The VAS Formalism in VASE*

S. Üsküdarlı

Programming Research Group
University of Amsterdam
Amsterdam, The Netherlands
http://adam.wins.uva.nl/~susan

T. B. Dinesh

Department of Software Technology
CWI
Amsterdam, The Netherlands
http://www.cwi.nl/~dinesh

Abstract

This paper discusses the visual algebraic specification (VAS) *formalism and its supporting interactive environment* (VASE). *The* VAS *formalism is intended for the definition of visual syntax and semantics and is aimed at generating language specific environments. The* collection *construct of* VAS *is emphasized in that it supports the convenient specification of language constructs involving arbitrarily many items. The* VAS *formalism and supporting environment are discussed by use of an example in which the syntax, semantics and interaction of a toy language are specified.*

1 Introduction

Specifications of languages while on one hand desirable for the purposes of analysis and tool generation are on the other hand are intimidating as they are often cryptic and difficult to comprehend. This is often caused by the use of a fixed specification language burdened with expressing complex languages features [12, 8]. This problem is further magnified in the case of visual languages as the specification language must also deal with the complex visual notation and spatial relationships – yielding very complicated specifications. The fact is, to promote the development of specifications we need formalisms and tools that ease both the task of specifying as well as their comprehensibility.

We advocate a specification formalism, VAS, for specifying visual language syntax and the language semantics in terms of that syntax. The specification formalism is claimed to be easier to use and understand because of its use of concrete syntax which raises the level of specification to the syntax of the language being defined rather than being at a level of a fixed language. Thus, the specifier is allowed to remain at the level of the language being specified and not forced to map onto another representation.

We also define an interactive environment VASE for developing VAS specifications and generating end-user environments to construct and execute terms of the specified languages. The syntax definition is used to generate a syntax directed editor for constructing terms and the semantic definition is used to generate a rewriting engine to evaluate (execute) the terms.

*To appear in the proceedings of the 1996 IEEE symposium on Visual Languages, Boulder CO (Sept 1996).

The focus of this paper is the demonstration of how visual languages can be specified using VAS with the support of an interactive environment. First the VAS formalism is introduced in Section 2, followed by the VAS Environment in Section 3. Section 4 gives a brief description of the generated visual programming environment. Section 5 provides an example detailing both the specification formalism and the environment. The paper concludes with a discussion of related and ongoing work.

2 The VAS Formalism

The *Visual Algebraic Specification Formalism* (VAS) is an algebraic specification formalism for the visual specification of syntax and semantics of visual languages. For textual languages, algebraic specifications have proven to be desirable due to their simplicity. In practice, most algebraic specifications can be executed by orienting them as rewrite rules. Also, tools such as compilers, program analyzers, type-checkers, program slicers, and language specific editors can be generated from algebraic specifications [1, 4] which renders them useful for prototyping languages.

The VAS formalism finds its roots in the ASF formalism [1] which has been extended for dealing with visual languages. VAS separates the various aspects of language specifications (lexical, context-free syntax and semantic). VAS formalism consists of two specification languages: one for the syntax and the other for the semantics. The latter uses the former utilizing the concrete syntax of a language in the semantic definition. In another words the semantic specification language is parameterized by the syntax of the language. This supports the specification of semantics at the level of a specific language rather than at a fixed specification language yileding more comprehensible specifications.

VAS relies on VODL (Visual Object Definition Language) [14, 13, 15] for defining the lexical elements to support use of visual notation in specification. VODL is a declarative constraint-based language for defining visual elements which are called *vods* (visual object definitions) which consist of a set of sub-*vods* with possible spatial constraints among them. It supports abstract definitions with parameterized *vods* which are highly utilized by the VAS formalism. Not that VODL only defines pictures and it is at the level of VAS that they get special interpretion as language elements. The details of VODL are not explained here, but rather its use is demonstrated.

Languages are specified in modules which have the following syntax:

module NAME
    imports NAME-LIST
    exports
        sorts SORT-LIST
        functions FUN-LIST
        variables VARDEC-LIST
    hiddens
        ...
    equations
        EQUATION-LIST

where the notation *x-LIST* is used to indicate a (sometimes "," separated) list of *x*s. The next two sections discuss the syntax definition and the semantic specification by defined the sorts *FUN* and *EQUATION* respectively.

2.1 Specifying Syntax

This section presents the highlights of the VAS syntax formalism. Syntax specification involves the definition of the syntax of the functions of the language (signature). The sort *FUN* (as appearing in *FUN-LIST*) declares the functions of the signature[1] whose syntax defined as follows:

$$\begin{array}{lll} LHS \text{ ``}\rightarrow\text{'' } SORT & \rightarrow FUN & \\ SVL & \rightarrow LHS & \\ SORT & \rightarrow SVL & \\ vod_{lib}(SVL_1, \cdots, SVL_n) & \rightarrow SVL & n \geq 0 \end{array}$$

The right hand side of a function declaration is a sort name (*SORT*) and the left hand side defines the function syntax which may be any combination of sorts and visual lexicals (*SVL*), and collections (see Section 2.1.2).

2.1.1 SVL

The predefined *vod*s (defined with VODL) are denoted by $vod_{lib}(\cdots)$. Having *SVL*s as arguments to parameterized *vod*s allows complex functions to be defined in terms of other *vod*s and/or sorts — the latter representing the arguments of the function. and 3.2.

The following is a syntax rule:

$$\boxed{LABEL} \quad C \quad \rightarrow LBC$$

where $\square$ is a lexical and *LABEL*, *C* and *LBC* are *SORT*s. This rule defines a function of sort *LBC* with one argument of sort *LABEL* and another of sort C. The abstract representation of this function is: $f(LABEL, C) \rightarrow LBC$. In the concrete syntax the function symbol rectangle along with any spatial properties correspond to f. The construction of such rules and equations are discussed in Sections 3.1.

2.1.2 Collection

The VAS formalism provides a collection primitive which is used to conveniently define an arbitrary set of language constructs in signature definition. It serves a similar purpose to lists in a specification formalism for textual languages. The *LHS* of *FUN* can also be *collection*:

$$\begin{array}{ll} SORT \text{ ``}*\text{'' ``}\triangleleft\text{'' ``}\{\text{''} \; COP\text{-}LIST \text{ ``}\}\text{''} & \rightarrow LHS \\ vod_{lib}(\text{``}\square\text{''}, \text{``}\square\text{''}) & \rightarrow COP \end{array}$$

Collections define of a group of items of a sort ($SORT*$) "constrained" by the spatial properties as given by the *COP*s in *COP-LIST*. *COP*s are predefined constraint operators and the symbol "$\square$" is used to indicate the parameters

[1]Note that the arrow ($\rightarrow$), used in the mix-fix signature rules is the reverse of BNF style. Also, sorts correspond to non-terminals and lexicals correspond to terminals.

of the constraint operations. The arguments to these operators is the sort given in the collection definition – one for each parameter of *vod*.

The VAS formalism supports associative matching of collection items. This means that, the items in the collection will match other collections irrespective of the order in which they are constructed. Visually, the collection is basically a multiset which allows the definition of constraints over its elements. In its simplest form the collection items have no constraints. This could be a collection representing windows, where there is no particular constraint among the windows.

For example, a collection signature of the form

$$T{*} \ \triangleleft \ \{ \square\, v_1 \,\square \ , \square\, v_2 \,\square \} \ \ \rightarrow \ \ S$$

means that two visual operations $T\, v_1\, T \ \rightarrow \ S_R$ and $T\, v_2\, T \ \rightarrow \ S_R$ are defined over a hidden sort S_R.

For representation and semantics of the collection, we use the relational grammar representation of Wittenburg [18]. A collection is converted into an indexed multidimensional multi-set which is of the form $(I, R_1 \cdots R_n)$ where I is an indexed-set of elements and the $R_1 \cdots R_n$ are binary relations over these indices. This implies that every instance of v_1 or v_2, in the example above, would be an R_i $(1 \leq i \leq n)$; and every instance of T in the collection would be an indexed element of I.

Section 5 provides an example discussing how such a structure is built using structure editing and how equations that use this structure are interpreted. Such a special sort also calls for a special kind of variable declaration so that special treatment could given to equations that rewrite parts of a collection. We choose a second-order style [4, chapter 8] of variable declaration that could be used to help match parts of a collection and help rewrite *in context*. Equations that deal with collections are translated to conditional equations over the underlying multidimensional multi-set. By using an indexed multi-set during rewriting we can realize sharing. Space limitations do not allow us to elaborate on specific details but we hope that Sections 5 and 6 is illustrative enough for the purposes of this paper.

2.2 Specifying Semantics

The semantics of a language is given by a set of equations relating the equivalent terms of the language. The rules use the concrete syntax defined for the language. For every sort in the language there is an equation:

$$\text{“[”}\, \mathit{ID} \,\text{“]”} \ \mathrm{S} \ \text{“ =”} \ \mathrm{S} \rightarrow \mathit{EQUATION}$$

where *ID* is a label identifying it. For example, the equation:

$$[l1] \qquad \boxed{\text{aa}}\bigcirc \quad = \quad \boxed{\text{a}}\bigcirc$$

is a semantic rule stating that algebraically the terms on the left and right of “=” are equal. The syntax follows from the definition given in Section 2.1 for *LBC* where *C* is a circle and *LABEL* is ‘aa’ and ‘a’ on the left and the right hand side respectively . For the purpose of term rewriting, this is interpreted as a rule to rewrite a term matching the left hand side with the one on the right hand side resulting in rewriting the label with two ‘a’s to one with a single ‘a’.

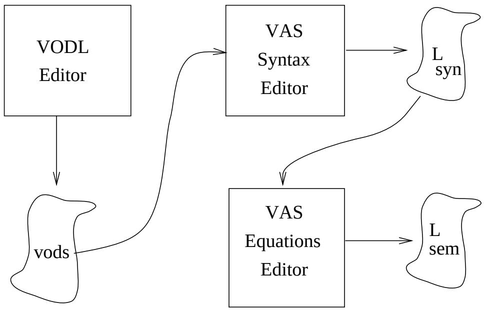


Figure 1: The VASE editors and their relations to various specifications.

In the above equation, the spatial layout of the rectangle and circle are always consistent with their syntactic definitions. This rule will only match terms possessing the same relations. Terms that may appear similar but do not have the same spatial relations will not match. Thus, the interpretation [17] of language constructs is always consistent with their syntactic definition.

3 VASE

The *Visual Algebraic Specification Environment* (VASE) is the interactive environment for developing VAS specifications. It consists of editors for the definition of lexicals, syntax, and semantics of visual languages based on the VAS formalism. Figure 1 shows the editors of VASE.

The VODL editor is used to define *vod*s, which are utilized by the VAS syntax editor in defining the signature of a language. The signature, in turn, used by the VAS equation editor for defining the semantics of that language. This implies that the necessary lexical definitions must exist before the syntax is defined and the syntax definition must precede the definition of the semantics. The example presented in Section 5 demonstrates these relationships.

All the editors are syntax directed editors. The terms are constructed by selecting meta-variables (place-holders in terms) which can be replaced by permitted replacements as defined by the syntax. The replacement can be selected from a panel of permitted choices or, alternatively, from the term (a subterm) that is being constructed. The latter option allows sharing, which is true sharing when constructing a collection, otherwise is pseudo sharing [15].

The editors use a constraint solver for placement as dictated by the constraints present in the syntax. Recall that lexicals may have constraints in their definitions. The user of the editor is allowed to move subterms around as long as the constraints permit.

3.1 VAS Syntax Editor

The VAS syntax editor supports the syntax of a visual language according to the grammar defined in Section 2.1. Here the construction of a single function specification is shown. The goal is to specify the function: $\boxed{\boxed{A}} \rightarrow$ B. Figure 2 shows a possible construction sequence. The first two steps simply follow

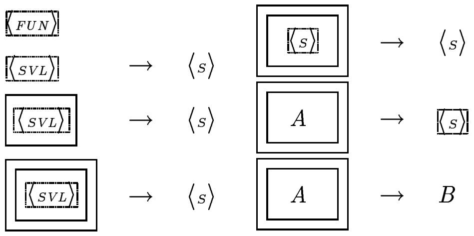


Figure 2: A syntax rule construction sequence.

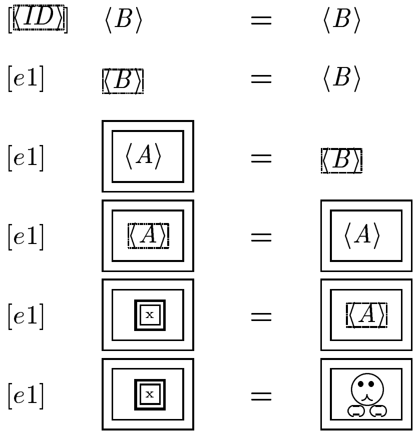


Figure 3: An equation construction sequence.

from the VAS grammar. Recall that VAS relies on *vod* definitions for its lexical elements. In this case we could use the *vod inbox*:

$$
\begin{array}{l}
\textbf{defv}\ \ \textit{inbox}(X)\{\\
\quad \mathrm{v}_1 : \mathrm{rectangle}(),\ \mathrm{v}_2 : \mathrm{X}\ \}\\
\quad \lhd\ \{\ \mathrm{v}_1\ \textit{contains}\ \mathrm{v}_2\ \}
\end{array}
$$

The behavior of the VAS editor when replacing and $\langle SVL \rangle$ with a lexical is to use $\langle SVL \rangle$ as arguments to the *vod*, which in this case results in $\boxed{\langle SVL \rangle}$.

3.2 The Equation Editor

The semantics of a language is defined with equations using the language's own syntax. The editor is parameterized by the syntax of the language as discussed in Section 2.2. The syntax of the equations is generated from the syntax definition.

Figure 3 shows the construction of an equation for the function defined in the previsou section. Here, "goofy" ( ) is a constant and is a variable both of sort A. A rewriting system generated from equations including the above would rewrite all terms of sort A which are contained within a double-box to a double-box containing the "goofy" character. It should also be mentioned that this equation causes rewriting to loop as the "goofy" character is itself of sort A.

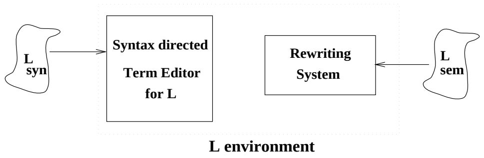


Figure 4: The end user environment for the language L, which is generated from the specification of L.

4 VPE for Specified Languages

The intention of specifying the languages is for the purpose of generating language specific tools. Figure 4 shows the use of these specifications in generating programming environments for languages. Note that, it is not necessary to have a semantic definition, in which case the end environment will have only an editor allowing the construction of terms. Both the components in this figure are generated from the corresponding specifications. The editor allows the construction of terms over the language in a syntax directed manner. The rewriting system makes the specifications executable. In fact these components interact through the use of interface definitions, which bind functions over the constructed term to a button in the editor [13]. Clicking a button, passes the term to the rewriter for execution. The resulting term may replace the term in place or be placed in another editor.

5 Example: Move-about Language

The move-about language is used as an example to demonstrate VAS formalism and VASE. Each of the following sections assumes that VASE is used for defining the specifications. All lexicals used are predefined in VODL. The move-about language is a simple language which consists of a collection of squares which contain game objects (*GO*s). Given a direction, the player can move in that direction only if the game piece in that direction is an empty square. We show how the syntax is specified (using the collection primitive) and how the semantic equations are written to specify its behavior as well as perform some static checking of initial game conditions.

5.1 Syntax Definition

Figure 5 shows the syntax of this language. The first eight functions are nullary functions (first of which is simply an empty *vod*). $\boxed{GO}$ is constructed from a unary *vod* that constrains its parameter to be contained within a square. Finally, $PIECE * \lhd \{ \boxminus\, , \;\square\;\square\; \}$ is a collection of *PIECE*s constructed with the binary *vod*s included in the bracket. The first *vod* defines 'above' and 'touches' and 'aligned' constraints over its parameters. The second one does the same but with 'right-of' instead of 'above'. The notation seen here is the *vod* applied to the placeholder symbols ($\square$). This collection is used to define the sort *GAME*.

The variables that will be used in equations must be declared. $\boxed{\boxed{\text{G}}} \rightarrow GAME$ and $\boxed{\boxed{\text{Dir}}} \rightarrow DIR$ are used in the equations in the following sections.

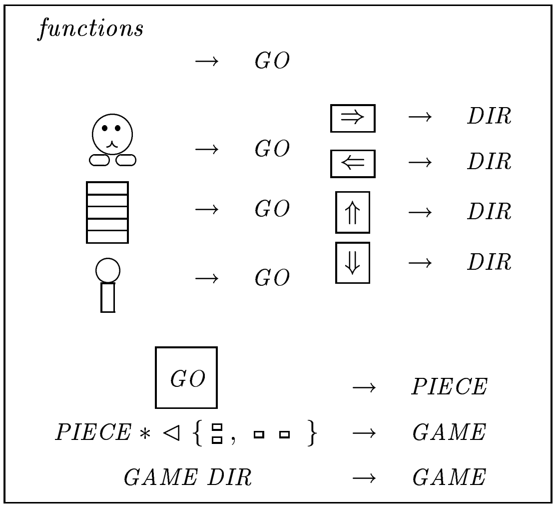


Figure 5: Syntax definition using VAS formalism

Higher order variables [4, chapter 8] are used for defining variables for the collection. These variables have a special interpretation during rewriting with respect to matching. The syntax of which are given by: $vod_{lib}(SORT{*} \lhd \{COP\text{-}LIST\}) \rightarrow SORT$. Using this, we define the collection variable used in the equations:

$$\left(\overline{\underline{PIECE{*} \lhd \{\square, \ \square\ \square\}}}\right) \rightarrow GAME$$

Here, the outer container matches the context in which a specific pattern is looked for. This allows only the relevant items of the collection to be specified in the equations. Note that the spacial layout of the constaint operators in the above rule come from their corresponding *vod* definitions. The placeholders serve as arguments to provide visual feedback.

5.2 The Move-about Editor

From the syntax definition of the move-about language, a term editor is generated. Figure 6 shows a representative term of the *move-about* game. The term is constructed by expanding the *GAME* sort, which is a collection. The expansion of collection sorts needs some explanation.

For the collection $T * \lhd \{\square v_1 \square, \overset{\square}{\underset{\square}{v_2}}\} \rightarrow S$, the term construction is defined using an example. The "$\Rightarrow_i$" indicates the ith modification of the term after a replacement.

$$\boxed{\langle S\rangle} \Rightarrow_1 \begin{array}{l}\langle T\rangle\ v_1 \langle T{*}\rangle\\ v_2\\ \boxed{\langle T{*}\rangle}\end{array} \Rightarrow_2 \begin{array}{l}\langle T\rangle v_1 \boxed{\langle T{*}\rangle}\\ v_2\\ \langle T\rangle\ v_1 \langle T{*}\rangle\\ v_2\\ \langle T{*}\rangle\end{array} \Rightarrow_3 \begin{array}{l}\langle T\rangle\\ v_2\\ \langle T\rangle\ v_1 \langle T{*}\rangle\\ v_2\\ \langle T{*}\rangle\end{array}$$

Collections are represented by the sort name followed an $*$. The meta-variable $\langle T{*}\rangle$ can be replaced

$\langle T \rangle \mathrm{v}_1 \langle T* \rangle$

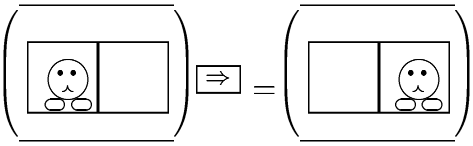
v_2

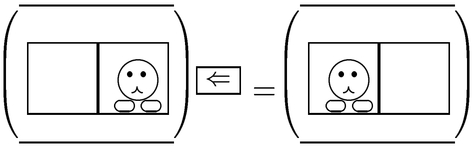
with: (1) nothing, (2) $\langle T \rangle$ or (3) $\langle T* \rangle$.

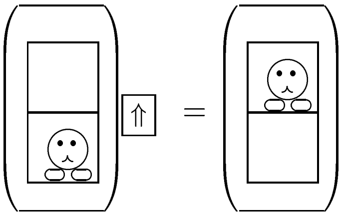
The editors try to maintain the constraints as much as possible during the construction of the term. But, while there are still meta-variables in the term – even for eventual correct terms – constraints may be violated. These cases are handled as an exception, where the term to be expanded that causes the violation is presented on the side, from where its relationships to the remaining term may defined.

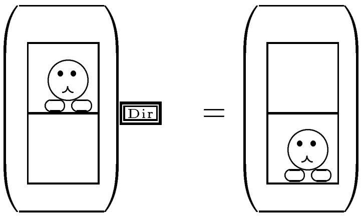
5.3 Evaluation Semantics

The evaluation semantics defines the behavior of the game. The following equations, define how the 'player' is moved from one scene part to the next given a direction.

[$s1$] 

[$s2$] 

[$s3$] 

[$s4$]  **otherwise**

[$s5$] $\boxed{\mathrm{G}}\boxed{\mathrm{Dir}} = \boxed{\mathrm{G}}$ **otherwise**

The rules [$s4$] and [$s5$] above are *default* rules [7] which are indicated with the keyword **otherwise**. The default rules are ordered with respect to the specificity of the left-hand sides—in this case [$s4$] has a higher priority than [$s5$].

5.4 Specification of Interaction

The equations of the prior Section describe how the player moves between scenes, given a direction. One or more directions for the game can be provided in

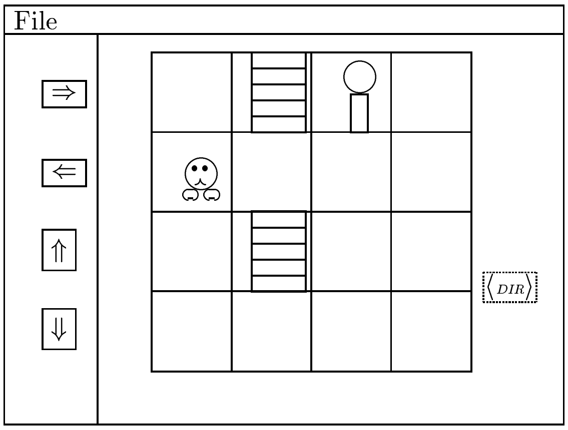


Figure 6: The game with interaction. The user selects the direction from the selection panel on the left.

the initially constructed term. Each of these directions, will cause one of the equations to be evaluated resulting in a (possibly) new game configuration.

Alternatively, the directions could be provided interactively during term rewriting [5][2]. Here the latter choice is considered. Briefly, we use meta-variables in the conditions of equations to indicate interaction. The presence of a meta-variable equated to a term with unbound variables causes the value for the variables to be solicited interactively. For the value needed a meta-variable of that sort is presented, prompting the user to provide a value (as usual only syntactically correct choices are presented). After the input is obtained, the rewriting resumes. Figure 6 shows a typical interaction window. We define this interaction using the function $play(GAME) \rightarrow \ GAME$. Interaction is specified as:

$$[p1]\ \frac{\langle GAME \rangle \quad = \quad \boxed{\mathrm{G}} \quad \boxed{\mathrm{Dir}}}{\mathrm{play}(\ \boxed{\mathrm{G}}\ ) \quad = \quad \mathrm{play}\ (\ \boxed{\mathrm{G}} \quad \boxed{\mathrm{Dir}}\ )}$$

This causes a direction to be repeatedly requested during rewriting.

5.5 Static checking

Finally, we define some checking for $GAME$ terms so that the game has only one player. To do this, we define a function $\mathbf{1?}GAME \rightarrow NAT$ which returns true when the game has only one player. An auxiliary function $\mathbf{n}(GAME) \rightarrow NAT$ is used to count the number of players. Sorts NAT and $BOOL$ are imported sorts.

equations

$$[t1]\ \mathbf{n}\left(\left(\boxed{\quad}\right)\right) = 1 + \mathbf{n}\left(\left(\boxed{\quad}\right)\right)$$

[2]Defining sub-terms externally, during rewriting, is uncommon for algebraic specifications. This is a recent extension to VAS formalism to accommodate the high demands of interaction of visual languages.

$[t2]$ $\mathbf{n}(\boxed{\boxed{G}}) = 0$ **otherwise**

$[t3]$ $\mathbf{1?}\boxed{\boxed{G}} = \mathrm{true}$ **when** $\mathbf{n}(\boxed{\boxed{G}}) = 1$

$[t4]$ $\mathbf{1?}\boxed{\boxed{G}} = \mathrm{false}$ **when** $\mathbf{n}(\boxed{\boxed{G}}) \neq 1$

This example, is greatly simplified for presentation purposes.

6 Related Work

VASE is inspired by the ASF+SDF Meta-environment [7] and uses many concepts and tools that are developed in its setting. The ASF+SDF formalism also has directly influenced some of the syntax and equation specification styles. The syntax specification is able to use visual lexicals in the construction of the signature. Our approach separates the constraint specification of visual lexicals from that of function specification. Thus we can define and use visual functions that are close to the intended interpretation. This is in contrast to that of many visual language syntax specification approaches (E.g., [6]). We have extended the term rewriting used in this context to include the collection sort which can be used for "graph rewriting" semantics.

The collection sort has been influenced by graph grammar formalisms. The Relational Grammars work [18] has influenced the selection of the underlying representation for the collection sort. The PROGRES graph rewriting approach [12] uses attributed graphs with explicit directives for manipulating attributes. The conditional set-rewriting [10] is more general than these approaches, but for representing collections, we think Relational Grammars are better suited.

Programming environments such as VAMPIRE [9], PROGRES [12] use a mix of pictures and text to define the intended semantics. They also use rewriting as the basic underlying operation. In our work rather than recognizing pictures, we have focused on the construction and working with correct terms. There is no parsing of pictures and rewriting is used to facilitate "execution" of a specified visual language. We have also concentrated on the use of concrete syntax in our tools.

7 Discussion and Future Work

The use of concrete syntax to specify the visual syntax of a language requires proper tool support that can help a user (specifier) who could get confused by a mix of over-loaded operation definitions and/or visual lexicals. This may seem irritating at first as we could have terms that appear to be as we desire but never match the rule we expect. However, this is important for the purpose of being able to define the intended semantics and get the intended interpretation [17]. Syntax directed editing is already helpful since the editor can help traverse the structures (being) constructed. There is a need for tools which present alternative views of the syntax definitions. There are many other user interface issues—such as improvement of interface specification, construction of collection terms—that need to be further studied.

Another problem of using concrete syntax is that sometimes the visual operation definitions might look different than when used. The assumptions made by a constraint solver when using a meta-variable while constructing a term might make it difficult to construct a term that could eventually be well defined. Also, the well-definedness of the algebraic semantics of the specification in this case would assume the certain well-behavedness of the function defintions. This well-behavedness would mean that, for any term that can be constructed during rewriting, the corresponding pictures do exist. A criteria that might be useful is the partial computability of the completion of a picture [3, Chapter 11].

During the process of rewriting, new terms are introduced that ultimately have to be presented to a user. This requires some smart analysis of the specification to realize a natural presentation. We are hoping that "origin-tracking" techniques [4, chapter 9] will be of use here. We have effectively graph rewriting, with collections, that has a notion of how the sharing of terms in a collection is preserved. We also have the case where a term is shared with another by the manner in which it was constructed in a term editor. This *pseudo-sharing* might mean that during rewriting the sharing is not considered. However, allowing this pseudo-sharing means that we can construct the desired visual syntax using the VAS formalism [15].

The use of concrete syntax at the specification language level is an interesting appraoch. In the cases of language constants it is much more expressive. However, when representing abstractions it becomes much more of a challenge to find good representations. For example, in this paper, we represented sorts and meta-variables with text, which makes it difficult to represent spatial relationships that involve overlapping. This arises from the clear conflict between the concreteness of visual notation and abstract representations.

The implementation, which is far from completion at present, relies on ASF+SDF Meta-environment [7] for meta level specifications, DeltaBlue [11] for constraint solving, Python/Tk [16] for interface development. ToolBus [2] which supports the interaction of heterogeneous tools is used for achieving tool communication.

8 Conclusions

The VAS formalism specifies visual language syntax and semantics. We have defined tools to interactively support VAS specifications, which in turn are used to generate language specific environments. The use of concrete syntax in these tools is investigated where visual notation is used both during the specification process as well as in the generated VPE.

Acknowledgments The authors gratefully acknowledge Paul Klint and the anonymous referees for their valuable input.

References

[1] J. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley,

1989.

[2] J. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[3] S.-K. Chang. *Principles of Pictorial Information Systems Design*. Prentice Hall, 1989.

[4] A. v. Deursen, J. Heering, and P. Klint. *Language Prototyping. An Algebraic Specification Approach*. AMAST series in Computing. World Scientific, 1996. To appear.

[5] T. B. Dinesh and S. Üsküdarlı. Specifying Input and Output of Visual Languages. In *Workshop on Theory of Visual Languages*, Gubbio, Italy, May 1996.

[6] E. J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.

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

[8] K. Marriott. Constraint Multiset Grammars. In *Proc. 1994 IEEE symposium on visual languages*, pages 118–25, St Louis, Missouri USA, 1994.

[9] D. W. McIntyre and E. P. Glinert. Visual Tools for Generating Iconic Programming Environments. In *Proc. 1992 IEEE Symposium Visual Languages*, pages 162–168, Seattle, WA., Sept. 1992.

[10] M. A. Najork and S. M. Kaplan. Specifying Visual Languages with Conditional Set Rewrite System. In *Proc. 1993 IEEE Symposium Visual Languages*, pages 12–18, 1993.

[11] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the deltablue algorithm. *Software–Practice and Experience*, 23:529–566, May 1993.

[12] A. Schürr, A. Zündorf, and A. Winter. Visual Programming with Graph Rewriting Systems. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[13] S. Üsküdarlı. Generating Visual Editors for Formally Specified Languages. In *Proc. 1994 IEEE Symposium Visual Languages*, October 1994.

[14] S. Üsküdarlı and T. B. Dinesh. Towards a Visual Programming Environment Generator for Algebraic Specifications. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[15] S. M. Üsküdarlı. *Algebraic Specification of Visual Languages*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, December 1996. In progress.

[16] G. van Rossum. Python Language Home Page. http://www.python.org/.

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

[18] K. Wittenburg. Earley-style Parsing for Relational Languages. In *IEEE Workshop on Visual Languages*, Seattle, Washington, 1992.