

# A PREDICATE-BASED APPROACH TO DEFINING VISUAL LANGUAGE SYNTAX

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

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

## Abstract

This paper presents an approach to the specification of visual language syntax. Based on attributed graphs as the notion of abstract syntax, syntactical correctness is specified by a set of predicates over that structure. The proposed technique facilitates natural embedding of other visual and textual notations, the definition of complex syntactical and static-semantical properties, as well as precise error diagnosis and locating. An editing environment supporting this technique is briefly discussed.

## 1 Introduction

The acceptance of visual notations in computer science enjoyed a significant boom in recent years. To the extent that the precise representation of scientific and engineering artifacts is often based on visual notations, for which the precise and unambiguous definition is an important issue. While the definition of textual notations has been well researched over the past decades, to the point where the resulting techniques are almost universally accepted, widely used, and well-supported by many tools, the situation for visual languages is very different. There are various techniques for defining visual languages, which differ significantly in the way they conceptualize a visual notation as well as in the way they describe its syntactic structure.

One of the key properties of the way textual languages are defined is its conceptual *simplicity*—it is based on a very simple notion of an instance of a language (a string of characters), and syntax definitions are based on the notion of replacement (of a non-terminal symbol for some string). Being conceptually simple does not imply that it is easy to define any given language, or to produce good, elegant, or reusable definitions, but it lowers the threshold for adoption.

Thus, we believe that for a specification technique

for visual languages to be successful, it should meet the following requirements:

1. It must be based on a simple model of a picture, i.e. an instance of a visual notation.
2. It must compose well, with specifications of other visual languages (which one may want to embed/reuse), and also with specifications of textual languages.
3. The concept of a specification should be representation-neutral, i.e. it should lend itself to various textual and graphical representations.
4. A specification should support good localization and reporting of erroneous inputs.

This paper presents an approach to visual language definition developed in the context of the Moses Project [1], which addresses these issues for a certain class of visual languages, viz. graph-like notations (bubble-and-arc diagrams). The Moses Project focuses on the modeling and simulation/execution of discrete-event systems which are *heterogeneous* in the sense that they are composed of subsystems with very different characteristics, which suggests that these be modeled in different (usually visual) notations. Consequently, in this application area it is of critical importance to be able to support a wide, and growing, variety of domain-specific visual languages. The tool suite developed in this project is geared towards making it easy for a user to change existing languages, or develop new ones. This paper introduces some of the fundamental techniques for syntax definition. The definition of visual language semantics for discrete-event modeling notations is discussed in [13].

Following an overview of some related work in section 2, section 3 reviews our notion of abstract syntax, upon which the syntax definition is based. In section 4 we introduce the specification technique and some of

its interesting features by walking through a small example specification. Section 5 then discusses some implementation aspects and tools developed to support the technique, before concluding in section 6.

## 2 Related work

One important class of syntax definition techniques may be described as 'grammar-based', i.e. syntactical properties are described by productions similar to the way this is done for textual languages, except that the basic structure is not a string but rather a graph (as in e.g. [16, 17]) or a hypergraph [15]. These descriptions are then fed to parsers, that recognize the language structures in a relatively unstructured collection of input data. Closer to our approach in the style of specification, though very different in the way it is processed, are those parsing algorithms that operate on a set of constraints rather than a grammar, e.g. [8], [4].

Another approach to the specification of visual languages is through the use of first order or other forms of mathematical logic. Here spatial logics are used to define the different possible topological relationships between objects. This approach has the benefit of allowing both the syntax and semantics to be specified in the same formalism, e.g. [7].

Graph grammars are an attractive specification technique, not only because they parallel the way textual syntax is defined, but also because of their relative conceptual simplicity and their descriptive power. There are several approaches to use them even in the definition of visual language semantics, e.g. [6], [2], [19].

On the other hand, parsing algorithms on graphs tend to be significantly more complex than those for strings, and much harder to make well-behaved. Sometimes, an alternative to parsing is to have users enter the pictures on the level of the abstract syntax, i.e. provide them with an editing environment that facilitates the direct manipulation of the conceptual language constructs (the language 'meta model'). In this case, parsing is not necessary and the tools only need to distinguish 'well-formed' structures from erroneous ones, which simplifies the tool environment considerably, at the price of a more structured and constrained user interaction. This seemed to be an acceptable tradeoff in our application domain. A survey of visual language specification and recognition techniques is given in [14].

The Domain Modeling Environment (DOME) [10] developed at Honeywell is such a meta-modeling based tool-suite supporting system/software development. In DOME a new visual notation is described as a high level specification of node and connector types, connection constraints, and additional syntax and seman-

tics. 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 initiated by user actions. This is in contrast to Moses where syntax checking is a non-intrusive background task.

GME [11] is probably closest to the approach presented in this paper. In GME the syntax of a visual language is described using a UML-like visual notation and an OCL-like constraint language. Similar to the approach taken in Moses, results of meta-modeling are used to configure the various tools. The main distinguishing features 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 has no notion of derived attributes, and it has no capabilities for handling embedded textual languages. GME and Moses also differ radically in the way they define visual language semantics, though this is beyond the scope of this paper.

## 3 Attributed graphs and graph types

The syntax definition technique that we are about to introduce is based on a notion of *abstract syntax*, which is a formal structure that represents the pictures we want to draw and interpret. In this work, we are mostly interested in pictures that represent graph-like structures—as in Fig. 1, which shows concrete representations of (different kinds of) graphs. Although both structures represent graphs, they differ in the decorations of the graph elements, their visual attributes (shape, size, location). Fig. 1b in fact depicts a hierarchical graph, where a vertex contains another graph (or rather two of them).

Following [5], we define the abstract syntax of a picture to be an *attributed graph* (or, more precisely, a directed multigraph, since it allows for 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),*

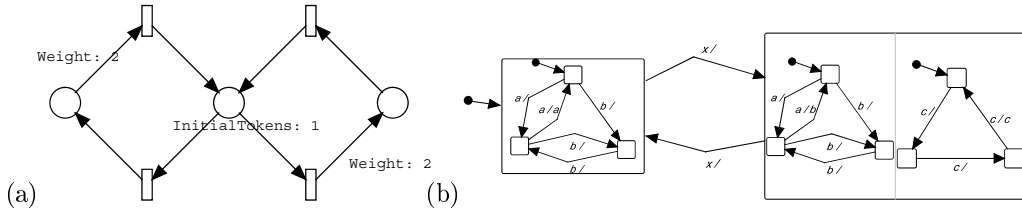


Figure 1: Concrete representations of different kinds of graphs.

and a special object  $\star \notin V \cup E$  representing the graph itself, and

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

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

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. 1b.

In principle, vertices and edges may of course represent any kind concrete object or relation, though in our context (and in the following examples) they will usually stand for 'bubbles' and graphical connections between them, respectively.

Obviously, 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 real-world 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.

## 4 Defining visual syntax

In this section, we will walk through a relatively small example, illustrating how to deal with some of the com-

<sup>1</sup>Obviously, 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 mirrors more closely the actual implementation and also allows us to talk about a graph violating a specific syntax predicate.

mon 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) [12], 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 [9] in Moses, the UML-variant in GME [11], or in the PROGRES system [18].

In this work, we will not be concerned with the *meaning* of the notations we specify (see [13] for a more comprehensive discussion of how to specify semantics), so we will omit any discussion of their 'behavioral' aspects. Likewise, we will omit all of the more advanced features of GTDL, such as user-defined types, 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. 2 shows a GTDL specification of the visual notation used in the example in Fig. 1a, a Petri net. 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 decoration in the picture. These properties are represented in lines 2-7 in the specification, where the 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 distinguishing between content and representation).

This is followed by a list of *predicates* that describe further syntactical constraints on the graph. For in-

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

```

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(integerWeight)
7     graphics (Head = "ClosedTriangle").
8
10  predicate "Arc weights must be positive."
11  forall a ∈ Arc : a("Weight") ≠ null ⇒ a("Weight") > 0 end
12  predicate "Initial token number must be non-negative."
13  forall p ∈ Place : p("InitialTokens") ≠ null ⇒ p("InitialTokens") ≥ 0 end
14  predicate "Arcs from places must end at transitions."
15  forall a ∈ Arc : src(a) ∈ Place ⇒ dst(a) ∈ Transition end
16  predicate "Arcs from transitions must end at places."
17  forall a ∈ Arc : src(a) ∈ Transition ⇒ dst(a) ∈ Place end
18 }

```

Figure 2: A syntax specification for a simple Petri net language.

stance, we require that arc weights be positive. The predicate in lines 10-11 represents this requirement. After the `predicate` keyword, it consists of a string describing the requirement as clear text (this can be used in diagnostic messages, cf. Sect. 5), followed by a predicate expression. In this case, that expression contains a quantifier over the set `Arc`, which in GTDL just represents 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")`, thus we directly use the graph object as its attribution function. In this example, we allow the attribute to be absent, which is implemented by the test `a("Weight") ≠ null` (a value of `null` indicating absence). Similarly, the requirement that the number 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 hierarchy as well as more complex textual attributes.

## 4.2 Hierarchical visual languages

Several refinement techniques have been suggested for Petri nets (see for instance [3]), most of which are usually best represented visually as hierarchical graphs, where the refined element (a place or transition, de-

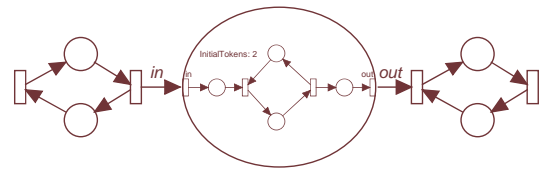


Figure 3: A hierarchical graph.

pending on the concrete refinement technique) visually contains its refinement (another Petri net). An example of a notation that allows the refinement of places can be seen in Fig. 3.

The corresponding changes to the syntax specification are shown in Fig. 4. Embedding another visual notation is achieved by simply declaring an attribute (in this case of a `Place` vertex in line 1) that can hold a graph (and optionally restricting the graph type of that graph, in this case to simply another `PetriNet`).

However, let us assume (without concerning ourselves here with the actual semantics of this notation) that the way we implement refinement in this example asks for more than this, viz. that we need to match the transitions surrounding the refined place with transitions inside the refining Petri net (its *interface transitions*). We identify the interface transitions of a Petri net by defining their `Name` attribute (declared in line 3), and match them with the transitions in the neighborhood of a containing place according to the `Name` attribute of the connecting arcs (declared in line 5). We thus need to specify a number of syntactical constraints, including the following:

1. names of interface transitions must be unique in-

```

1 vertex type Place(..., graph("PetriNet")Refinement)
2 ...
3 vertex type Transition(stringName)
4 ...
5 edge type Arc(..., stringName)
6 ...
7 derived InterfaceTransitions = {t : for t ∈ Transition, t("Name") ≠ null}
8 ...
9 predicate "Interface transitions must have unique labels."
10 forall t1, t2 ∈ G("InterfaceTransitions") :
11     t1("Name") = t2("Name") ⇒ t1 = t2
12 end
13 predicate "Place refinement interface must match arc labels."
14 forall p ∈ {v : for v ∈ Place, v("Refinement") ≠ null} :
15     {t("Name") : for t ∈ p("Refinement")("InterfaceTransitions")} =
16     {a("Name") : for a ∈ Arc, src(a) = p ∨ dst(a) = p}
17 end

```

Figure 4: Defining a hierarchical notation.

- side a Petri net,
- names of connecting arcs must be unique for all arcs going to or from a given place,
  - all arcs around a refined place have a non-null Name attribute,
  - arc names around a refined place must match the interface transition names of the refining Petri net.

We will focus on the first and the last property, because these present the more interesting challenges.

The most elegant way to approach this in GTDL is by introducing an attribute of the *graph* itself containing the set of its interface transitions. This attribute differs from those we encountered before not only because it belongs to the graph itself rather than an edge or a vertex, but also because its value is completely *determined* by the rest of the graph. Such an attribute is called a *derived* attribute, and in GTDL is declared together with the expression that defines its value (line 7), in this case the set of all transitions whose Name attribute is defined.

Using this attribute, lines 9-12 specify the first property, uniqueness of interface transition names. Here,  $G$  denotes the graph itself, thus the expression  $G(\text{"InterfaceTransitions"})$  yields the set of interface transitions for this current graph.

In order to define the last property, i.e. matching arc labels in this graph with the interface transitions in an embedded graph, we need to access the `InterfaceTransitions` attribute of that embedded graph. The predicate in lines 13-17 shows how this is done in GTDL: the expression

$p(\text{"Refinement"})$  yields the refining graph of the place  $p$ , and  $p(\text{"Refinement"})$ (`InterfaceTransitions`) is just the set of its interface transitions.

### 4.3 Embedding textual notations

In the previous examples we encountered text fragments that served as simple strings to name things, as well as text representing numbers. However, many real-world 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.

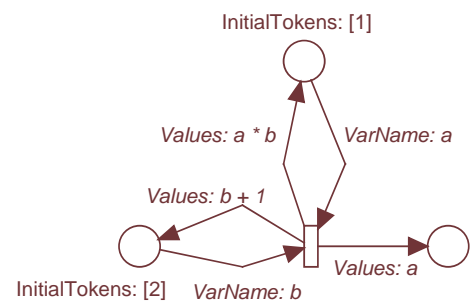


Figure 5: Embedded textual notation.

Once again, we will extend the Petri net notation defined in Fig. 2 by a capability to perform computation on the values of tokens. Fig. 5 depicts such a Petri

```

1 vertex type Place(expressionInitialTokens)
2 ...
3 edge type Arc(..., stringVarName, expressionValues)
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 6: Static properties of embedded textual notations.

net, the corresponding changes to the syntax specification are shown in Fig. 6. 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>

One syntactical 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 previous constraints.

The more interesting constraints 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 expression "[1]" (a one-element list of the integer 1) contains no free variables, it is said to be *closed*.

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

1. The expressions defining the initial tokens of places (if present) do not contain any free variables.
2. The expressions in the **Values** attribute of an arc from a transition  $t$  to a place can only be free in

<sup>3</sup>In GTDL, **expression** refers to the built-in expression language, for which it provides built-in support for parsing and basic operations such as extraction of free variables etc. Users may, however, embed any textual notation, providing their own code (usually written in the host language and only called from GTDL) to manipulate it.

the variables declared on the arcs leading to  $t$ .

The predicate in lines 5-8 expresses the first property—it requires for each place that if it has an initialization expression associated with it, the set of free variables of that expression must be empty. 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 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 Editing pictures

Similar to both DOME and GME, the results of meta-modeling are used to configure the tools in the Moses tool-suite. The Moses Graph Editor, for instance, is a general purpose graph editor that is configured by a *graph type*. The graph editor provides general purpose graph manipulation functionality such as vertex and edge insertion and deletion and layout. It extracts

<sup>4</sup>Again, 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 provide respective functions that operate on instances of that notation.

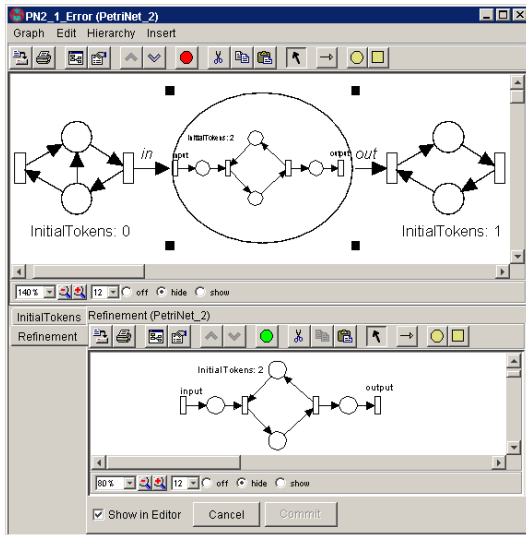


Figure 7: The graph editor showing the graph and the *Refinement* graph attribute of the selected place. Note the red icon indicating that syntax errors are present

from the graph type information about the existing types of vertices and edges, as well as their appearance, 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 Moses expression language or the Java API. The 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 (see Fig. 7).

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 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.

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

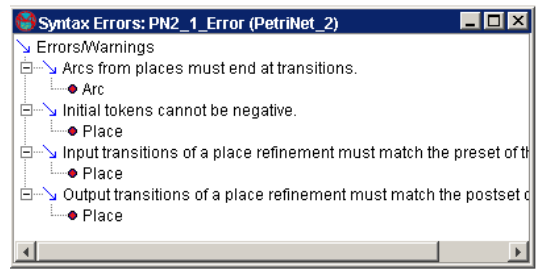


Figure 8: The syntax error pane showing errors and responsible elements.

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 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.

## 6 Conclusion

In this paper, we introduced a technique for specifying syntactical properties of graph-like visual languages. We applied it to a series of small specification tasks, and elaborated its integration into a tool environment for visual language editing/processing.

The visual language specifications created with this technique satisfy the requirements mentioned in the introduction:

1. The underlying concept of an attributed graph is almost canonically simple, yet flexible and sufficiently rich to support a wide variety of notations.
2. A visual language specification composes very naturally with other specifications of visual and textual notations, without having to resort to new conceptual models.
3. Although in our presentation the specifications were written in one (textual) language, a visual alternative exists already [9], and others are easily conceivable.
4. Finally, the way predicates are formulated makes it very easy to use them for precise error localization. Also, because each predicate typically represents exactly one syntactical property, its failure

allows good diagnosis and reporting of the erroneous situation.

While we believe that the proposed specification style is very appropriate to the kinds of languages we are dealing with, there are still a number of open questions to be answered. One interesting field is the *modularity* of syntax specifications: Having applied our technique to a wide variety of visual languages, some properties and structures keep recurring—e.g. bipartiteness, uniqueness of labels, definedness of variables. As language definitions become more complex, it is tempting to factor out these specification parts and reuse them. The question here would be the proper conception for this kind of module and its use, and its integration into a specification language.

Another interesting problem is the specification language itself, whether textual or visual. The textual format we presented here (GTDL) is very fundamental, almost canonical. A more visual notation might be desirable that facilitates a better visualization of higher-level syntactic structures in the specified language. VisualGTDL [9] is a first step in this direction, but more work needs to be done, especially in specifying real-world visual notations and learning from this experience.

Finally, more work needs to be done in exploring the interaction paradigms suited to these specifications—e.g. integrating them with visual language parsers in a sketch-based environment, having them either guide the parsing process or filter its output or both.

## References

- [1] The Moses Project. Computer Engineering and Communications Laboratory, ETH Zurich (<http://www.tik.ee.ethz.ch/~moses>).
- [2] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In *Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.
- [3] Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of petri nets. In Rozenberg, G., editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 1–46, Berlin, Germany, 1991. Springer-Verlag.
- [4] S. Chok and Kim Marriott. Parsing visual languages. In *Proceedings of 18th Australasian Computer Science Conference*, 1995.
- [5] Martin Erwig. Abstract visual syntax. In *1997 International Workshop on Theory of Visual Languages*, pages 15–25, 1997.
- [6] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.
- [7] 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.
- [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] Jörn W. Janneck. Graph-type definition language (GTDL)—specification. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2000.
- [13] 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.
- [14] 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.
- [15] Mark Minas. Diagram editing with hypergraph parser support. In *Proc. 13th IEEE Symposium on Visual Languages*, pages 230–237. IEEE Computer Society Press, 1997.
- [16] J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *Proc. 11th IEEE Symposium on Visual Languages*, pages 195–202, 1995.
- [17] J. Rekers and Andreas Schürr. Defining and parsing visual languages with layered graph grammars. *JVLC*, 8(1):27–55, 1997.

- [18] 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.
- [19] G. Taentzer, C. Ermel, and M. Rudolf. The AGG approach: Language and tool environment. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.