Generating Visual Editors for Formally Specified Languages

Susan M. Üsküdarlı
Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands
e-mail: `susan@fwi.uva.nl`

Abstract

We discuss the generation of structured visual editors for formally specified languages. Our concept of structured visual editor is a structure–oriented editor which supports the interactive construction of programs as well as provide language specific behavior. This paper describes an approach to the specification of the visual syntax of a language and the generation of a structured visual editor from the formal specification of that language. A simple visual *expression* editor is presented in order to demonstrate our approach. Our prototype implementation for the structured visual editor and visual definition language is presented at the end of the paper.

1 Introduction

In recent years, a great variety of visual languages have evolved. One of the concerns related to visual languages is supporting environments for the construction of programs. Visual programming environments should provide a set of tools that deal with the software development process for visual languages. There are many programming environments for textual languages and several systems that can generate such environments their specifications [13, 7, 10].

We have also seen an increasing number of visual programming environments for a great variety of languages. Some of these are special purpose languages and others are general purpose visual programming languages. We are interested in the generation of visual environments for visual languages. Section 8 discusses some other work related to visual programming environments generation. Our current focus is the specification of visual languages and the generation of language specific visual editors.

We propose a constraint–based visual language specification in which we make a distinction between the semantic description of a language and its visual syntax. In essence this allows us to treat the visual syntax as an interface to a language and thereby allow different views of the same language. This also makes it possible to define visual syntax for already existing textual languages and construct visual editors for them.

The construction of visual editors is typically addressed in two ways: (1) structure–oriented (syntax–oriented) visual editors, and (2) general–purpose visual editors. While the first approach considerably restricts the user freedom, the second approach requires visual parsing which still remains a considerable problem[1]. We choose to develop structure–oriented visual editors, not only because of the difficulties related to visual parsing, but more from our interests in generating language specific components. Ideally, we would prefer hybrid editors that support both structured and free form editing.

Section 2 gives an overview of the setting of our research. Section 3 briefly discusses formal specifications and introduces the example language referred to throughout this paper. Section 4 introduces the visual specification language. Section 5 discusses the specification and generation of visual editors. In Section 6 we describe our prototype implementation which is based on an algebraic specification system and an user–interface development environment. Section 7 describes the visual editor for expressions. Section 8 discusses related work. Section 9 discusses the future work, and finally, Section 10 draws conclusions regarding our current work.

2 Overview

As stated in the introduction, we approach visual syntax and visual editors from the programming environment perspective. Programming environments are comprised of several components, such as editors, type-checkers, debuggers, animators, etc. These components are integrated with the intention of providing its user an environment that supports the software development process for a given language. Programming environment generators, generate such environments from the formal specification of languages. The medium of communication among these tools is typically an abstract syntax tree.

In addition to the context-free syntax and the dynamic semantics of a language, other behavior such as type-checking and pretty-printing can also be specified. One such aspect that can be specified, relevant to our purposes, is the visual representation of the syntactic units. Our research focuses on such specifications and how they can be utilized in the generation of visual editors.

Structured visual editing mainly requires two kinds of specification: visual appearances and editing behavior. Thus, two specification languages, VDL (Visual Definition Language) and EDL (Editor Definition Language), are developed to address these needs.

[1]Visual parsing is an active research area [15, 4, 6], however we feel that the results are not yet satisfactory enough for us to depend on them.

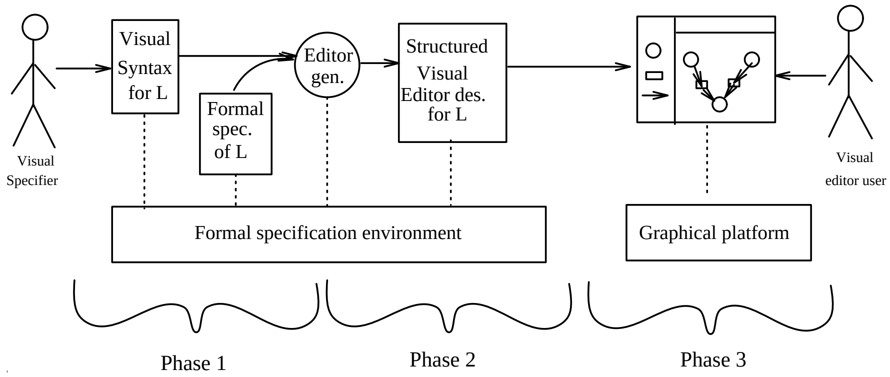

Figure 1: The overall view of a structured visual editor generation. Dotted lines indicate the underlying system and solid lines indicate the source of input.

Before describing the specification languages, let's outline the overall approach to generating structure–oriented visual editors. We start with a formal specification of a language L. Then the visual syntax of L is specified. At this stage the formal specification of L is enriched with the visual description of its syntactic units. Next, this specification is processed to generate a structure–oriented visual editor description, which contains all the necessary information for the implementation of the user–interface for the visual editor. Finally, the graphical user–interface for the editor is implemented on the desired graphical platform. This implementation uses the editor description for the language specific parts of the structured editor. The overall view of this process is shown in Figure 1.

We can identify three major phases in the editor generation process (also shown in Figure 1). The nature of each phase is somewhat different. The first phase involves the specification of L. For our purposes, we are particularly interested in the visual syntax specification. The second phase is the generation of a visual editor description. The final phase is the implementation of the graphical user–interface based on the desired graphics platform. Note that all specifications and the editor generator are specified using a *formal specification system*, whereas the visual editor interface is implemented over some graphical platform.

The emphasis of this work is on the specification of the information relevant to visual syntax and visual editing. Thus, the implementation of the user–interface of the structured visual editor is decoupled from the specification aspects. This allows the freedom of choosing any graphical platform for the user–interface implementation.

3 The formal specification of a language

The formal specification of a language involves the specification of the behavior of the language such as its context–free syntax and dynamic semantics. We use a context–free syntax to describe our language.

Now, let us consider a very simple language of integer expressions that supports two operations: division and addition. The context–free syntax of integer–expressions can be given as follows:

$$\begin{array}{lcl} \text{EXP} + \text{EXP} & \longrightarrow & \text{EXP} \\ \text{EXP} \ / \ \text{EXP} & \longrightarrow & \text{EXP} \\ \text{INT} & \longrightarrow & \text{EXP} \\ \text{eval(EXP)} & \longrightarrow & \text{EXP} \end{array}$$

According to this specification $1 + (<EXP>/60)$ and $(1 + <EXP>)/60$ are syntactically correct expressions. In these expressions non–terminals are denoted by the name of the non-terminal enclosed in angle brackets. The context–free syntax of integers is imported from another specification.

Now let us consider a two-dimensional syntax for expressions. For example, we may want to visualize the above expressions as follows:

$$1 + \frac{<EXP>}{60}$$
$$\frac{1 + <EXP>}{60}$$

Although this example is very simple, it is easy to imagine extensions of it for the definition of integration, summation, factorial, limit expressions and so on where this sort of layout is particularly suitable. Such layouts are required in most computer algebra programs.

In order to obtain such two dimensional layout, we first need to specify the desired visual syntax. The following section describes the visual definition language, VDL, which is used to specify such visual syntax.

4 Visual Definition Language

Visual Definition Language (VDL) is a constraint–based declarative specification language for describing visual appearances of syntactic units. By syntactic units we mean the visual building blocks required to construct visual programs.

The advantages of a declarative style are hiding of low–level geometric details from the specifier and the ease of defining spatial relationships among visual objects [9, 2, 1].

VDL addresses three major aspects of visual specification:

- primitive visual object definitions

- composite visual object definitions
- visual layout definition

We refer to visual object definitions as *vods*. Primitive *vods* serve as a small foundational set of *vods*, whereas composite *vods* are defined in terms of lesser complex *vods*. The visual layout definition deals with the specification of visual relationships among visual objects.

The remainder of this section provides an outline of VDL to give a flavor of the language. We describe VDL in terms of the three aspects mentioned above.

4.1 Primitive visual object definitions

Primitive *vods* form the foundation of all visual descriptions. This set consists of simple geometric objects such as lines, rectangles and circles; text; bitmaps and so on. Primitive *vods* intend to be a small set of basic visual descriptions from which more complex visual descriptions can be specified.

Primitive *vods* are specified using the following syntax:

$$\text{VOD-TYPE (VOD-ATTRIB}*) \longrightarrow \text{VOD}$$

A rectangle can be specified as $rectangle(width = 50)$, where *rectangle* is a VOD-TYPE and 'width = INT' is an attribute valid for a rectangle. In this example the remaining attributes such as height use default values.

4.2 Composite visual object definitions

Visual representations vary immensely depending on the nature of a language as well as user preference, which makes it impossible to provide built–in *vods* suitable for arbitrary languages. The *compose* operator is provided specifically for this purpose.

The compose operator is similar to the *picture* specifications defined in [9], where a picture is defined in terms of visual objects and the relationships among them. The relationships are similar to Golin's [6] *shape composition operators* which define relationships among two objects. These operators are used in shape composition to combine two elements. Our *compose* operator differs somewhat from Golin's shape compositions in that we define composition to be over an arbitrary number of elements involving several relationships. The reason for this decision is to enable a composite *vod* to represent a visual abstraction without having to define intermediate compositions which do not correspond to something meaningful in the domain of interest.

The compose operator constructs a composite *vod* based on two sets: *vod* set and visual layout description set (explained in Section 4.3). The *vod* set specifies the *vods* that make up the composite object and the visual layout description set provides the visual relationships among those *vods*.

The syntax of the compose operator is:

$$\begin{array}{llll} \textbf{compose} \; (& \textbf{vods:} & \{\ \mathrm{VOD+}\} \;, & \\ & \textbf{layout:} & \{\ \mathrm{VLD}*\} &) \longrightarrow \mathrm{VOD} \end{array}$$

The compose operator enables us to maintain a relatively small set of primitive *vods*, since we can extend *vods* as desired.

4.3 Visual Layout Description

Visual layout of *vods* involves the description of the spatial relationships among *vods*. These are similar to Golin's constraints defined in his picture layout grammar [6]. Layout descriptions specify spatial relationships among visual objects and are defined in terms of bounding boxes which encompass the objects. The bounding box for a visual object is defined by the points $(top, left)$ and $(bottom, right)$.

The syntax of a visual layout description (VLD) is:

$$< \mathrm{VOD\text{-}REF\ V\text{-}REL\ VOD\text{-}REF} > \rightarrow \mathrm{VLD}$$

where VOD–REF refers to a *vod* and V–REL is a spatial relationship. For example, the layout specification $<\ aRect\ \textbf{left–of}\ aText\ >$ states that the *vod* referenced by aRect is left of the *vod* referenced by aText, where the left–of constraint is specified as:

$$< vobj_1\ \textbf{left-of}\ vobj_2 > \;=\; vobj_1.\textbf{right} <= vobj_2.\textbf{left}$$

There are basic spatial relationships such as *touches, left–of, over, contains,* and *tiling.* These constraints can also be composed to yield new user defined spatial relationships such as touches–left which would be a composition of left-of and touches.

The reasons for using references to *vods* is for readability and more significantly for the need of specifying multiple layout descriptions related to the same construct within the same *vod*. This can be easily observed in the *vod* specifications for expressions with respect to operators (see Section 4.4). The *op* is referenced twice in the layout specification of the *plus* and *divide* expressions. In general, we simply use the reference name when discussing *vods*.

In order to avoid any confusion it might be useful to clarify our use of the word layout. Our interests lie within the specification of visual syntax. With respect to layout this implies the manner of specifying layout and not layout algorithms. We are aware of the abundant work in the area of complicated layout algorithms [3] and intend to take advantage of this work in the future.

4.4 Visual Expression

Now let us consider a visual syntax specification for our expression language. The following specification defines a very simple visual syntax[2]:

$$\nu(Int) = \textbf{text}(\textbf{string} = \textbf{int2str}(Int))$$

$$\begin{array}{lll} \nu(Exp_1 + Exp_2) = \textbf{compose}(& \textbf{objects}: \{ & < e1 : \nu(Exp_1) > \\ & & < op : \nu(+) > \\ & & < e2 : \nu(Exp_2) > \} \\ & \textbf{layout}: \{ & < \mathrm{e1}\ \textbf{followed–by}\ \mathrm{op} > \\ & & < \mathrm{op}\ \textbf{followed–by}\ \mathrm{e2} > \}) \end{array}$$

$$\begin{array}{lll} \nu(Exp_1 / Exp_2) = \textbf{compose}(& \textbf{objects}: \{ & < e1 : \nu(Exp_1) > \\ & & < op : line(\textbf{width} = \\ & & \quad max(e1.\textbf{width}, e2.\textbf{width})) > \\ & & < e2 : \nu(Exp_2) > \} \\ & \textbf{layout}: \{ & < \mathrm{e1}\ \textbf{over}\ \mathrm{op} > \\ & & < \mathrm{op}\ \textbf{over}\ \mathrm{e2} > \}) \end{array}$$

$$\nu(+) = \textbf{text}(\textbf{string} = "+", \textbf{font} = \textbf{bold})$$

When the underlying representation of the picture is a tree (like above) the visual specification is simple, since the *vod*s for the syntactic units are local to the specification. However, when the underlying structure of a picture is a graph matters are more complicated as we need to be able to reference non–local constructs. This problem is addressed by distinguishing remote–constructs and allowing constraint specifications between local and remote visual objects. This is analogous to Golin's [6] remote symbols which reference symbols that are not local to a production.

5 Editor Definition Language

Once we have the specification of a language including its visual syntax description, we need an editor to construct programs in that language. The language specification encompasses sufficient information to generate a structured visual editor for that language.

First we define a very simple editor definition language, EDL, for the purposes of representing the abstract definition of a structure–oriented visual editor (SVE). A structure–oriented visual editor can be defined in terms of its three major aspects:

- visual syntax
- replacement behavior

[2] Exp_1, Exp_2 are variables of sort EXP, Int is a variable of sort INT and ν stands for visual description.

- functions over programs

The description of a SVE is basically a composition of the specifications of these three aspects. The visual syntax consists of the visual description of the syntactic units as was described in section 4. The remaining two are explained in the following two sections.

5.1 Replacement behavior

SVE is a structure–oriented editor, which means that programs are constructed via a series of appropriate replacements until the desired program is obtained. The replacement behavior is defined by specifying the set of permissible replacements for each non-terminal of the language. The replacement behavior for a language can be obtained from the syntax definition of a given language.

The context–free syntax definition of the language is processed in the following manner to yield the necessary information. Let the context-free syntax of L be specified by rules:

$$\alpha_i \rightarrow S_k$$

where S_k is a sort (non–terminal) of language L. The visual syntax of a language is defined in terms of the visualization of its syntactic units:

$$\nu(\alpha_i) \rightarrow S_k$$

Plus the visual appearance of a meta–variable corresponding to each sort:

$$\nu(S_{Meta}) \rightarrow \mathrm{S}$$

The default visual specification for the meta–variable is the name of the sort enclosed in angle brackets:

$$\nu(S_{Meta}) = text(string = " < S > ")$$

where S_{Meta} acts as a constant of the sort S[3]. The purpose of this is to provide a meta–variable (place holder) for the sort S. This is necessary for structure–oriented editors as the program is developed starting from a very abstract representation. Meta–variables can be replaced by terminals, other meta–variables, or a combination of both as specified by the visual syntax of the language. Section 7 shows the use of meta–variables in the contruction of an expression.

Terminals have a concrete representation as specified by the visual syntax. Some special terminals such as numbers and strings have an infinite number of representations. These are considered as primitive terminal values. These terminals have a different replacement behavior in that their values are provided with some input device (usually keyboard) instead of making a selection from the replacement menu.

[3]The default *vod* for meta–variables is very simple can be redefined by the specifier if desired.

5.2 Functions on programs

Finally, we are interested in attaching some semantic behavior to the editor [11]. This is done by providing access to the functions defined for the language. These functions are defined as part of the formal specification of the language, such as *eval* for expressions.

Currently, we consider the application of functions to the entire program or to the focus of the program. The focus of a program is some part of the program which is selected via user–interaction. The user can select components as dictated by the specification of the languge.

We specify functions to be included in the editor as follows:

$$< \text{SVE-Name, F-Name, SORT} > \rightarrow \text{FD}$$

where SVE-Name and F-Name are strings representing the name to be used for the function in the editor and the name of the function defined as part of the language specification respectively. SORT is the sort which the function is defined for.

5.3 Generating editor descriptions

Given the formal specification of a language we can generate a structured editor description for that language. This description is a composition of:

- the replacement behavior computed as described in section 5.1,
- the visual syntax specification for the language,
- the desired functions for the editor.

These aspects are extracted from the specification of the language and composed into a structured visual editor description. The syntax of a SVE description is:

$$\begin{array}{llll} \{ & \textbf{repl-beh:} & \{ \text{ REP+ } \} & \\ & \textbf{functions:} & \{ \text{ FD}* \text{ } \} \text{ } \} & \longrightarrow \quad \text{SVE} \end{array}$$

where

$$\text{SORT} : \{ \text{ VOD+} \} \rightarrow \text{REP}$$

The replacement behavior part of the specification embodies both the replacement behavior as well as the visual syntax. For each sort a list of *vods* is supplied as valid replacement for that sort.

For the integer-expressions the replacement behavior is as follows:

$$
\begin{array}{l}
\{\ \textbf{repl-beh:} \\
\qquad \text{EXP: } \{\ \nu(EXP + EXP) \\
\qquad\qquad\quad \nu(EXP/EXP) \\
\qquad\qquad\quad \nu(EXP) \\
\qquad\qquad\quad \nu(INT)\ \} \\
\qquad \text{INT: } \{\ \textbf{primitive: integer}\ \} \\
\\
\quad \textbf{actions:} \\
\qquad \{\ <\ "Evaluate", "eval", EXP > \ \}\ \}
\end{array}
$$

where EXP and INT are abbreviations for EXP_{Meta} and INT_{Meta} respectively.

Note that the *vod*s in the replacement part will in reality be the full expansion of the *vod*s which would be much more detailed than shown here. The full description is derived from the specification of visual syntax for the language which is further refined according to the description of VDL.

Should the visual syntax not be specified, it is conceivable to generate a default visual specification for a language, say as corresponding to a syntax–tree. Although, such an approach has not yet been attempted.

6 Implementation

What we have discussed thus far has been the abstract description of the visual syntax and the structured visual editor. What is missing from the picture is the graphical editor for creating programs.

All aspects of the language and the editor, with the exception of the implementation of the user–interface of the editor, are specified. Furthermore, the user–interface itself uses the generated editor descriptions to realize the visual editor. This approach is intentional and is motivated by the desire to remain within one formalism to specify the language and examine how we can generate visual tools for the language. This separation also enables us to use any user–interface development system, thereby refraining us from forming a strong dependence on or being influenced by any particular graphical package.

The implementation work is done with two software systems: ASF+SDF [10] and Garnet [12]. The following two sections briefly describe these two systems as well as motivate the reasons for their choice.

6.1 ASF+SDF

All the specifications dealt with in this work are developed with an algebraic specification formalism called ASF+SDF (Algebraic Specification Formalism and Syntax Definition Formalism). ASF+SDF is a programming environment generator, which generates programming environment components from the formal

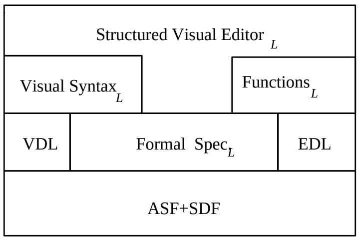

Figure 2: The uses dependencies of the various specifications.

specification of a language. The ASF+SDF system provides an interactive environment for the development of specifications. It supports the modularization of language specification, where each module consists of a context–free description and optional semantic specification implemented as a term rewriting system (thus the specifications are *executable*). It provides a very good medium both for developing specification languages (i.e., VDL and EDL) as well as specifying the languages we want to visually specify such as expressions, finite state automata or hydraulics systems.

Figure 2 shows the dependencies among the specifications relevant to the structured visual editor. The figure shows the *uses* dependencies by placing a specification over other specifications that it uses. The ASF+SDF layer is at the very bottom as it forms the basis of all specifications.

6.2 Graphical User Interface

Our prototype implementation of the user–interface to SVE is implemented using Garnet [12]. Garnet is a user–interface development environment which supports the construction of interactive, graphical interfaces. The main attraction of Garnet for us was its graphical constraints and its interpretive environment which supports fast prototyping.

We use Garnet to implement the user–interface aspects like windows, menus and so on. Garnet also provides a powerful graphical constraint system to deal with the layout constraints. We should point out that the translation from *vod*s to garnet components is also easily specified using ASF+SDF.

6.3 The Structured Visual Editor

The graphical user–interface for SVE consists of three major parts.

- *Expansion selection* is a menu for selecting the desired replacement for a non–terminal while constructing a program.

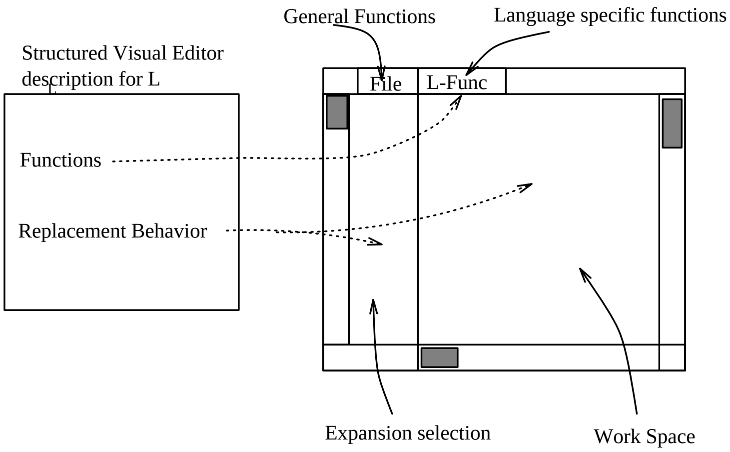

Figure 3: The structured visual editor and how it relates to the editor description.

- *Functions* consist of general editor level commands like save, print, load as well as language specific functions as described in Section 5.2.
- *Work Space* is the area where the visual program is constructed.

The graphical user–interface for SVE uses a generated editor specification (result of phase 2 in Figure 1) to create a language specific editor. Essentially, there are two distinct parts of the editor. First, there is the shell of the editor which is constant across all SVEs. This consists of the graphical components such as the window, menu bar and the general editor functions. The second part forms the language specific part of the editor, such as the replacement behavior and the language specific functions. The language specific aspects of the editor are obtained from the generated editor description for the given language. Figure 3 shows the structured visual editor and its relationship to the editor description.

7 Visual Expression Editor

Now let's take a look at the visual expression editor. The expression editor description given in section 5.3 provides information about two sorts: EXP and INT. The expression editor, which initially has an empty workspace, allows the creation of meta–variables corresponding to these two sorts. Figure 4 shows the beginning sequence of an expression creation. Part (a) shows the initial selection menu and the result of the choice of the $<EXP>$ meta–variable from the selection menu. Part (b) shows the selection menu after the $<EXP>$ in

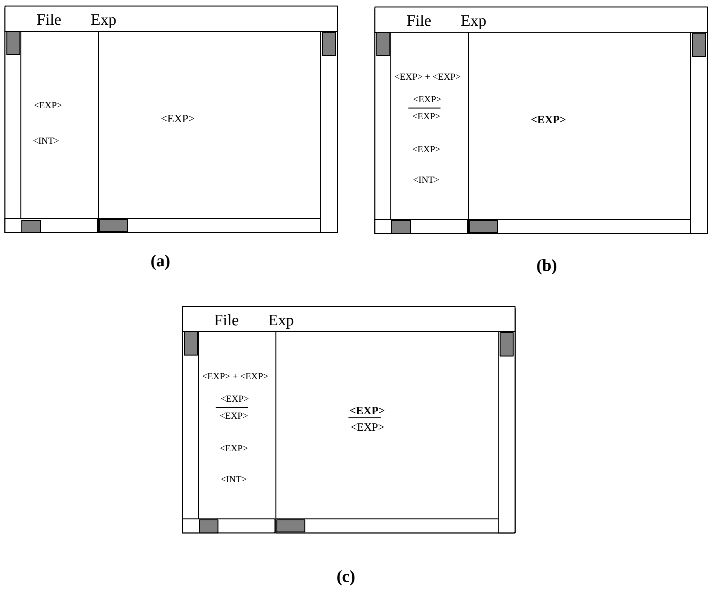

Figure 4: The structured visual expression editor.

the workspace is selected. This selection menu is computed based on the replacement behavior associated with the EXP sort. Finally, part (c) shows the selection of the division expression followed by the selection of the numerator expression. The selection menu remains the same as the selected meta–variable has not changed. To fully construct an expression the user successively expands the expressions as desired. Eventually each expression is expanded to an $<INT>$ which is replaced with an integer entered from the keyboard (since it is a primitive type).

The *Exp* menu consists of two items: *Evaluate* as given in the editor specification, and *reduce* which is the standard language specific behavior. *Reduce* simply takes the current program and reduces it to its normal form.

Language specific functions are evaluated by the ASF+SDF system. This is done by maintaining textual terms which correspond to the visual terms being constructed. This process is transparent to the SVE user. The results of function evaluations are translated into their visual appearance (according to their visual syntax specification) and displayed within a separate window.

8 Related Work

Golin and Reiss [5, 6] define a language–independent environment (GREEN) for visual programming which consists of a general purpose editor, a grammar driven spatial parser and a visual specification formalism based on *picture layout grammars*. Their *picture grammar* is an attributed multiset grammar where the productions are picture composition operators. In our formalism we have have separated the specification of the semantics of the language definition from the visual syntax.

The style of our visual syntax definition is influenced by Helm and Marriott [9] where they define *pictures* as being either primitive or complex. Complex pictures are specified in terms of its sub–pictures and a set of constraints. They provide a rigorous declarative semantics both for the specification and the recognition of pictures. We remain more interested in the specification of *vods* (pictures) and the specification of how they relate to syntactic units.

The Visual Programming Workbench (VPW) [14] is a collection of tools that synthesizes environments for the specification, analysis, and execution of visual programs. They specify visual syntax with Golin's picture layout grammar and generate components from the grammar using various tools written in different languages. We have tried to remain within one formalism where we specify the language features as well as the generation of visual tools.

Bjorn Backlund et. al. [2, 1] define the generation of visual language–oriented environments where they distinguish two layers: *derivation* and *structured presentation*. The derivation layer defines the concrete syntax in terms of graphical data types and the structured presentation layer concretizes these views. Their formalism combines attribute grammars and graphical constraints. Attribute grammars are used to define edit semantics at the derivation layer. A predefined set of graphical types and constraints at the presentation layer is used for specifying pictures and their constraints. Hagsand [8] extended this work by defining the dynamic semantics of visual languages using operational semantics.

9 Future Plans

Our work is in a preliminary stage and leaves for much interesting future work. Foremost, is to gain additional experience with a variety of visual languages.

Our goal is to firstly provide visual environments of textually defined languages, and then to bootstrap this approach to the underlying specification system to obtain a visual programming environment generator.

One of our shorter term goals is to create an editor for specifying the *vods*. This will be achieved by generating an editor for the VDL specification language in the manner described in this paper.

In relation to the visual syntax specification there is work to be done in the area of default representations. Defaults provide a very convenient way to allow the visual specifier to concentrate only on the relevant aspects. In addition to default *vod* parameters complete default *vod* specifications may be useful which would allow partial definition of visual syntax.

With respect to the editor specification, one of the areas to be dealt with is user–interaction specification. This is particularly important with functions that require several arguments and when the expansion behavior would require the identification of visual objects already in existence in the editor. The user–interaction specification would specify how the user makes such identifications.

Another interesting study is how to achieve multiple views of a program and the transformation among different views of a program.

10 Conclusions

We present a general approach for the generation of a structured visual editor for formally specified languages. We define a visual syntax specification language (VDL) for the purposes of specifying the visual syntax and describe the generation of a structure–oriented editor from the specification of a language.

In attempts to focus on the characteristics of the visual syntax and visual editing we separate their specification from the implementation of the user–interface of the editors.

While the work presented in this paper is currently quite simple, we believe that the underlying concepts provide a good foundation for studying visual languages and programming and consequently the generation of environments for them.

Acknowledgments: I wish to extend my sincere thanks to T. B. Dinesh and Jan Rekers for intriguing discussions and reading drafts of this paper.

References

[1] B. Backlund. Visual programming languages and how to generate syntax–oriented environments for them. Technical Report TRITA-NA9003, Royal Institute of Technology, Sweden, 1990.

[2] B. Backlund, O. Hagsand, and B. Pehrson. Generation of visual language–oriented design environments. *Journal of Visual Languages and Computing*, 1:333–354, 1990.

[3] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. Technical report, Brown University, 1993.

[4] G. Costagliola, M. Tomita, and S.-K. Chang. A generalized parser for 2–D languages. In *IEEE 1991 Workshop on Visual Languages*, pages 98–104, 1991.

[5] E. Golin and S. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, 1:141–157, 1990.

[6] E. J. Golin. *A method for the specification and parsing of visual languages.* PhD thesis, Brown University, 1990.

[7] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[8] O. Hagsand. A framework for generating language–oriented environments for visual programming languages. Technical Report R92:06, Swedish Institute of Computer Science, March 1992.

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

[10] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, September 1993.

[11] J. Koorn. Connecting semantic tools to a syntax-directed user-interface. Report P9222, Programming Research Group, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92a.ps.Z.

[12] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[13] T. Reps. Generating language-based environments. Technical report TR 82-514, Cornell University, Ithaca, 1982. Ph.D. Thesis.

[14] R. V. Rubin, J. Walker II, and E. J. Golin. Early experiences with the visual programmer's workBench. *IEEE Transactions on Software Engineering*, 16(10):1107–1121, 1990.

[15] K. Wittenburg. Early–style parsing for relational grammars. In *IEEE 1992 Workshop on Visual Languages*, pages 192–199, 1992.