Specifying Visual Syntax

Susan Üsküdarlı

Faculty of Mathematics and Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, email: susan@fwi.uva.nl

Abstract. This paper discusses visual syntax definition in a framework of generating programming environments for visual languages. In particular, the work is set in the context of the ASF+SDF Meta-environment [4]. For the generation of a visual programming environment we need to be able specify visual languages. This paper discusses how we can define visual syntax for existing languages, and then outlines an approach to defining a visual specification formalism.

The basic idea is that given a language specification, such as *sets*, and a mapping from this language to a graphical syntax (using a visual definition language), we obtain a visual language definition from which we can generate visual tools. This approach can be exploited to define tools for a visual specification formalism yielding a *visual* environment for specifying visual languages.

Key Words & Phrases: Visual Languages, Picture Specification Languages, Visual Editors, Visual Programming Environments, Algebraic Specification, Tool generation

1 Defining Visual Syntax

In order to define visual syntax we need to be able to describe graphical representations. For this purpose a constraint-based declarative language VODL [6, 7] (Visual Object Definition Language) is defined. The entities defined by VODL are referred to as *vods* (visual object definitions). It provides a set of primitive *vods* along with a set of expressive *vod* operations for defining new *vods*. Composite *vods* are constructed from sub-*vods* and sets of constraints among the sub-*vods*. The basic construction involves declaring a set of *vod*-references (*vod-refs*) that refer to other *vods*, and then specifying the constraints between these *vods* using the *vod*-refs. Any specific attributes and their values can also be specified (with the $\oplus$). Instead of fully describing VODL we illustrate its use in defining visual syntax by way of example.

2 Example

This section outlines the definition of visual syntax by way of an example. The example is to define a visual syntax for the *Set* language. First, a very simple specification for the *Set* language is presented. Then a set of *vods* to be used in defining the visual syntax is presented. That is followed by a mapping associating the two. Finally, an application of the visual syntax definition is discussed with a visual structured editor.

2.1 SET: The Algebraic Specification

The specification of *Set* is specified using the ASF+SDF formalism. The syntax refrains from too much use of SDF to suggest that the mapping could be from an abstract (or other) syntax definition.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

2.2 SET

module SET
imports Booleans Identifiers
exports
sorts SET ELEM
context-free syntax
"{" {ELEM ","}* "}" → SET
union(SET, SET) → SET
diff(SET, SET) → SET
inter(SET, SET) → SET
elem(ELEM, SET) → BOOL
ID → ELEM
variables
Set [*0-9′*]* → {ELEM ","}*
Elem [*0-9′*]* → ELEM
equations

[1] $\{Set_1,\ Elem,\ Set_2,\ Elem,\ Set_3\} = \{Set_1,\ Elem,\ Set_2,\ Set_3\}$

[2] $union(\{Set_1\},\ \{Set_2\}) = \{Set_1,\ Set_2\}$

[3] $elem(Elem,\ \{Set_1,\ Elem,\ Set_2\}) = true$

[4] $diff(\{Set_1,\ Elem,\ Set_1{}'\},\ \{Elem,\ Set_2\}) = diff(\{Set_1,\ Set_1{}'\},\ \{Elem,\ Set_2\})$

[5] $$\frac{elem(Elem,\ \{Set_1\}) \neq true}{diff(\{Set_1\},\ \{Elem,\ Set_2\}) = diff(\{Set_1\},\ \{Set_2\})}$$

[6] $diff(\{Set\},\ \{\}) = \{Set\}$

[7] $inter(\{Set_1\},\ \{Set_2\}) = diff(\{Set_1\},\ diff(\{Set_1\},\ \{Set_2\}))$

2.3 Defining *vods*

In order to define the visual appearance of sets, we first define the appropriate graphical tokens. So, we define some new *vods* for *Sets*. Table 1 shows the chosen visual representations for the *Set* tokens. The *vods* below are defined using *defv* which defines new *vods* via composition or *vod*-operations.

The *v-set vod* is a parameterized (by elements) definition, consisting of a circle containing a collection of elements.

```
defv v-set(Elems)
  {  cir : circle(),
     elems : Elems
  } ◁ {
    cir contains elems   }
```

When we consider defining the function symbols, we see that there are a number of similarities present in all of them (the border and the two overlapping circles). To avoid

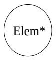
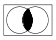
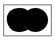
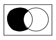
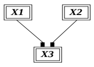
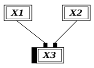
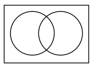
vod	representation
v-set	
intersect-sym	
union-symbol	
diff-symbol	
commf	
noncommf	

Table 1: *vod* definitions and their visual appearances.

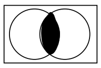
repeating this definition for each function symbol, we define a *vod* corresponding to these similarities and call it *two-sets*:



In this definition, there are six sub-*vods*, three of which (set_1, set_2 and border) are ordinary predefined sub-*vods* and the other three (inter, diff_1, and diff_1) are emergent-*vods* recognized in this composition. set_1 and set_2 are defined using the v-set abstraction just defined (where ϕ_c denotes the empty collection) and border is defined to be a *rectangle()* which is a primitive *vod*. Emergent-*vods* are determined by graphical operations present in VODL, like the *overlap* and *difference* operations. Note the use of these emergent-*vods* in subsequent *vod* definitions (intersect-sym, union-sym, diff-sym). Also, note the lack of attributes. Many attributes are defined as a result of constraint solving and the others are assigned default values. This style of specification allows the specifier to concentrate only on the essential properties of the definitions.

defv *two-sets*()
{ set_1 : v-set(ϕ_c) ,
set_2 : v-set(ϕ_c) ,
inter : overlap(set_1,set_2) ,
diff_1 : difference(set_1,set_2) ,
diff_2 : difference(set_2,set_1) ,
border: rectangle()
} $\lhd$ {
$\mathrm{set}_2.\mathrm{left} > (\mathrm{set}_1.\mathrm{left} + \mathrm{set}_1.\mathrm{right})/2$,
border contains set_1 ,
border contains set_2 }

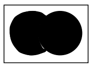
The function symbol for *intersection*:



is defined with the $\oplus$ *vod*-operation, which redefines *two-sets* by altering the highlight attribute of the *inter* component.

defv *intersect-sym*()
two-sets()$\oplus$[inter.highlight = true]

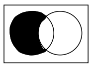
Similarly, the function symbol for *union*:



is defined by altering the highlight attributes of the components set_1 and set_2.

defv *union-sym*()
two-sets()$\oplus$ [set_1.highlight = true ,
set_2.highlight = true]

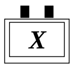
The function symbol for *difference*:



is also similarly defined by altering the highlight attribute of the diff_1 component.

defv *diff-sym*()
two-sets()$\oplus$[diff_1.highlight = true]

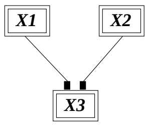
Now, we define three auxiliary parameterized *vod*s for representing the set functions. First, we define a *vod* (fsym-ports) which attaches two small rectangles on the top of a *vod*.



Later, we will use this for representing input ports for functions.

defv *fsym-ports*(*fsym*)
{ f: *fsym* ,
port_1 : rectangle()$\oplus$[height = 5, width = 5] ,
port_2 : rectangle()$\oplus$[height = 5, width = 5]
} $\lhd$ {
port_1.bottom = f.top ,
port_1.left = (f.right - f.left) * 0.3 ,
port_2.bottom = f.top ,
port_2.left = (f.right - f.left) * 0.6 }

Next, we define two *vod*s for representing binary functions with their operands. For commutative functions we define *commf*:

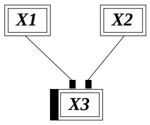


defv *connector*(v_1,v_2)
{ from: v_1, to : v_2, cl : line
} $\lhd$ {
cl.left touches from ,cl.right touches to }

defv *commf*(*fsym*,vod_1,vod_2)
{ f : fsym-ports(*fsym*) ,
v_1 : vod_1 ,
v_2 : vod_2 ,
con_1 : connector(v_1,f.port_1) ,
con_2 : connector(v_2,f.port_2)
} $\lhd$ {
v_1 over f , v_2 over f }

For non-commutative functions we define *noncommf*[1], which adds a strip on the left side of a function symbol:

[1]The black strip on the left side is for representing orientation, which is similar to *dog ears* in Holt's [3] *viz* language.



defv *noncommf(fsym,*v_1,v_2)
commf(fsym, v_1, v_2)$.f \otimes$
$(\{tag : rectangle() \oplus [highlight = true, width = 20]\}$
$\lhd$ { tag.right = f.left ,
tag.top = f.top ,
tag.height = f.height })

Note that the effect of this definition is analogous to inheritance in object oriented systems, where an earlier definition is extended. Here, *commf* is extended with an additional *vod*-ref, tag, and some additional constraints, giving us the desired *vod* for *noncommf.*

2.4 Mapping to VODL

Using VODL we create the necessary *vod*-types (lexicals) required to specify visual syntax. However, this does not constitute a language definition and therefore can not be used in defining tools. In order to obtain a language definition we should either use these as lexicals in a visual syntax definition formalism where we can define signatures using the *vod*s or define a relation using the *vod*s to a syntax definition thereby providing a visual syntax for that language. In this paper, we consider the latter where we take advantage of the ASF+SDF formalism and Meta-environment in defining languages and mappings. This approach consists of associating visual lexicals to language constructs. When using SDF it implies replacing the textual lexicals with visual ones.

For example, for the *Set* language visual lexicals can be specified using, among other *vod*s, the *vod*s described in the previous section. A mapping from *Set* to VODL defining the visual appearance of the constructs in *Set* could be:

$\nu(\{Elem^*\})$= *v-set*(ν-*coll*(*Elem**))
$\nu(inter(S_1,\ S_2))$= *commf*(*intersect-sym*(),$\nu(S_1)$,$\nu(S_2)$)
$\nu(union(S_1,\ S_2))$= *commf*(*union-sym*(),$\nu(S_1)$,$\nu(S_2)$)
$\nu(diff(S_1,\ S_2))$= *noncommf*(*diff-sym*(),$\nu(S_1)$,$\nu(S_2)$)

ν(elem(Elem, S))= { $e : \nu(Elem)$,
s : $\nu(S)$,
c : $connector(e, s)$,
t : *text*("?") $\oplus$ [*bold* = *true*]} $\lhd$
{ t over c }

$\nu(Id\ )$= $text$(id-to-string(Id))

The last two *vod* mappings define *vod*s directly unlike the previous ones which used abstractions. Note the similarities among the mapping rules. Each rule shares the same variables on both sides of the equations. Section 4 discusses how a tool can support such mapping definitions by generating a significant part of the mapping.

3 Use of visual languages definitions

In the previous two sections we defined visual lexicals and a mapping from a language definition to VODL, together which consider as forming a visual syntax definition for a language. A problem here is that we map everything to a *vod*, which on one hand allows much flexibility in specifying visual description, but on the other hand does not really provide a structure for the visual language. After all, we want to be able to use the visual syntax definition to generate some tools to support programming in that language. This requires some structure in the resulting language. However, by defining the mapping we

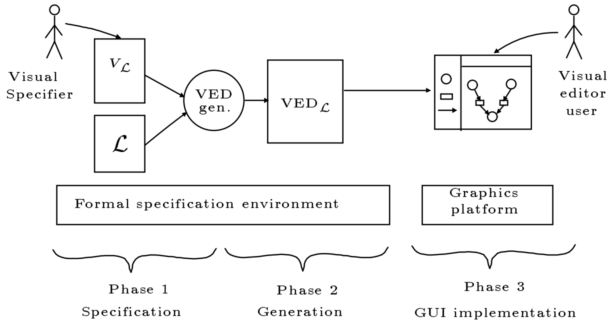


Figure 1: The overall view of a structured visual editor generation. $\mathcal{L}$ denotes the specification of a language, $V_{\mathcal{L}}$ is the visual specification (mapping) for that language, and $VED_{\mathcal{L}}$ is the visual editor description.

not only give visual appearances of the language constructs, but also provide a language structure.

To illustrate, consider one of the mapping rules:

$$\nu(\text{diff}(\text{S}_1,\ \text{S}_2)) = \text{noncommf}(\text{diff-sym}(), \nu(\text{S}_1), \nu(\text{S}_2))$$

The right hand side is parameterized by the visual appearances of S_1 and S_2 both of which are of sort *Set*. We can consider the above mapping indicating a context-free function:

$$\text{noncommf}(\text{diff-sym}(), \nu(SET)\ , \nu(SET)\ ) \quad \to \nu(SET)$$

where $\nu(SET)$ represents the sort *Set*. Currently the language definition is processed first and then mapped to visual appearances, but it follows the above idea.

The most important tool for visual languages is a visual editor allowing the graphical construction visual programs. Using the *Set* language which defines the context-free syntax of the language and the mapping which describes the visual appearances of its constructs, we can generate a structured editor for a *visual Set* language [6]. The generation, in fact, produces an abstract definition of the structured visual editor. This definition consists of the syntactic replacement behavior, visual representations of the language constructs, and functions to be included in the editor (which are defined by the semantics of the source language). The SVE description is utilized by a graphical user-interface that implements the editor. Figure 1 shows the overall view of the visual language definition and editor generation process.

In such an editor we construct visual programs via selecting *vod*s and replacing them with appropriate *vod*s until the desired program is constructed. The structured visual editor (SVE) consists of two regions, one in which the program is constructed and the other a panel where suitable replacements are presented. Each SVE has a language independent and language dependent part. Figure 2 shows the relation between the generated SVE description and the editor. The language-independent part remains the same across all SVEs, such as user interface components and input/output behavior. The language-dependent part deals with the language specific editing behavior as well as semantic behavior as defined in the source language. The semantic behavior is handled by maintaining a corresponding term over the source language while the term over the target language is constructed in the editor. The underlying term is transparent to the editor user. When an evaluation is requested, the source term is evaluated and then mapped to its visual appearance before being presented.

For example, Figure 3 shows an SVE for the language *Set* as defined by the mapping. The SVE allows two kinds of replacements: by an item from the replacement panel and by sub-terms of the term being constructed. The latter allows the construction of pictures with an underlying graph structure such as shown in Figure4.

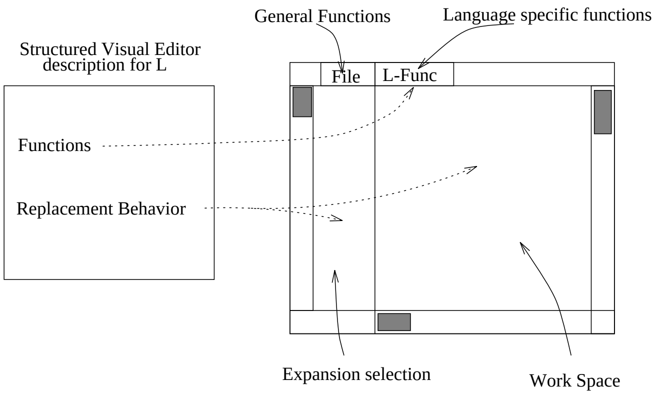


Figure 2: The structured visual editor and how it relates to the editor description.

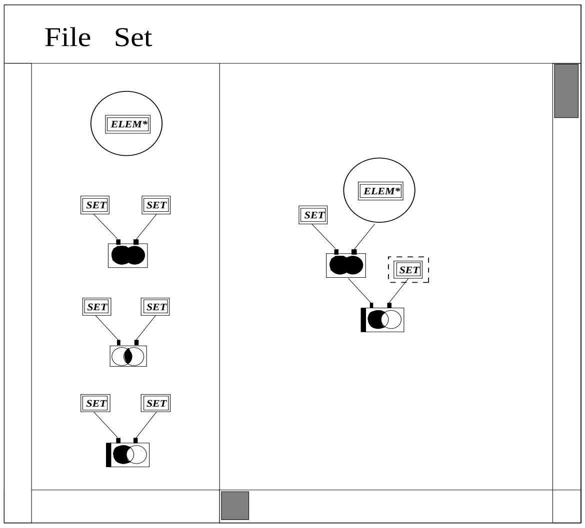


Figure 3: The construction of a set term in the visual set editor.

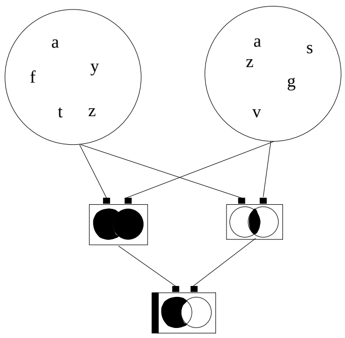


Figure 4: A visual term over the Set language representing $(\{a, f, y, t, z\} \cup \{a, s, z, g, v\}) - (\{a, f, y, t, z\} \cap \{a, s, z, g, v\})$.

4 Tool to support the mapping

The mapping defined in section 2.4 for defining an alternate syntax representation for sets is very uniform, where each construct in the source language (*Set*) is mapped to a construct in the target language (VODL). What we need is a tool that, given two languages and a mapping name, yields an editor for defining the mapping. Such an editor would contain a template for the appropriate translations that need to be defined. Such a tool can be used to specify different kinds of syntactic representations (as in our goal), but also provide basis support the definition of programming transformations. This section outlines the generation of such a tool.

The mappings of concern here is a homomorphism among languages, where a translation is defined for each language construct. According to Gougen [2], the definition of a homomorphism for many-sorted signatures is:

Definition: A *many-sorted signature* is a pair (S, Σ) where S is a set and Σ is an indexed set, $\{\Sigma_{w,s} | w \in S^* \text{ and } s \in S\}$. w is called the arity and $< w, s >$ the rank of $\sigma \in \Sigma_{w,s}$

A *many sorted* Σ-*algebra* A is an indexed set $\{A_s | s \in S\}$ of carriers and an indexed family of $i_{w,s} : \Sigma_{w,s} \rightarrow [A_w \rightarrow A_s]$ of interpretation functions for operation symbols.

A *many sorted* Σ-*homomorphism* h*:* $A \rightarrow B$ is an indexed set $\{h_s : A_s \rightarrow B_s | s \in S\}$ of functions such that

$$h_s(\sigma_A(a_1, ..., a_n)) = \sigma_B(h_{s_1}(a_1), ..., h_{s_n}(a_n))$$

for $\sigma \in \Sigma_{w,s}$ and $a_i \in A_{s_i}$ when $w = s_1 ... s_n$.

In line with this definition, a tool for supporting such homomorphisms can be generated given the following information:

- L_1 : source language
- L_2 : target language
- h: name of the homomorphism function

Given this information the following template for the mapping is generated:

- $h(s_i) = \langle \text{s} \rangle | s_i \in S$ $\langle \text{s} \rangle$ is a meta-variable of sort $SORT$
- $h(\sigma_{L_1}) = \sigma_{L_2}$, where
  σ_{L_1} is a term of L_1 generated by replacing each sort occurrence by a unique variable of the same sort, and
  σ_{L_2} is a term of the sort $h(s_i)$ where s_i is the sort of σ_{L_1}.

What we need is a tool that, given two language definitions and a mapping name, yields an editor for defining the mapping. Such an editor would contain the left-hand-sides and right-hand-sides (with place holders) as appropriate. For example, let us consider applying this framework to the *Set* example. First we need to specify the input for the generator:

$$\begin{array}{l} h \leftarrow \nu \\ L_1 \leftarrow Set \\ h_2 \leftarrow \text{VODL} \end{array}$$

There are two sort mappings for the *Set* language corresponding to the two sorts SET and ELEM:

$h(SET) = \langle \text{SORT} \rangle$ $h(ELEM) = \langle \text{SORT} \rangle$

In each case we replace $\langle$SORT$\rangle$ with VOD.

The mapping equations are as follows:

$$\begin{array}{ll} \nu(\{\text{Elem*}\}) & = \langle \text{VOD} \rangle \\ \nu(\text{union}(Set_1\ ,\ Set_2)) & = \langle \text{VOD} \rangle \\ \nu(\text{diff}(Set_1,Set_2)) & = \langle \text{VOD} \rangle \\ \nu(\text{inter}(Set_1,\ Set_2)) & = \langle \text{VOD} \rangle \\ \nu(\text{Id}) & = \langle \text{VOD} \rangle \end{array}$$

This provides a basic template but is not so interesting. What happens when we want to replace the right-hand-side *vods*? There are a few ways we can specify *vods*. We could use predefined *vods* like seen in section 2.3 or define them on the fly. At any rate what we want is to have some assistance in choices such that the mapping conforms to a homomorphism.

To make this editor more useful we not only rely on the syntactic definition of the target language, but also introduce syntax based on the left-hand-side of the mapping. For each, variable v_i on the left-hand-side we introduce a replacement option $h(v_i)$ for the meta-variables on the right-hand-side with the appropriate sort. For a homomorphism to hold, for each v_i on the left-hand-side there must be a $h(v_i)$ on the right-hand-side. A type-checker could be specified to verify this property.

For example, for the mapping

$$\nu(\text{inter}(Set_1,\ Set_2)) = \langle\text{VOD}\rangle$$

the replacements we want for ⟨VOD⟩are:

$$\begin{gathered}\{\ \langle\text{VODDEC*}\rangle\ \}\\ \langle\text{VOD}\rangle \oplus [\langle\text{ATTR*}\rangle]\\ \{\ \langle\text{VOD}\rangle\ \} \lhd \{\ \langle\text{CON*}\rangle\ \}\\ \langle\text{VODNAME}\rangle(\langle\text{VOD*}\rangle)\\ \nu(Set_1)\\ \nu(Set_2)\end{gathered}$$

The replacement behavior of ⟨VODDEC*⟩ is:

$$\langle\text{VODREF}\rangle : \langle\text{VOD}\rangle$$

These replacements allow us to define the mapping:

$$\nu(\text{inter}(\text{S}_1,\ \text{S}_2)) = \text{commf}(\text{intersect-sym}(), \nu(\text{S}_1), \nu(\text{S}_2))$$

This is achieved by replacing the ⟨VOD⟩ with ⟨VODNAME⟩ (⟨VOD*⟩) and then replacing ⟨VODNAME⟩ by *commf* and ⟨VOD*⟩ with diff-sym(), $\nu(Set_1)$, and $\nu(Set_2)$.

In this manner the replacement behavior depends on the syntax of the language L_2 as well as some context dependent syntactic replacement based on the the left-hand-side of the mapping. This provides a somewhat more sophisticated choice of replacements in defining language translations.

We can also define additional checks such as checking whether each construct is mapped to a different *vod*, which could be desirable for visual syntax if we don't wish to have different syntactic constructs share the same visual appearance. We could also impose a ordering on sorts as in [8] to decide on the appropriateness of the mappings. Another interesting point to investigate is how the replacement choices can be further refined by the specifier providing some additional information, such as a set terms over the target language. For example, if the only *vods* allowed in the mapping were the ones defined in 2.4 we could further refine the choices for defining a homomorphism by selecting the appropriate *vods* from that set based on sort information and number of parameters required.

5 Discussion

For the definition of a mapping tool we need to write meta-level specifications, where we define a mapping syntax over two language definition modules. AsFix [5] is specifically designed to write meta-level specifications for ASF+SDF modules. It uses prefix notation for representing the abstract signature of languages. We can define a tool as defined in 4 using AsFix, which defines such mappings using the AsFix representation for the source and target languages. We can also define the type checker for homomorphisms.

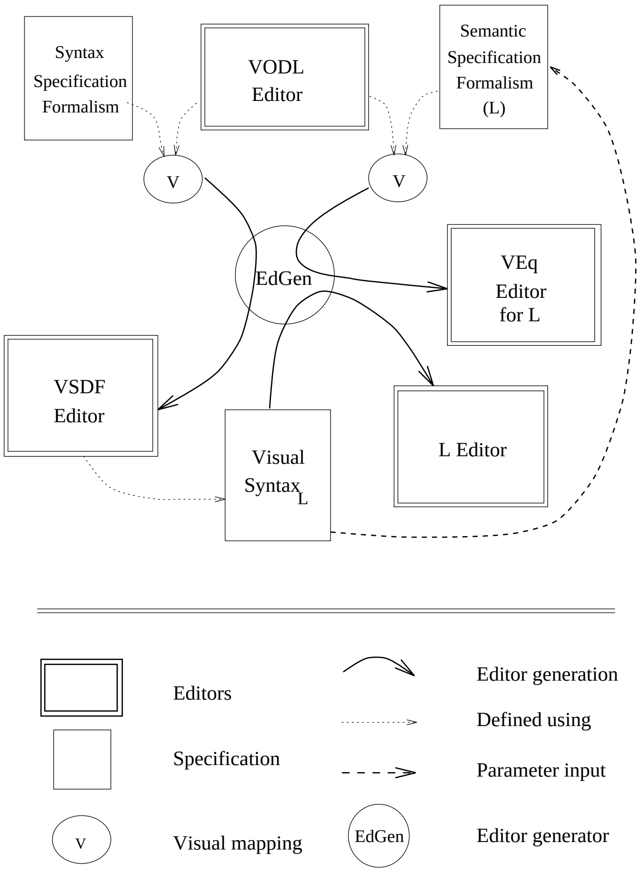


Figure 5: An overview of the visual specification environment

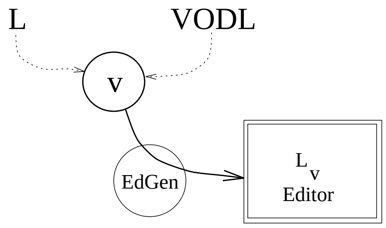
6 Future Work

The main focus of my research is to deal with visual languages and tools for them. The work presented here deals with how to describe visual syntax by relating it to an existing language specification (abstract or otherwise). The real goal is to define a visual language specification environment which parallels the ASF+SDFMeta-environment. Thus, to eliminate the mapping process (as defined in this paper) from the visual language specification process.

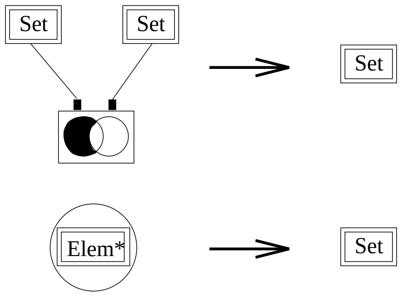
So, is the point of this paper futile? Not necessarily. Firstly, it will be used in the construction of the visual specification environment. And secondly, language translations are a very common activity when dealing with programming languages.

The idea in constructing the visual specification environment is to define a mapping from a language specification formalism. Then to generate an editor for specifying visual syntax and semantics. To do this, first a mapping from VODL to VODL must be defined. This allows the graphical construction of *vods*. Secondly, a mapping from an abstract specification to VODL must be defined. The overview of this approach is shown in Figure 5.

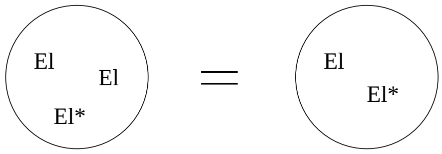
Recall that editor generation is achieved by having a visual language definition which is obtained via a mapping:



The top of Figure 5 shows this process being applied to the syntax definition formalism and semantic definition formalism (which is parameterized by a language). This yields an environment parallel to the ASF+SDF Meta-environment for visual languages, where in the VSDF editor we can define syntax rules like:



and in VEq editor for the *Set* language we can write semantic rules like:



The specification environment consists of the VSDF and VEq editors (which are obtained via a mapping) along with the editor generator and other tools and manager components. The editors for languages are generated free from mappings, since they can be directly specified in the VSDF and VEq editors. An SVE for a language L is generated from its specification defined in VSDF editor resulting in SVE_L, where terms over the language L can be constructed.

One of the significant components missing from this picture is a parser for visual languages. Ideally, we would like to have a hybrid editor as we have for the textual languages. Having a parser would ease the program construction for the end-user and also help in language specification. For example, it is not clear which strategy would be best in handling the effects of syntax alteration on the equations. With a parser, at least, the specifier is forced to update the equations with some help from the parser indicating the location of the erroneous construct.

7 Conclusions

In this paper we explore syntax translation for defining visual syntax. We discuss how such a language definition can be utilized in generating a visual editor. We also describe a tool which can aid in the language translation definition (mapping). Such a tool provides a template for defining the mappings and uses the language definitions to provide guidance in appropriate replacements. Such a tool could also be useful in serving as a basis for program transformations and pretty printer generation [1].

References

[1] M. v. d. Brand and E. Visser. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994. Postscript version available by anonymous ftp as `ftp.fwi.uva.nl:/pub/prg/reports/1994/P9420.ps.Z`.

[2] J. Goguen and R. Diaconescu. An oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.

[3] C. M. Holt. *viz*: A visual language based on functions. In *Proc. 1990 IEEE Workshop Visual Languages*, pages 221–226, Skokie, Illinois, October 1990.

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

[5] P. Klint. Writing meta-level specifications in asf+sdf. unpublished document, November 1994.

[6] S. M. Üsküdarlı. Generating visual editors for formally specified languages. In *Proc. 1994 IEEE Symposium Visual Languages*, pages 278–285, St. Louis, Illinois, October 1994.

[7] S. M. Üsküdarlı and T. B. Dinesh. VODL: A picture description language. Technical report, University of Amsterdam, 1995. To appear.

[8] D. Wang. *Studies on the Formal Semantics of Pictures.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, January 1995.