

# A Framework for Defining Domain-Specific Visual Languages

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

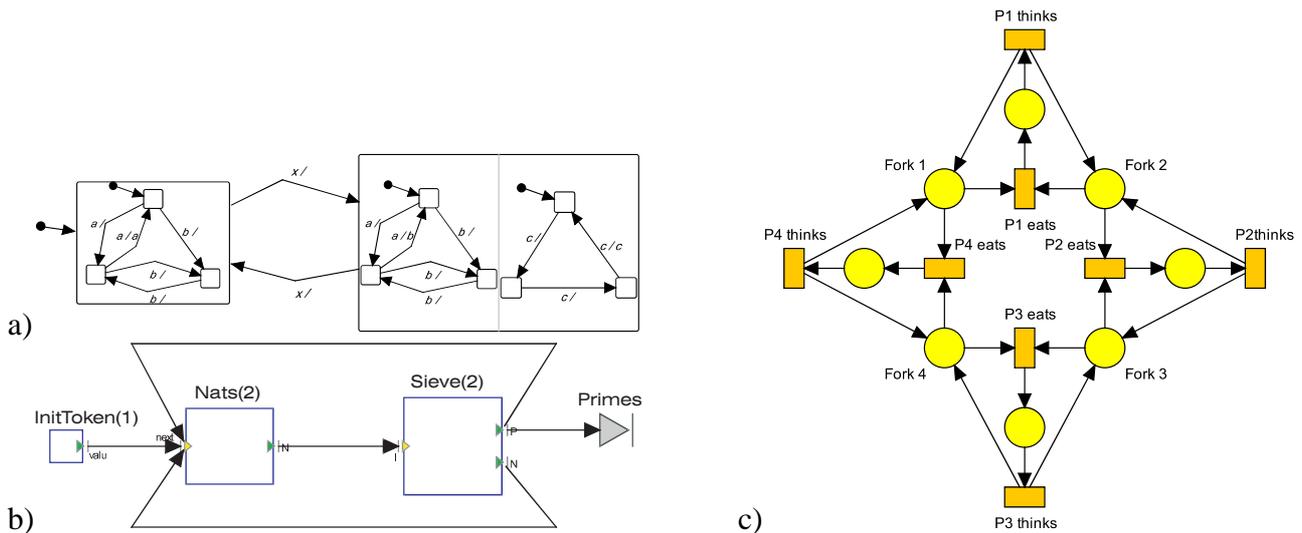
## Abstract

*For many problem domains domain-specific languages (DSLs) offer users more appropriate notations and abstractions in which to model systems when compared with general purpose programming languages. These benefits can often be amplified if a visual notation is used instead of textual notations. In many problem domains visual notations are preferred by practitioners as they often are the most intuitive representation of a problem. However, the lack of supporting infrastructure for constructing, implementing, and maintaining visual languages in general and domain-specific visual languages (DSVLs) in particular has been an impediment to gaining wider acceptance. This paper describes techniques used in the Moses tool-suite for defining the syntax and semantics of DSVLs, which are very general, yet are built on a few very simple concepts and are therefore easy to apply.*

## 1 Introduction

Domain-specific languages (DSL) are those that are tailored to a particular problem domain. Through the appropriate use of notations and abstractions they provide the expressive power to better describe specific solutions to problems in that domain [2]. This is often because the solutions can be expressed at an appropriate level of abstraction, and in the language of the problem domain, employing the concepts familiar to practitioners. Hence these domain experts can understand, validate, and develop DSL programs, whereas developing equivalent solutions in a general-purpose language (GPL) is often too daunting a task for people typically not trained as software engineers. In addition it is often possible to validate and optimize at the level of the domain rather than at the level of the GPLs where detail may obfuscate important features [4].

Examples of DSLs abound, including well-known and widely-used languages such as PIC, LEX, YACC, Make, SQL, and HTML [3]. All these languages are designed to address the problems from a well defined application domain. For example, in the case of the hypertext mark-up language (HTML), it is intended to describe the contents of a hypertext documents in a platform independent way, and in the case of the *Make* language it describes dependencies between objects and commands on objects in order to optimally rebuild a set of software artifacts.



**Figure 1. Models in different visual languages: a) hierarchical finite state machine, b) process network, c) Petri net**

Along with the advantages of DSLs listed above there are a number of disadvantages. These include the costs associated with designing, implementing and maintaining a DSL, the costs of education for DSL users, the difficulty of balancing between domain-specificity and general-purpose programming language constructs and the potential loss of efficiency when compared with hand-coded software.

In principle, visual languages (as opposed to textual ones) would often be ideal for use as domain-specific notations. For a large number of specialist application or problem domains there exists a natural and intuitive visual representation of artifacts in these domains, which is often what practitioners in these fields have been using all along, which reduces training costs and lowers the barriers to acceptance.

A good example is the field of designing embedded systems. Here, systems come with very different behavioral characteristics, roughly corresponding to different application domains. They may be state- and control-oriented, algorithmic and computational, they may be primarily data- or material-flow systems or they may combine some or all of these characteristics. Traditionally, different kinds of visual notations have complemented textual notations such as C or C++, e.g. StateCharts, Petri nets, dataflow graphs (see figure 1). These notations reflect the modeling requirements of their specific domain.

One key impediment to the use of visual languages in very domain-specific contexts is the high initial cost of creating a useful baseline environment (editor, possibly compiler/interpreter and debugger etc.) compared to textual languages, especially when a 'little' visual language is intended for a very specific application, a very short life span, or a very small target audience.

Textual languages are based on a very simple common data structure (character strings) and simple common notions of language syntax (usually context-free grammars), and the design and translation of textual languages is well-supported by a variety of tools to efficiently construct parsers, static analyzers, or even context-sensitive editing environments. We believe that the success of visual notations as commonly used domain-specific languages is contingent on making similar tools and concepts for visual languages a commodity that can be readily used and understood by a wide audience, effectively lowering the initial hurdle to adoption.

Unfortunately this is currently not the case—existing VL tools are often built on advanced notions of syntax and syntactical correctness, are difficult to use, or are too specific to be attractive to a wider audience. In many cases they also do not interoperate well enough with other tools. Perhaps this is partly an indication of the additional complexity inherent in visual languages, which translates in higher requirements on the basic infrastructure, and also of the wide range of different kinds of visual languages that are used.

This paper shows how the Moses tool suite [1] addresses these issues, and discusses the definition of visual language syntax and also their semantics for a specific field (discrete-event systems modeling) in this environment. Its key properties are these:

- It is based on a very simple structural model of an instance of a visual language.
- Describing syntactic properties of visual languages is fairly straightforward, and simple to use.
- The tool suite is open: It integrates well with other (visual and textual) languages, and a simple basic structure and storage formats facilitates data exchange with other tools.

The rest of this paper is structured as follows. In section 2 a brief overview will be given of DSLs in general and DSVLs in particular. In Section 3 the notion of an attributed graph and graph type in the context of the Moses tool-suite is presented, this is followed in section 4 by a worked example showing how a DSVL can be defined. Section 5 outlines a method for defining the semantics of a DSVL, while section 6 describes how other tools are configured by a GTDL description and how the user interacts with these tools. Finally we will present some conclusions in section 7.

## 2 Related work

### 2.1 Domain Specific Languages

Domain-specific languages have been designed and used for many years. However it is only relatively recently that DSLs have been studied systematically. A DSL is usually a "small" language as by its very nature it is not required to be applicable to a wide range of domains. It is often integrated into a general purpose language and may rely on libraries to provide support for the small set of concepts they support. In general DSLs offer only a restricted set of notations and abstractions and are also referred to as *micro-languages* and *little languages* [3].

There are many examples of DSLs in practice and as is often the case with the familiar, users may not even realize that they are dealing with a DSL. A taxonomy of application domains for DSL is given in [2]. Associated with DSLs is a design methodology [6, 25] in which the *Analysis* step essentially captures the domain specific knowledge, notations and abstractions as well as the design of the DSL itself. This paper is based on the premise that DSLs need not be restricted to text based languages. For very many domains a DSVL is a far more intuitive representation for capturing domain specific notations and abstractions.

### 2.2 Visual languages

There are many approaches to the specification and recognition of visual languages including grammar-based [23], (first-order) logic-based [11], graph grammars [21], constraint based [13], etc. (see [20] for

a survey of these approaches). In this section we will concentrate on the area of environments for the definition of visual languages as they typically include those tools that are essential to specify, recognize, create and interpret instances of these languages.

Ptolemy [5, 10] introduces the 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.

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 [8] 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 if the formalisms are not relatively close to the base formalism.

In DOME [15] (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 6).

The approach to syntax definition taken by GME [16] 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 with the resulting meta-model used to configure the various tools. The main difference in syntax definition between GME and Moses is the way in which hierarchical models are constructed – they are embedded models in GME, whereas in Moses they are simply element attributes. Also Moses syntax specification uses derived attributes and capabilities for handling embedded textual languages etc. which GME does not have. GME and Moses also differ in the way they define visual language semantics – Moses provides a layered plug-in architecture (see section 5) and an advanced simulation and animation engine, while GME has no formalized support for visual language semantics.

### 3 Attributed graphs and graph types

In this section we define the basic structure of an instance of a visual language. In order to make this easy to use from the perspective of a potential language author, this structure must be simple and in a straightforward way related to the *visual* elements of the language.

The formal structure that represents the pictures we want to draw and interpret, their *abstract syntax*, is what language authors will use to specify the syntactic properties of a visual language. In this work, we discuss pictures that represent graph-like structures of the kind shown in Fig. 1. Although all are graphs, they differ in the decorations of the graph elements and their visual attributes (shape, size, location). Fig. 1a in fact depicts a hierarchical graph, where a vertex contains other subgraphs.

Following [7], we define the abstract syntax of a picture to be an *attributed graph* (or, more precisely, a directed multigraph, allowing more than one connection between any two vertices) in the following manner:

**Definition 1 (Attributed graph.)** Assume fixed sets  $A$  of (infinitely many) attribute names and  $U$  of possible attribute values, containing an 'undefined' value  $\perp$ . We define the abstract syntax of a picture as an attributed graph, i.e. as a structure

$$(V, E, s, d, \mu)$$

such that  $V$  and  $E$  are disjoint sets of vertices and edges, respectively (collectively called graph objects), and a special object  $\star \notin V \cup E$  representing the graph itself, and

$$s, d : E \longrightarrow V$$

$$\mu : (E \cup V \cup \{\star\}) \longrightarrow A \longrightarrow U$$

The set of all attributed graphs is written as  $\Gamma$ .

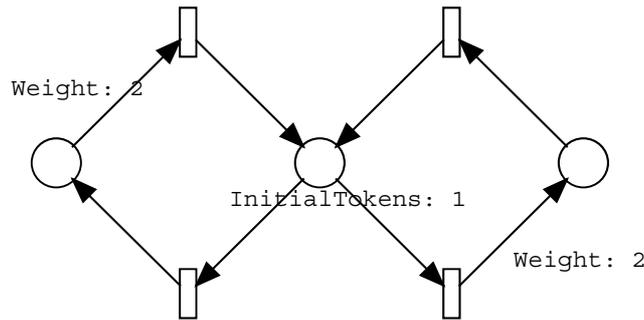
The attribution function  $\mu$  contains all information except for the connection structure of the graph. It is important to choose  $U$  to contain a relevant set of different kinds of attribute values – e.g. with  $\Gamma \subset U$ , attributes may contain subgraphs as in the example in Fig. 1a.

For any given visual language we need to restrict the admissible graph structures and/or attributions. We do this simply by specifying a set of predicates over the attributed graphs:<sup>1</sup>

**Definition 2 (Graph type, visual language.)** A set  $\mathbf{P}$  of predicates over  $\Gamma$  is called a graph type. It defines a visual language  $\mathcal{L}(\mathbf{P}) \subset \Gamma$  as follows:

$$\mathcal{L}(\mathbf{P}) = \{G \in \Gamma \mid \forall P \in \mathbf{P} : P(G)\}$$

When specifying realistic visual languages, this simple definition gives rise to several non-trivial issues concerning the concrete specification of the constraints/predicates, and their realization in the context of a visual programming environment. We will now address some of these issues.



**Figure 2. An instance of a simple Petri net**

<sup>1</sup>This is formally equivalent to just one predicate that is simply the conjunction of the  $P \in \mathbf{P}$ . We chose to present it in this way because it allows us to talk about a graph violating a specific syntax predicate.

```

1 graph type PetriNet {
2   vertex type Place(integer InitialTokens)
3     graphics (Shape = "Oval", ExtentX = 24, ExtentY = 24).
4   vertex type Transition()
5     graphics (Shape = "Rectangle", ExtentX = 8, ExtentY = 24).
6   edge type Arc(integer Weight)
7     graphics (Head = "ClosedTriangle").
8
9   predicate "Arc weights must be positive."
10    forall a ∈ Arc : a("Weight") ≠ null ⇒ a("Weight") > 0 end
11  predicate "Initial token number must be non – negative."
12    forall p ∈ Place : p("InitialTokens") ≠ null ⇒ p("InitialTokens") ≥ 0 end
13  predicate "Arcs from places must end at transitions."
14    forall a ∈ Arc : src(a) ∈ Place ⇒ dst(a) ∈ Transition end
15  predicate "Arcs from transitions must end at places."
16    forall a ∈ Arc : src(a) ∈ Transition ⇒ dst(a) ∈ Place end
17 }
18 }

```

Figure 3. A syntax specification for a simple Petri net language.

## 4 Defining visual syntax

In this section we will present a simple yet flexible infrastructure for defining the syntactical properties of a visual notation. We do this by walking through a relatively small example, illustrating how to deal with some of the common issues arising in the specification of visual modeling languages. We use a textual format for doing this<sup>2</sup>, called *Graph Type Definition Language* (GTDL) [17], because it is much closer to common mathematical notations and can thus be understood without much explanation of the notation itself. In practice, one might want to use a visual notation to specify visual syntax, as e.g. VisualGTDL [14] in Moses, the UML-variant in GME [16], or in the PROGRES system [24].

For reasons of brevity, we will omit most of the more advanced features of GTDL, such as user-defined types, hierarchical graphs, rule dependencies, function declarations, its host-language interface, and most of the facilities for tuning the graphical appearance. Instead we will focus on some of the underlying principles of our specification technique, and its composition with other visual and textual language specifications.

### 4.1 A simple visual language

Fig. 3 shows a GTDL specification of the visual notation used in the example in Fig. 2, Petri nets. This notation consists of two kinds of vertices, circular ones called *places* and rectangular ones called *transitions*, and one kind of edge. In the figure, one place has an "InitialTokens" attribute (an integer), while some edges have a "Weight" attribute (also an integer). These attributes are represented textually as decorations in the picture. These properties are represented in lines 2-7 in the specification, where the

<sup>2</sup>For readability, we use a few special symbols rather than pure ASCII text.

types of vertices and edges present in a graph are listed, together with their attributes and graphical properties (which are really also just attributes, although GTDL separates 'semantically' relevant attributes from graphical ones, to help distinguish between content and representation).

This is followed by a list of *predicates* that describe further syntactical constraints on the graph. For instance, we require that arc weights be positive. The predicate in lines 10-11 formulates this requirement. After the `predicate` keyword, a string describes the requirement as clear text (this can be used in diagnostic messages, cf. Sect. 6), followed by a predicate expression. In this case, that expression contains a quantifier over the set `ARC`, which in GTDL is a set that contains all graph objects of this type (as is the case for all vertex and edge type names). Note that the arcs are applied to their attribute names, as in  $a("Weight")$ , directly using the graph object as its attribution function. In this example, we allow the attribute to be absent, hence the test  $a("Weight") \neq \text{null}$  (a value of `null` indicating absence). Similarly, the requirement that the number of initial tokens be not less than zero is formulated in lines 12-13.

Petri nets are also bipartite graphs, i.e. arcs always go from a place to a transition and vice versa, but never between places or transitions. This slightly more complicated requirement is expressed in the two predicates in lines 14-17. Here, the functions `src` and `dst` represent the  $s$  and  $d$  structural functions in Def. 1.

In the following subsections we will extend this basic visual language to incorporate more complex textual attributes.

## 4.2 Embedding textual notations

Often a textual DSL is embedded into a general purpose programming language allowing the resources and capabilities of the GPL to be used within the DSL. This may result in the benefits of DSLs being diluted when users make use of GPL language features. In Moses we provide mechanisms to embed program fragments into a visual language. These fragments may themselves be textual DSLs or if necessary a GPL. The decision of what textual language to use is left to the DSVL designer but in general we believe it is preferable to use a small dedicated language (DSL) rather than a GPL. If it is necessary to provide the functionality that only a GPL can provide, we believe that it may be preferable to define a standard interface to a GPL and avoid the pitfalls of an ad hoc integration while maintaining the advantages of DSLs.

In the previous examples we encountered text fragments that served as simple strings to name things, as well as text representing numbers. However, many realistic visual languages contain text in much more intricate roles, and often these text fragments are highly structured—e.g. the OCL constraints language in UML, or the expressions in StateCharts. We will now show how to address issues arising from the interaction between structured text and the visual/graphical parts of a visual language in our framework.

This time we extend the Petri net notation defined in Fig. 3 with the capability to perform computation on the values of tokens. Fig. 4 depicts such a Petri net with the corresponding changes to the syntax specification are shown in Fig. 5. As can be seen, the arcs are now decorated with two new attributes, `VarName` containing a string, `Values` containing an expression (line 3). Furthermore, the `InitialTokens` attribute of `Place` vertices now contains an expression rather than an integer (line 1).<sup>3</sup>

---

<sup>3</sup>In GTDL, *expression* refers to the built-in expression language, for which it provides built-in support for parsing

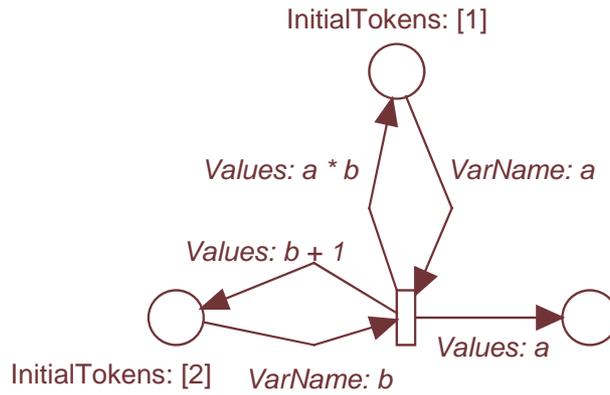


Figure 4. Embedded textual notation.

```

1 vertex type Place(expression InitialTokens)
2 ...
3 edge type Arc(..., string VarName, expression Values)
4 ...
5 predicate "Initialization expressions must be closed."
6   forall p ∈ Place :
7     p("InitialTokens") ≠ null ⇒ freeVars(p("InitialTokens")) = ∅
8   end
9 predicate "Free variables in expressions must be incoming arc labels."
10  forall a ∈ {v : for v ∈ Arc, src(v) ∈ Transition, v("Values") ≠ null} :
11    freeVars(a("Values")) ⊆ {v("VarName") : for v ∈ Arc, dst(v) = src(a)}
12  end

```

Figure 5. Static properties of embedded textual notations.

One syntax rule is that the `Values` attribute of arcs leading from places to transitions must be `null`, and that the `VarName` attribute of arcs from transition to places must be `null`. Similarly, for any transition, we require that all arcs leading to it be labeled with *different* variable names. We will skip the discussion of these rules since they are only slight variations of predicates discussed above.

The more interesting properties concern the free variables of the expressions, i.e. those variables which need to be defined outside of the expression because they are referenced but not defined inside it. For instance, the expression  $a + b$  contains two free variables,  $a$  and  $b$ , while  $\{a : \text{for } a \in S, a < t\}$  contains three variables,  $a$ ,  $S$ , and  $t$ , only two of which ( $S$  and  $t$ ) are free because  $a$  is bound inside the expression. The expressions  $[1]$  (a one-element list of the integer 1) and  $\{2 * a : \text{for } a \in \{1, 2\}\}$  contain no free variables, they are said to be *closed*.

For our simple visual notation we specify the following two constraints (among others):

1. The expressions defining the initial tokens of places do not contain any free variables.

---

and basic operations such as extraction of free variables etc. Users may, however, embed any textual notation, providing their own code to manipulate it.

2. The expressions in the `Values` attribute of an arc from a transition  $t$  to a place can only be free in the variables declared on the arcs leading to  $t$ .

The predicate in lines 5-8 expresses the first property—it requires each initialization expression of a place to be closed. The predicate relies on the `freeVars()` function which computes the set of free variables in an expression.<sup>4</sup>

The second property is a little more complicated, because the environment of a textual attribute (the defined variables which an expression may refer to) is the result of the structure of the graph as well as other textual attributes in it (the `VarName` attributes of other arcs). The predicate in lines 9-12 demonstrates how this is expressed in GTDL, again using the `freeVars()` function and testing for inclusion of the resulting set in the set of all variable names attached to arcs going to the respective transition.

Experience has shown that this approach of factoring out the treatment of textual languages and embedding them into the syntax and static semantics checking of an enclosing visual language is a very versatile and powerful technique, facilitating very natural and straightforward syntax specifications.

## 5 Specifying VL semantics

Since creating models of discrete-event systems in a variety of different modeling languages is the an important application area for the Moses tool suite, it provides extensive support in this area.

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. [18] for an extensive discussion of this topic.

On a very fundamental level, the semantics of a visual notation are 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, and which behaves according to the model the 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*<sup>5</sup> (ASM) [12], a very general language for specifying operational semantics. As a language, ASMs are 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 [18] 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

---

<sup>4</sup>This is a predefined function specific to the built-in expression language—if users need to specify similar properties for some other textual notation, they need to define those functions for their notation.

<sup>5</sup>Abstract State Machines are also known as *Evolving Algebras*.

```

1 class PNInterpreter[G] is
2 function M arity 1 ;
3
4 initialize :
5 once
6 do forall  $p \in V(G, "Place")$  :
7    $M(p) := p("InitialTokens")$ 
8 end
9 end
10
11 rule step :
12 once
13 choose
14    $t \in \{t \in V(G, "Transition") \mid,$ 
15      $\forall e \in Arc :$ 
16        $dst(e) = t \Rightarrow$ 
17        $e("Weight") \leq M(src(e))\}$  :
18
19   do forall  $e \in E(G, "Arc")$  :
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 6. Schema definition of a Petri net interpreter**

order to give an impression of the technique, Fig. 6 shows an ASM describing the semantics of the Petri net language defined syntactically in section 4.

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 [22]—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).

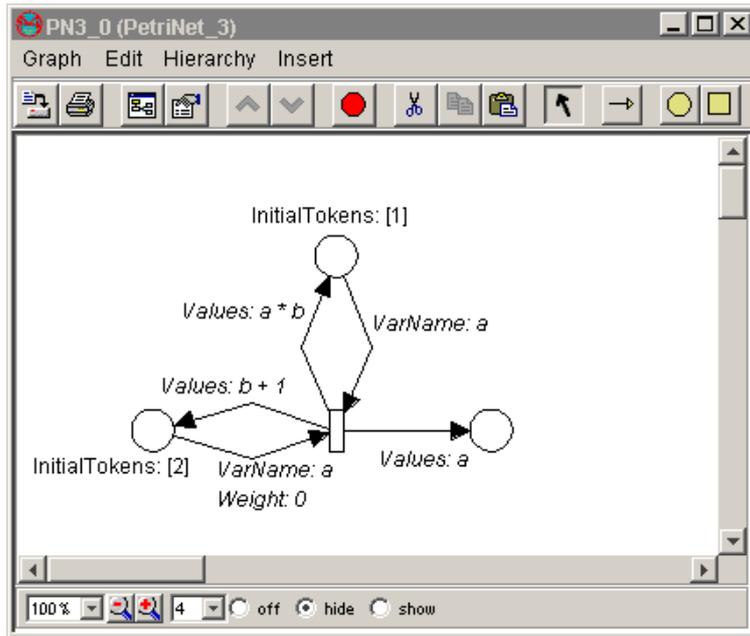


Figure 7. The graph editor showing a Petri net model containing syntax errors.

## 6 Editing pictures

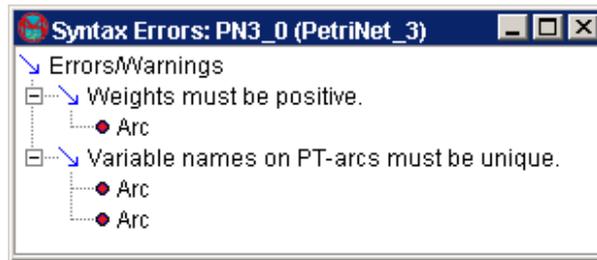
A graph type definition is used to configure tools in the Moses tool suite, e.g. the Moses graph editor [9]. The graph editor provides general-purpose graph manipulation functionality such as vertex and edge insertion and deletion and layout. It extracts the types of the vertices and edges from the graph type, as well as their appearance and attributes, and of course the syntax rules (see below). An important part of the graph editor is its ability to view and edit attributes associated with the graph, its vertices and edges. In Moses attributes are typed allowing the graph editor to instantiate the appropriate attribute editor. Naturally editor functionality, including support for new types and associated attribute editors, can be easily extended using the built-in expression language or the host language API.

The Moses editor is non-intrusive with respect to syntactical correctness. It checks the syntax of a graph in the background, allowing users to ignore syntax errors until they are ready to deal with them (see Fig. 7). This approach has been found to be very valuable in practical modeling, where the order of editing operations is a matter of the user's preference and modeling convenience, and should not be constrained by the necessity to maintain a valid structure at all times.

Error reporting happens in a separate pane that is continuously updated as editing proceeds (see Fig. 8). All syntax errors are associated with a set of graph elements that can be easily located in the graph editor by selecting them in the error report pane. For example, consider the following predicate expression:

**forall**  $a \in Arc$  :  
 $src(a) \in Place \Rightarrow dst(a) \in Transition$   
**end**

Each  $a \in Arc$  for which the expression  $src(a) \in Place \Rightarrow dst(a) \in Transition$  fails is reported



**Figure 8. The syntax error pane showing errors and responsible elements.**

as an error location. Each universally quantified error predicate may generate a list of such locations, which are displayed under the common message text associated with each predicate.

## 7 Conclusion

This paper presents the Moses tool suite, and the way it facilitates the definition of domain-specific visual languages, with a focus on, but not limited to, languages for modeling discrete-event systems.

The syntax definition technique is sufficiently general, easy to use, and sufficiently expressive for a wide variety of visual notations. Apart from visual languages for discrete-event modeling it has been used to define visual notations for various tasks, such as describing the structure of a system of data types, for specifying file and project dependencies (similar to make files), and for defining the syntax of visual languages. Every language whose syntax is defined in this way inherits the entire range of generic functionality, e.g. editor, syntax checking and reporting, support for various file formats (including XML), printing, layout tools and so on.

The semantics definition in turn supports the specification of a variety of DSVLs in the field of discrete-event modeling—every language defined in this way automatically interoperates with all other discrete-event notations, can be animated, simulated, instrumented, etc.

Even though a baseline tool support is part of the tool suite, it is critical that other tools, e.g. tools performing specific tasks, or tools improving on existing generic tools for specific cases, can easily be added. This is facilitated by the simplicity of the underlying structures and formats.

We believe that this kind of visual language architecture, i.e. providing useful baseline support for a sufficiently large set of notations, while allowing for easy extension of the tool set and integration of new and legacy tools, will be the key to a wider acceptance of visual languages as 'small' domain-specific notations.

Apart from these architectural issues, which are the focus of this paper, there are other important factors which play a role in using visual notations more casually in domain-specific contexts. For example:

- Composing DSVL syntax (and, ideally, semantics) specifications from a library of existing specification fragments. This not only has the potential for making VL definition significantly more efficient, it may also make VLS more consistent.
- There is clearly a tradeoff between simply configuring a generic tool and writing a language specific tool from scratch. By designing generic tools to be more configurable and to support a wider variety of interaction styles and visual notations, one can make them more reusable and reduce the need for creating new tools from scratch.

The experience inside the Moses project shows that good tool support and an open architecture may shift the balance between textual and visual notations toward the latter at least inside a small group of developers. However, much more is needed to do the same for a more general audience.

## References

- [1] The Moses Project. Computer Engineering and Communications Laboratory, ETH Zurich ([http : //www.tik.ee.ethz.ch/ ~ moses](http://www.tik.ee.ethz.ch/~moses)).
- [2] Paul Klint Arie van Deursen and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
- [3] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.
- [4] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In S. Kamin, editor, *DSL'97 – First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL'97*, pages 17–35, Paris, France, January 1997. University of Illinois Computer Science Report.
- [5] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulations*, 4:155–182, April 1995.
- [6] J.C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.
- [7] Martin Erwig. Abstract visual syntax. In *1997 International Workshop on Theory of Visual Languages*, pages 15–25, 1997.
- [8] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.
- [9] Robert Esser and Jörn W. Janneck. Moses – a tool suite for visual modeling of discrete-event systems. In *Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, HCC'01*, 2001.
- [10] J. Davis et al. Ptolemy II - heterogeneous concurrent modeling and design in JAVA. Memo, UCB/ERL, EECS UC Berkeley, CA 94720, September 2000.
- [11] 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.
- [12] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [13] R. Helm and Kim Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.

- [14] Manuel Hilty. Graphical definition of visual syntax. Term project report, Computer Engineering and Networks Lab, ETH Zurich, 2000.
- [15] Honeywell, Inc. *Dome Guide, Version 5.2.2*, 2000.
- [16] Institute for Software Integrated Systems, Vanderbilt University. *GME User's Manual, Version 1.0*, March 2000.
- [17] Jörn W. Janneck. Graph-type definition language (GTDL)—specification. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2000.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] Mark Minas. Diagram editing with hypergraph parser support. In *Proc. 13th IEEE Symposium on Visual Languages*, pages 230–237. IEEE Computer Society Press, 1997.
- [22] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.
- [23] J. Rekers and Andreas Schürr. Defining and parsing visual languages with layered graph grammars. *JVLC*, 8(1):27–55, 1997.
- [24] A. Schürr. PROGRES: A VHL-language based on graph grammars. In *in Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science*, number 532 in LNCS, pages 641–659. Springer-Verlag, 1991.
- [25] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.