Moses – a tool suite for visual modeling of discrete-event systems

Robert Esser
Department of Computer Science
Adelaide University
Adelaide, S.A. Australia
esser@computer.org

Jörn W. Janneck
EECS Department
University of California at Berkeley
Berkeley, CA, U.S.A.
jwj@acm.org

Abstract

This paper gives an overview of the Moses tool suite, a set of tools for visual language programming. In Moses, visual language syntax is defined by first-order predicates over the abstract syntax of a picture–an attributed graph. One way of specifying language semantics in Moses is by writing an Abstract State Machine that interprets a given attributed graph. This paper shows how the editor is parameterized with a description of a visual language, and discusses briefly the generic architecture used to animate and debug visual programs.

1 Introduction

Modern software systems are becoming more and more *heterogeneous*, i.e. they are composed of a number of subsystems with very different behavioral characteristics: they may be state- and control-oriented, they may be algorithmic and computational, they may primarily be data- or material-flow systems, or they may combine some of these characteristics. Traditionally, very different notations (visual and otherwise) have been developed to describe such systems (e.g. StateCharts, Petri nets, dataflow graphs), reflecting the modeling requirements of their respective domain. The challenge in writing heterogeneous systems is to reconcile these different programming styles and incorporate them into a common platform.

As modeling domains develop, so do their notations. Furthermore, new domains may need to be incorporated as new kinds of subsystems are used in building these heterogeneous systems. Rather than a large but fixed set of predefined modeling notations, the most suitable architecture for a modeling environment is an open platform where adding or adapting existing modeling languages is made as easy as possible.

This paper presents the results of the Moses project [1], a tool suite that supports a wide variety of notations for discrete-event programming/modeling, focusing primarily on *visual* programming languages. Apart from giving an overview of the Moses tools as a modeling environment, this paper will outline the way visual notations are defined, both in their syntactical as well as their semantical aspects. For an in-depth discussion of these issues, cf. [14].

The remainder of this paper is structured as follows. After discussing some other contributions pertinent to this work, we will sketch the way visual language syntax is defined, and then show how this definition is used to configure the editing environment. After this we will give an outline of how the semantics of modeling languages can be defined, followed by a presentation of the animation/debugging/simulation environment.

2 Related work

There are many approaches to the specification and recognition of visual languages including grammar-based [19], (first-order) logic-based [6], graph grammars [17], constraint based [8], etc. (see [16] for a survey of these approaches). In this section we will concentrate on the area of heterogeneous system modeling.

Heterogeneous system modeling is not a new area of research. MOOSE [2] introduces the notion of a *multi-model*, where models can be composed of submodels of a set of different types. This set is mostly predefined, and MOOSE does not provide support for user-defined visual languages.

Ptolemy [12, 5] introduces the powerful notion of *domain* to describe semantically different interactions between modeling components. Both syntax and semantics of visual languages need to be defined in the host programming language, hence modifying or adding the definition of a visual notation involves considerable coding.

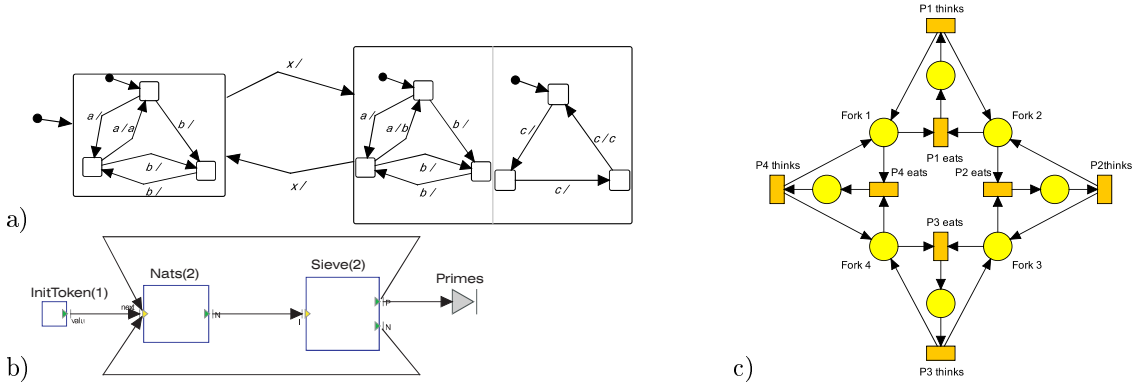

Figure 1: 3 different visual languages: a) hierarchical finite state machine, b) process network, c) Petri net

The Moses tool suite is closer to *meta-modeling* environments such as CodeSign, DOME, or GME. These expose an underlying 'meta-model' of the (in these cases, visual) notations they support that users may configure to specify a particular notation. CodeSign [4] defines a visual language by using a graph grammar that transforms it into a base formalism (a timed Petri net language). It provides extensive support for configuring the graph editor, although this usually involves the host language, Smalltalk. Also defining the semantics using graph grammars can be awkward unless the formalism to be specified is relatively close to the base formalism.

In DOME [10] (Domain Modeling Environment) a new visual notation is described as a high level specification of node and connector types, connection constraints, and additional syntax and semantics. Similar to Moses, nodes and connectors can also be attributed with sub-diagrams. Syntax checking is achieved by attaching boolean expressions to predefined *alter* methods. These methods are called from the editor and are initiated by user actions. This is in contrast to Moses where syntax checking is a non-intrusive background task (cf. section 4).

The approach to syntax definition taken by GME [11] is probably closest to that taken in Moses. In GME the syntax of a visual language is described using a UML-like visual notation and an OCL-like constraint language, the resulting meta-model is then used to configure the various tools. The main differences on the syntax definition side between GME and Moses are the way in which hierarchical models are constructed—they are embedded models in GME, whereas in Moses they are simply element attributes. Also GME is lacking some of the more advanced features of Moses syntax specification such as derived attributes, capabilities for handling embedded textual languages etc. GME and Moses also differ in the way they define visual language semantics—Moses provides a layered plug-in architecture (section 5 and an advanced simulation and animation engine, while GME has no formalized support for visual language semantics.

3 Defining visual syntax

The visual languages that we are concerned with here are *graph-like*, i.e. they consist of vertices (represented as closed geometrical shapes) and edges (represented as lines) connecting them. Fig. 1 shows 3 examples of models written in a variety of languages used to describe discrete-event systems. These languages differ in the kinds of visual elements they use, the way elements are connected, and the presence and kinds of (textual) decorations used.

Specifying visual language syntax in Moses is divided into two parts: (1) a definition of the *appearance* of the visual elements (e.g. colors, shapes, line types), and (2) a specification of the *well-formedness* of a picture in the language. While the appearance of visual elements is (mostly) a context-free property,[1] well-formedness may be a very complex property, in principle based on any kind of interdependency between vertices, edges, the connection structure and the attributes of edges and vertices. We can only sketch the syntax definition here, cf. [14, 15] for a more complete discussion.

Syntactic properties of a kind of graph (a *graph type*) are defined on the *abstract syntax* of the graphs, i.e. the mathematical structure used to represent them. The next section will first introduce this structure, which we will then use to specify appearance and well-formedness.

[1] There are exceptions, of course, but we will ignore them here for simplicity. We will later shortly sketch how these can be dealt with in Moses.

3.1 The structure of a graph

As in [3], we define the abstract syntax of a picture to be an *attributed graph*, or more precisely a multigraph, since we allow any number of distinguishable edges between any pair of vertices. Vertices, edges, and also the graph itself carry (named) *attributes*. The names, types, structure, and meaning of these are naturally dependent on the graph type, although some are used in all graphs.

The attributes contained in a graph not only represent formalism dependent information but also information that is relevant to the graphical representation of the graph. Figure 2 shows a simple Petri net containing just 3 elements which are annotated with their name and element-specific attributes.

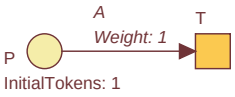

Figure 2: A simple Petri net

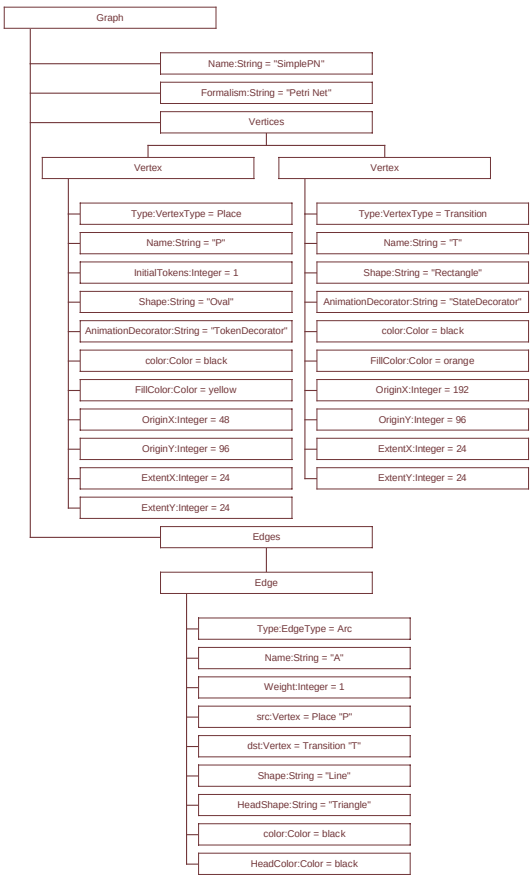

Figure 3: The attributed graph for Fig. 2

The complete attributed graph for the Petri net component is shown in figure 3. Note that the graph contains 4 attributes that are used to represent the name of the component, the formalism in which the component was modeled, the set of two vertices and the set containing the edge. The vertices and edges themselves have attributes representing information about their graphical representation including size and position. The source and destination vertices of an edge are simply represented as attributes of the edge. Obviously, for tools to be able to meaningfully operate on this structure, certain attribute names (such as that of the source and destination vertices of an edge) need to be agreed upon.

3.2 Defining a graph type

The Moses term for a visual notation is *graph type*, reflecting its graph-like nature. The syntax of a graph type is defined in the *Graph Type Definition Language* (GTDL) [13], a textual notation developed for this purpose.[2] The definition of a graph type basically consists of two parts:

1. A list of the kinds of vertices and edges in graphs of this type, together with definitions of their attributes and information about their graphical appearance.

2. A list of constraints/predicates on an attributed graph which must be fulfilled for the graph to be well-formed.

Consider e.g. Fig. 4 which shows the GTDL definition for a Petri net language. Lines 2-9 list the kinds (types) of vertices and edges, their attributes and graphical appearance. Lines 10-15 then contain the *syntax predicates* which formulate conditions on the structure of the graph or the values of the attributes. Each of these predicates consists of a text describing its purpose and a predicate expression (which may be quantified over elements in the graph) that formally specifies the condition.

Going into the details of this specification is beyond the scope of this paper, however it suffices to say that this simple description technique is able to express complex contextual properties of a language in a concise and yet readable manner. It also allows the integration of textual languages and hierarchical graph structures – cf. [15] for a more extensive discussion.

4 The editing environment

As models of discrete-event systems are typically described using visual formalisms, the main editing tool

[2] Moses also features a visual language to define graph types (VisualGTDL [9]) not discussed here.

```
1  graph type PetriNet {
2      vertex type Place(integer InitialTokens)
3          graphics (Shape = "Oval", AnimationDecorator = "TokenDecorator",
4          ExtentX = 24, ExtentY = 24).
5      vertex type Transition()
6          graphics (Shape = "Rectangle", AnimationDecorator = "StateDecorator",
7          ExtentX = 8, ExtentY = 24).
8      edge type Arc(integer Weight)
9          graphics (Head = "ClosedTriangle").
10     predicate "Arc weights must be positive."
11         forall a ∈ Arc : a("Weight") ≠ null ⇒ a("Weight") > 0 end
12     predicate "Arcs from places must end at transitions."
13         forall a ∈ Arc : src(a) ∈ Place ⇒ dst(a) ∈ Transition end
14     predicate "Arcs from transitions must end at places."
15         forall a ∈ Arc : src(a) ∈ Transition ⇒ dst(a) ∈ Place end
16 }
```

Figure 4: A syntax specification for a simple Petri net language.

is the *Graph Editor*. The graph editor is *generic* in the sense that it does not depend on any specific graph type and can manipulate any model that is represented as an attributed graph. The role of the graph editor is to provide an intuitive, easy to use environment in which to edit these graphs. Naturally this is not only limited to the insertion/deletion/layout of vertices and edges but also includes the editing of attributes associated with vertices, edges and the graph itself. In addition it also presents the status of the syntax predicates to the user. The following sections describe how the graph editor is configured by the graph type and how it manages user interaction during the editing process.

4.1 Configuring the editor

As we saw in section 3.2, the graph type description contains attributes that define the appearance of a vertex or edge. As can be seen from Fig. 4 a vertex typically has a *Shape* attribute. This attribute identifies a piece of code that is used to build the picture representing the respective vertex. Moses has a rich library of predefined shapes which are highly customizable. All shapes can be parameterized using other attributes of the vertex, such as *Color*, *FillColor* etc. However, if the author of a graph type requires a vertex appearance that cannot be generated in this way, the shape library can easily be extended.

The graph type also configures the graph editor with respect to the editing of attributes. The attribute type is used to select an appropriate editor for the respective attribute (these are mostly text attributes of one or more lines, but may also contain colors, numbers, expressions, other graphs (cf. next section) and in fact arbitrarily complex data structures, for which the graph type author may choose to provide a custom attribute editor plug-in.

Fig. 5 shows the configuration of the attribute editor pane (in the lower half of the window) and its dependence on the selected graph object(s).

4.2 Support for hierarchy

The graph editor also supports hierarchical notations where a vertex or edge contains attributes of type *graph*. The editor allows the user to navigate and edit a hierarchy that may be described by graph types representing different visual languages. In Fig. 6 a hierarchical finite state machine is shown. Note that the selected state contains another hierarchical finite state machine. The attribute editor shows the attributes defined for the current selection, in this case one of the contained hierarchical states is selected. As the *Show in Editor* check box is checked the graph also appears within the parent state. The child state machine can either be edited in the attribute editor or the user can navigate down and edit the net directly in the main editor pane.

4.3 Syntax checking

Unlike other systems where the syntax of a visual notation is enforced during the editing process, the Moses editor takes a less intrusive approach where the syntax of a graph is checked in the background and users can choose to ignore syntax errors until they are ready to deal with them. This approach has been found to be very valuable in practical modeling, where the order of

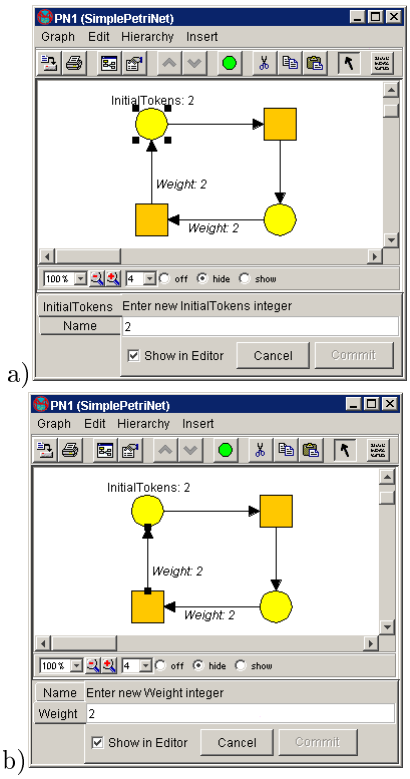

Figure 5: Object-dependent attribute editors.

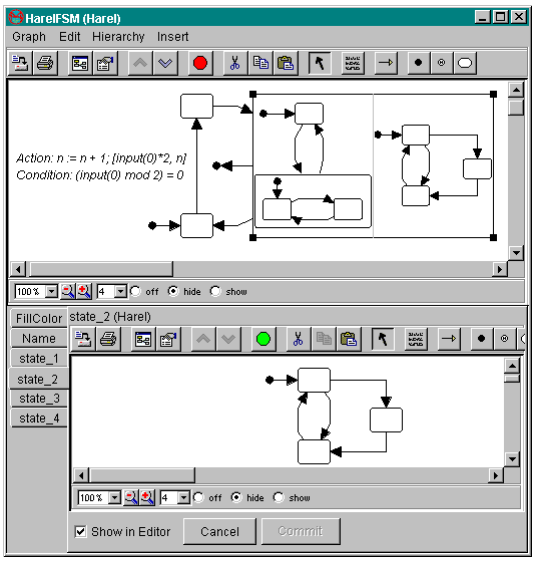

Figure 6: The graph editor showing a hierarchical finite state machine and and the *State* graph attribute of the selected state.

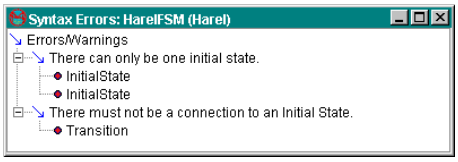

Figure 7: The syntax error pane showing errors and responsible elements.

editing operations is a matter of the user's preference and modeling convenience, and is not constrained by the necessity to maintain a valid structure at all times. Obviously, this is of particular importance in visual notations where it is in fact impossible to maintain a well-formed picture throughout the editing process, but it also facilitates much more rapid model construction, as users can choose the most efficient interaction with no restrictions imposed by the language rules.

Error reporting happens in a separate pane that reflects the result of background check and is continuously updated as editing proceeds (see Fig. 7). All syntax errors are associated with a set of graph elements that can easily by located in the graph editor by selecting them in the error report pane. For example, consider the following predicate expression:

forall $t \in Transition$:
$\sim dest(t) \in InitialState$
end

Each $t \in Transition$ for which the expression $\sim dest(t) \in InitialState$ fails can be reported as an error location. This way, each universally quantified error predicate may generate a list of such locations, which are displayed under the common message text associated with each predicate.

5 Specifying VL semantics

Specifying visual language semantics in a heterogeneous environment is not a trivial problem, and we can only give a very rough impression here of the way this is done in Moses –cf. [14] for an extensive discussion of this topic.

On a very fundamental level, the semantics of a visual notation is defined by specifying a compiler for it, which transforms a picture in that language (or rather the attributed graph that represents it) into a host-language program. This allows for very language-specific compilers to be added that exploit the structures of a specific notation for optimization of the resulting code.

However, very often, especially when initially developing a visual notation, being able to conveniently and clearly specify the language behavior takes precedence over run-time efficiency of the resulting code. Rather than specifying a code generator, a notation specific *interpreter* can be defined. Basically, this is an automaton that is parameterized with a particular attributed graph having behavior appropriate to the model the

```
1  class PNInterpreter[G] is
2  function M arity 1 ;
3
4  initialize :
5    once
6    do forall p ∈ V(G,"Place") :
7      M(v) := v("InitialTokens")
8    end
9  end
10
11 rule step :
12 once
13   choose
14     t ∈ {t ∈ V(G,"Transition") |,
15       ∀e ∈ Arc :
16         dst(e) = t ⇒
17         e("Weight") ≤ M(src(e))} :
18
19     do forall e ∈ E(G,"Arc") :
20       if t = dst(e) then
21         M(src(e)) := M(src(e)) − e("Weight")
22       end,
23       if t = src(e) then
24         M(dst(e)) := M(dst(e)) + e("Weight")
25       end
26     end,
27   end
28 end
```

Figure 8: Schema definition of a Petri net interpreter

graph represents.

Like compilers/code generators, these interpreters may of course also be written in the host language. While this may yield relatively efficient interpreters, it is usually a complex task, and the resulting interpreter is very likely to be large, hard to understand and even for simple visual notations, close to impossible to verify. As a consequence the Moses tool suite offers the option to specify the interpreter as an *Abstract State Machine*[3] (ASM) [7], a very general language for specifying operational semantics. As a language, ASMs are very simple yet very expressive having simple formal semantics that, at least in principle, are amenable to formal analysis.

The variant of ASMs used in Moses [14] has facilities for component communication, scheduling, a notion of time etc. The scope of this paper does not allow a detailed exposition of this language, but in order to give an impression of the technique, Fig. 8 shows an ASM describing the semantics of the Petri net language defined syntactically in section 3.

The G parameter represents the graph, and expressions such as $V(G,"Place")$ compute the set of all graph objects of the specified type (here: all vertices of type "Place"). Line 2 declares the structure of the state as is usual for simple Petri nets [18] – a simple unary function that maps the *Place* vertices to the number of 'tokens' residing upon them. This function is initialized in lines 6-8, where the *InitialTokens* attribute of the *Place* vertices is used to set the initial value of the M function. Finally, the step rule defines the state transition behavior: non-deterministically picking an activated *Transition* (lines 13-16) and updating the state according to the standard Petri net definition (lines 19-26).

6 Animating the visual program

Animating a visual program involves displaying the state of the executing program during its run as part of its visual representation, i.e. to decorate the visual program with additional graphical clues. These decorations are by their very nature dependent on the concrete language the program was written in. In Petri nets dots are drawn on places to represent the number of tokens residing upon them, in state machines the active state may be drawn with a contrasting border, and in dataflow networks the number of tokens are displayed as dots on the arcs.

The key challenge in the design of an animator and (visual language) debugger in a heterogeneous environment is to keep the animation infrastructure generic so it works for any visual language. In Moses, animation is basically set up in the same way as the static visual appearance of a visual program – in the graph type definition, the *AnimationDecorator* attribute, if present for a given edge or vertex type, identifies the code that represents the state of the particular object inside the graph, just as the *Shape* attribute controls the static appearance of a vertex. The decorator attributes for Petri net places and transitions can be seen in Fig. 4.

The generic animator assumes that each graph object (vertex or edge) that has an animation decorator attribute produces *state change* messages, which it then forwards to the decorator for display. As with shapes, if the set of predefined and parameterizable animation decorators provided by Moses is not sufficient, visual language designers are free to define more specialized ones.

When animating complex systems that consist of more than one component, the animator maintains a global view of the system structure as well as a hierarchical containment-tree, which can be navigated to locate specific components and animate them. It is im-

[3] Abstract State Machines were previously known as *Evolving Algebras*.

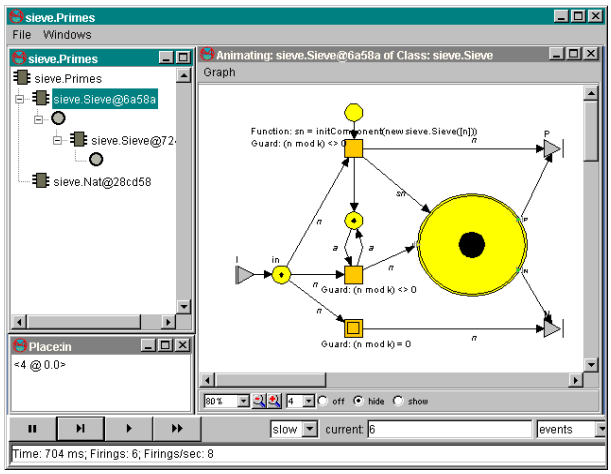

Figure 9: Animation framework showing the containment hierarchy, the state of the selected Petri net component and an inspector of the place *in* of a model of a recursive sieve.

portant to note that an executable model is dynamic in that components may be created or destroyed at runtime. Thus the structure of a model may change over time. The animation framework can extract the containment hierarchy of an executable model by executing the value of a graph attribute containing a function that extracts the contained components from the current state. A visualization of a containment hierarchy is shown in figure 9.

Assuming that the animation framework is instantiated and a containment hierarchy has been extracted it is possible to open a graph editor on a particular instance of any animatable component. The following steps enable the graph editor to visualize a component and its current state:

1. The executable component is checked whether it can be animated.

2. The graph and the locations that provide state change events are extracted from the component.

3. Using the graph type the graph is rendered in the graph editor as for editing. Animation decorators are installed into the graph at the appropriate vertices and edges – it creates a visual object that augments the icon produced by the shape and is registered as listener for state change events.

4. During execution state changes in the execution model are propagated to all registered listeners.

5. The occurrence of a state change event will cause the elements decorator to be redrawn to indicate the change in state.

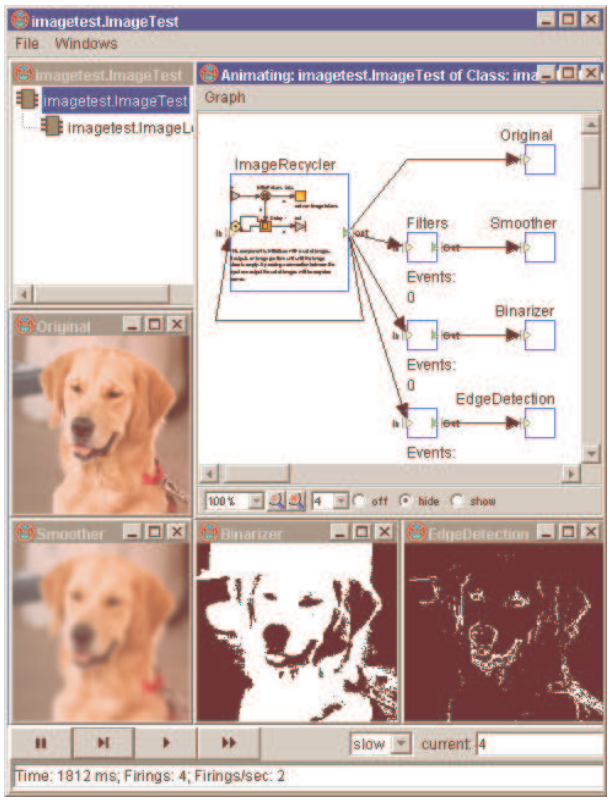

Figure 10: Animation framework showing a model of a image processing system with output windows showing the processing results

Finally a component in a formalism may represent an input to the system. In this case this component may be associated with a graphical object in which inputs are generated. As it is a component like any other it will be scheduled and when given control will cause events to propagate to other components.

Similarly a component in a formalism may represent an output from the system. These elements can be considered either to be another form of decorator that listens for state changes or a listener for inter-component communication. Either way information is may be visualized, output to the file system etc.

7 Conclusion

This paper gives an overview of the Moses tool suite, an environment that supports a number of visual programming languages and in addition provides an extensive infrastructure for customizing existing languages and developing and adding new ones. Tools such as the editor or the debugger/animator are parameterized by an abstract description of the visual language. Experience has shown that the creation of new visual

languages is made very much easier by the relatively short development cycles. This is due in part to the high-level scripting languages provided for the definition of visual language syntax and semantics.

Current research in the Moses project focuses on the following issues:

- Development of domain-specific visual languages for various application domains.
- Techniques for making both syntax and semantics descriptions more modular. In practice, when specifying visual notations, similar syntactical structures occur over and over (bipartiteness of graphs, connectedness, directedness, ...). Similar observations hold for semantics definition. Making solutions to these problems available as ready-made, parameterizable, and reusable specification components would make language definition much easier. It may also help create a set of best practices and patterns (similar to software engineering patterns) that make visual languages more understandable, robust, and elegant.
- Extending the tool support for visual languages defined in this framework: code generator support, verification, state space creation, model checking, etc.

References

[1] The Moses Project. Computer Engineering and Communications Laboratory, ETH Zurich (*http* : *//www.tik.ee.ethz.ch/ ∼ moses*).

[2] Robert M. Cubert and Paul A. Fishwick. MOOSE: An object-oriented multimodeling and simulation application framework. *Simulation*, June 1997.

[3] Martin Erwig. Abstract visual syntax. In *1997 International Workshop on Theory of Visual Languages*, pages 15–25, 1997.

[4] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design.* PhD thesis, ETH Zurich, 1996.

[5] J. Davis et al. Ptolemy II - heterogeneous concurrent modeling and design in JAVA. Technical report, UCB/ERL, EECS UC Berkeley, CA 94720, September 2000.

[6] J.M. Gooday and A.G. Cohn. Using spatial logic to describe visual languages. In *Proc. International Workshop on Theory of Visual Languages, Gubbio, Italy*, 1996.

[7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[8] R. Helm and Kim Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.

[9] Manuel Hilty. Graphical definition of visual syntax. Term project report, Computer Engineering and Networks Lab, ETH Zurich, 2000.

[10] Honeywell, Inc. *Dome Guide, Version 5.2.2*, 2000.

[11] Institute for Software Integrated Systems, Vanderbilt University. *GME User's Manual, Version 1.0*, March 2000.

[12] E. Lee J. Buck, S. Ha and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulations*, 4:155–182, April 1995.

[13] Jörn W. Janneck. Graph-type definition language (GTDL)—specification. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2000.

[14] Jörn W. Janneck. *Syntax and Semantics of Graphs—An approach to the specification of visual notations for discrete event systems.* PhD thesis, ETH Zurich, Computer Engineering and Networks Laboratory, July 2000.

[15] Jörn W. Janneck and Robert Esser. A predicate-based approach to defining visual language syntax. In *Symposium on Visual Languages and Formal Methods, HCC'01*, 2001.

[16] B. Meyer K. Marriott and K. Wittenburg. A survey of visual language specification and recognition. In Marriott, K. and Meyer, B.(Eds), Visual Language Theory, Springer-Verlag, 1998.

[17] Mark Minas. Diagram editing with hypergraph parser support. In *Proc. 13th IEEE Symposium on Visual Languages*, pages 230–237. IEEE Computer Society Press, 1997.

[18] Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[19] J. Rekers and Andreas Schürr. Defining and parsing visual languages with layered graph grammars. *JVLC*, 8(1):27–55, 1997.