynamic inheritance). Each time we modify the trait, the behavior of all processors is affected. The trait can be created by executing the followng code.

```
  processorTrait ← AtomicDEVSTrait new.
  processorTrait addMethod: 'intTransition    ........'.
  processorTrait addMethod: 'extTransition ........'.
  processorTrait addMethod: 'outputFnc ........'.
  processorTrait addMethod: 'timeAdvance ........'.
```

The intTransition, extTransition, outputFnc, timeAdvance defi nitions in the trait are the same as in the previous example. Together with the trait, we need to specify also a prototype of a processor. The processorPrototype defi nition looks like this:

```
  processorPrototype ← AtomicDEVSPrototype new.
  processorPrototype addSlots: {
        'queue' -> OrderedCollection new.
        'queueSize' -> 5.
        'processorStatus' ->  #idle.
        'currentJob' ->  nil.
        'timeSpent' -> 0 }.
  processorPrototype addInputPorts: {#in}.
  processorPrototype addOutputPorts: {#out. #discard}.
  processorPrototype addDelegate: 'defaultTrait' withValue: processorTrait.
```

By delegating the behavior to the trait we ensure that the behavior is shared by all clones of the processorPrototype. Now we can specify a system with several processors with the same, shared behavior.

```
  system ← CoupledDEVSPrototype new.
  system addOutputPorts: { #out. #discard }.
  system addComponents: { #generator -> generator }.
  previousPort ← {#generator. #out}.
  1 to: 3 do: [ :i |
    newProcessor ← processorPrototype copy.
    newProcessorName ← (#processor, i printString) asSymbol.
    system addComponents: { newProcessorName -> newProcessor }.
    system addCouplings: {
      previousPort -> {newProcessorName. #in}.
      {newProcessorName. #out} -> {#self. #out} }.
    previousPort ← {newProcessorName. #discard} ].
  system addCouplings: {
    {newProcessorName. #discard} -> {#self. #discard} }.
```

Note that the traits can also delegate parts of their behavior to other traits. This way, traits can play the role of classes and the delegation can play the role of inheritance. Also note that multiple delegation is possible as well as runtime changes of delegates. We can see that no feature of class-based object-orientation has been lost. What is more, the prototype-based object-orientation offers more flexibility needed for interactive modeling and simulation.

5 Reflective features and operating system

Reflectivity is an essential feature of SmallDEVS—we can not only build a model incrementally but we can also inspect what has been actually built (what is really needed if we allow models to evolve automatically) and in which state the simulation is. Anything we can do interactively, the models can do themselves, as well. This opens an interesting area of reflective systems modeling and simulation. Anyway, we need an operating system in which the modeling and simulation may take place.

Smalltalk represents the basic layer of the operating system for SmallDEVS. It offers virtual machine, incremental compiler, mutitasking, rich class library, input/otput, and interactive development tools. The basic Smalltalk tools which are important for SmallDEVS, are Workspace and Inspector. Workspaces allow for editing texts and executing code snippets. Resulting objects can be examinated by inspectors. This way the models can be edited, inspected and manipulated incrementally. It is also possibe to start a simulation in background and see the log in Transcript. The following code works with the previously created model pointed by the variable $system$:

```
s ← system getSimulatorRT.
s stopTime: Float infinity.
s RTFactor: 1.
s start.
```

Then we can communicate with the simulator (pause the simulation, let it continue etc.), inspect and edit any part of the simulated model during the simulation (the system ensures that the editing operations take place at the right time between simulation steps) interactively, in the same way as we have created and examined the model in the previous sections. Models, as well as simulations, can be cloned:

```
aModel2 ← aModel copy.
aSimulation2 ← aSimualtion copy.
```

An important task of the SmallDEVS operating system is persistency. Models and simulations can be stored in workspace variables. Nevertheless, much better solution of persistency can be offerd by a hierarchical fi lesystem-like structure. MyRepository represents a hierarchy of folders and objects. This tree is unique in the system and is rooted in Smalltalk as a global variable. Generally, MyRepository can hold any object that understans some basic protocol allowing for hierarchical composition. MyRepository is used here as a container for models and simulations.

Although MyRepository resembles a filesystem in a traditional OS, the main difference from files is the fact that objects are live entities residing in Smalltalk object memory, while files are nothing but named strings of bytes lying on some external media. Although the SmallDEVS objects can be ”externalized” using XML or as a storeString (a Smalltalk code which, when executed, recreates an exact copy of the original object), their primary form is the live form in the object memory of Smalltalk which can be stored and restored at once as it is in Smalltalk obvious. Objects (simulations, models, as well as their components) in MyRepository can be accessed in the following way:

```
MyRepository at: '/Simulations/GeneratorAnd3Processors' put: system.
aComponent ← MyRepository at: '/Simulations/GeneratorAnd3Processors'.
aComponent addComponents: { name1 -> object1. name2 -> object2 }.
```

Note that a copy of a model can be made at any time during the simulation, of course. What is important, any copy of the system made during the simulation can be used as an initial state for another simulation, and/or saved as text for possible editing the code by hand.

6 Conclusion

The paper has demonstrated an alternative to mainstream approach to DEVS-based modeling and simulation. In the prototype-based OO, the focus is on concrete objects which are allways ready for run (can be simulated). Possible shared behavior can be easily extracted from them and put to shareable traits. Objects which are considered to become patterns for cloning can be put among other well-known objects (and available by pathname in MyRepository) as prototypes. Models can be edited during simulation and any state of the simulation can be considered to be a model – its “source code” can be generated from the live model at any time. These features are suitable for experiments with more dynamics in modeling and simulation.

The interactivity and feeling of concretness can be significantly amplified by an appropriate GUI. SmallDEVS GUI has been higly influenced by the GUI of Self [3]. Current version of SmallDEVS can be downloaded from its website [4].

This work has been supported by the Grant Agency of Czech Republic grant No. 102/04/0780 “Automated Methods and Tools Supporting Development of Reliable Concurrent and Distributed Systems”.

Bibliography

1. B. P. Zeigler, H. Praehofer, T. G. Kim: *Theory of Modeling and Simulation Second Edition*, ACADEMIC PRESS, 2000
2. Bolduc, J. S. and H. Vangheluwe: *A modeling and simulation package for classic hierarchical DEVS.* Internal document for the MSDL, School of Computer Science, McGill University, 2002
3. Self, http://research.sun.com/self
4. SmallDEVS, http://www.fit.vutbr.cz/˜janousek/smalldevs