EXPLORATORY MODELING WITH SMALLDEVS

Vladimír Janoušek
Elöd Kironský
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 61266 Brno, Czech Republic
e-mail: {janousek | kironsky}@fit.vutbr.cz

KEYWORDS

DEVS, prototype object, trait, delegation, clonig, reflectivity, Smalltalk, Self, GUI

ABSTRACT

This paper is an introduction to the simulation and modeling framework and tool SmallDEVS. It is a new modeling and simulation framework for Smalltalk. SmallDEVS is different from other tools of its category, because of its openness and reflective features. It supports class-based as well as prototype-based object-oriented model construction. Its meta-object protocol allows the models to be constructed from scratch and inspected and edited during run-time. Interactive modeling and simulation is supported by a graphical user interface which has been highly influenced by the user interface of Self.

INTRODUCTION

This paper introduces a new modeling and simulation tool for the programming language Smalltalk named SmallDEVS. It is an experimental software. SmallDEVS has been designed mainly for experiments with evolving, self-modifying models and with interactive modeling under simulation. The tool is being developed and tested since 2003. As one could already foretaste, SmallDEVS is based on the DEVS (Discrete Event System Specification) formalism. DEVS was introduced in 1976 by Bernard P. Zeigler (University of Arizona). The formalism specifies a system hierarchically. 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 its state variable and four functions – external transition δ_{ext}, output function λ, internal transition δ_{int}, and time advance ta. The theory behind DEVS comprises also abstract simulators for atomic and coupled models.

DEVS makes systems modeling and simulation clear and easily understandable whereas it keeps a relatively simple structure. There are many variants of the DEVS formalism. This text will consider only the classical version of DEVS.

Since the invention of the DEVS formalism, new implementations for various programming languages are coming up. Most of these programming languages are object oriented like C++ or Java. SmallDEVS package is a DEVS implementation for Smalltalk. While SmallDEVS allows us to implement a model as a class in a traditional fashion, we prefer the use of prototype objects to create models because of a higher flexibility of this approach. The creation of the models and the experimentation with them is supported by a graphical user interface which is highly influenced by the user interface of Self, a prototype-based object oriented language and system (Ungar and Smith 1989). More concretness and more interactivity in the construction of models of discrete-event systems—these are the main ideas behind SmallDEVS development. It is our believe that the implementation of these ideas can significantly contribute to the quality of the "understanding by modeling".

Our motivation to design and implement a new simulation and modeling tool is discussed in the next sections. We will explain, why did we choose the Smalltalk programming language and what are the innovative features of SmallDEVS. We assume, that the reader is already familiar with the details of the DEVS formalism (Zeigler at al. 2000) and therefore we will skip it.

CLASS-BASED DEVS IMPLEMENTATION

The majority of DEVS modeling and simulation tools is implemented in C++ (Zeigler at al. 1996) or Java (Zeigler at al. 1997). An implementation of a DEVS model in a class-based object-orented languages obviously leads to the modeling by subclassing the existing models. The subclasses can define new instance variables for the representation of state; redefine the methods corresponding to the four main functions (internal transition, external transition, output function and time advance function) of atomic models; and specify a component list and a coupling relation for the coupled models. An initialization method is responsible for creating input and output ports.

SmallDEVS supports the class-based approach to the modeling in a way that is very similar to that of Python DEVS (Bolduc and Vangheluwe 2002). Both of these frameworks are very close because they are implemented in dynamically typed languages. Nevertheless, this paper deals with a more flexible approach—a prototype-based model construction, which is explained in the next section.

PROTOTYPE-BASED DEVS IMPLEMENTATION

The class-based modeling brings some complications into the play when we deal with evolving and self-modifying models. Especially the DEVS implementations which are built using 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 to a model during a simulation are limited to the structural changes only. Every time we want to modify the behavior of an atomic model, we must recompile the code of the coresponding class and restart the simulation.

The traditional approach to the dynamic model implementation relies on a well-designed set of fine-grained atomic models. The dynamics is then expressed by the structural changes of the coupled models. Dynamic languages are more flexible. They allow also the atomic models to be dynamicaly changed during runtime. Nevertheless, even the dynamic class-based object-oriented languages do not offer enough flexibility. For example, if we have several instances of the same model and we want to change only one of them in a specific way, most likely we have to define a separate class for it. It is not an essential problem, but it is a complication which can be easily eliminated by switching to the prototype-based (i.e. classless) approach.

SmallDEVS is implemented in Squeak Smalltalk (Ingalls et al. 1997) using an extension, that allows to modify the structure and behavior of the individual instances. This extension is installed with the package Prototypes. This package makes possible to create prototype objects. A prototype object can be created as an instance of the class PrototypeObject, or as a clone of another prototype object:

$aPrototypeObject := PrototypeObject\ new.$
$anotherPrototypeObject := aPrototypeObject\ clone.$

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:

$aPrototypeObject\ addSlots : \{$
$'name1' -> anObject.$
$'name2' -> anotherObject\}.$

$aPrototypeObject\ addMethod :$
$'messageSelector\ ...(method\ body)...'.$

Values of the slots can be accessed by sending the appropriate messages to the objects, e.g. $self\ slotName$, or $self\ slotName : aValue.$

Shared behavior can be specified by means of *traits*. Traits are prototype objects which contain methods which are intended to be shared (dynamically inherited) by other objects (models). Other objects can delegate messages to them (it is also refered to as the dynamic inheritance or the instance-based inheritance). The delegates (traits) can be specified by the delegation slots:

$aPrototypeObject\ addDelegates : \{$
$'name1' -> aTrait.$
$'name2' -> anotherTrait\}.$

Note that the traits can also delegate parts of their behaviour to some other traits. This way, the traits can play the role of classes and the delegation can play the role of inheritance. Also note that a multiple delegation is possible, as well as a runtime changes of the delegates. We can see that no feature of class-based object-orientation has been lost. What is more important, the prototype-based object-orientation offers more flexibility which is needed for interactive modeling and model refactoring. The prototype objects can behave completely differently depending on their slots and methods which can be incremetally edited at run-time (we can add and also remove slots, methods, and delegates). This feature opens a huge box of possibilities. The atomic models can be created in a very simple way from prototypes by adding slots, delegates and methods and then they can be modified dynamically. Such a degree of flexibility is needed to support

- reflective and evolving systems modeling – as an example, we can mention anticipatory systems (Rosen 1985), which perform nested simulations of themselves, possibly with some modifications, in order to support their decisions about their next actions and self-improvements;
- interactive and incremental construction of the models under simulation – we call it *exploratory modeling*, similarly to the notion of exploratory programming which represent the way of programming in Smalltalk which is based on wast exploration of the actual state of a running program, together with the program modifications during runtime.

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 leads to an interesting area of reflective systems modeling and simulation.
When we used class-based approach, we needed classes to be created and discarded during the model evolution. Smalltalk can be used this way (classes can be created even on the fly, without registering in the smalltalk class repository), but the prototype-based approach is much simpler (we don't have to deal with classes if we don't need them) and more flexible (thanks to the dynamic inheritance and individual objects modifications).

SmallDEVS allows an atomic DEVS to be created by executing the following expressions:

$model := AtomicDEVSPrototype\ new.$

$model\ addSlots : \{'name' - > value....\}.$
$model\ addInputPorts : \{'name1'.\ 'name2'....\}.$
$model\ addOutputPorts : \{'name1'.\ 'name2'....\}.$
$model\ addDelegates : \{'name' - > aTrait\}.$
$model\ intTransition :\ '...(method\ body)...'.$
$model\ extTransition :\ '...(method\ body)...'.$
$model\ outputFnc :\ '...(method\ body)...'.$
$model\ timeAdvance :\ '...(method\ body)...'.$

A coupled DEVS can be created by executing the following expressions in SmallDEVS:

$model := CoupledDEVSPrototype\ new.$
$model\ addInputPorts : \{name1.name2....\}.$
$model\ addOutputPorts : \{name1.name2....\}.$
$model\ addComponents : \{name- > aComponent....\}.$
$model\ addCouplings : \{$
$\#(component1\ port1) - >\ \#(component2\ port2).$
$\#(component3\ port3) - >\ \#(component4\ port4)....\}.$

OPERATING SYSTEM

SmallDEVS has been designed specifically for the experimenting with several interesting techniques such as multi-simulations, reflective systems simulation studies, interactive modeling and simulation, and model based design. The following requirements were set on SmallDEVS:

- manipulation with models (create, delete, inspect, edit),
- manipulation with simulations (run, stop, resume, inspect, cloning, saving and restoring simulation state, nested simulations),
- it should be no difference between manipulating models in a running simulation or separately,
- interactivity and visualization.

As to the manipulation with models, the abstract examples in the previous section showed how to create models incrementally. Beside that we are able to aquire any detail about the model - slot names and content, method sources, ports, delegates, components, couplings and we can also remove them and/or edit them. This makes full inspecting and editing of our model possible. These changes can be made either interactively or programatically. The same operations are also available during a running simulation. SmallDEVS makes sure, that the editing operations are executed safely between the simulations steps. One can also synchronize the simulation with real-time to support interactive and HIL simulations.

Simulations can be started and controled in a following way:

$aSimulation := aModel\ getSimulatorRT.$
$aSimulation\ stopTime :\ Float\ infinity.$
$aSimulation\ RTFactor :\ 1.$
$aSimulation\ start.$
$aSimulation\ stop.$

The simulation runs on the background and it is possible to start and control more simulations simultaneously. The models, their parts, as well as the simulations, can be cloned:

$aModel2 := aModel1\ copy.$
$aSimulation2 := aSimualtion2\ copy.$

A copy of a simulation creates a copy of the complete state of the simulation. 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 model made during the simulation can be used as an initial state for another simulation, and/or saved as a text for possible editing of the code by hand.

An important responsibility of SmallDEVS operating system is persistency. On the basic level, any object pointed to by some Smalltalk Workspace variable is persistent (and can be stored as a part of the Smalltalk object memory). Nevertheless, for serious work it is not sufficient and SmallDEVS offers a better solution. The models, as well as the simulations (either running, or ready-to-run) can be stored and organized in a structure named MyRepository (the name is relatively general because it is intended not only for models and simulations). MyRepository represents a hierarchy of folders and objects. Objects which are considered to become patterns for cloning can be put among other well-known objects, into the objects tree (and available by a pathname in MyRepository) as prototypes. This tree is unique in the system and is rooted in Smalltalk as a global variable. Generally, MyRepository can hold any object, that understands a protocol allowing for hierarchical composition of objects and folders. The inspiration came from filesystems of traditional operating systems. The main difference from files in such systems is the fact that objects are live entities residing in Smalltalk object memory, while files are nothing but named strings of bytes lying passively 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. Thus, they can be stored and restored at once—in a form of the so-called image—as it is in Smalltalk obvious. Objects (simulations, models, as well as their components) can be accessed in the following way:

$MyRepository\ at :\ '/Sims/TestSim'\ put :\ system.$
$aComp := MyRepository\ at :\ '/Sims/TestSim'.$

The overall structure of the SmallDEVS system is depicted in Figure 1. The lowest level of the SmallDEVS

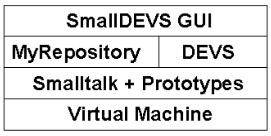


Figure 1: SmallDEVS system

system is the Squeak Smalltalk Virtual Machine. It is responsible for the interpretation of Smalltalk Image (the place where all the objects reside). VM is implemented in a small portion of the C language, being completely portable to almost all known platforms. Smalltalk image with Prototypes package represent another level of the system. The core parts of SmallDEVS are MyRepository and the DEVS simulator. The core contains all the classes that implement the real time simulator and the wrapper classes, that define the reflective interface to the inner prototype objects (traits, atomic models, etc.). The topmost level represent the SmallDEVS GUI which is described in the next sections.

VISUAL TOOLS FOR EXPLORATORY MODELING

The feeling of concretness of the prototype-based approach can be significantly amplified by an approporiate graphical user interface. The SmallDEVS GUI has been higly influenced by the GUI of Self, a prototype-based object oriented language. Self's GUI with direct manipulation of objects significantly amplifies the concretness which is inherent in prototype-based programming. Self's objects can be inspected and modified by the so-called outliners. In fact, the outliner is a merge of inspector and browser, which follows the fact that a prototype object is a standalone object – it has its own data and methods. Self's GUI is able to visualize inter-object relations and modify them in a drag-and-drop manner. Object refactoring (moving slots between objects) is also supported in the same, intuitive and concrete way.
Another inspiration for the SmallDEVS GUI came from the file managers known from the traditional operating systems - they allow fow creating, copying (by cut/copy/paste actions), renaming and opening files which are organized in folders. In our case, we use this approach to the objects organized in MyRepository.

SmallDEVS allows the models to be created by the GUI or without it. They can be generated by a program. The user interface can visualize and manipulate a model despite the way how the model has been created. The visualization if completely transparent in SmallDEVS. The model components are primary entities, while the user interface is secondary. GUI can be opened on any component of a model at any time thanks to the reflective interface of the models. Each component under investigation has its own, independent GUI instance. The main components of the SmallDEVS graphical user interface are described in the following sections.

MYREPOSITORY BROWSER

MyRepository Browser provides an access to the context menus of the objects. MyRepository is used mainly as a container for models and simulations, but it can contain basically any object (for example documents, pictures, binary data, etc.). Figure 2 shows the window of a MyRepository Browser. You can see the hierarchical tree of objects as well as several simulations in the 'Simulation' directory (subtree).

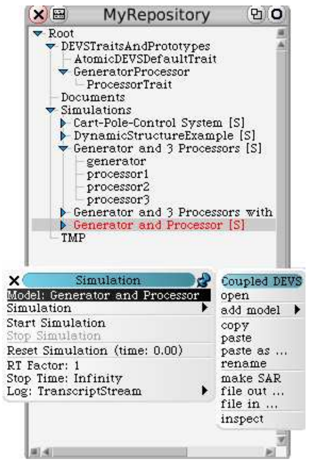


Figure 2: MyRepository browser

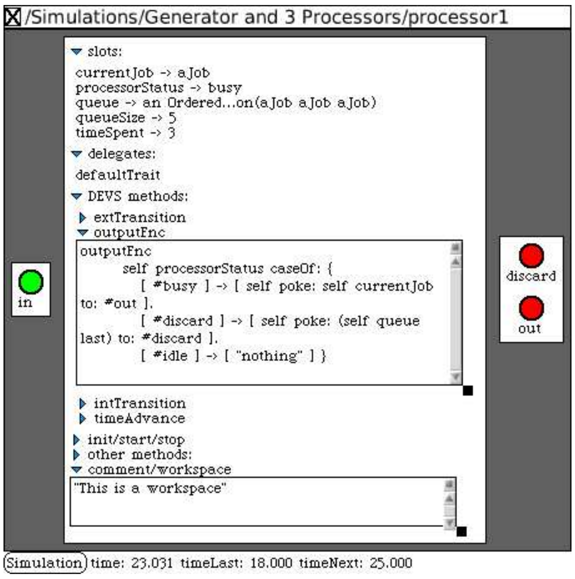
At the bottom of the figure is an opened context menu of the simulation named 'Generator and Processor [S]' that was stopped at the time of the screenshot. From the simulation context menu, the coupled DEVS context menu is poped up, where you can see the operations available on the coupled model.

ATOMIC MODEL INSPECTOR

One of the two main SmallDEVS tools is the atomic model inspector and editor. It is a tool that makes the implementation of atomic models more user friendly, but one still has to write the implementation of the model's behaviour. The design of this editor is heavily inspired by the Self language and its outliner. It allows us to define slots for the model, assign vaues to them, define the four basic methods of a DEVS model and to add other methods when they are needed. A special initialization method is prepared for the user, that is executed at the moment, when the simulation is restarted (or at the first run). This method ensures, that every model in a simulation can be returned to initial state at once or separately when needed. A so called "Workspace" is also part of the tool, where arbitrary expression can be evaluated interactively in the context of the inspected object. This way, the values of the slots can be changed (among other things like the execution of scripts, etc.). A screenshot from a editor of a simple atomic model is in Figure. 3. Notice the expandable editors of methods. The header contains the full path within the MyRepository hierarchy. It is possible to add or remove input ports on the left side and output ports on the right. Also, the bottom status bar provides information about the simulation and the simulation control is accessible from here, too.

COUPLED MODEL INSPECTOR

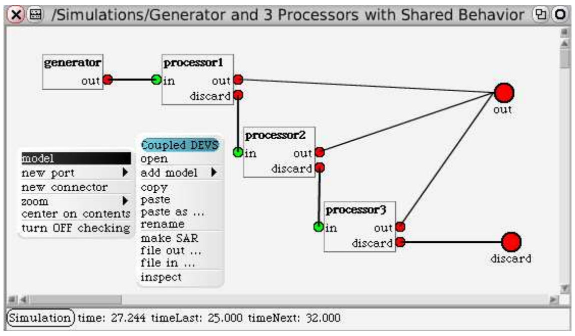
The coupled models inspector is a tool to build, inspect and edit coupled models. The connections editiong, together with copying, cutting, pasting, and renaming of the models are supported. The viewable area can be zoomed in or out. Ports and new atomic/coupled models can be added and removed. Also there is an option to choose a model from existing mod-



```
/Simulations/Generator and 3 Processors/processor1
▼ slots:
  currentJob -> aJob
  processorStatus -> busy
  queue -> an Ordered...on(aJob aJob aJob)
  queueSize -> 5
  timeSpent -> 3
▼ delegates:
  defaultTrait
▼ DEVS methods:
  ▶ extTransition
  ▼ outputFnc
   outputFnc
         self processorStatus caseOf: {
            [ #busy ] -> [ self poke: self currentJob
   to: #out ].
            [ #discard ] -> [ self poke: (self queue
   last) to: #discard ].
            [ #idle ] -> [ "nothing" ] }
  ▶ intTransition
  ▶ timeAdvance
▶ init/start/stop
▶ other methods:
▼ comment/workspace
  "This is a workspace"
Simulation time: 23.031 timeLast: 18.000 timeNext: 25.000
```

Figure 3: Inspector of atomic models

els in the hierarchy of models and copy that particular model. Like in the editor of atomic models, here is also a status bar with the same function. Figure 4 shows an editor over a simple coupled model.



```
Simulation time: 27.244 timeLast: 25.000 timeNext: 32.000
```

Figure 4: Inspector of coupled models

SUMMARY

The paper showed why and how the prototype-based object orientation can help us to build a tool which can support structurally dynamic and evolving DEVS models and exploratory modeling. SmallDEVS is a highly interactive tool for modeling and simulation. Its real power is in rapid prototyping of DEVS models. It supports model modifying during simulation (interactively as well as programmatically). Generally, SmallDEVS is designed to allow vast experimentations with a model without having to recompile it and start over the simulation each time the model changes. Persistency of models and simulations is also supported, as well as interconnecting models with real components (hardware in the loop). An interesting topic of the future research and development is a meta-language, that could describe DEVS models independently on the underlying software and hardware architecture. This would allow us to develop models and debug them in SmallDEVS and then simulate them on a performance optimized simulator in C++ or on a distributed simulation engine. For intelligent systems simulation, we plan to develop a library of soft-computing components. We also plan to allow the atomic models to be specified by other formalisms such as Petri nets and state charts. Other fields of interests are some applications of the model continuity concept in the intelligent systems development. The current version of SmallDEVS is available on its web site *http://www.fit.vutbr.cz/˜janousek/smalldevs*.

ACKNOWLEDGEMENT

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”.

REFERENCES

Bolduc, J. S. and H. Vangheluwe. 2002. ”A modeling and simulation package for classic hierarchical DEVS”. *Internal document for the MSDL*, School of Computer Science, McGill University

Ingalls, D.; Kaehler, T.; Maloney, J.; Wallace, S.; Kay, A. 1997. ”Back to the future. The story of Squeak, a practical Smalltalk written in itself.”. *OOPSLA ‘97 Conference Proceedings*, 318-326.

Ungar, D. and Smith, R. 1989. ”SELF: The Power of Simplicity”. *OOPSLA ‘87 Conference Proceedings*, 227-241.

Zeigler, B. P.; Y. Moon; D. Kim; J. G. Kim. 1996. ”DEVS/C++ A High Performance Modelling and Simulation Environment.”. *29th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, 350-359.

Zeigler B. P. 1997. ”DEVS-JAVA User’s Guide”. *Technical Report*, AI & Simulation Lab, Department of Electrical and Computer Engineering, University of Arizona, Tucson.

Zeigler, B. P.; H. Praehofer; T. G. Kim. 2000. ”Theory of Modeling and Simulation Second Edition”. Academic Press. ISBN 0-12-778455-1.

BIOGRAPHY

VLADIMÍR JANOUŠEK received the Ph.D. degree from the Faculty of Information Technology, Brno University of Technology in 1999. He is an assistant professor in the Department of Intelligent Systems at the Faculty of Information Technology, Brno University of Technology. His research focuses on simulation-driven development, pure object orientation and reflective architectures.

ELŐD KIRONSKÝ received the M.S. degree from the Faculty of Information Technology, Brno University of Technology in 2005. He is a Ph.D. student in the Department of Intelligent Systems at the Faculty of Information Technology, Brno University of Technology. His research focuses on modeling and simulation tools, robotics and exploratory modeling.