MODELLING TOOLS FOR A COMMON LISP OBJECT SYSTEM ENVIRONMENT

J. M. Farrow
S. Sevinc
The University of Sydney

Abstract

DEVS-CLOS is an object-oriented, hierarchical modelling environment for the Common Lisp Object System based on the Discrete Event System Specification propounded by Zeigler. Every part of the modelling environment is open for inspection and alteration. Some tools for coupling models based on the generation of Lisp symbols has been developed. As well, a method to traverse the model structure to find models fitting specific criteria has been developed. The CLOS class hierarchy is used to provide inheritance of behaviour for models.

Introduction

A simulation environment should closely parallel the modelling paradigm being used. This enables simulations of interesting situations to be created quickly and altered easily. Well developed environments allow information to be extracted from models in a clear and easy manner. A clear modelling paradigm is useless to the practical modeller without an effective implementation with which to simulate models and retrieve results.

The DEVS formalism

DEVS-CLOS[3] derives its name from the modelling formalism used, the Discrete Event System (DEVS) formalism propounded by Zeigler[4,5] and the implementation language, the Common Lisp Object System. This formalism provides a means for a mathematical object called a system. A system has a time-base, inputs, states, outputs, and functions for determining the next state and output given the current state and inputs[5].

The DEVS formalism is of interest because of the formal representation it provides for model construction and its use in constructing simulations. The formalism provides sound semantics upon which to design simulation languages and environments. Such a framework also allows model behavioural analysis and simplifications — a complicated model constructed from a set of component models may be replaced by a simplified (or single) model which has the same, or very nearly the same, behaviour. This is of use in studying situations of interest at a high level without going into great detail for 'uninteresting' components[2].

The hierarchical nature of the DEVS formalism leads to a model with a part/subpart structure. Models are either atomic, the basic model unit, or consist of a collection of component child models coupled together to form larger models. Model descriptions are self-contained and are connected together via these coupling specifications. If A and B are models then their coupling together to form a new model AB is also in a modular form. This process is repeated to give hierarchical model descriptions.

The modular models used in the DEVS formalism have labelled inputs and outputs called ports. The coupling specifications state connections between the output ports of one model and the inputs of another, or, in the case of nested models, between a model's input ports and one of its child component input ports, or a child's output port to one of its parent's. It is allowable to have multiple output ports connected to the same input port or to have one output going to multiple inputs.

In figure 1 the coupled model AB is shown. The model AB consists of the component models A and B and the couplings AB.in→A.in, B.out→AB.out, A.out→B.in1, and A.out→B.in2.

It is natural to represent models as individual objects when they are described in this form by the formalism and the choice of CLOS as the implementation language allows this. CLOS also provides other features in terms of behavioural inheritance for models, described in detail later.

The simulation environment is important since it is the would be modeller’s interface to the modelling paradigm. If the environment does not fit the paradigm then describing models becomes more difficult. If the environment
The use of CLOS classes

The modelling classes available to the modeller in DEVS-CLOS form a tree rooted at the class entity, which is therefore a superclass of all DEVS-CLOS classes either directly or indirectly. It supplies, through inheritance, the attributes parent and name to all model components. Each model or submodel in an hierarchical modelling system has a parent and also a name, and so, being the most general attributes are supplied by the most general class. The DEVS-CLOS class tree is shown in figure 2.

Specifying models as part of a class tree leads to explicit statements about relationships between different model types both structurally and behaviourally. Through slot inheritance a new model class can inherit features of a parent model. Implementing the different state transition functions, which define the behaviour of the model, as CLOS methods specialised to a particular class also provides behavioural inheritance for models.

Models with similar behaviour, that is, models which share state manipulation functions such as the internal-transition, external-transition or output functions, may be declared as part of the same branch of the class tree. Specialised classes may be built from a common root. Usually this root is the class atomic-models. An example is shown in figure 3 of how the DEVS-CLOS model class tree could be extended.

Special behaviour for a model which otherwise behaves as a perfectly ordinary member of an existing model class can be achieved by defining a new model class which has as an ancestor a model of the required type and then replacing the different state transition functions by 'shadowing' them with more specialised methods on the new class.

As an example leading from the specialisations shown in figure 3, take tellers in a bank. The majority of tellers process customers and take a predetermined amount of time for each, either some given minimum transaction time or the length of the customer's request. One new teller, whom is still being trained, takes longer. In terms of the model all the tellers behave in the same way with respect...
to their internal-transition functions when a transaction is complete — they ask for the next customer — but in terms of their external-transition function, which calculates how long the processing for each customer will take and hence how long to remain in a particular processing phase, they will be different. A set of methods do describe the usual behaviour may be declared and specialised on the class \textit{teller}. To take into account the new teller's iniquity, a new class may be created, class \textit{new-teller}, as a child of class \textit{teller} and the differing external-transition function attached to this class. For all methods not specialised on \textit{new-teller} the methods on \textit{teller} will be applied. The altered behaviour will take over whenever a new customer is supplied to the trainee teller because of this more specialised external-transition function.

Such explicit statements of model relationships by the formation of a model class tree is much clearer than attaching individual copies of functions to model instances since binding functions to individual models means that the association of similar model types relies on noticing that the same function is used in more than one model. Using method specialisation and behavioural inheritance, groups of models may be defined with the same or similar behaviour more easily. Having this information stated explicitly highlights similarities between models in a simulation — models descended from a common parent have similar behaviour. As one moves further down the class tree, model behaviour becomes more specific. It remains possible to construct a special model case by defining a new 'one-off' model class and accompanying appropriately specialised methods. The usefulness of the method specialisation approach may be seen most in the case of groups of models all required to behave in the same manner, \textit{broadcast-models} for instance. The former implementation of DEVS-CLOS was lacking this ability for the DEVS state transition functions.

This method of model specification, a CLOS method specialised to a particular model class as opposed to an explicitly stored function, does not change the behaviour exhibited by a particular model. Rather two orthogonal hierarchies have been created. The first is the \textit{model structure hierarchy} consisting of the model tree of coupled \textit{atomic-} and \textit{coupled-models}. The expression of the model couplings describes this tree. \textit{Coupled-models} have children which may have further children until the leaves of the tree, the \textit{atomic-models}, are reached.

The classes of the models in this tree form another hierarchy, the \textit{model class hierarchy} which is an extension of the tree given in figure 2. Since this tree is different and separate from the first it allows model components of the same type to be used in different parts of a model. Taking as an example a shopping mall, if one is examining the flow of customers through the mall, it is necessary to find how the employees handle the customers. An atomic model class \textit{employee} is created to handle customers,-stalling them for a certain time at the 'register'. Such \textit{employee} models form a part of many of the components of the mall and in each case an instance of \textit{employee} may be declared and each would exhibit the same behaviour. They are related by their class although their place in the model structure are wildly different. In this case a method specified for \textit{tellers} may share many aspects of behaviour with \textit{employees} and it would be expected that the \textit{teller class} be a descendant of the more general \textit{employee} class (see figure 3).

![Figure 3 — Extending the DEVS-CLOS class tree](image)

**Creating model instances**

When a model instance is created, be it some type of \textit{atomic-} or \textit{coupled-model} it is necessary for DEVS-CLOS to attach a \textit{processor} to it before it can proceed with a simulation. The \textit{processor} instance is responsible for bookkeeping associated with those aspects of the simulation separate from the model such as the time of the next and last events and, for coupled-models, the child in which the next internal event will occur. Previously, in a similar manner to DEVS-Scheme\cite{4,5} such a correspondence was created with the macro \texttt{make-pair}. The user provided the name of the model and its corresponding model class and instances of the model and its processor would be created. The naming scheme varied according to whether an \textit{atomic-} or \textit{coupled-model} was being created and it was necessary for the user to remember this. Such low-level features should be hidden from the average user, but remain
accessible if needed in keeping with the open design of DEVS-CLOS.

DEVS-CLOS now performs all model initialisation, including creation of a corresponding processor instance, through the CLOS method initialize-instance, an after method specialised on the class model. The user now creates a model, say an instance of atomic-models, using exactly the same procedure as an instance of any other CLOS class, with a call to make-instance. DEVS-CLOS takes care of any bookkeeping necessary to integrate that model as part of the simulation under development.

The call to initialize-instance may be specialised as with any other CLOS method. Any keyword arguments to this method (those able to be specified with a keyword such as :name or :couple) become allowable arguments to the make-instance method. The default behaviour is to create a model with the name given, bind it to that symbol in the environment and attach to it an instance of co-ordinator or simulator. The last two options may be overridden by the modeller if so desired by providing nil values for the keyword arguments :bind-to-name or :link-to-processor. The form of the initialize-instance method is then

\[
\text{initialize-instance} \quad \text{[After method]} \quad \text{:after}
\]

\[
\text{(instance model) &rest initargs &key bind-to-name link-to-processor initialize-form}
\]

The final keyword argument initialize-form can be used to supply a Lisp form to carry out any one-off manipulations on the new model.

Coupled-model classes allow two other keyword arguments to be specified :child-list, the list of component models, and :couple, a list of model-port pair coupling specifications.

Symbol generation

Model couplings, be they components of a parent model to each other or a component to its parent, involves specifying mappings between a single model-port pair and a destination pair or list of pairs. A message appearing at an output port will be copied to all input ports specified in such mappings.

It is often the case when large numbers of models need to be connected together that these lists of coupling specifications can become large and unwieldy. Consider the simulation of a multi-processor architecture with tasks being farmed out to 'free' processors. In these cases the names of the component models are similar and a symbolic pattern generator can be used to specify couplings. This has the advantage of collapsing many lines of code detailing explicit couplings to a few lines, or even a single line, of a pattern and substitution ranges of numbers or characters.

In some situations it is not possible to specify beforehand the names of the children in such a manner. They may be wildly different and an expression could be too complex or impossible. In an extreme case the names of the children may not be known until later. Such couplings are not set up at model creation but at a later stage once all the required children of the parent have been inserted in the parent's child list. It is possible to use this list as a substitution range for symbol generation.

Model names are generated, not by matching regular expressions to internal symbols, but rather by substitution into a template. Given the template "FRED#-a", where the "-a" indicates the place of substitution, the modeller generates a list of symbols by supplying a suitable substitution range to the function expand. Each element of the list is substituted in turn into the template. The form

\[
\text{(expand "FRED#-a" '(1 2 3 4 5))}
\]

will return the list of symbols

\[
\{\text{FRED#1}, \text{FRED#2}, \text{FRED#3}, \text{FRED#4}, \text{FRED#5}\}.
\]

More than one substitution point may be given, for instance

\[
\text{(expand "EL#-a.-a" '(1 2) '(a b))}
\]

generates

\[
\{\text{EL#1.A}, \text{EL#1.B}, \text{EL#2.A}, \text{EL#2.B}\}.
\]

The signature of the expand method may be given (in a notation derived from [1])

\[
\text{expand (pattern string) \quad [Function]}
\]
For convenience, this function has also been implemented as a read macro, hence the notation

```
#!("EL#-a.-a" '(1 2) '(a b))
```

is a synonym for the second expression above which was

```
(expand "EL#-a.-a" '(1 2) '(a b)).
```

**Traversing the model structure hierarchy**

Symbol generation as a means of determining child models can also be coupled with the method `model-children`. This method allows a search of the model tree to be conducted to find specific model types, or more generally, models satisfying a given predicate. The method `model-children` searches the model tree from a given starting root model and finds all children satisfying the given predicate on the child’s model class. The default behaviour of this method examines all first-level (immediate) children of a model. Its full signature is thus

```
(model-children parent)
```

The root node is taken to be level zero of the tree. This means that children at level one are the immediate children of the root. Level two children are the children of the children of the root, and so on. This enables a list of children at the same depth of the tree to be found. In the case where an instance of `root-co-ordinators` is supplied to `model-children`, the result of the search is defined to be the result of a call with the same arguments beginning at the root-co-ordinator’s immediate child, i.e., the top level of the model. The special value `inf` may be supplied at the keyword argument corresponding to `:end` and results in a search terminating only at leaves of the model structure hierarchy.

Note that the test `#:typep model class` returns `t` if `model` is an instance of `class` or a subclass of `class`. Exact class matches can be found by specifying a suitable CLOS predicate for the `:key` keyword parameter.

This method, `model-children`, can be used in combination with the `expand` method to generate lists of model-port pairs. To connect all output ports of child models to the output port of a model `parent` the port-pair list may be generated with

```
(expand "((~a out) (parent out))"
  (model-children parent)).
```

For a model with children named `CPU#1`, `CPU#2`, `FPU`, and `FTU`, this would generate following coupling list

```
(((CPU#1 OUT) (PARENT OUT))
 ((CPU#2 OUT) (PARENT OUT))
 ((FPU OUT) (PARENT OUT))
 ((FTU OUT) (PARENT OUT))).
```

Model coupling can be carried out, and facilitated by the use of the `expand` function, either at model creation time, with the `:couple` keyword argument, or later with the `couple` macro and the use of the `model-children` method, whichever is more convenient to the modeller or the situation. The `couple` macro has the same effect as the `:couple` keyword argument and inserts the given couplings between model-port pairs into a model coupling specification list.

**Conclusion**

We believe, by paralleling the DEVS formalism so closely and by providing a totally open modelling environment with all the features of CLOS, DEVS-CLOS is a useful and powerful modelling tool. Part of the success of a simulation environment hinges on its ability to be used. Unobtrusive environments which provide a set of tools for modelling let the modeller concentrate on what is being modelled as opposed to remembering how to express it in the simulation environment.

**References**
