Algebraic Specification of Visual Languages

Susan M. Üsküdarlı

Algebraic Specification of Visual Languages

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr J.J.M. Franse
ten overstaan van een door het college van dekanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op maandag 24 maart 1997 om 13:30 uur

door

Susan Michele Üsküdarlı

geboren te Kirksville, U.S.A.

Promotor: Prof. dr P. Klint
Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam

CIP gegevens

Üsküdarlı, Susan M

Algebraic Specification of Visual Languages
Susan M. Üsküdarlı
Thesis Universiteit van Amsterdam. - With ref.
ISBN: 90–74795–65–X

Printed and bound by Print Partners Ipskamp.

to my dear friend T. B. Dinesh

Contents

Words of Thanks

My doctoral research started as a result of a short visit to Amsterdam, which turned into a rather extended stay. During my visit I was offered an assistantship from my promotor (advisor) Prof. dr Paul Klint and the rest, as they say, is history. I am deeply grateful both to Prof. Klint and to the programming research group which I have been a part of at the University of Amsterdam. They have graciously provided me with the opportunities to conduct my research.

I would like to extend my gratitude to my committee members: Prof. dr K. R. Apt, Prof. dr J. A. Bergstra, dr W. Citrin, Prof. dr P. Klint, Prof. dr ir F. C. A. Groen, dr J. Rekers, and dr D. Wang. Their diligent efforts in reading my manuscript and their excellent input has been very much appreciated.

Dr T. B. Dinesh is undoubtably the person who has had the most impact on my work. We have had many collaborations and lots of fun. He has been a great friend and kept me on my toes. I shall always remain grateful for his input and challenges!

I owe a very special thanks to Prof. Lambert Meertens for reading my thesis draft very thoroughly and providing me with detailed and humorous feedback. Teşekkürler dostum! Jan Heering has been instrumental in helping me keep perspective and lifting my spirits which have made a great difference to me.

I have had the opportunity to have great colleagues at the University of Amsterdam and at CWI (Centrum voor Wiskunde en Informatica) . I would especially like to acknowledge: Huub Bakker, Mark van den Brand, Arie van Deursen, Casper Dik, Joris Hillebrand, Wilco Koorn, Emma van der Meulen, Pieter Olivier, Chris Verhoef, and Eelco Visser. Among them I would like to acknowledge Arie van Deursen for his constant support both personally and professionally. He has provided me with very detailed and useful feedback. He has also translated the summary of this thesis into Dutch. Also, my former office mate and good friend Eelco Visser for his support and good company. Also, my thanks goes to Pieter Olivier who has been very helpful and amazingly quick with his technical help.

During the Spring of 1996 I had the great opportunity to visit dr Kim Marriott and dr Bernd Meyer at Monash University in Australia. I would like to thank them for all the support and fun they provided during my visit.

I have also had the pleasure to assist two masters projects one with Mike Taiafat and another with Harold Breebart. I would like to thank them both. I am still working with Harold who has been very fun to work with.

I hope I am not alone in having tortured my family and many friends especially while writing a dissertation. Their support has been phenomenal and has touched me deeply. I am told I shall be under observation to see if I ever return to being human $\ddot\smile$! I hope that I prove to do so...

I will start by thanking my parents Janet and Ayet Üsküdarlı for believing in me and supporting and putting up with me during my long academic path. Can Üsküdarlı has been a great brother and good friend flooding me with e-mail and phone calls (international) of constant support. He has lightened my spirits with his crazy sense of humor. I warmly remember my grandmothers, Wilma Dubberke and Irene Üsküdarlı, who were great women and who have inspired me in many ways and I am grateful for having known them and experienced their love. Also, my grandfather Ayet Üsküdarlı has always encouraged me to pursue higher education.

My dear friend Nurshen Bakır has been a great help and supporter. We have had great conversations and grown together as foreigners in a strange land. Her presence has made a great difference to my life. Patricia Griffin has been an inspirational friend who has always given me new ways to think about situations. She has been a very good friend and great great fun to be with. A very warm thanks goes to my dear friend Hikmet Salih Özkan, otherwise known as Chief Crazy Blanket, who has been a constant support and who has kept me in good spirits with his immense kindness and *great* sense of humor.

There are so many more to thank... Among them I would like to acknowledge Hamideh Afsarmanesh, Harun Aksu, Farhad Arbab, Lon Barfield, Pınar Elmasoğlu, Murat Fadıloğlu, Lene Gravesen, Lynda Hardman, Wendelynne Heelis, Ece Heper, Imagine, Sjoerd Mullender, Günsu Oğuztüzün, Carol Orange, Cocky Rotteveel, Chris Scheel, Jacintha Santen, Machteld Vonk (who has also been very helpful in reading parts of my dissertation) and Els Willems.

Finally, I have to acknowledge a place that I have frequented during my stay here. It is a nearby Turkish bakery called Saray, where Ibrahim Karabaş and Bekir Aleş work. It has always been a great place where I could escape when I needed. They always asked me what I was doing to get so tired. And I would answer "I am writing a thesis". It so happens that in Turkish the word for 'thesis' and for 'quick' are the same (tez). I had never noticed this until one fine day Bekir, who did not know what a thesis was, finally bursted out saying "Sister! This sure is not going very fast!!! What exactly were you doing?". He was right! And boy did I ever have a good laugh...

February 1997

Chapter 1

Introduction

1.1 Main Goals

For textual languages many techniques have been developed for generating tools to support programming in these languages. Examples of such techniques are parser and editor generators yielding an environment of language specific tools for the end-user. The objective of these techniques are to reuse the methods employed in constructing the commonly desired tools by generating them from a description of a language.

Several visual languages have graphical programming environments, sometimes referred to as visual programming environments, which have been crafted to support the construction, manipulation and evaluation of programs of those languages. This thesis is concerned with visual programming environment generation and the specification of visual languages. The next section discusses environment generation to better position this research in the programming environments spectrum.

Techniques for generating programming environments for visual languages have also been developed. One such technique is also the aim of this work. Our work finds its roots in the ASF+SDF formalism for specifying textual languages, and its supporting environment the ASF+SDF Meta-Environment [42]. ASF+SDF is an algebraic specification formalism for specifying textual languages and the ASF+SDF Meta-Environment is an interactive specification environment supporting the specification of languages using the ASF+SDF formalism. We explore the possibilities of extending the ASF+SDF formalism and environment to handle visual notation and specify visual languages. Accordingly, we can summarize our goals as the following:

- To extend textual, algebraic, language definitions with textual definitions of visual notation.

- To give visual support for the definition of visual notation.
- To generate tools based on the definitions for both the visual notation as well as visual language definitions.
- To support the previous three goals by means of experimental implementations.

Much like the ASF+SDF Meta-Environment this work is intended for providing support to the specifier and end-user of the specified language. Both the specifier and the end-user are in need of visual programming environments (VPEs) which are a set of tools that assist the development of visual program construction. Such tools can be editors, parsers, and analyzers. Programming environments have been highly successful for the case of textual languages. Considering the complexity of visual languages, programming environments for them are considered even more valuable.

1.2 Programming Environments and Generators

To build a highly sophisticated programming environment, indeed, requires much care and work as tools that bring about the particular features of the language of interest must be designed. On the other hand there are some basic tools that remain the same from one language to another. For example, editors and parsers share the same behavior but operate on different syntax. For such common tools it is sensible to avoid repetitive work by generating them from specifications. Given a specification formalism and tool generators for this formalism, any language that can be defined with that formalism can automatically obtain all generated tools. This means that tools may not be highly specialized but standard tools can be obtained effortlessly. This is particularly useful in language prototyping.

We sketch the context of our research by relating this work to that of others in the area of programming environments. This work focuses on generating programming environments for visual languages as well as on a programming environment supporting the definition of visual languages. It addresses two main issues: visual programming environments and visual languages. More specifically, it is concerned with the former for the latter.

In order to generate tools, a description of the language must exist. Tool generation is based on the description language. This is typically done with a specification formalism although occasionally frameworks have also been used as well. Specification formalisms are more formal and lend themselves to better analysis.

To set the context of this work, it is useful to have a taxonomy (Figure 1.1) relating the wide range of work in the area of visual languages as well as programming environments. The leaves of the tree represent programming environments or generators which are listed as representative environments/formalisms.

This taxonomy first divides programming environments along the line of language-specific and generic environments. It is very likely that nearly all programmers have used some language-specific programming environment. They are in demand since they ease the development and perhaps the maintenance of programs for a specific language with a set of tools specifically addressing the tasks required for that language.

Generic environments on the other hand are not language-specific, but rather environments with the intention of generating language-specific environments. The landmark environment for textual languages is the Cornell Synthesizer Generator [64] which uses attribute grammars to specify languages. The ASF+SDF Meta-Environment [42] uses an algebraic formalism for specifying textual languages and has been the inspiration for the VASE environment and will be explained in more detail in the next section and in Chapter 5.

Our work falls within the area of the dashed box, which concerns itself with the generation of programming environments for visual languages. Within this box different kinds of definition languages for specifying the visual languages which range from textual to visual languages. In the next section we shift our attention from programming environments to the definition of visual languages on which the generators rely. In doing so we look at two approaches: frameworks and grammar based approaches concentrating more on the latter as our work follows this approach.

1.3 Frameworks

Frameworks consist of a set of predefined data and behavior that provide the basis for a set of applications. They became popular in light of the graphical user-interfaces for applications that involved lots of similar work but were different in application specific behavior. By providing a large part of the base of an application the user becomes free to concentrate on customizing the framework with the application specific details. Frameworks require the language designer to adhere to a more specific style and are typically built for a class of languages which are to be customized with the language specific parts. Since frameworks are specialized they tend to yield more sophisticated tools. On the other hand if the language does not conform to the framework it is difficult or impossible to define the needed language.

McIntyre's VAMPIRE [51] is an object-oriented framework for defining iconic languages. It consists of a set of tools for editing class hierarchies, rules, and icons; and generates a run time environment that allows building and executing programs in the defined language. The general behavior of the icons is defined in the class hierarchy where the leaf nodes represent the icons of the language. Their

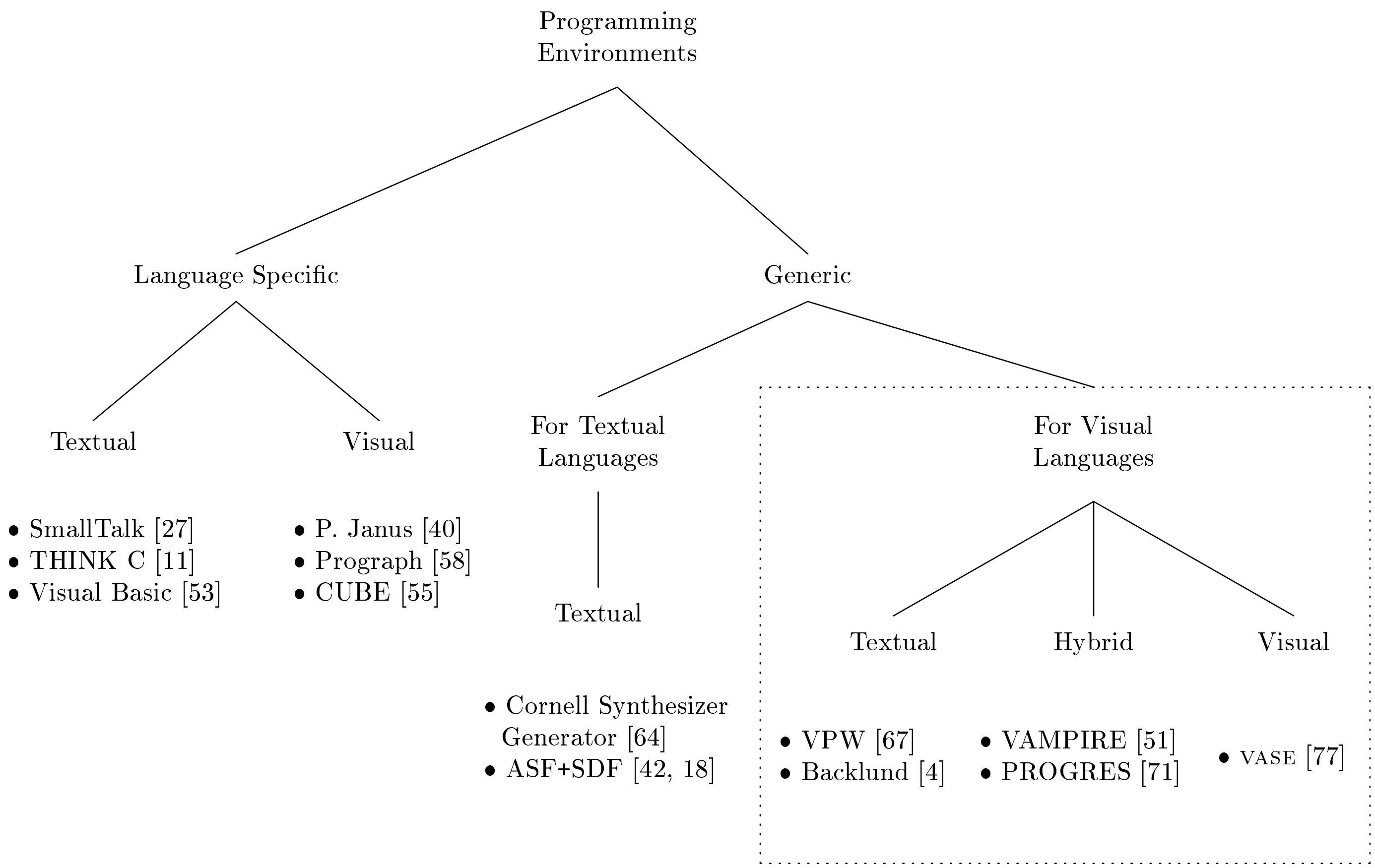
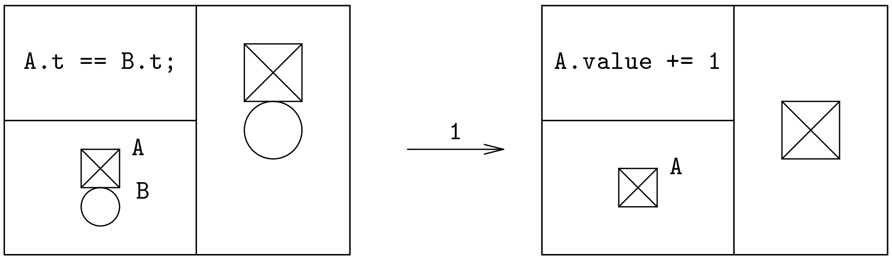


Figure 1.1: Taxonomy of programming environments and their generators. The region enclosed within the dashed box indicates the research concerning this work: visual environments for visual language specification.

behavior is represented as much as possible with graphical rules that look like:



on the left side the rule shows the icon and any attributes to be matched and the right side shows the icon to replace it and possibly new values for its attributes. Each box consists of three parts: the upper left part contains attributes that have no visual representation, the lower left part shows a labeled depiction of the icon of interest to be used in the attribute box, and the right box contains the icon to be matched. The expressions in the attribute box are considered as constraints if they are on the left side of the rewrite rule and as attribute assignments if they are to the right of the rewrite rule. The number above the arrow indicates a preference to resolve any conflicts when multiple rules match the icons in the icon box. The above rule matches the icons in the left box when the t attribute of the two icons are equal. These icons are then replaced with just the icon labeled `A` and the *value* attribute of the icon is incremented by one.

1.4 Grammars

A more formal approach to the specification of languages is grammars. There have been a number of visual language grammar formalisms which themselves may be textual, hybrid, or visual. Grammar formalism are favored for the reasons of semantic clarity and verifiability. On the down side they are often rather difficult to comprehend by the observer due to detail or cryptic notation. In this section we will survey some of these formalisms.

1.4.1 Attribute Grammars

For the textual case, the most well known environment generator is the Cornell Synthesizer Generator [64]. It uses attribute grammars to specify languages from which it generates syntax-directed editors. The attributes are used for type checking which is also incorporated into the editor supporting not only syntactically but also type correct program construction.

For visual languages also several attribute grammars have been proposed [5, 28, 50] where the definition of the graphical syntax is defined with spatial attributes. These grammar rules are themselves textual.

Visual Programmer's WorkBench(VPW) [67] is a collection of tools that synthesizes environments for the specification, analysis, and execution of visual programs. It generates visual programming environments for languages which are specified in terms of their syntax, abstract structure and static and dynamic semantics. The programming environment consists of a general purpose editor, a grammar driven spatial parser, and some other analyzers.

The syntax of the language is defined with Golin's *picture layout grammar* [28, 29]. The abstract structure is defined as an *object graph* and serves as the basis for the static and dynamic semantics. The object graph defines a set of relations specifying interactions among objects. The static semantics is defined by processing visual programs in terms of extracting, analyzing or synthesizing the static properties of a visual program. The dynamic semantics defines the execution properties such as interpretation, simulation, and dynamic verification. It is specified with *action routines* in terms of an external method. *Action equations* define relationships that must hold between objects and external methods.

A *Picture layout grammar* is an attributed multi-set grammar where the productions are picture composition operators. Each grammar symbol has attributes representing spatial information. The constraints represent relationships among the components and the semantic functions compute attributes for the aggregate object. A production is defined as follows:

$$
\begin{array}{l}
A \rightarrow \{B, C\} \\
\quad A.attr = func_{op}(B.attr, C.attr) \quad \#\text{semantic function} \\
\text{where:} \\
\quad pred_{op}(B.attr, C.attr) \quad \#\text{constraint}
\end{array}
$$

where $\{B, C\}$ is a multi-set, the semantic function describes the attribute transfers and the constraints describe the graphical layout between the production elements. VPW supports a predefined set of *production operators* that consist of a constraint and a semantic function. This permits the shorthand notation: $A \rightarrow op(B, C)$ which is equivalent to the above production.

A fragment of a grammar of a language called *PetriFSA* obtained from [67] is seen in Figure 1.3. The operators *set_of, touches, contains, tiling, points_to, points_from,* and *over* are grammar production operators of the picture layout grammar. The underlined non-terminals represent remote symbols which allow directed graphs to be described. A remote symbol is not considered to be part of the production but one that is defined somewhere else. It serves as a non-tree edge in the parse tree making the parse structure a directed graph (which are restricted to be acyclic). The attributed multi-set grammar rule for the *STATE* is shown in Figure 1.2.

STATE → STATE_INITIAL
    STATE.ID = STATE_INITIAL.ID
    STATE.LABEL = STATE_INITIAL.LABEL
    STATE.EVAL = STATE_INITIAL.EVAL

TRANSITION → tiling(TRANSITION_IN,TRANSITION_OUT)
    TRANSITION.ID = TRANSITION_IN.ID
    TRANSITION.LABEL = TRANSITION_IN.LABEL
    TRANSITION.trigger = TRANSITION_IN.trigger
    TRANSITION.action = TRANSITION_IN.action
    TRANSITION.from = TRANSITION_IN.from
    TRANSITION.to = TRANSITION_IN.to

Where:
    TRANSITION_IN.ID == TRANSITION_OUT.ID

Figure 1.2: Attributed multi-set grammar for STATE.

PFSA → set_of(STATES_AND_TRANSITIONS)
STATES_AND_TRANSITIONS → STATE | TRANSITIONS
STATE → STATE_INITIAL | STATE_FINAL | STATE_NORMAL
STATE_INITIAL → touches(CARAT,STATE_NORMAL)
STATE_FINAL → contains(circle,STATE_NORMAL)
STATE_NORMAL → touches(circle,string)
TRANSITION → tiling(TRANSITION_IN,TRANSITION_OUT)
TRANSITION_IN → points_to(TRANSITION_ARROW,<u>STATE</u>)
TRANSITION_OUT → points_from(<u>TRANSITION_ARROW,STATE</u>)
TRANSITION_ARROW → touches(TRANSITION_BOX_CONNECT,arrow)
TRANSITION_BOX_CONNECT → touches(line,TRANSITION_BOX)
TRANSITION_BOX → contains(box,TRIGGER_AND_ACTION)
TRIGGER_AND_ACTION → over(string,ACTION)
ACTION → over(line,STRING)

Figure 1.3: The grammar and corresponding attributed multi-set grammar of PetriFSA.

The parser *Vizier* [28] takes a picture created in a general purpose editor with a picture layout grammar and produces a parse graph corresponding to the syntactic structure of the input picture. The abstract structure is obtained from an *object-graph system* which defines all the object-graph types with definitions which maps each abstract structure to an object-graph type. VPW then uses Awk [1], definite clause grammars (DCG's) in Prolog [85], and synthesized attribute-evaluation [64] over the parse graph to yield an abstract representation. It is this abstract structure which is processed in various ways when defining the static and dynamic semantics of a language. The various tools that make up the VPW are integrated with a *Message Backplane* which supports distributed programming. The Message Backplane is based on the *Linda* [25] model and its implementation was derived from FIELD [60].

Bjorn Backlund et. al. [5, 4] 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 are used for specifying pictures and their constraints. Hagsand [34] extended this work by defining the dynamic semantics of visual languages using operational semantics.

1.4.2 Graph Grammars

Graph grammars have been proposed as a formalism for specifying visual languages due to their "natural" correspondence to graph representations. A good example of an interactive specification environment for visual languages is PROGRES [70, 71] which is a *multi-paradigm* language based on graph rewriting. It has a mixed textual and graphical representation, where various aspects of a language can be defined. Language specification consists of defining graph schemata, declaration of attributes, atomic graph rewrite rules, and the imperative programming of graph transformation rules. It uses a graphical syntax for defining the graph schemata, graph queries and atomic graph rewriting steps. It provides an editor and analyzer for checking static semantic errors of the PROGRES language.

With this style of specification it is argued that the most relevant aspects of a visual language are specified with a visual language — namely the graph rewrite and graph schemata. The specifier is aided with various analyzers to assist in yielding correct specifications. However, the specifier must switch from one context to another using different tools to specify different parts of a language. While the graphical editor for rewrite rules does show the relationships among language components it does not really bear any similarity to the end language. The syntax of the language is completely disjoint from the semantic specification.

Abstract syntax of the graphical part of hybrid languages can be specified by means of a PROGRES graph grammar [2]. Rekers and Schürr consider the definition of syntax of visual languages and the parsing of pictures according to such syntax definition [61]. Such specifications are written in a formalism that provides labeled boxes and labeled arrows to define the productions. Figure 1.4 shows a specification for *entity relationship diagrams.* The grey nodes are called *context nodes* which must exist on both sides of the production (e.g. they must be common). The grey boxes allow the definition of embedding rules. For example, the production number 2 allows the introduction of an entity, where the diagram remains connected. Graph grammar productions can get very involved and hard to read as they can be graphs represented with a great many boxes and arrows. Examples of such productions can be found in [61].

1.4.3 Algebraic Specifications

Algebraic language specifications have been around for a long time and promote the benefits of abstract data type specifications. They specify the functions using equations which may be conditional. Algebraic specifications declare a set of *sorts*, which are the types used in describing the language, and functions

$$f : s_1^a \times ... \times s_n^a \rightarrow s_1^b \times ... \times s_m^b$$

where each s_i^j is a sort and m is usually 1 and for constants n=0. A *term* is a syntactically correct function which has terms as arguments. A term may also be a variable.

Equations when oriented can be interpreted as rewrite rules. An equation:

$$f(t_1, ..., t_n) = g(t_{n+1}, ..., t_m)$$

means to rewrite a term that matches the pattern of f with the function g. Matching involves the matching of tree structures representing the abstract syntax of a term.

Conditional equations are used to specify language semantics and occur very frequently in realistic specifications. *Conditional rewrite rules* [8, 43, 17] are used to execute conditional equations.

A conditional rewrite rule:

$$\frac{s_1 = t_1, \cdots, s_n = t_n}{s_0 = t_0}$$

with $n \geq 0$, and s_i, t_i $(0 \leq i \leq n)$ terms. There are usually well-definedness constraints imposed on the variables of the conditions [81, p.16] in order to ensure their definedness during execution.

The VAS formalism and the VASE environment, which will be discussed in Chapter 5, find their roots in the algebraic specification formalism ASF+SDF. For

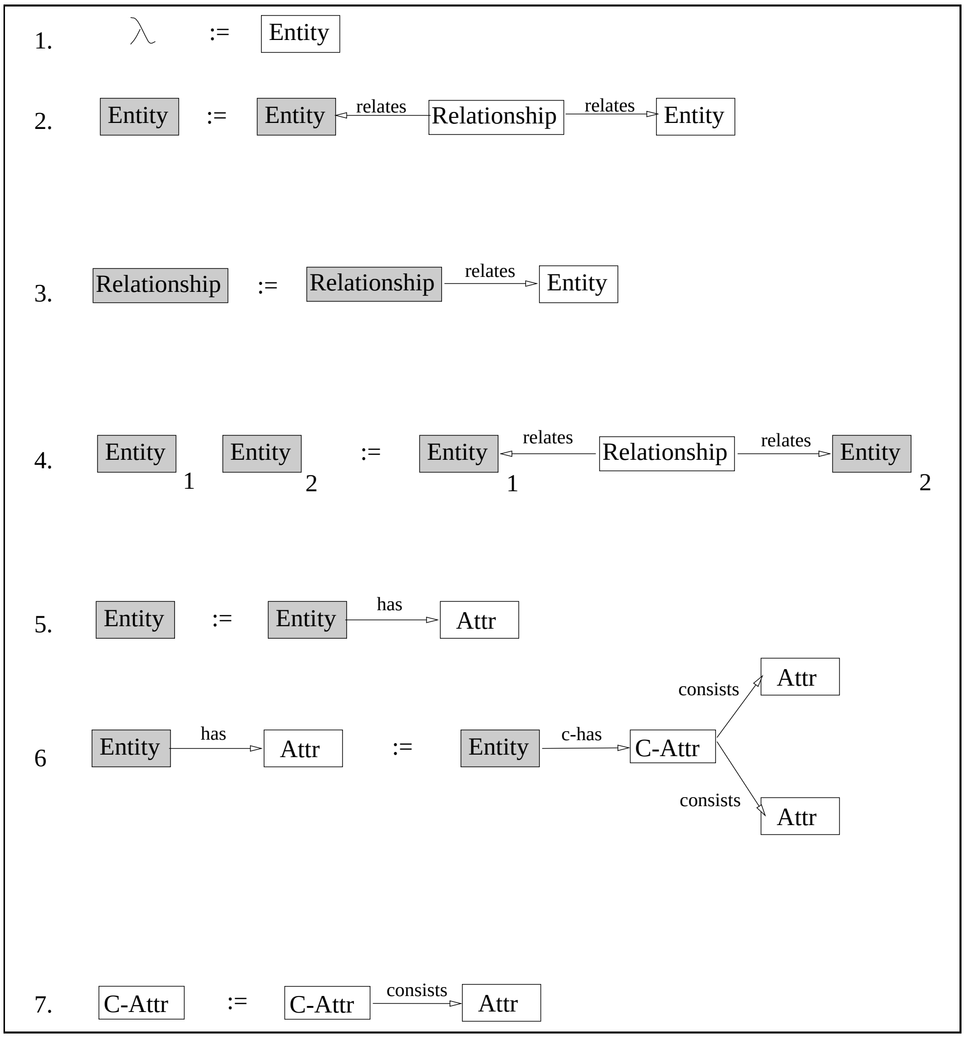


Figure 1.4: A PROGRES specification for entity relationship diagrams.

this purpose this formalism is explained in greater detail especially since we rely on the ASF for the underlying representation and for the term rewriting machinery.

The ASF+SDF formalism and Meta-Environment

The ASF+SDF formalism [42] is a many-sorted algebraic specification formalism for textual languages. The ASF+SDF Meta-Environment supports the modular interactive specification of languages and the generation of programming environments for the specified languages. We find algebraic specifications attractive in that they allow the definition of equational semantics which lend themselves to analysis, verification, and the semi-automatic and automatic generation of tools based on specifications. Tools that can be generated are among others editors, term rewriters (for executable specifications), pretty printers, and type-checkers.

The ASF+SDF Meta-Environment is well suited for specifying arbitrary abstract data types (traditional algebraic specifications), as well as the definition of any (formal) language. The specification environment is based on the ASF+SDF formalism which combines the **A**lgebraic **S**pecification **F**ormalism (ASF) with the **S**yntax **D**efinition **F**ormalism (SDF). This environment, called the ASF+SDF Meta-Environment, is an interactive environment that supports the specification of languages. The Meta–Environment provides *module editors* for the development of modular specifications. Each module editor consists of two parts: one for the syntax and one for the semantics.

The syntax definition formalism (SDF) supports the definition of *free syntax* which allows specifiers the freedom to choose any syntax for their language. Thus, the same program can be represented in a variety of ways (see Section 1.4.3).

The semantics of a language is specified with conditional equations (both positive and negative) over terms defined in the syntax of the language. Furthermore, these equations are implemented as a term rewriting system yielding executable specifications.

Given a specification, the system automatically generates a parser and a syntax-directed editor for that language. The equations are always type correct as they must conform to the specified syntax. The editor enables the specifier to immediately write terms of the specified language. The terms are then parsed with the generated parser. Finally, these terms can be executed according to the specified semantics.

The ASF+SDF Meta-Environment is an incremental development environment. When a specification is modified, its associated tools are automatically adapted rather than regenerated from scratch, saving significant regeneration time that would otherwise be required after each modification. This makes interactive development of specifications, that usually involves a lot of modification, feasible in practice.

The SDF formalism For the case of textual languages, the ASF+SDF Meta-Environment has demonstrated that the use of concrete syntax for defining an algebraic specification of a language – both the signature and the semantics, is not only feasible but also desirable as it enhances the comprehensibility of the specification. The free syntax is particularly useful in specifying a language when following a written specification as the specification can be written in a similar form. SDF [35] is the Syntax Definition Formalism used for defining "grammars" for context-free languages. The syntax definition is utilized in generating a term editor for writing terms over the language and an equation editor for defining the semantics of the language.

The language syntax is defined by a signature consisting of syntax rules of the form:

$$\alpha \quad \rightarrow \quad S$$

where α is either a sequence of any combination of sorts and literals or a literal followed by parenthesized sequence of sort and S is a sort (non-terminal) of the language. We can define the syntax for a function called *plus* as follows:

$$\mathrm{plus}(N,N) \qquad\qquad \rightarrow \quad N$$

which resembles abstract syntax. In this rule N is a sort and "plus()" is the pre-defined notion of a function in SDF. We could also have defined the plus function as:

$$N \text{ ``+'' } N \qquad\qquad \rightarrow \quad N$$

where N is a sort name and "+" is concrete syntax representing the plus function. In the generated term editor, the user may enter the term: 4+5.

Semantic equations use the concrete syntax of the languages which makes the association between the syntax and semantics visible to the specifier who is not forced to make a mental translation into some abstract representation. Equations are written over terms of equal sorts. The following equation can be one of the several equations required to define the "+" function:

$$[1] \quad Nat + 0 = Nat$$

where Nat is a variable of sort N. In the above equation [1] is a label used for identification purposes.

SDF also has a built in notation for associative lists having the syntax:

$$\{\ SORT\ [\ \text{“}\ SEPARATOR\ \text{”}\ ]\ \}\ \ * \mid +$$

defining list of zero or more items with an optional separator. We can define, for instance, a sort *SET* as a comma separated list of integers:

$$\text{“\{ ” } \{N \text{ “,”}\}* \text{ “\}”} \qquad\qquad \longrightarrow \text{SET}$$

which defines a “,” separated list of items of sort *N*. Here we also see the “{” and “}” as part of the syntax definition which defines the bracket as part of the *SET* syntax. The syntax for a union function of the sort *SET* can be defined as:

$$\text{SET “}\cup\text{” SET} \qquad\qquad \longrightarrow \text{SET}$$

allowing the term: { 3,2,7,6 } ∪ { 6,1,3,9 } to be constructed. Finally, we can write an equation defining ∪ that simply adjoins them into a single list representation and another rule that defines the equality of sets in the presence of multiple items which together result in a set without duplications in a normal form:

$$[union]\ \ \{Nat_1*\}\cup\{Nat_2*\} = \{Nat_1*, Nat_2*\}$$

$$[dupl]\ \ \{Nat_1*, Nat, Nat_2*, Nat, Nat_3*\} = \{Nat_1*, Nat, Nat_2*, Nat_3*\}$$

where Nat_1* and Nat_2* are variables over the “list of Nat” sort and Nat is of sort N.

Example: Binary Trees In order to give a flavor of the specification environment, let us consider a specification for binary trees. To specify this in ASF+SDF we need to define two parts: the syntax and the semantics. Figure 1.5 shows a module specifying the modules for the binary tree language (modules *BTree* and *RLinear-BTree*).

1. The top part defines the syntax and consists of the definitions of new sorts, lexical and context–free syntax and variables that are used in defining the semantics. This section also defines the modules to be imported for the specification (in this case the module *Integers*) which specified the syntax and semantics of integers). The binary trees use the *INT* from this module to represent the leaf nodes. It is the import facility which enables the modularization of specifications.

module: BTree
imports Integers
exports
  sorts Leaf Node
  context-free syntax
    INT → Leaf
    Leaf → Node
    node(Node, Node) → Node

Module: RLinear-BTree
imports BTree
exports
  context-free syntax
    rl(Node) → Node
  variables
    Lf → Leaf
    Nd [*1-3*]∗ → Node
equations

$$\mathrm{rl}(\mathrm{node}(\mathrm{node}(Nd_1,\ Nd_2),\ Nd_3)) = \mathrm{rl}(\mathrm{node}(Nd_1,\ \mathrm{node}(Nd_2,\ Nd_3))) \quad [1]$$

$$\mathrm{rl}(\mathrm{node}(Lf,\ Nd)) = \mathrm{node}(Lf,\ \mathrm{rl}(Nd)) \quad [2]$$

$$\mathrm{rl}(Lf) = Lf \quad [3]$$

Figure 1.5: The modules containing the specification for the BTree language.

2. The lower half defines the semantics by defining equations over the syntax. The module *BTree* has none and the module *RLinear-BTree* has three equations defining the *rl* function which transforms a binary tree into another one that is right linear. A right linear binary tree has only leaf nodes in the left branches of each sub-tree.

After writing the specification one can immediately test it by executing a term over the specification. The term

$$\mathrm{node}(\mathrm{node}(1,\ \mathrm{node}(2,\ 3)),\ \mathrm{node}(\mathrm{node}(4,\ 5),\ \mathrm{node}(7,\ 3)))$$

is entered in the editor. The buttons on the editor allow the user to create a LaTeX representation of the term, or to apply the *rl* function to the term. A button called "*right-linear*" is bound to the *rl(N)* function defined in *RLinearBTree* module. The result of applying this function to the term is:

$$\text{node}(1, \text{node}(2, \text{node}(3, \text{node}(4, \text{node}(5, \text{node}(7, 3))))))$$

This style of specification allows the concentration to be on the language specific issues. The syntax and semantics are defined within one formalism making the entire specification accessible for understanding, maintaining, and testing. The semantic definition does not rely on external functions defined in some other language. While the equations provide a nice declarative manner of expressing the functions, they may still not be so easy to understand. Some languages benefit greatly from a visual representation. In Chapter 2 we will reexamine the right linear binary tree again with a visual syntax.

The process of developing language specifications is a tedious task which can be significantly aided with interactive tools facilitating language development and testing. The executability of the specifications seems to be the crucial factor for realizing the utility of specifications for prototyping languages and their environments.

1.4.4 VASE

The *Visual Algebraic Specification Environment* (VASE) is an interactive visual specification environment for visual languages and is based on the *Visual Algebraic Specification* formalism (VAS) and both are the main topic of this thesis. VASE is defined with the express intention of providing a specifier-friendly specification environment. For achieving this, it is our belief that interactive tools as well as the use of concrete syntax in specifying static and dynamic semantics as well as having executable specifications are very useful. The VASE environment has been based on an uniform formalism for the specification visual languages and the generation of environments for them. In our formalism we have separated the specification of the semantics of the language definition from the visual syntax.

Rekers in [3] advocates the utility of a clear separation of graphical lexical description from syntax definition. This is a confirmation that our approach is indeed desirable. But they do not use concrete pictures in the specification and thus it would still involve specifying attribute manipulation explicitly.

The style of the lexical syntax definition is influenced by Helm and Marriott [36] who define *picture*s as being either primitive or complex. Complex pictures are specified in terms of their sub–pictures and a set of constraints. They provide a rigorous declarative semantics both for the specification and the recognition of pictures. We have limited our work to the specification of pictures and define the picture specification language VODL (Chapter 4) and how it may be used in specifying visual languages and in end-user environments.

The work on VASE enhances this by allowing one to build binary trees with visual representations that reflect the tree structure.

1.5 Organization of Thesis

The remainder of the thesis is organized as follows: Chapter 2 provides an overview of the VAS specification formalism by use of an example. Chapter 3 discusses editor generation which forms the basis of all interactive tools described in this thesis. Chapter 4 introduces a picture definition language called VODL which is the foundation for describing all "visual" elements of both end-user and specification languages. Chapter 5 describes a "visual" visual algebraic specification language called VAS formalism and a framework for an interactive specification environment for it called VASE. Chapter 6 introduces a technique called "Share-Where maintenance" for maintaining the information regarding shared sub-terms that are introduced by editors and how this technique is used in pretty printing rewritten terms. Chapter 7 describes an extension to the VAS formalism for specifying input and output behavior during execution. Chapter 8 describes our various implementation experiments. Finally, in Chapter 9 we draw our conclusions and suggest possible future directions.

Several chapters in this thesis are revised versions of articles that have appeared elsewhere. Most revisions made were to cut out some duplication of introductory material and provide continuity. The chapters 3, 4, and 5 have been published in the proceedings of the *Symposium of Visual Languages* in [75], [76] and [77] respectively. Part of Chapter 4 was also discussed in the *Eurographics workshop on Programming Paradigms in Graphics* [20]. Ideas in Chapter 7 were presented both at the *Workshop on the Theory of Visual Languages* [21] and the ERCIM Workshop on user Interfaces for All [19]. A version of Chapter 6 will appear in a book that forms the collection of articles that originate from *Workshop on the Theory of Visual Languages, 1996.*

Chapter 2

Visual Formalisms

This thesis concerns itself both with the specification of visual languages as well as the generation of end-user environments for them. Similar to the arguments that favor programming environments for programmers we believe that language specifiers need specification environments. This is perhaps more relevant with the increase of special purpose language. Having tools for the generation of language-specific environments, at least for prototyping, is very useful.

We propose that the specifier as well as the visual language researcher need interactive tools which allow them to specify and interact with visual languages. In this line we propose a number of tools that support the definition of graphical lexicals, visual syntax, syntax-directed visual editors, and visual term rewriters. We believe that there is much to be gained by providing visual support for the definition of visual languages. The visual representations both at the specification level and at the end-user programming level allow the user of such tools to remain closer to the language of interest.

This thesis focuses on visual environments which are based on the abstract syntax of visual languages but support the definition at a concrete syntax level. Since the syntax of visual languages is often semantically relevant, it is important to have tools that provide access to their appearances. The concrete notation allows the specification to be made at a level close to the language of interest, facilitating comprehensibility, and the abstract syntax, hidden from the user, represents the interpretation which is used by the underlying tools.

The specification formalism promotes two main themes: the use of visual syntax and the use of concrete syntax in language specification. In the next three sections we motivate these themes by first examining the choice of visual syntax and in the following two sections the use of language-specific syntax in program specifications and language specifications respectively.

2.1 Why a *Visual* Formalism

The visual language users as well as specifiers need tools to assist them in their tasks. Much of the research regarding visual languages has focused on end user issues (language and/or environments). However, there has been little effort in providing support for the language designer, which is very important for language prototyping.

Visual notation is much more complicated both in its underlying representation as well as its concrete notation. In Section 1.2 various specification formalisms were discussed. These were mostly textual formalisms which prevent the specifier from being directly in touch with the syntax of the language. Although these grammar rules yield the desired results and can be utilized in generating tools for the end user, for the language specifier there is no view of graphical notation [28, 86]. In many cases the syntactic representations chosen are very relevant. This is particularly true for special-purpose languages as they typically relate to some real-world entities or some conventionally accepted notation in that problem domain. Having the visual notation accessible during the specification process would be valuable since it would facilitate the recognition of the language elements.

In our approach, we view the language designer also as a user who requires tool support. Visual notation is often different from textual notation since the physical attributes and spatial relations often are semantically significant.

Notational freedom is a powerful tool for expressing problems and solutions. This is utilized by mathematicians who define their notation in a domain and express their definitions, problems and solutions using that notation.

Still we must assure that we do not loose the proven benefits of abstract representations. While visual notation frequently lends itself to ambiguous representations, the meaning of the specification must be unambiguous – there must be a single interpretation of a specification or program. This is usually achieved with a parser or constructing the program in a syntax directed manner. The interpretation relies on the syntax definition, where all concrete representations are removed. We chose to construct the program in a syntax directed manner to avoid dealing with parsing issues.

2.2 Language Specific Notation in Programming

Visual rules allow the reader to comprehend the semantics of languages or programs by bringing the visual representations to the foreground. For example, AgentBuilder [65] is a tool developed for defining agent rules for Agentsheets[1]. This tool provides support for defining the actions for the icons of an Agentsheet. These rules define the actions which define what happens to depictions of agents

[1]Agentsheets is a tool for generating iconic programming environments based on a spatio-temporal metaphor of communicating agents [62].

Figure 2.1: Part of a graphical rule as would be entered in Agent Builder.

under certain conditions. The actions applied to depictions are movements in different directions.

Figure 2.1 shows a part of a rule for KidSim [72] in Agent Builder [65].

In the rule  denotes a depiction of an agent,  denotes a query which is used in conjunction with a depiction to query the depiction of an agent, the  denotes a direction modifier to query the icon below the one in consideration, and  denotes a downward movement. The rule uses these language constructs along with the icons  (for a gorilla) and  (for empty space) to state that the gorilla is to move down when it is above empty space. More specifically, the rule states that if the icon that is being queried is a depiction of a gorilla (first three symbols), and the icon below it is a depiction of empty space, then the gorilla should move down. Without the Agent Builder tool these rules would be encoded in common lisp, which is generated by this tool.

Granted that users must learn some visual syntax, this syntax is closer to the domain of interest. The icons of interest are presented in the same manner as in the execution of the program, for example a gorilla icon is used rather than an icon name. This makes the connection between the definition and the execution of the program more obvious by reducing the amount of translation needed between the problem being defined and the language in which to express this problem.

2.3 Language Specific Notation in Specification

The AgentBuilder tool demonstrates the support offered for the programmer by using graphical rules. It is a tool defined for a specific language. In the VAS formalism we define a specification language for defining visual language syntax which is then used to define the semantics of that language. Thus, the language-specific notation is used in semantic definitions as well, but in this case the method is generic – i.e. applicable to all languages specified with the VAS formalism.

Language specifications define the properties of languages by means of some formalism so that they can be analyzed and used for generating tools such as editors, parsers, and evaluators. The advantages of abstracting away from concrete properties of languages are to have one formalism capable of representing many languages with an uniform abstract representation. While abstract representations are well suited for capturing the language characteristics, they are usually too difficult for a user to comprehend and maintain.

The cryptic nature of abstract representations makes it difficult for the specifier to relate the specification to the real representation (concrete syntax) of the language. Concrete syntax is used precisely to alleviate this problem by introducing non essential syntax that provides some cues and/or context for understanding. The more concrete syntax the language uses the greater the gap between its concrete and abstract representation. In the case of visual languages, the gap is typically larger due to high dependence on graphical and spatial representation which is lost in an abstract representation.

In order to reduce this gap we promote the use of concrete syntax of the language being specified during language semantic specification. This allows the language specification to be much more accessible to unsophisticated users (specifiers). The success of the use of concrete syntax in the ASF+SDF Meta-Environment is attributed to the fact that the language specifier is much more comfortable in dealing with a syntax that represents the language of interest. While the specification is developed using the concrete syntax the underlying abstract syntax is constructed automatically, which distills the term representation from irrelevant concrete information leaving semantically relevant properties which generic tools are based on.

2.4 Visual Algebraic Specifications

The Visual Algebraic Specification (VAS) formalism is inspired by ASF+SDF. VASE is the proposed specification environment for language specification. It consists of editors for defining lexicals, language syntax and semantics, and an editor generator. In this section, we give a very small specification of a visual language for binary trees. Languages are specified in modules that define some syntax and/or some semantic equations. The syntax is defined using the VAS formalism which is covered in Chapter 5. The VAS formalism requires the definition of visual lexicals which are defined using the picture specification language VODL which is explained in Chapter 4.

2.4.1 Visual Lexicals

Visual Object Definitions(*vod*s) define the lexical representations of a language. They are defined with the picture definition language VODL which is a constraint-based declarative picture specification language supporting user-defined types.

Graphical lexicals are defined by *vod*s which consist of sub-*vod*s and their spatial relationships. Each *vod* may also have a set of attributes defined. The idea is to set the values that are relevant and leave the others to be determined by the system (by constraint satisfaction or default values for unconstrained attributes). We need two classes of values: "don't care" values (user preference or default) and semantically relevant values. Attributes that are set by preference ($=_p$) or default ($=_d$) are needed for rendering. Attributes that are set as relevant ($=_r$) are considered as significant and are reflected in the abstract syntax.

When defining new *vod*s the specifier can compose a new type from previously defined *vod*s and define only the significant attributes. Leaving irrelevant attributes undefined results in default values to be used in initial presentation. Since, at the time of rendering, some basic values must be known (such as position, width, and height) default values are used. As these values are not semantically relevant it is important that they are not reflected in the abstract syntax. Considering the KidSim example, we might want to set apart the case when the gorilla is red (indicating super-powerful) in which case it can walk through obstacles which if encountered otherwise would cause it to turn around. In this case the color is relevant and must be reflected in the definition as such.

Most *vod*s are under-specified, rather under-constrained, leaving a great many concrete presentations that are consistent with their specification. Thus, when we show some presentation, it is almost certainly one of many possible acceptable presentations. The interactive tools deal with representative-*vod*s which are *vod*s that are consistent with their definitions. Typically there are numerous *vod* instances that are consistent with *vod*s. Users may manipulate interactively these representations as long as the manipulation does not violate the constraints in the definition – in which case the interaction is not permitted. This approach allows for a single *vod* definition to cover a large set of graphical descriptions.

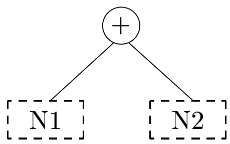
For the binary tree we define two new *vod*s using VODL (Chapter 4). The first *vod* represents the "plus" symbol. This *vod* is named *Plus* and is a composite *vod* consisting of two sub-*vod*s.

```
defv Plus ( )
  { c : circle ( ),
    l : text ( ) ⊕ [ strval = "+" ] }
  ◁
  { c contains l }
```

A representative appearance for this definition could be: $\oplus$. A *vod* for defining a node plus can be defined using the just defined *Plus vod*.

```
defv Node ( N1, N2 )
  { n1 : N1,
    n2 : N2,
    p : Plus ( ),
    c1 : Connector ( p, n1 ),
    c2 : Connector ( p, n2 ) }
  ◁
  { p over n1,
    p over n2 }
```

Node is a composite *vod* with two parameters and five sub-*vod*s. The first two sub-*vod*s correspond to the parameters and the last three to previously defined *vod*s. *Connector* is a *vod* with two parameters that defines a line that touches the two parameters. The following picture is a representative *vod* where the parameters are represented with dashed boxes containing the parameter names.



2.4.2 Syntax Definition

The syntax for binary trees is defined using the visual lexicals and the VAS formalism which is directly analogous to the SDF formalism explained in Section 1.4.3. The "Integers" module which defines integer numbers is imported. Two sorts named L and N representing leaf and a node are introduced. Two variables of each of these sorts which are coincidentally represented with the same name as the sorts are also introduced. These variables will be used when defining semantic equations in the next section.

module BTree

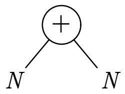
imports Integers

sorts L N

functions

INT $\rightarrow$ L

L $\rightarrow$ N



$\rightarrow$ N

2.4.3 Semantic Definition

In our work we are interested in graphical equations which define language semantics. These equations, when oriented (left to right), are considered as graphical rewrite rules. Graphical rewriting rules have been used to define iconic language semantics in [24, 51]. We consider the rewrite rules for the VAS formalism which is parameterized with a particular language specification. Semantic equations use the context-free syntax of a langauge. Thus, rewrite rules may be written only for specified languages.

The semantics are defined using conditional equations. The equations use the syntax specified for the language. Such syntax includes the syntax introduced in the same module as well as any imported module. Here, we define some semantic equations that produce right-linear binary trees. To do this first we introduce a new function for the right-linearalization function which uses the function symbol ${}^{\wedge}\!\lambda$, takes an argument of sort "N" and returns a results of sort "N".

module RLinear-BTree

imports BTree

sorts

functions

$${}^{\wedge}\!\lambda\,(N) \quad \rightarrow \quad N$$

variables

$$\mathcal{L} \quad \rightarrow \quad L$$

$$\mathcal{N} \quad \rightarrow \quad N$$

equations

$$[1] \quad {}^{\wedge}\!\lambda\left(\begin{array}{ccccc} &&&\oplus&\\ &&\diagup&&\diagdown\\ &\oplus&&&\mathcal{N}_3\\ \diagup&&\diagdown&&\\ \mathcal{N}_1&&\mathcal{N}_2&& \end{array}\right) = {}^{\wedge}\!\lambda\left(\begin{array}{ccccc} &\oplus&&&\\ \diagup&&\diagdown&&\\ \mathcal{N}_1&&&\oplus&\\ &&\diagup&&\diagdown\\ &&\mathcal{N}_2&&\mathcal{N}_3 \end{array}\right)$$

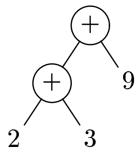
$$[2] \quad {}^{\wedge}\!\lambda\left(\begin{array}{ccc} &\oplus&\\ \diagup&&\diagdown\\ \mathcal{L}&&\mathcal{N} \end{array}\right) = \begin{array}{ccc} &\oplus&\\ \diagup&&\diagdown\\ \mathcal{L}&&{}^{\wedge}\!\lambda\,(\mathcal{N}) \end{array}$$

$$[3] \quad {}^{\wedge}\!\lambda\,(\mathcal{L}) = \mathcal{L}$$

One can observe that the use of tree representation in the equation visually reflects how the tree is manipulated to get a right linear tree. When a term of the language is constructed, it will be matched against the set of rewrite rules defined for the language. The matching policy is with respect to the abstract syntax of the language. It is possible that a term matches several rules, in which case the most specific one is chosen. The specificity of the rules are determined by the number of variables present in a rule.

2.4.4 Term editor

Given the language specification, domain specific tools are generated – some of which are interactive tools using the syntax of the language. The major tool being an editor which the end-user uses for constructing programs (terms over the syntax). These programs can be evaluated (rewritten) by the term rewriter. The abstract representation of the syntax definition allows the terms in the editor to match the terms in the equation even though their physical appearances are not identical. Figure 2.2 shows an instance of a *BTree term editor* while constructing the term:



Terms are constructed in a syntax directed manner where the focus of a term is replaced with a set of permissible replacements presented in the *selection panel.* The focus is indicated with a surrounding dashed box and the selection panel is the left panel of the editor. The term shown in the editor is a normal term conforming to an underlying tree representation.

Frequently, a strict tree representation is not sufficient and we need a graph-like structure to represent multi-dimensional relationships. For this, the term editors allow the sharing of sub-terms when the sorts are appropriate. For example, a binary tree sharing the same leaf can be constructed as:

(a) (b)

where ⟨N⟩ is replaced with the "2" present in the term instead of making a selection from the selection panel.

2.5 A comment on structured editors

We, thus, start our journey with the consideration of obtaining editors for visual languages. Editors allow the construction of specifications/programs and allow interaction with them. They can be used for presenting animations, presenting results of computations or analysis, or for dialogs for obtaining and presenting

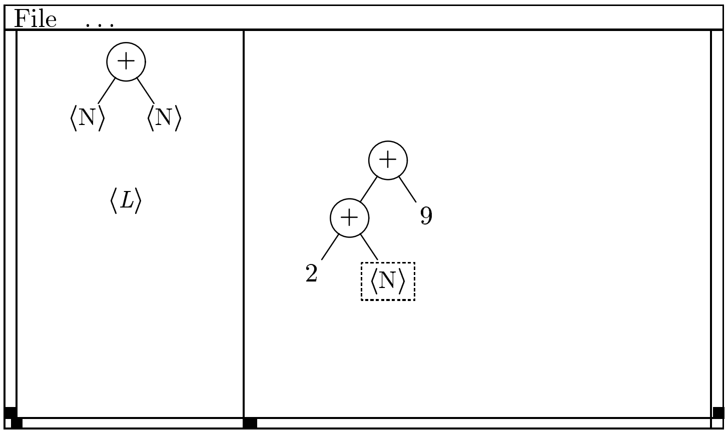


Figure 2.2: An instance of the BTree term editor.

input and output.

In this work, we regard the existence of structured visual editors as the basis of the interactive tools that make up the end-user as well as the specification environment. By choosing structured editors we have quite intentionally tried to set aside any parsing issues of visual languages. This decision has no bearing in any manner on the desirability or appropriateness of parsers[2]. It simply reflects the main focus of this research: kinds of tools that can be generated from specifications and the support provided for creating these specifications. By describing some kind of simple editor we can at least be assured that we can construct the intended terms in some manner. One can imagine that one of the desirable tools to generate would be a parser for the specified languages. However, this is not in the scope of this research.

[2]Visual parsing is an active research area [86, 28, 61]; however we feel that the results are not yet satisfactory enough for us to depend on them.

Chapter 3

Generating Visual Editors

3.1 Introduction

There are an increasing number of visual programming environments for a great many languages [26]. 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 aid the software development process for visual languages. There are many programming environments for textual languages and several systems that can generate such environments given their specifications [63, 42]. We are interested in the *generation* of visual environments for visual languages. We concentrate on the specification of visual languages and the generation of language specific visual editors for them.

This chapter covers two themes which the rest of the work relies on: editor generation and the specification of visual notation. Editors are what form the interactive environment for both the language-specific editors as well as the development environment. This chapter addresses only language specific editors. In subsequent chapters we will look at editors for developing the specifications themselves – which can also be considered to be language specific where the language of interest is the specification language.

The construction of visual editors is typically addressed in two ways: (1) structure-oriented (syntax-directed) visual editors, and (2) general–purpose visual editors. While the first approach considerably restricts the user freedom, the second approach requires visual parsing. Ideally, we would prefer hybrid editors that support both structured and free form editing. We chose to develop structure–oriented visual editors, not only because of the difficulties related to visual parsing, but also since our concentration is on generating language specific components.

All the editors in consideration are visual editors. This makes it necessary for us to be able to define visual notations. For this, we introduce a picture definition language called VODL (visual object definition language) which will be explained in detail in Chapter 4. In this chapter, we examine how visual notation definitions can be used in conjunction with textual language definitions for obtaining visual editors. We start from textual languages since we are interested in extending such an existing textual specification environment and the approach given in this chapter is the first step towards that end. This is basically done by mapping the language constructs onto *vod*s (visual object definitions). This approach results in modifying the appearance of the language constructs during visual sentence construction. In chapters 4 and 5 we will describe how visual language syntax can be defined directly without any such mappings.

3.2 Overview of the Approach

We approach visual syntax specification and visual editors from the programming environments 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 users with an environment that supports the software development process for a given language. Programming environment generators are capable of deriving 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 aspects of a language such as type-checking and pretty–printing can be specified. One such aspect, 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. Visual notation definition is done with VODL (Visual Object Definition Language) which is eventually reflected in the visual syntax of the editor. Editor behavior is generated from the syntax definition of the language of interest which is mapped onto the defined visual descriptions.

Figure 3.1 shows the overall approach to generating structure–oriented visual editors. We start with a formal specification of a language $\mathcal{L}$. Then a visual syntax for $\mathcal{L}$ ($V_{\mathcal{L}}$) is specified. At this stage the formal specification of $\mathcal{L}$ 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 the language–specific information necessary for the implementation of the user–interface for the visual editor. Finally, the graphical user–interface for the editor is constructed – using the editor description.

We can identify three major phases in the editor generation process, of which the first two are formally specified and the last is based on some graphical platform. The nature of each phase is somewhat different. The first phase involves

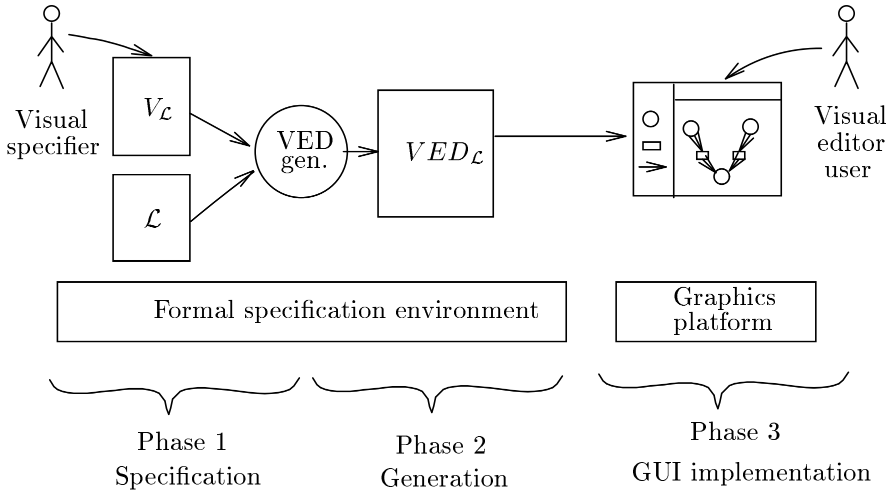


Figure 3.1: The overall view of a structured visual editor generation. $\mathcal{L}$ denotes the specification of a language, $V_{\mathcal{L}}$ is the visual specification (in VODL) for that language and $\mathrm{VED}_{\mathcal{L}}$ is the visual editor description.

the specification of $\mathcal{L}$. 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, implements the graphical user–interface, using the visual editor description for the language–specific parts, over the desired graphics platform.

The emphasis of this work is on the specification of the information relevant to visual syntax and visual editing. 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.

3.3 Formal specification of a language

The aim of the formal language specification in our work is to describe the syntax and semantics of languages. The semantics can be defined in a denotational, operational, or any other style so long as certain functions over the abstract syntax trees are defined, specifying, e.g., statement execution, function evaluation, and so on, or even program analysis (e.g. type checking). We use the ASF+SDF specification formalism to explore this problem.

Let us consider a very simple language of expressions that supports two operations: division and addition. The syntax of this language can be defined as shown in Figure 3.2. This specifies a syntax for expressions with the operations $plus$, div, and $eval$. The expressions are over the digits $0-4$ with the operations

imports Layout
exports
sorts
EXP DIGIT
context-free syntax

plus(*EXP*,*EXP*)	$\rightarrow$	*EXP*
div(*EXP*,*EXP*)	$\rightarrow$	*EXP*
DIGIT	$\rightarrow$	*EXP*
eval(*EXP*)	$\rightarrow$	*EXP*
0	$\rightarrow$	*DIGIT*
1	$\rightarrow$	*DIGIT*
2	$\rightarrow$	*DIGIT*
3	$\rightarrow$	*DIGIT*
4	$\rightarrow$	*DIGIT*

Figure 3.2: The syntax definition module for *EXP*.

being modulo 5. This specification imports the definition of layout characters[1] from the *Layout* module. According to this syntax we can create sentences like:

$$plus(1, div(1,3)) \quad \text{and} \quad div(plus(1,1),3)$$

This language will be our running example throughout this chapter.

For these expressions a two-dimensional representation such as the following is frequently used:

$$1 + \frac{1}{3} \quad \text{and} \quad \frac{1+1}{3}$$

3.4 Visual Object Definition Language (VODL)

In order to obtain two dimensional notation, we first need a language for specifying the desired visual notation. Visual Object Definition Language (VODL) is a constraint-based declarative specification language which we use to describe visual appearances of syntactic constructs of a language. In this chapter we only provide an outline of VODL to give a flavor of the language to help in the understanding of the examples covered here. The details of VODL are presented in Chapter 4. VODL specifies visual notation in terms of:

- primitive visual object definitions
- composite visual object definitions

[1] Characters that are not part of the language such as white space and comments.

- spatial relations 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 less complex *vods*. The spatial relationships define the spatial constraints that exist among the sub-*vods*.

3.4.1 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. For example, a rectangle having the width of 50 can be defined as:

$$rectangle() \oplus [\text{width} = 50]$$

where $\oplus$ $[width = 50]$ defines the value of the width attribute of the rectangle. Only the desired attribute values need be specified. For unspecified variables default values are used.

3.4.2 Composite visual object definitions

Composite *vods* consist of lesser complex *vods* and are defined with two sets: a *vod* set and a spatial relations set. The *vod* set defines the *vods* that make up the composite *vod* and the spatial relations set provides the spatial relationships among those *vods*.

3.4.3 Spatial Relations

The 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 [28]. There are some basic spatial relationships such as *touches, left-of, over, contains,* and *followed-by.*

References to *vods* (*vod*-refs) are used to refer to *vods* in the spatial relations set. The *vod* set and the relations set are separate so that multiple relations between *vods* can be defined as is the case in the example in Section 3.4.4. We also use the reference names when discussing *vods*. For example a spatial relation could be:

$$aRect \ \textbf{left-of} \ aText$$

where $aRect$ and $aText$ are *vod*-refs and would constrain the rectangle to the left of the text, such as: Hello World!!!

$$
\begin{array}{lll}
\nu(plus(Exp_1, Exp_2)) = & \{ & e1 : \nu(Exp_1) \\
 & & op : text() \oplus [\textbf{string}=\text{“+”}] \\
 & & e2 : \nu(Exp_2)\} \\
 & \lhd \{ & e1 \ \textbf{followed-by} \ op \\
 & & op \ \textbf{followed-by} \ e2\} \\
\nu(div(Exp_1, Exp_2)) = & \{ & e1 : \nu(Exp_1), \\
 & & op : line() \oplus [\textbf{width}= \max(\text{e1}.\textbf{width},\text{e2}.\textbf{width})], \\
 & & e2 : \nu(Exp_2), \} \\
 & \lhd \{ & e1 \ \textbf{over} \ op, \\
 & & op \ \textbf{over} \ e2\}
\end{array}
$$

$$
\begin{array}{ll}
\nu(0) = & text() \oplus [\textbf{string}=\text{“0”}] \\
\nu(1) = & text() \oplus [\textbf{string}=\text{“1”}] \\
\nu(2) = & text() \oplus [\textbf{string}=\text{“2”}] \\
\nu(3) = & text() \oplus [\textbf{string}=\text{“3”}] \\
\nu(4) = & text() \oplus [\textbf{string}=\text{“4”}]
\end{array}
$$

$$
\begin{array}{lll}
\nu(eval(Exp)) = & \{ & e : \nu(Exp), \\
 & & op : \textbf{rectangle}()\} \\
 & \lhd \{ & op \ \textbf{contains} \ e\}
\end{array}
$$

Figure 3.3: Visual mappings for the expression language.

3.4.4 Example: A visual expression

We define the *visual syntax* of a language $\mathcal{L}$ with a function ν, which maps the abstract syntax of $\mathcal{L}$ to the domain of *vod*s's. ν is defined by specifying a mapping for every syntactic sort in $\mathcal{L}$. Figure 3.3 shows the visual mappings giving a simple visual syntax for expressions where Exp_1, Exp_2 are variables of sort *EXP* and $Digit$ is a variable of sort *DIGIT*.

The following visual expression is a valid expression according to this mapping:

$$\boxed{\frac{1 + 0 + 3}{2 + \frac{4}{3}}}$$

It is important to realize that most visual representations shown here are only representative ones since there is no unique representation corresponding to the *vod* definitions as they are generally under-constrained.

When the underlying representation of the picture is a tree (like above) the visual specification is simple, since the *vod*s are local to the equations.

3.5 Editor Definition Language

Given a specification of a language, and its corresponding visual syntax description, an editor for constructing visual sentences of that language can be generated. The generated part is actually an editor definition which we call a structured visual editor (SVE) definition for a given language.

A SVE definition is defined in terms of its three major aspects:

- visual syntax
- replacement behavior
- functions over programs

The description of a SVE is a combination of these three parts. The visual syntax consists of the visual description of the syntactic units as was described in section 3.4. The remaining two aspects are explained in the following two sections.

3.5.1 Replacement behavior

In a structure–oriented editor, programs are constructed via a series of appropriate replacements until the desired program is obtained. The replacement behavior is defined as the set of permissible replacements for each non-terminal of the language. The replacement behavior for a language can be obtained from its syntax definition.

Let the context-free syntax of $\mathcal{L}$ be $\mathcal{L}_{\text{sig}}$ and let it be specified by rules of the form:

$$\alpha_i \rightarrow S_k$$

where S_k is a sort (non-terminal) of language $\mathcal{L}$. For each sort s , a meta-variable (placeholder) is introduced as a special syntactic construct: $S_M \rightarrow S$. The default visual specification for meta-variables is:

$$\nu(S_M) = text() \oplus [\text{string} = \text{“}\langle\text{S}\rangle\text{”}]$$

The visual syntax for this language is defined in terms of the visual mappings of its syntactic units : $\nu(\alpha_i)$. A meta-variable of sort S may be replaced with any $\nu(\alpha_i)$, such that $\alpha_i \rightarrow S$ is in $\mathcal{L}_{\text{sig}}$.

When creating visual sentences in a syntax-directed editor meta-variables bearing the name of corresponding non-terminals (sorts) are used such as:

$$plus(1, div(EXP_M, 3))$$

which corresponds to:

$$1 + \frac{\langle \text{EXP} \rangle}{3}$$

3.5.2 Semantic behavior

In addition to constructing visual sentences, we would like to have semantic behavior accessible from the editor. Thus, we attach some semantic behavior to the editor. There are two ways by which terms can be executed. One is to create a sentence including the function such as: $eval(plus(2,2))$. The other way is to have a special user interface for such a function such as an *eval* button [44, 45]. With the latter method the sentence created in the editor is an expression like $plus(2,2)$ and selecting the appropriate function from the menu causes it to be applied to the sentence. The semantic functions are defined as part of the formal specification of the language – such as *eval* for expressions which evaluates an expression. For example, two of the rules defining *eval* are:

$$[1]\ eval(plus(0, Digit)) = Digit$$

$$[26]\ eval(div(Digit, 0)) = div(eval(Digit), 0)$$

Semantic functions are either applied to the entire program or to a focus in the program. A focus in a program is some part of a program which is selected via user interaction. The functions to be included in the editor are designated with the signature:

$$\langle\ FN_{\text{SVE}},\ FN_{\mathcal{L}},\ SORT\ \rangle \rightarrow \text{FD}$$

where FN_{SVE} and $FN_{\mathcal{L}}$ 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 for which the function is defined informing the editor of when the function is applicable.

3.5.3 Generating editor descriptions

Given the formal specification of a language we can generate a SVE description for that language. The signature of a SVE description is:

$$\begin{array}{lcl} \{\ \textit{REP-LIST}\ \}\ \{\ \textit{FD-LIST}\ \} & \rightarrow & SVE \\ \textit{SORT}\ \text{“:”}\ \{\ \textit{VOD-LIST}\ \} & \rightarrow & REP \end{array}$$

where *REP-LIST* and *FD-LIST* are lists of *REP*s and *FD*s, and *VOD-LIST* is a comma separated list of *VOD*s.

The replacement behavior is, simply, the set of valid substitutions for each sort of the language. Note that the replacement behavior embodies also the visual

repl-behavior(	**imports** Layout		
	exports		
	sorts		
	EXP DIGIT		
	context-free syntax		
	plus(*EXP*,*EXP*)	$\rightarrow$	*EXP*
	div(*EXP*,*EXP*)	$\rightarrow$	*EXP*
	DIGIT	$\rightarrow$	*EXP*
	eval(*EXP*)	$\rightarrow$	*EXP*
	0	$\rightarrow$	*DIGIT*
	1	$\rightarrow$	*DIGIT*
	2	$\rightarrow$	*DIGIT*
	3	$\rightarrow$	*DIGIT*
	4	$\rightarrow$	*DIGIT*
	$DIGIT_M$	$\rightarrow$	*DIGIT*
	EXP_M	$\rightarrow$	*EXP*)

Figure 3.4: The application of the *replacement-behavior* function on the *EXP* language syntax.

syntax to be used. This can be observed from the fact that the replacement behavior consists of a collection of *vod*s.

The generation of the expression editor uses the expression signature and the visual mapping. The syntax-directed editor uses meta-variables for constructing sentences. So, we generate special constants for each sort introduced by the language. In this case they are $DIGIT_M$ and EXP_M. We add these constants to the initial syntax specification and use it as an argument to the editor behavior generating function *replacement-behavior* (Figure 3.4). This function first finds all the syntactically valid replacements for each sort, the result of which can be seen in Figure 3.5. Then, it maps the replacements to their visual representations and attaches their corresponding textual syntax as annotations as shown in Figure 3.6.

The SVE description shown in Figure 3.6 has the desired visual syntax for each sort. Meta-variables are used here since we are dealing with structured visual editors. The annotations next to each replacement choice are the corresponding syntax constructs in the initial language. Sentences are constructed in this manner to execute them with the term rewriter generated for the *EXP* language. A function definition is also given for including the *eval* function as a menu item called *Evaluate*.

If a visual syntax specification for a language does not exist, it is conceivable that a default visual specification could be generated for a language, say as corresponding to a syntax-tree. See [13] for such an approach for textual languages.

EXP :	{	$plus(EXP_M, EXP_M)$,
		$div(EXP_M, EXP_M)$,
		$DIGIT_M$,
		$eval(EXP_M)$,
		EXP_M }
DIGIT :	{	$DIGIT_M$,
		0,
		1,
		2,
		3,
		4 }

Figure 3.5: The replacement behavior without the visual representations.

$$
\begin{array}{ll}
\textbf{\{}\ \text{EXP:}\ \textbf{\{} & \langle\text{EXP}\rangle + \langle\text{EXP}\rangle\ \textbf{\{}\ plus(EXP,EXP)\ \textbf{\}}\ , \\
 & \frac{\langle\text{EXP}\rangle}{\langle\text{EXP}\rangle}\ \textbf{\{}\ div(EXP,EXP)\textbf{\}}\ , \\
 & \langle\text{EXP}\rangle\ \textbf{\{}\ EXP_M \textbf{\}}\ , \\
 & \langle\text{DIGIT}\rangle\ \textbf{\{}\ DIGIT\ \textbf{\}}\ \textbf{\}} \\
\text{DIGIT:}\ \textbf{\{} & \langle\text{DIGIT}\rangle\ \textbf{\{}\ DIGIT_M \textbf{\}}\ , \\
 & 0\ \textbf{\{}\ 0\ \textbf{\}}\ , \\
 & 1\ \textbf{\{}\ 1\ \textbf{\}}\ , \\
 & 2\ \textbf{\{}\ 2\ \textbf{\}}\ , \\
 & 3\ \textbf{\{}\ 3\ \textbf{\}}\ , \\
 & 4\ \textbf{\{}\ 4\ \textbf{\}}\ \textbf{\}}\ \textbf{\}} \\
\textbf{\{}\ \langle\ \text{“Evaluate”,“eval”, EXP}\ \rangle\ \textbf{\}} &
\end{array}
$$

Figure 3.6: The editor description for *EXP* language.

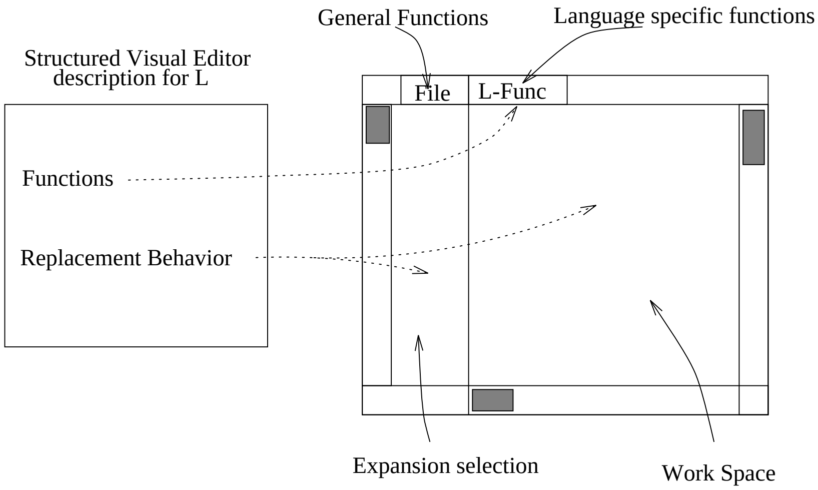


Figure 3.7: The structured visual editor and how it relates to the SVE description.

3.5.4 The Structured Visual Editor

The user interface for SVE consists of three major parts:

- *Replacement panel* is a menu for selecting the desired replacement for a non–terminal during the construction of a program
- *Functions* consist of general editor level commands like save, print, load as well as language specific functions as described in Section 3.5.2. Chapter 7 discusses another approach to realizing this functionality.
- *Work Space* is the area where the visual program is constructed

The user interface for SVE uses a generated editor specification (result of phase 2 in Figure 3.1) for the language specific part of editors. Essentially, there are two distinct parts of the editor. First, there is the shell of the editor which is constant across all SVEs. This part consists of the graphical components such as the window, menu bar and the general editor functions. The second part, is 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 editor description generated for a given language. Figure 3.7 shows the structured visual editor and its relationship to the SVE description.

Finally, all language specific functions are to be executed by the ASF+SDF system. This is done by constructing an abstract syntax of the visual sentences being constructed. This process is transparent to the SVE user.

3.6 Visual Expression Editor

The visual expression editor, which initially has an empty workspace, allows the creation of two kinds of meta–variables (for sorts *EXP* and *DIGIT*), which correspond to the two sorts in the SVE description given in section 3.5.3. Figure 3.8 shows the beginning sequence of an expression creation. (A) shows the initial selection menu and the choice of the ⟨EXP⟩ meta-variable from the selection menu. (B) shows the selection menu when the ⟨EXP⟩ is selected. This selection menu is obtained from the replacement behavior associated with the EXP sort. Finally, (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 type has not changed and is still ⟨EXP⟩. To fully construct an expression the user successively expands the expressions as desired. Eventually, each expression is replaced with a ⟨DIGIT⟩ which is then replaced by one of $0, 1, 2, 3, 4$.

The *Exp* menu consists of one item: *Evaluate* as is specified in the editor specification. When this item is selected, the expression in the editor is passed on to the ASF+SDF system with the semantic function *eval* applied. The resulting value is translated to its visual representation and presented to the user.

3.6.1 Abstract syntax tree

As we already mentioned the abstract syntax of the visual sentences is created by the editor while the user constructs the sentences. The abstract syntax of the sentence can thus be executed by the system. The resulting term must again be presented to the user by means of mapping it to visual representation.

The visual editor has to maintain the path information to the corresponding abstract syntax node at every *vod* instance of a ν map. This path information in combination with tree movement operations supported by a tree processing tool and current node information suffices to build language specific editors. This can also be supported by having special attributes in the tree that serve the purpose of being a pointer to a *vod* instance of a ν map.

The tree for the expression in Figure 3.8-c is:

$$
\begin{array}{c}
\mathit{div} \\
\mathit{EXP} \quad \mathit{EXP}
\end{array}
$$

3.6.2 Sharing

In order to build reasonable visual sentences we must allow sub-sentence to be shared. After all, multi-dimensional syntax permitting multiple relationships among language constructs is the real power of visual expressions. In this work, we have

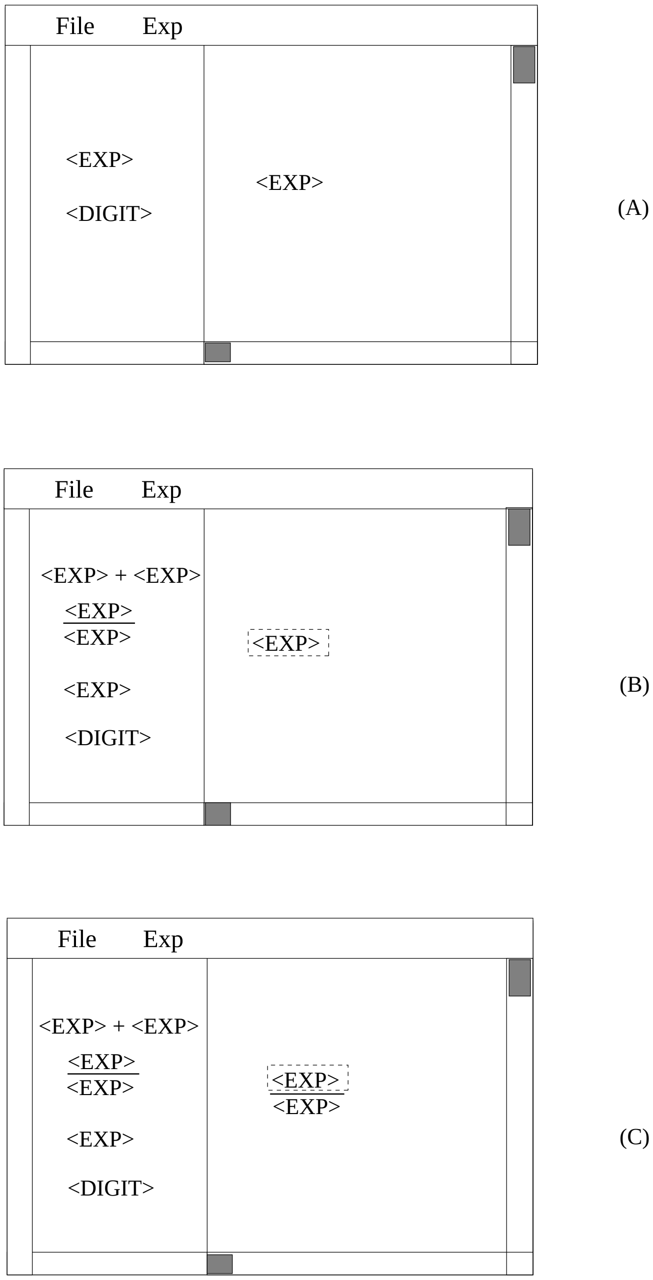


Figure 3.8: The structured visual expression editor.

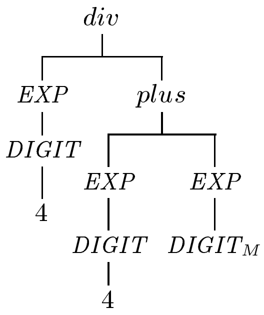
made very conservative extensions which allow pseudo-sharing while still being based on trees. The sharing is created and manifested at the editor level and thus is called pseudo-sharing. At the abstract syntax representation this results in copy semantics.

Sharing is allowed in the following manner:

- when replacing a meta-variable instead of choosing only from the replacement panel, a sub-sentence having the appropriate sort may be chosen as well.
- a replacement by a sub-sentence that results in vertical sharing is not permitted.

Vertical sharing occurs when a of a node is chosen. This results in cyclic terms which can not be resolved and are thus not permitted. Such replacements are rejected by the editor by testing for this condition.

The syntax presented before does not have any allowance for sharing, not for reasons of vertical sharing but due to its spatial constraints. With a different visual mapping we can demonstrate a sentence with sharing. Figure 3.9 shows such a sentence, where the left ⟨EXP⟩ seen in the top window is replaced with the existing subterm $\boxed{4}$ which is shown in the window on the bottom. The abstract syntax for this sentence is:



where the shared left expressions are repeated by copying. If ⟨DIGIT⟩ is replaced by $\boxed{2}$ and the "Evaluate" menu is selected the corresponding term will be rewritten to 4 (modulo 5 arithmetic). This term is then mapped to its visual representation with the appropriate ν mapping resulting in $\boxed{4}$, which is presented to the user in an editor.

3.7 Example: Set Editor

Figure 3.10 shows a view of the Set editor and the beginning of the construction of a term of the Set language. The replacement panel (the left part) provides

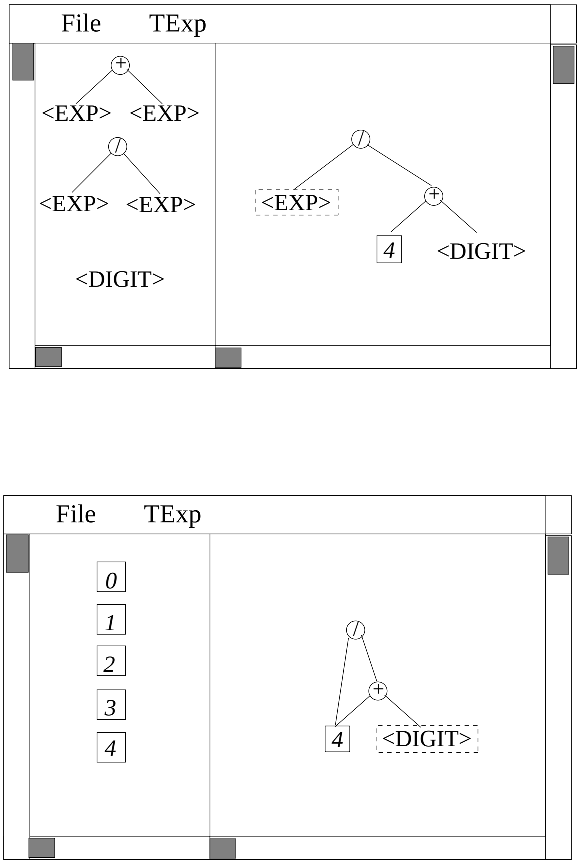


Figure 3.9: The structured visual expression editor with an alternate syntax.

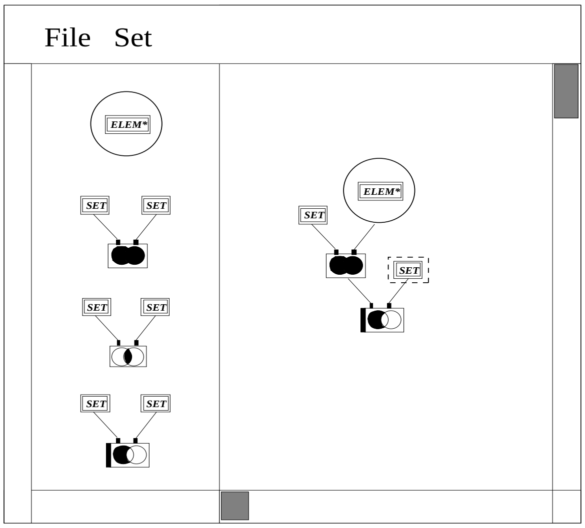


Figure 3.10: The construction of a set term in the visual set editor.

the appropriate replacements for the selected part (focus) of the term. In this case the selected meta-variable is of sort SET and the panel shows the allowed replacements. The "Set" menu provides semantic behavior for the language (such as reducing the constructed term). A completed term is shown in Figure 3.11.

The replacement behavior is enhanced such that replacements are not restricted only to the replacement panel. Replacements by *vod*s of appropriate sorts present in the constructed term are also allowed. This allows the sharing of *vod*s among different parts of the term as seen in Figure 3.11. The editor generation is of great importance as it allows the construction and execution of programs over specified languages. We exploit this generation for defining a VPE generator which is discussed in the next section.

3.8 Towards a Visual Specification Environment

Thus far we have described a setting where visual notation can be defined for textual language contructs. This was done in order to obtain visual editors using the newly defined visual notation. In fact, employing such a method to achieve visual syntax definition would likely be undesirable and we would prefer have to an implicit mapping from the visual notation to some underlying representation, thus, allowing the definition of the syntax of visual languages directly. To obtain a graphical environment for a VPE generator, which allows the definition of visual syntax and semantics, we need *visual* editors for each of these specifications. We

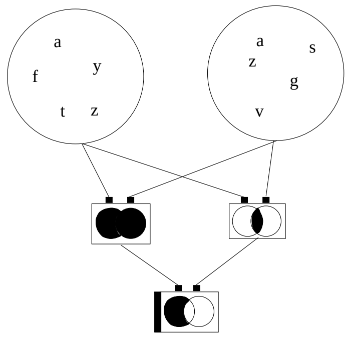


Figure 3.11: A visual term over the Set language representing $(\{a, f, y, t, z\} \cup \{a, s, z, g, v\}) - (\{a, f, y, t, z\} \cap \{a, s, z, g, v\})$.

have already described how to specify visual syntax and generate editors for them. By applying this process to a language-specification language, we get a visual editor for specifying visual languages. In this chapter, we discuss how visual editors could be generated for the specification of visual languages. In a way this section paves the way for the specification environment VASE described in Chapter 5. It is addressed in this chapter due to its relation to editor generation.

3.8.1 Visual Syntax

We use the ideas described thus far as the starting point, where the general scheme of generating visual editors is:

$$L_a \xrightarrow{\nu_L} L_v$$

Given the (abstract) syntax of a language (L_a) and a mapping (ν_L) of the constructs in L_a to visual lexicals, we get the visual language syntax (L_v). Given L_v we can generate visual tools such as a structured editor for L_v.

We can now use this approach at a meta-level, for a language that can specify other languages (language specification language). Doing so would not only allow direct specification of the visual language syntax, but also provide the abstract language definition as its (underlying) representation as well as the implicit mapping function between the two.

For the definition of algebraic specifications, we choose the Algebraic Specification Formalism (ASF) [7], which is a formalism for specifying the syntax and semantic definitions of languages, albeit abstractly. To support the interactive specification of visual languages, we need to generate two tools from this formalism: one for syntax definition and one for semantic definition.

In other words, having a visual mapping from the specification formalism of ASF:

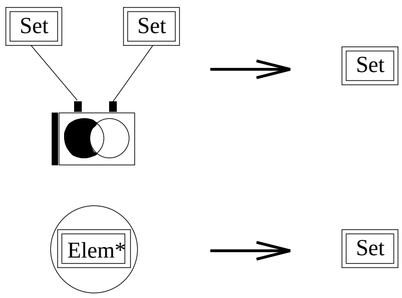
$$\mathrm{ASF} \xrightarrow{\nu_{\mathrm{ASF}}} \mathrm{VASF}$$

would enable using VASF for directly describing the visual syntax of the language of interest (L_v). We then have the diagram:

$$\begin{tikzcd}
L_a \arrow[r, "\nu_L"] & L_v \\
\mathrm{ASF} \arrow[u, dotted] \arrow[r, "\nu_{\mathrm{ASF}}"'] & \mathrm{VASF} \arrow[u, dotted]
\end{tikzcd}$$

where L_a would serve as the abstract representation of the "visually specified" L_v, giving us the necessary information for the generation of an L_v editor and other tools.

Visual syntax can be defined by a signature that consists of a set of syntax rules, where each rule consists of a collection of lexicals and sorts, an arrow, and a resulting sort (all having visual representations defined in VODL). For example, in the following visual signature definition for the *Set* language:



the function symbol for difference, connectors, and the circle are lexical elements defined as *vods*. $\boxed{\boxed{Elem*}}$ and $\boxed{\boxed{Set}}$ represents sorts. The first rule states that the difference function takes two arguments of the sort $\boxed{\boxed{Set}}$ and that this function itself has the sort $\boxed{\boxed{Set}}$.

Given a syntax definition, we can generate two editors: (1) a visual structure oriented editor for the language, and (2) a visual equation editor for defining the semantics of the language.

3.8.2 Visual semantics

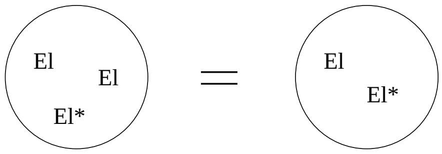
The second tool for language specification is the *visual semantics* editor. The visual semantics editor is parameterized by the syntax of a language and is used for defining the semantics of that language. The semantics is defined by a set of conditional equations between terms of the same sort. Thus, the signature of the semantic specification language for a given language L_v, consists of (1) a signature of the form $S_k = S_k \rightarrow Eq$ for each sort S_k in L_v and (2) the signature of L_v. The sort "Eq" is a sort in the equations language.

For the Set example we have:

$$\boxed{SET} \quad = \quad \boxed{SET} \quad \rightarrow \quad \boxed{Eq}$$

$$\boxed{ELEM} \quad = \quad \boxed{ELEM} \quad \rightarrow \quad \boxed{Eq}$$

along with the rest of the Set syntax. These equation signatures allow the specifier to write equations over the SET and ELEM sorts. Due to the replacement behavior of the editor, each of the above sorts can be replaced with corresponding terms from the Set language. For example, the equation[2]



specifies the elimination of duplicate items in a set. Since the semantic equations use the visual representation of the language, they are quite easy to read (see Section 4.2.2).

3.8.3 Overview of the VPE generation

Figure 3.12 shows the overview of the creation of an interactive VPE generator environment. The top portion of the figure shows the generation of editors for the syntax and semantic specification. This is done by defining a visual mapping for the syntax specification formalism and semantics specification formalism (parameterized by a language specification). This allows the generation of visual editors VASF (for syntax specification) and VEq(L) for semantic specification.

The VASF and VEq(L) along with the editor generator make up a visual VPE generator environment, by which a visual language is specified visually. Finally, from the language specification a visual editor for that language is generated. Programs constructed in the editor are evaluated by term-rewriting as dictated by the semantic eqations. This forms a small VPE for the specified language. Clearly,

[2]The variables could also be mapped to appropriate visual constructs.

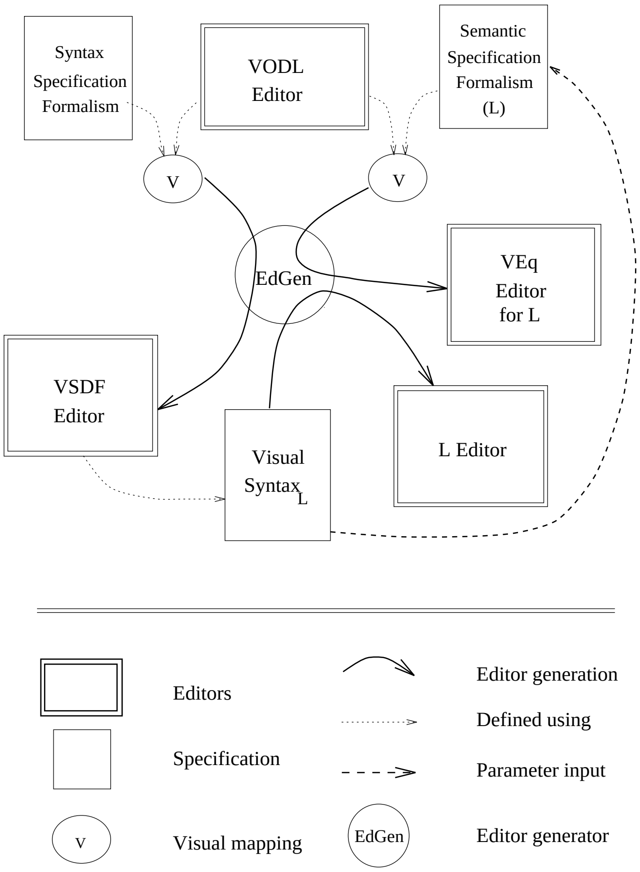


Figure 3.12: An overview of the visual specification environment

many more tools can be generated based on the language specification for a richer VPE (see Chapter 6).

3.9 Implementation

In Section 8.2 we report on some implementation experiments regarding the ideas presented in this chapter. These experiments involve the construction of a simple editor shell using the graphical user-interface development software Garnet [54], the definition of a language, mapping it to visual representations and then to generate corresponding Garnet code for these representations.

3.10 Summary

Editors form the basis for the interactive tools for end users of languages as well as language specifiers. We have shown how language specifications can be used in generating language-specific editors. We have aimed at focusing on the characteristics of the visual syntax and visual editing, and thus separated the specification aspects from the implementation of the user-interface of the editors.

In this chapter we have discussed only the end user editors which are generated from language specifications. We have described this process for the case with an explicit mapping from a textual language constructs to visual notation. Then we outlined how to apply the editor constuction to the specification formalism to yield visual specification editors. Finally, we outlined how such editors could be generated, using implicit mapping from abstract constructs to visual notation.

For the purpose of defining the visual notation we have defined a visual specification language called VODL. In this chapter we have used *vod*s for defining the visual notation for a language. We provided only enough information about VODL to follow the examples. The next chapter addresses the details of VODL.

Chapter 4

VODL: Visual Object Definition Language

4.1 Introduction

Special purpose languages with rich programming environments have allowed so-called "non-programmers" to effectively carry out programming tasks. Such languages are designed to be well suited for particular tasks for end-users and are typically accompanied by user-friendly graphical interfaces to ease programming tasks.

Graphics is widely used in programming environments for easing the programming process for the end-user. This use may involve graphical browsers, animation, debugging, and other tools. Or it may be that the programming language itself is graphical (visual language). One of the approaches to make programming more accessible to novice users is through the use of visual languages which use pictures to represent programs. One of the interesting research areas in visual programming is regarding visual programming environments for special purpose languages.

The demand for special purpose languages and their programming environments has led to research in the *generation* of language-specific environments based on formal language specifications. The specification of languages consists of specifications of syntax, semantics and other language features. Work in programming environment generation includes [64, 42] for textual languages, and [28, 5, 34, 52, 75] for visual languages.

For whatever reason graphics may be used, there is a need for languages which specify graphical constructs. Such languages must describe pictures appropriate for some use. There are a variety of such languages used in document preparation, visualization, pretty printing, visual languages, etc. Some examples include languages based on attribute grammars such as [30, 5], procedural descriptions like [41], or constraint-based declarative languages such as [36, 75, 83].

Constraints have been successfully used in graphical interfaces such as [12, 54] and are very natural in expressing spatial relationships among graphical constructs. We define a constraint-based declarative picture specification language (VODL), where each picture is defined in terms of its sub-components along with a set of constraints expressing the relationship between them. Such descriptions are easy to define, since the specifier need only describe sub-components and their relationships. Instead of the specifier describing some procedure to achieve a certain layout, the underlying system is responsible for finding a solution (via constraint solving).

Our intention is to define a language that can itself be visualized, which can be easily analyzed, which aids in building the picture by accommodating partial evaluation and incremental evaluation; which can be used at an intermediate level for building tools and for communicating with other tools. We intend to achieve this with a clear separation of concerns: objects, constraints, attributes (default/non-defaults). The objects define the structure of a picture composition, attributes define their qualities and constraints define their relative layout.

There are many powerful graphical toolkits and many environments for developing special purpose textual languages and programming environments. However, there are not many attempts to reuse the well developed techniques for textual languages in a graphics supporting environment. In general, it takes a considerable understanding of the internals of the graphical toolkits to attempt using them in programming environment generation since a (semi) formal description of the graphical language (that these toolkits are effectively processing) is not available. Also, it requires a re-examination of the "understanding" whenever a new update is announced.

There are many tools that could have satisfied our purpose, such as the commonly available Xfig [73] or FrameMaker [23], or CSE [80], had they allowed general constraint specification and had they provided a well defined (or published) representation language for interfacing with other tools.

It may be appropriate to clarify that we tried to design VODL expressedly as a small language with constructs that lend to the definition of graphical entities and in no way have we attempted to model a sophisticated graphical tool.

By formally specifying a picture specification language, we hope to study its utility in:

- creating a visual editor for constructing new graphical descriptions
- partial evaluation of pictures
- incremental behavior of pictures
- programming environment generation for visual languages

- graphical tools in programming environments such as animators, visualizers, debuggers, presentation tools such as import graphs and class hierarchies.

4.1.1 The VODL Language

The VODL language specifies visual entities called *visual object definitions* (*vod*s). It provides a set of primitive *vod*s along with a set of expressive *vod* operations for defining new *vod*s. It has a graphical theory, as suggested in [83], providing standard graphical operations which are defined on all *vod*s. Such operations (i.e., overlap) are useful in determining emergent-*vod*s which are discussed in Section 4.2.5.

*vod*s specify visual entities having certain attributes and being comprised of sub-*vod*s, with constraints between these sub-*vod*s. Sub-*vod*s may be: primitives, composites or emergent objects. Constraints are binary relations among sub-*vod*s, defining spatial relationships. Attributes define properties of a *vod*, of which, some might be inherent in every *vod* and others specific to certain *vod*s. See section 4.2 for an example that uses various VODL constructs.

Composite *vod*s allow the definition of new *vod*s in terms of other *vod*s (sub-*vod*s) which may have spatial constraints among them. Sub-*vod*s may be: primitives, composites or emergent objects. Each *vod* may have attributes which define their properties.

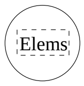
The following section describes various aspects of VODL and gives a brief description of the language. The signature of an interesting subset of the VODL language is shown in Figure 4.1. To assist in describing the language we use a simple visual *Set* language, which is described in the following sections.

4.2 Example: SET

The *Set* language is used as a running example to illustrate the definition of visual tokens and language specification. Table 4.1 shows some visual lexicals for the *Set* language, where Elems denotes a parameter. We will define corresponding *vod*s while describing various concepts of VODL. We often provide graphical appearances of *vod*s, in discussions, which are representative pictures consistent with the constraints—there could be other pictures that also conform to the same set of constraints.

The first *vod*, V-Set, is parameterized (by elements):

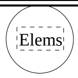
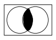
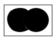
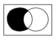
defv V-Set (Elems)
{ cir : **circle** ( ),
elems : Elems }
◁
{ cir **contains** elems }



sorts	$\mathcal{V}$	(*vod* expressions)
	$\mathcal{L}$	(*vod* reference labels)
	$\mathcal{C}$	(constraints)
	$\mathcal{P}_{\mathcal{V}}$	(primitive *vods*)
	$\mathcal{N}$	(names usable for *vod* defs)
	$\mathcal{X}$	(names usable for variables)
	$\mathcal{A}$	(attribute definitions)
	$\mathcal{V}_{\mathcal{A}}$	(attribute values)
	$\mathcal{N}_{\mathcal{A}}$	(attribute names)
	$\mathcal{E}_{\mathcal{A}}$	(attribute expressions)
	$\mathcal{O}_{\mathcal{E}}$	(emergent object operations)
	$\mathcal{O}_{\mathcal{C}}$	(constraint operators)
	$\mathcal{E}_{\mathcal{C}}$	(constrainable expressions)
	$\mathcal{P}_{\mathcal{E}}$	(functions on primitive vods)
	$\mathcal{D}_{\mathcal{V}}$	(defined *vod* abstraction)
	$\mathcal{P}$	(VODL program)
functions	$\{\mathcal{L} : \mathcal{V}, \cdots\} \rightarrow \mathcal{V}$	(*vod* ref declarations)
	$[\vec{\mathcal{L}}.\mathcal{N}_{\mathcal{A}} = \mathcal{E}_{\mathcal{A}}, \cdots] \rightarrow \mathcal{A}$	(define attributes)
	$\{\mathcal{E}_{\mathcal{C}}\ \mathcal{O}_{\mathcal{C}}\ \mathcal{E}_{\mathcal{C}}, \cdots\} \rightarrow \mathcal{C}$	(binary constraints)
	$\mathcal{V} \lhd \mathcal{C} \rightarrow \mathcal{V}$	(constrained *vod* refs)
	$\mathcal{V} \oplus \mathcal{A} \rightarrow \mathcal{V}$	(set attribute values)
	$\mathcal{A}\_\mathcal{V} \rightarrow \mathcal{V}$	(default attribute values)
	$\mathcal{P}_{\mathcal{V}} \rightarrow \mathcal{V}$	(primitive *vods*)
	$\mathcal{L} \rightarrow \mathcal{V}$	(*vod* reference)
	$\mathcal{V}.\mathcal{L} \rightarrow \mathcal{V}$	(label dereferencing)
	$\mathcal{N}(\mathcal{V}_1, \cdots, \mathcal{V}_n) \rightarrow \mathcal{V}$	($n \geq 0$; *vod* instance)
	$\mathcal{X} \rightarrow \mathcal{V}$	(formal arguments)
	$\mathcal{V} \otimes \mathcal{V} \rightarrow \mathcal{V}$	(*vod* merge)
	$\mathcal{O}_{\mathcal{E}}(\mathcal{L}, \mathcal{L}) \rightarrow \mathcal{V}$	(emergent object)
	$\mathcal{L}_1. \cdots .\mathcal{L}_n \rightarrow \vec{\mathcal{L}}$	($n \geq 1$) (*vod* ref dereferences)
	$\vec{\mathcal{L}} \rightarrow \mathcal{E}_{\mathcal{C}}$	(dereferenced *vod*)
	$\mathcal{E}_{\mathcal{A}} \rightarrow \mathcal{E}_{\mathcal{C}}$	(attribute expression)
	$\mathcal{V}_{\mathcal{A}} \rightarrow \mathcal{E}_{\mathcal{A}}$	(attribute values)
	$\mathcal{N}_{\mathcal{A}} \rightarrow \mathcal{E}_{\mathcal{A}}$	(attribute names)
	$\vec{\mathcal{L}}.\mathcal{N}_{\mathcal{A}} \rightarrow \mathcal{E}_{\mathcal{A}}$	(dereferenced attribute)
	$\mathcal{P}_{\mathcal{E}} \rightarrow \mathcal{E}_{\mathcal{A}}$	(predefined functions)
	defv $\mathcal{N}(\mathcal{X}_1, \cdots, \mathcal{X}_n)\ \mathcal{V} \rightarrow \mathcal{D}_{\mathcal{V}}$	($n \geq 0$) (define abstract *vod*)
	$\mathcal{D}_{\mathcal{V}}\ \cdots\ \mathcal{D}_{\mathcal{V}}\ \mathcal{V} \rightarrow \mathcal{P}$	(VODL picture)

Figure 4.1: Signature of VODL Terms

Table 4.1: Some visual lexicals for the *Set* language.

vod	representation
V-Set	
Intersect-Sym	
Union-Sym	
Diff-Sym	

This *vod* describes a circle containing a collection of elements. In this definition we see the use of the **defv** abstraction which defines a composite *vod* consisting of two sub-*vod*s constrained by the *contains* constraint.

4.2.1 *vod* abstraction

Abstraction facilitates the definition of *vod*s with parameters. These can be used in defining auxiliary *vod*s or to effectively extend VODL by building a *vod* library. Abstraction allows the independent definitions of new *vod*s to be used at a later time. The syntax is given by

$$\begin{array}{rcll} \mathbf{defv}\ \mathcal{N}(\mathcal{X}_1, \cdots, \mathcal{X}_n)\ \mathcal{V} & \rightarrow & \mathcal{D}_{\mathcal{V}} & \text{(define abstract } vod\text{)} \\ \mathcal{N}(\mathcal{V}_1, \cdots, \mathcal{V}_n) & \rightarrow & \mathcal{V} & (vod \text{ instance}) \\ \mathcal{D}_{\mathcal{V}}\ \cdots\ \mathcal{D}_{\mathcal{V}}\ \mathcal{V} & \rightarrow & \mathcal{P} & (\text{VODL picture}) \end{array}$$

where a VODL picture is a list of *vod*-abstractions followed by a *vod* term that could be built by their instantiations. The example in Section 4.2 shows the use of abstraction in defining visual syntax for sets.

4.2.2 Primitives

Primitive *vod*s are the *vod*s that are provided in the basic VODL and can be used as the basic building blocks for constructing composite *vod*s. There are a number of primitive *vod* types, such as Point, Line, Circle, Text, Polygon and Collection. These types have primitive functions for building useful attribute expressions and also have various constraint operations defined on them. They also have some weak constraints (e.g., the elements in a collection are non-overlapping) and some default attributes.

Furthermore, a collection provides a way of gathering many *vod*s, without the need for a specific shape in which they have to be contained. A collection is a set of *vod*s gathered together by an associative and commutative operation[1]. The

[1] ϕ_c denotes the empty collection.

constraints defined over collections are abbreviations for constraining all *vod*s in a collection. Thus many constraint operations are meaningless when both operands of a constraint happen to be collections.

Textual specification languages generally include a *list* construct for this use [35]. Since visual syntax is often unordered, collections are more appropriate. However, visual lists are useful with visual languages as well. For example, Holt uses them for argument sequences [38], and Wang [83] defines a list to be a *pictorial concept* which is used in defining complex pictures. A finite list is easily defined, in VODL, since VODL is based on records – albeit not an arbitrary but finite list.

4.2.3 Composites

Composite *vod*s are constructed from sub-*vod*s and sets of constraints among the sub-*vod*s. The basic construction involves declaring a set of *vod*-references (*vod-refs*) that refer to other *vod*s, and then specifying the constraints between these *vod*s using the *vod*-refs. Any specific attributes and their values can also be specified (with the $\oplus$). See Section 4.2.7 for operations defined over (composite) *vod*s.

The basic structure of a composite *vod* can be characterized by the signature:

$$\begin{array}{rcll}
\{\mathcal{L}:\mathcal{V},\cdots\} & \rightarrow & \mathcal{D} & \text{(\textit{vod} ref declarations)} \\
\{\mathcal{E}_{\mathcal{C}}\ \mathcal{O}_{\mathcal{C}}\ \mathcal{E}_{\mathcal{C}},\cdots\} & \rightarrow & \mathcal{C} & \text{(binary constraints)} \\
\mathcal{D} \lhd \mathcal{C} & \rightarrow & \mathcal{V} & \text{(constrained \textit{vod} refs)} \\
\mathcal{V}\oplus[\vec{\mathcal{L}}.\mathcal{A} = \mathcal{E}_{\mathcal{A}},\cdots] & \rightarrow & \mathcal{V} & \text{(set attribute values)}
\end{array}$$

The constructor *vod* forms are the cases:

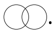
$$\begin{array}{ll}
\{\mathcal{L}:\mathcal{V},\cdots\} \lhd \{\mathcal{E}_{\mathcal{C}}\ \mathcal{O}_{\mathcal{C}}\ \mathcal{E}_{\mathcal{C}},\cdots\} & \text{and} \\
(\{\mathcal{L}:\mathcal{V},\cdots\} \lhd \{\mathcal{E}_{\mathcal{C}}\ \mathcal{O}_{\mathcal{C}}\ \mathcal{E}_{\mathcal{C}},\cdots\})\oplus[\mathcal{A} = \mathcal{V}_{\mathcal{A}},\cdots] &
\end{array}$$

The attributes ($\mathcal{A}$) differ from one primitive type to another. However, certain attributes are inherent to all *vod*s, such as width and height.

4.2.4 Constraints

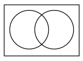
Constraints, $\mathcal{O}_{\mathcal{C}} \in \{\textbf{contains},\textbf{over},\cdots,=,\neq,<,\geq\}$, are binary relations among *vod*s. They are based on the geometric properties and the attributes of *vod*s. Geometric constraints deal with the relative sizes and positions of *vod*s. For example, the constraint *over* concerns the relative positions of *vod*s (the left one must be positioned over the right one), and *contains* which concerns the size of *vod*s (the left one must be greater than the right one). Other kinds of constraints are defined in terms of the attributes of *vod*s, such as *width*, and *color*.

4.2.5 Emergent objects

The sub-*vod*s can be special *vod*s which are *emergent objects* as in [83]. Emergent objects are the new *vod*s that appear as a result of composing *vod*s where some overlap occurs. For example, consider a *vod* which consists of two overlapping circles, . This composition results in the emergence of the four sub-*vod*s: . The recognition and access to such emergent objects can be very useful in the specification of constraints (see Section 4.2). The emergent objects are identified when $\mathcal{O}_{\mathcal{E}} \in \{$**overlap**,**difference**,$\cdots\}$ operations are used in the specification.

Returning to the *Set* example, when considering the function symbols in the table, it is apparent that there are considerable similarities between all of them. Each function symbol shares the same graphical representation. By defining the auxiliary *vod*, *Two-Sets*, corresponding to this representation, the repetition of this definition in each function symbol can be avoided.

defv Two-Sets ( )
{ set1 : V-Set (ϕ_c),
set2 : V-Set (ϕ_c),
inter : **overlap**(set1,set2),
diff1 : **difference**(set1,set2),
diff2 : **difference**(set2,set1),
border : **rectangle** ( ) }
$\lhd$
{ set2.**top** = set1.**top**,
set2.**left** = set1.**left** + set1.**width** $*$ 2 / 3,
border **contains** set1,
border **contains** set2 }



The definition of Two-Sets consists of six sub-*vod*s, three of which (set1, set2 and border) are ordinary predefined sub-*vod*s and the remaining three *vod*s (inter, diff1, and diff2) are emergent-*vod*s identified by using the *overlap* and *difference* graphical operations defined in VODL. These emergent-*vod*s are used in subsequent *vod* definitions (Intersect-Sym, Union-Sym, Diff-Sym).

4.2.6 Attributes

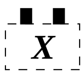
Attributes for *vod*s can be specified with the $\oplus$ operation. Some attributes such as width and height are defined over all *vod*s, whereas others are specific to *vod*s.

The auxiliary parameterized *vod* Fsym-ports attaches two small rectangles on top of a *vod* [2] to represent input ports for functions.

[2]The *vod* is visually represented as 

.

defv Fsym-ports (Fsym)
  { func : Fsym,
    port1 : **rectangle** ( ) ⊕
        [**height** = 5, **width** = 5],
    port2 : **rectangle** ( ) ⊕
        [**height** = 5, **width** = 5] }
  ◁
  { port1.**bottom** = func.**top**,
    port1.**left** = func.**left** + func.**width** / 3,
    port2.**bottom** = func.**top**,
    port2.**left** = func.**left** + 2 ∗ func.**width** / 3 }



4.2.7 *vod* operations

New *vod*s can be defined as described in Section 4.2.3 or by using *vod* operations. The *vod* operations define new *vod*s based on existing ones by extending them with attributes and/or constraints, or by merging two existing *vod*s. These operations are described below. Our intention is to realize a *vod* from its seamingly different descriptions by identifying some syntactically differing forms. For instance, "color parent red" is the same as traveling to the parent with "color red". The equations are not complete and thus are only indicative of the semantics. We use the following variable declarations and abbreviations[3]:

$$
\begin{array}{lll}
v \rightarrow \mathcal{V} \quad l \rightarrow \mathcal{L} \quad a \rightarrow \mathcal{N}_{\mathcal{A}} \quad o \rightarrow \{\ \} & & \\
e \rightarrow \mathcal{E}_z \qquad (z = \mathcal{A} \text{ or } z = \mathcal{C}) & & \\
\vec{d} \;=\; l_1 : v_1, \cdots, l_n : v_n & & (n \geq 0) \\
\vec{l} \;=\; l_1 . \cdots . l_n & & (n \geq 0) \\
\vec{x} \;=\; \vec{l_1}.a_1 = e_1, \cdots, \vec{l_n}.a_n = e_n & & (n \geq 0) \\
\vec{c} \;=\; e_1\ o_1\ e_1', \cdots, e_n\ o_n\ e_n' & & (n \geq 0)
\end{array}
$$

Also, "$v[l : v_1]$" is an abbreviation for "v has a *vod-ref* l with an associated *vod* v_1", i.e.:

$$
\begin{array}{ll}
v = (\{\vec{d_1}, l : v_1, \vec{d_2}\} \ \lhd \ c) & \text{or} \\
v = (\{\vec{d_1}, l : v_1, \vec{d_2}\} \ \lhd \ c) \oplus [\vec{x}] &
\end{array}
$$

Given this notation we will now define the operations ***attribute-set***, ***constraint-add***, and***vod-merge***.

[3]Note that the equals symbol (=) is overloaded. We use

1. a meta-level "=" for identifying equal terms (semantics),
2. an "=" for setting attribute values, and
3. an "=" for indicating a binary constraint.

attribute-set

The attribute-set operation

$$\mathcal{V}\oplus[\vec{\mathcal{L}}.\mathcal{N}_{\mathcal{A}} \;=\; \mathcal{E}_{\mathcal{A}}, \cdots] \;\rightarrow\; \mathcal{V} \quad \text{(set attribute values)}$$

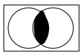
is defined over a *vod* and a list of attribute-value pairs. Its intended meaning is to assign values to some attributes of a given *vod* and it can be formally defined as follows.

$$\begin{aligned} v\oplus[] &= v & \text{(A0)}\\ v[l : v_1]\oplus[l.\vec{l}.a = \alpha] &= v[l : (v_1\oplus[\vec{l}.a = \alpha])] & \text{(A1)}\\ (v\oplus[\vec{x_1}])\oplus[\vec{x_2}] &= v\oplus[\vec{x_1}, \vec{x_2}] & \text{(A2)}\\ (v_1[l : v_2].l.\vec{l})\oplus[\vec{x}] &= v_1[l : (v_2.\vec{l}\oplus[\vec{x}])].l.\vec{l} & \text{(A3)} \end{aligned}$$

The equation (A1) specifies that the attribute is to be defined on a sub-*vod* of v, the equation (A2) indicates merging the attribute lists (assume left-prefer) and the equation (A3) shows the travel[4] to a sub-*vod*.

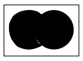
As an example consider a definition of a function symbol for *intersection* using the $\oplus$ *vod*-operation, which redefines Two-Sets by altering the highlight attribute of the *inter* component. Note that *vod*-refs are accessible from outside of the defining *vod*.

defv Intersect-Sym ( )
  Two-Sets ( ) $\oplus$ [inter.**filled** = **true**]



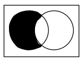
Similarly, the function symbol for *union* is defined by altering the highlight attributes of the components set1 and set2.

defv Union-Sym ( )
  Two-Sets ( ) $\oplus$
    [set1.**filled** = **true**, set2.**filled** = **true**]



Also, the function symbol for *difference* is defined by altering the highlight attribute of the diff1 component.

defv Diff-Sym ( )
  Two-Sets ( ) $\oplus$ [diff1.**filled** = **true**]



constraint-add

The *constraint-add* operation

$$\mathcal{V} \;\lhd\; \{\mathcal{E}_{\mathcal{C}}\; \mathcal{O}_{\mathcal{C}}\; \mathcal{E}_{\mathcal{C}}, \cdots\} \;\rightarrow\; \mathcal{V} \quad \text{(add constraints)}$$

[4]Note that the equation $v_1[l : v_2].l = v_2$ does not exist. We do not want to say that a *vod* with "color parent red" is same as the parent *vod* with "color red", but we want to capture that a *vod* with "color parent red" is same as traveling to the parent *vod* with "color red".

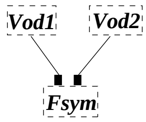
is defined over a *vod* and a set of constraints and is intended to add constraints to a given *vod*. The semantics is given by rules:

$$
\begin{array}{rcll}
v \triangleleft v & = & (\mathrm{C0}) & \\
(v \triangleleft \{\vec{c_1}\}) \triangleleft \{\vec{c_2}\} & = & v \triangleleft \{\vec{c_1}, \vec{c_2}\} & (\mathrm{C1}) \\
(v_1[l : v_2].l.\vec{l}) \triangleleft c & = & v_1[l : (v_2.\vec{l} \triangleleft c)].l.\vec{l} & (\mathrm{C2}) \\
(v \oplus [\vec{x}]) \triangleleft c & = & (v \triangleleft c) \oplus [\vec{x}] & (\mathrm{C3})
\end{array}
$$

The equation (C1) indicates the merge of constraints; (C2), the travel to a sub-*vod* and (C3), the commuting of the $\triangleleft$ and the $\oplus$ operations.

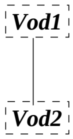
To represent the binary functions, in our *Set* example, two *vod*s are defined. *Commf* represents commutative and *Non-Commf* represents non-commutative functions. *Commf* is defined using the constraint-add operator, whereas the definition of *Non-Commf* will follow shortly.

defv Commf (Fsym, Vod1, Vod2)
{ func : Fsym-ports (Fsym),
v1 : Vod1,
v2 : Vod2,
con1 : Connector (v1, func.port1),
con2 : Connector (v1, func.port2) }
$\triangleleft$
{ v1 **over** func,
v2 **over** func }



where Connector is defined as:

defv Connector (Vod1, Vod2)
{ first : Vod1,
second : Vod2,
cl : **line** ( ) }
$\triangleleft$
{ cl.**left touches** first,
cl.**right touches** second }



vod-merge

The *vod-merge* operation

$$\mathcal{V} \otimes \mathcal{V} \rightarrow \mathcal{V} \quad (vod \text{ merge})$$

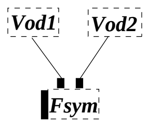
is defined over two (composite) *vod*s. The semantics is given by:

$$
\begin{array}{rcll}
(\{\vec{d_1}\} \triangleleft \{\vec{c_1}\}) & & & \\
\otimes\, (\{\vec{d_2}\} \triangleleft \{\vec{c_2}\}) & = & \{\vec{d_1}, \vec{d_2}\} \triangleleft \{\vec{c_1}, \vec{c_2}\} & (\mathrm{V1}) \\
(v_1[l : v_2].l.\vec{l}) \otimes v_3 & = & v_1[l : (v_2.\vec{l} \otimes v_3)].l.\vec{l} & (\mathrm{V2}) \\
v_1 \oplus [\vec{x}] \otimes v_2 & = & (v_1 \otimes v_2) \oplus [\vec{x}] & (\mathrm{V3})
\end{array}
$$

The equation (V2) indicates the travel to a sub-*vod* and the equation (V3) indicates the commuting of $\otimes$ and the $\oplus$ operations. The effect of (V3) is to make the attributes defined in v_2 preferred over that of v_1—assuming a left-prefer rule.

Continuing our Set example, *Non-Commf* is defined by adding a strip on the left side of a function symbol[5]:

defv Non-Commf (Fsym, Vod1, Vod2)
  Commf (Fsym, Vod1, Vod2).func $\otimes$
    { tag : **rectangle** ( ) $\oplus$
        [**filled** = **true**, **width** = 5] }
    $\triangleleft$
    { tag.**right** = func.**left**,
      tag.**top** = func.**top**,
      tag.**height** = func.**height** }



This definition uses the *vod*-merge operation to redefine the *Commf vod*. Note that the effect of this definition is analogous to inheritance in object oriented languages, where an earlier definition is extended. Here, *Commf* is extended with an additional *vod*-ref (tag) and some additional constraints.

4.3 VODL editor

We have defined the VODL language as a textual language. It would likely be useful to have visual *vod* editors which would allow the graphical interactive construction of *vod*s. In defining *vod*s as such we have to take some things into account. First of all, as we have seen, there are usually numerous representations that correspond to each *vod*. Graphical representations, by their nature, are highly concrete and not very good at representing abstractions such as in definitions. However, it is still useful to have visual feedback that corresponds to the *vod*s that are being defined. On the other hand most of the details of the definition will be invisible – such as the precise structure of the composite *vod*s and the constraints. Thus, we would need at least one more editing view that reveals the particulars of the *vod*. For example, Figures 4.2 and 4.3 indicate possible visual editor views corresponding to the *vod*s Two-Sets and Intersect-Sym defined earlier.

The view in Figure 4.2 shows a graphical definition (**defv**) of the *vod* Intersect-Sym where Two-Sets is extended by altering an attribute (filled = true) of the overlapping *vod* ("inter," in the textual definition). The *vod* that is being extended is above the new *vod* being defined, which are connected with the attribute-set operation. The specifier has chosen the *define vod* from the Operations menu to define a new *vod* and then chosen the *attribute-set* operation, which requires a *vod* and a set of attributes.

[5]The black strip on the left side is for representing orientation, which is similar to *dog ears* in Holt's [38] *viz* language.

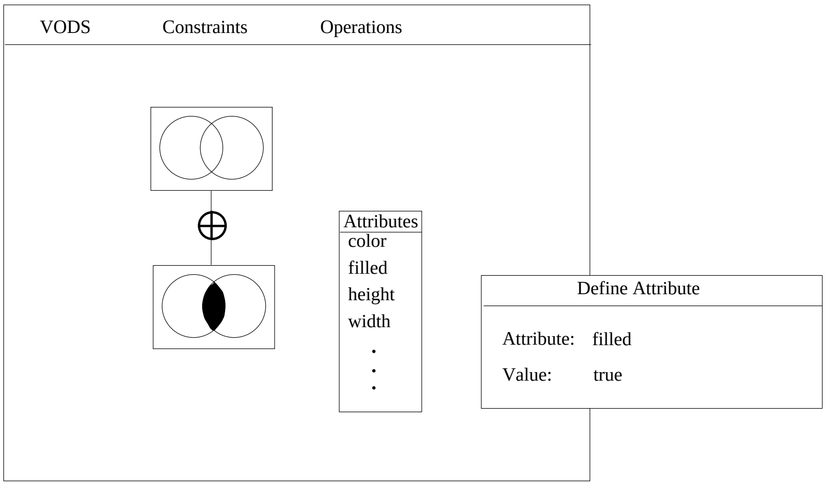


Figure 4.2: VODL editor: Defining the *vod* Intersect-sym

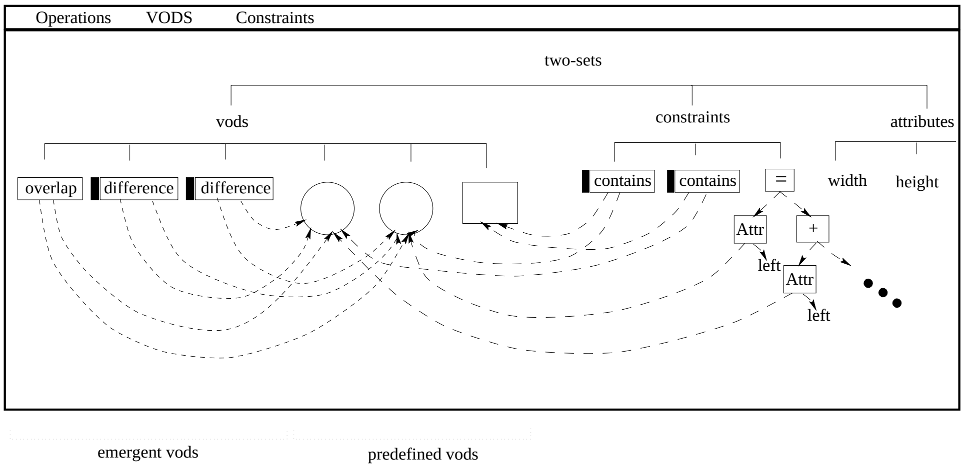


Figure 4.3: VODL graph view editor: Definition of Two-sets

The editor in Figure 4.3 shows a different view of the Two-Sets *vod* which shows the relationships among the components using a graph representation and reveals much more information about the structure of the *vod* as opposed to the previous editor which presents its visual appearance. Such an editor is useful in understanding the relationships among the components and can be useful in accessing some components that may be difficult to access by pointing and clicking (i.e. due to overlapping, layering etc.). It may be useful to hide uninteresting components and their relations from the view in order to be able to concentrate only on the components of interest.

4.4 Implementation

In Section 8.2.2 and Section 8.3.1 we describe implementation experiments regarding *vod*s. In Section 8.2.2 the *vod*s are specified and their translation is defined for graphical objects in Garnet [54]. In Section 8.3.1 we show an interactive editor for defining *vod*s as defined in this chapter.

4.5 Summary

We have presented a picture specification language (VODL) and discussed its role in generating programming environments for visual languages. VODL supports the easy definition of graphical elements by providing composition, extension and abstraction in defining new *vod*s.

VODL has been introduced for the purpose of defining the necessary *vod*-types (lexicals) required for specifying visual syntax. In order to incorporate these lexicals in a language definition, we can relate them with the abstract syntax of a language via a mapping as was discussed in Section 3.4.4. Alternatively, we could incorporate *vod*s into a visual specification formalism as will be described in Chapter 5. In this formalism the *vod*s defined are directly used when specifying the language syntax which makes an explicit mapping unnecessary.

Chapter 5

The VAS Formalism in VASE

5.1 Introduction

Specifications of languages are on the one hand desirable for the purposes of analysis and tool generation but are on the other hand they are intimidating and cryptic and difficult to comprehend. This is often caused by the use of a fixed specification language burdened with expressing complex languages features [71, 48]. 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 use of specifications we need formalisms and tools that ease both writing and understanding them.

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.

The focus of this chapter 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 5.2, followed by the VAS Environment in Section 5.3. Section 5.4 gives a brief description of the generated visual programming environment. Section 5.5 provides an example detailing both the specification formalism and the environment.

5.2 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 [6, 18] which renders them useful for prototyping languages.

The VAS formalism finds its roots in the ASF formalism [6] 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 other 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, yielding more comprehensible specifications.

VAS relies on VODL (Visual Object Definition Language) (Chapter 4) for defining the lexical elements to support the use of visual notation in specifications. 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. Note that VODL only defines pictures and it is at the level of VAS that they get a special interpretation as language elements.

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
        sorts SORT-LIST
        functions FUN-LIST

> *variables VARDEC-LIST*
> *equations*
> *EQUATION-LIST*

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

5.2.1 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 is 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.

SVL

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

The following is an example of a syntax rule using visual lexicals:

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

where $\boxed{}$ is a lexical and *LABEL*, *C* and *LBC* are *SORT*s. This rule defines a function of sort *LBC* with first argument of sort *LABEL* and second argument of sort C. The abstract representation of this function is: $f(LABEL, C) \rightarrow LBC$. In the concrete (visual) syntax the function symbol $\boxed{}$ along with any spatial properties corresponds to f. The construction of such rules and equations is discussed in Section 5.3.1.

[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.

Collection Sort

Collections correspond to lists in one-dimensional (1D) syntax specification languages. Lists, as primitives, are available in many languages. In algebraic specification languages the analogous facility is the ability to express that an operation is associative. We have built our collection syntax based on the list syntax used in ASF+SDF. Briefly, what one desires from an associative list are the following:

- The ability to construct an empty list. When using an associative operator, one might declare a particular element as the identity element.
- Every two adjacent elements are related, either by an adjacency operator or by the declared associative operator. Another way to think of this is that two adjacent operators *share* an element.
- The editor provides support to construct this list in a natural manner, while constructing the intended abstract structure (for example, a flattened structure) for the underlying machine.
- Associativity is understood by the underlying machinery. For algebraic specifications, this implies that the underlying rewriting machinery uses associative matching so that the user is not worried about the manner in which the term was constructed. This is referred to as rewriting modulo associativity.

Our desire is to experiment with an analogous facility for the two-dimensional (2D) case. The problems that we have to consider for the 2D case are:

- What is an appropriate ordering of the elements in 2D? Adjacency was used in the case of 1D to construct the ordering. A related question is, how do we identify the elements? Ordering was used implicitly in the 1D case—say, to traverse a list.
- How can we help the user construct such a term in an editor?
- What is the abstract representation of this term? Also, how can the editor term be rebuilt from it?
- How do we do 2D matching? Or what is the intended interpretation, during rewriting?

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. This is very useful in writing rewrite rules. Visually, the collection is basically a multi-set which allows the definition of general constraints over its elements. In its simplest form the collection items have no constraints. This could, for instance, be a collection representing windows, where there is no particular constraint among the windows.

The VAS formalism provides a collection primitive which is used to conveniently define an arbitrary set of language constructs in a signature definition.

In this chapter we examine the use of collections in specifying visual syntax. The list collection is commonly used in textual language specification for defining language constructs consisting of arbitrarily many repetitions (such as statement lists in a standard procedural language). The multi-dimensional nature of visual syntax renders the use of lists, for similar purposes, in most cases not useful. To specify such languages we need a specification construct that is not ordered, but yet groups a set of language constructs. For this, we define the collection primitive which serves as a primitive for grouping items.

It is introduced to provide flexibility for the language specifier in defining language constructs consisting of arbitrarily many items. Such language constructs are very common in practice.

Without the existence of such constructs the specifier is forced to define these constructs in an elaborate manner. This forces the specifier to be painfully aware of the manner in which the terms could be built. More importantly, the specifier now has to take care of detecting the potentially shared terms in order to handle the intended semantics, while defining operations over these items. Suppose one specifies the sort C as follows:

$$
\begin{array}{ccccc}
A & \_ & B & \to & B \\
 & & A & \to & B \\
 & & | & & \\
 & & B & &
\end{array}
\qquad
\begin{array}{ccccc}
 & & A & \to & B \\
A & \_ & B & \to & B \\
| & & & & \\
B & & & &
\end{array}
\qquad
\begin{array}{ccc}
 & \to & C \\
B & \to & C
\end{array}
$$

Now lets say a term is built like:

$$
\begin{array}{ccc}
A_1 & \_ & A_2 \\
| & & | \\
A_3 & \_ & A_4
\end{array}
$$

where A_i is of sort A. This term can be constructed with pseudo-sharing by causing the Bs that are eventually replaced with A_4 to be shared. As this is pseudo-sharing, it is equivalent to terms where sharing does not exist. Thus, the term above is the same as:

$$
\begin{array}{ccc}
A_1 & \_\_\_\_ & A_2 \\
| & & | \\
A_3 & \_A_4 & A_4
\end{array}
$$

5.2.2 Collections and VAS

The idea behind collections is to provide an easy manner for defining a collection of program units and to provide operations on them that make it easy to define the semantic rules as well as defining end-user tools. These will be explained along with the example presented in Section 5.5.

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

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

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 constraint operators and the symbol "$\square$" is used to indicate the parameters of the constraint operations. The arguments to these operators are of the sort given in the collection definition – one for each parameter of *vod*. For example, a collection signature of the form

$$T* \;\lhd\; \{\square\, v_1 \square \,, \square\, v_2 \square\,\} \quad\rightarrow\quad S$$

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. A collection for the problem introduced in the previous section is:

$$A* \;\lhd \{\square\square \;, \begin{smallmatrix}\square\\ \square\end{smallmatrix}\} \quad\rightarrow\quad S$$

the v operator does not show here and instead its effect is displayed by using the $\square$ as arguments.

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 multi-set which allows the definition of constraints over its elements. In its simplest form the collection items have no constraints.

For representation and semantics of the collection, we use the relational grammar representation of Wittenburg [86]. 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 5.5 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 [18, 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.

5.2.3 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 a corresponding equation as follows:

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

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

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

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 5.2.1 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'.

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 [83] of language constructs is always consistent with their syntactic definition.

5.3 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 5.1 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 5.5 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

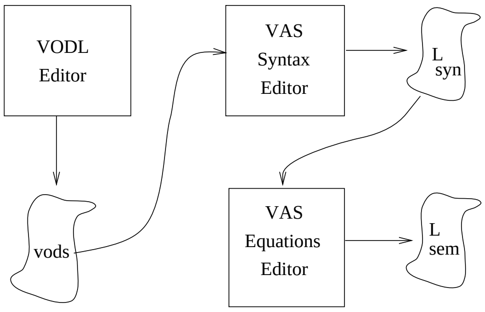


Figure 5.1: The VASE editors and their relations to various specifications.

being constructed. The latter option allows sharing, which is true sharing when constructing a collection, otherwise it is pseudo sharing (Chapter 6).

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.

5.3.1 VAS syntax editor

The VAS syntax editor supports the syntax of a visual language according to the grammar defined in Section 5.2.1. Here the construction of a single function specification is shown. The goal is to specify the function: $\boxed{\boxed{A}} \rightarrow$ B. Figure 5.2 shows a possible construction sequence. The first two steps simply follow from the VAS grammar. Recall that VAS relies on *vod* definitions for its lexical elements. In this case we could use the *vod inbox*:

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

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

5.3.2 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 5.2.3. The syntax of the equations is generated from the syntax definition.

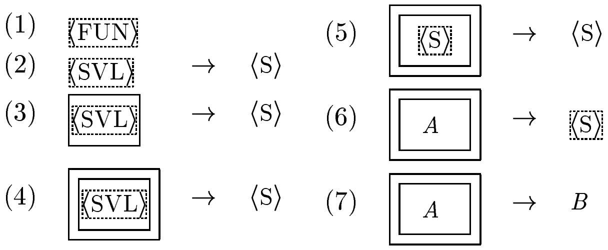


Figure 5.2: A syntax rule construction sequence.

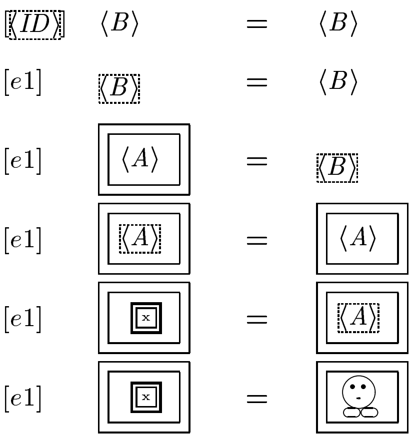


Figure 5.3: Equation construction sequence.

Figure 5.3 shows the construction of an equation for the function defined in the previous 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.

5.4 VPE for specified languages

The intention of specifying languages is for generating language specific tools. Figure 5.4 shows the use of 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.

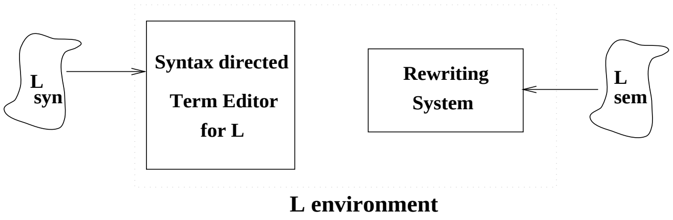


Figure 5.4: Generated L environment.

The components shown in the dotted region are generated from the specifications indicated with an arrow. The editor allows the construction of terms of the language in a syntax directed manner. The rewriting system makes the specifications executable. These components are united in a single interface [75] where the rewriter "evaluates" the term constructed in the editor (see Section 5.5.4). Functions are bound to interface components which pass the term with the function to the rewriter. The term resulting from execution may replace the term in place or be placed in another editor. The example presented in Section 5.5 discusses such an environment.

5.5 Example

The *move-about language* is used as an example to demonstrate the VAS formalism and the supporting environment 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.

5.5.1 Syntax definition

Figure 5.5 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* $* \lhd \{\genfrac{}{}{0pt}{}{\Box}{\Box}, \Box\;\Box\ \}$ 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 ($\Box$). This collection is used to define the sort *GAME*.

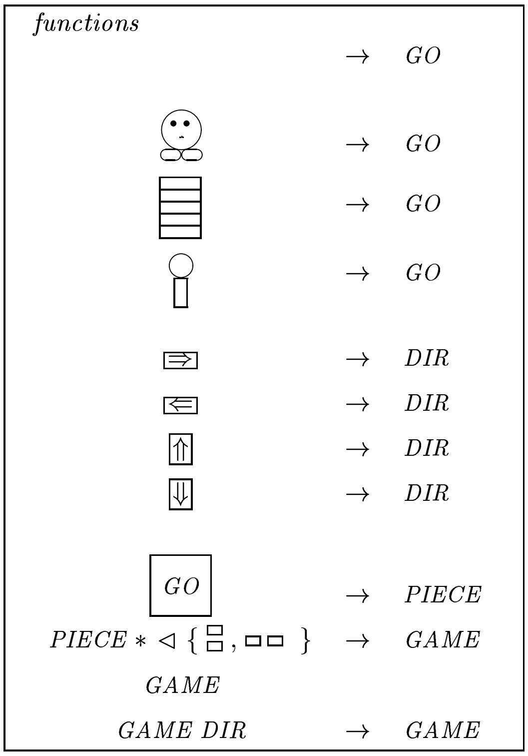


Figure 5.5: Syntax definition using VAS formalism.

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

Higher order variables [18, 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 is given by: $vod_{lib}(SORT* \lhd \{COP\text{-}LIST\}) \rightarrow SORT$. Using this, we define the collection variable used in the equations:

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

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 spatial layout of the constraint operators in the above rule comes from its corresponding *vod* definition. The placeholders serve as arguments to provide visual feedback.

5.5.2 The move-about editor

From the syntax definition of the move-about language, a term editor is generated. Figure 5.6 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 $T * \lhd \{\square\, \mathrm{v}_1 \square, \overset{\square}{\underset{\square}{\mathrm{v}_2}}\} \rightarrow S$, the term construction is defined using an example. The "$\Rightarrow_i$" indicates the ith modification of the term after a replacement.

$$\boxed{\langle S\rangle} \Rightarrow_1 \begin{array}{l}\langle T\rangle\ \mathrm{v}_1 \langle T*\rangle\\ \mathrm{v}_2\\ \boxed{\langle T*\rangle}\end{array} \Rightarrow_2 \begin{array}{l}\langle T\rangle \mathrm{v}_1 \boxed{\langle T*\rangle}\\ \mathrm{v}_2\\ \langle T\rangle\ \mathrm{v}_1 \langle T*\rangle\\ \mathrm{v}_2\\ \langle T*\rangle\end{array} \Rightarrow_3 \begin{array}{l}\langle T\rangle\\ \mathrm{v}_2\\ \langle T\rangle\ \mathrm{v}_1 \langle T*\rangle\\ \mathrm{v}_2\\ \langle T*\rangle\end{array}$$

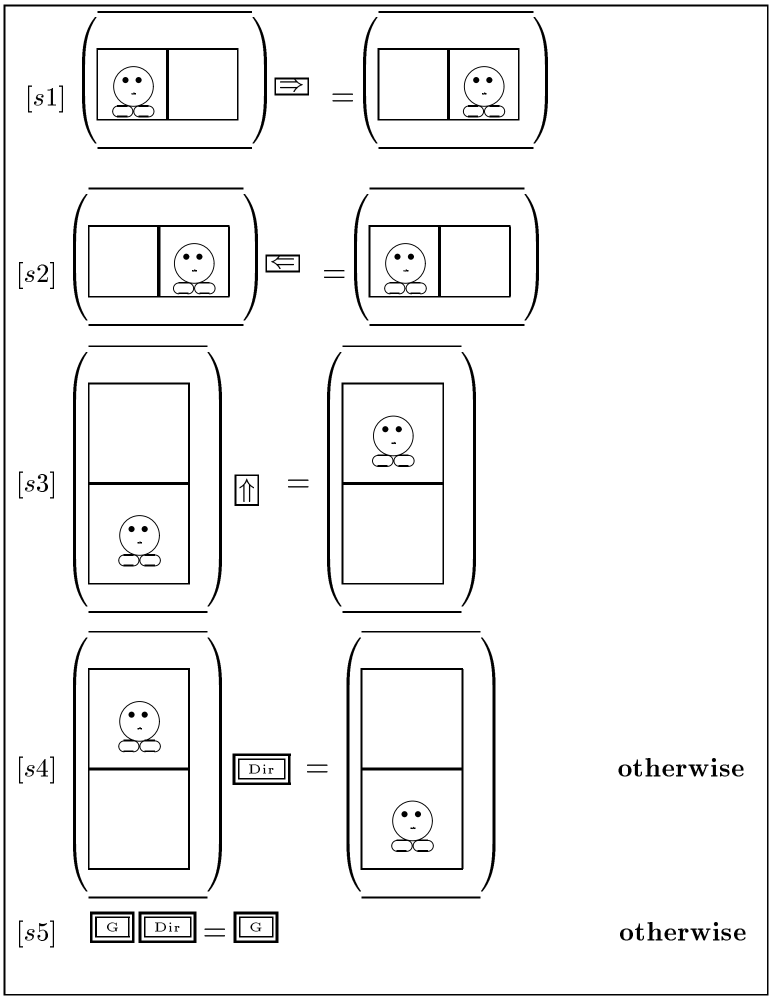
Collections are represented by the sort name followed an $*$. The meta-variable $\langle T*\rangle$ can be replaced with:

1.	nothing
2.	$\langle T\rangle$
3.	$\begin{array}{l}\langle T\rangle \mathrm{v}_1 \langle T*\rangle\\ \mathrm{v}_2\\ \langle T*\rangle\end{array}$

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 eventually 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.

5.5.3 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.



The rules [$s4$] and [$s5$] above are *default* rules [42] 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 [$s5$].

5.5.4 Specification of interaction

The equations of the previous section describe how the player moves between scenes, given a direction. One or more directions for the game can be provided in 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 [21][2]. Here the latter choice is considered. We use meta-variables to solicit values interactively. is a function which returns the value that replaced the meta-variable. Other terms inside the provide the user with the context for interaction. 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 5.6 shows a typical interaction window. We define this interaction using the function $play(GAME) \rightarrow \; GAME$. Interaction is specified as:

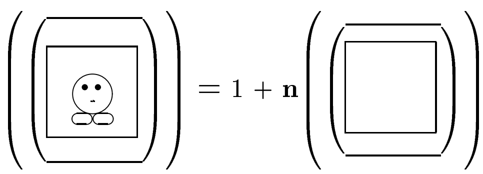
$$[p1] \quad \text{play}(\ \boxed{G}\ ) \quad = \quad \text{play}\ (\ \boxed{G}\ \boxed{\boxed{G}\langle DIR\rangle}\ )$$

which causes a direction to be repeatedly requested during rewriting.

5.5.5 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

[$t1$] **n** 

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

[2]Defining sub-terms externally, during rewriting, is uncommon for algebraic specifications. This is a recent extension to the VAS formalism to accommodate the high demands of interaction of visual languages (See Chapter 7).

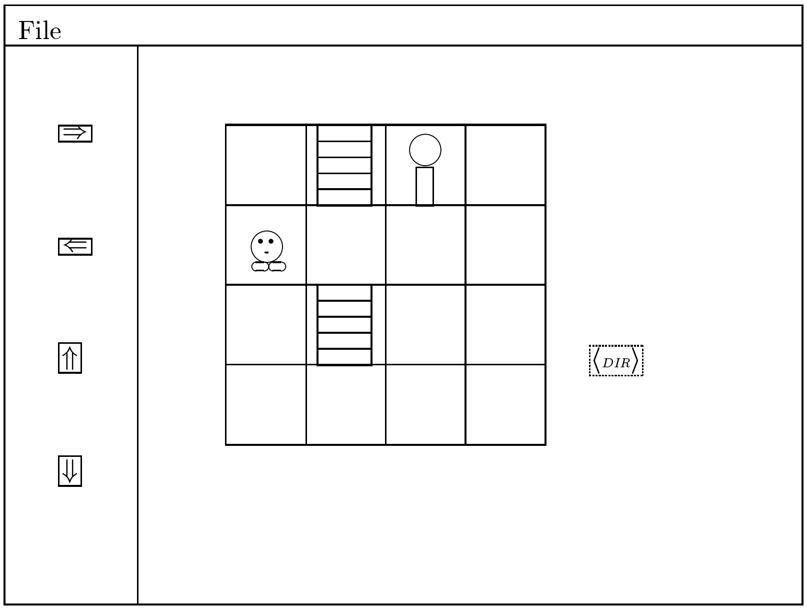


Figure 5.6: The game with interaction. The user selects the direction from the selection panel on the left.

$$[t3]\ \mathbf{1?}\,\boxed{\boxed{G}} = \text{true} \quad \mathbf{when\ n(}\,\boxed{\boxed{G}}\,\mathbf{)} = 1$$

$$[t4]\ \mathbf{1?}\,\boxed{\boxed{G}} = \text{false} \quad \mathbf{when\ n(}\,\boxed{\boxed{G}}\,\mathbf{)} \neq 1$$

It is intended to have a token of sort *GO* marking a finish location, where the game would terminate. Several other consistency checking equations, such as for the existence of a valid path from the player's location to the finish and the existence of the finish token. These are all very straightforward equations.

5.6 Related work

VASE is inspired by the ASF+SDF Meta-Environment [42] 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., [29]). 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 [86] has influenced the selection of the underlying representation for the collection sort. The PROGRES graph rewriting approach [71] uses attributed graphs with explicit directives for manipulating attributes. The conditional set-rewriting [56] is more general than these approaches, but for representing collections, we think Relational Grammars are better suited.

Programming environments such as VAMPIRE [52], PROGRES [71] 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.

In general many formalisms developed for visual language specifications are context-sensitive [49]. The VAS formalism allows basically context-free syntax definitions where the 2D lexicals are defined using VODL. The context-sensitive aspects are handled by a separate specification that serves as a "type-checker" (see Section 5.5.5).

5.7 Discussion

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 [83]. 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 and construction of collection terms—that need to be further studied.

The use of concrete syntax at the specification language level is an interesting approach. 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 chapter we represented sorts and meta-variables with text, which makes it difficult to represent spatial relationships that involve overlapping. This problem arises from the clear conflict between the concreteness of visual notation and abstract representations.

Another problem caused by using concrete syntax is that sometimes the function definitions might look different than when used. This is usually caused by the need for abstractness in the specifications. For specifications to be comprehensible they should be reasonably compact. Thus, the concrete syntax used must prefer representations that are small yet clear enough to represent the desired relations. At the same time care must be taken not to choose misleading representations that suggest relationships that are not part of the language. There are some interesting points in [84, 33] regarding consequences of representation choices. Analysis and detection of some spatial properties that could be misleading should be investigated at least for diagnostic purposes.

The well-definedness of the algebraic semantics of the specification in this case would assume certain well-behavedness of the function definitions. This well-behavedness would mean that, for any term that can be constructed during rewriting, the corresponding pictures do exist – a criterium that might be useful is the partial computability of the completion of a picture [16, 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 [18, 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 one by the manner in which it was constructed in a term editor (Chapter 6). 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.

5.8 Summary

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.

Chapter 6

Share-Where Maintenance

6.1 Introduction

The interactive construction of terms was discussed in Chapter 4, without much insight into the abstract representation or how rewritten terms are presented back to the user. In this chapter, we focus on the presentation of visual languages which involves the interaction with constructed sentences. This chapter discusses how we pretty print rewritten terms. In the next chapter we cover another aspect of interactive behavior which is input and output during term execution.

Until now we have examined how the syntax and semantics of two dimensional visual languages can be specified. The goal of specifying languages was for the generation of visual programming environments. We have introduced the VAS formalism for specifying visual languages. This formalism relies on abstract syntax trees for the underlying representation of visual sentences. Visual sentences are executed by rewriting their corresponding abstract syntax trees.

The result of a rewritten visual sentence is a term which must be pretty printed before it is presented back to the user. There are two main problems that arise when we want to pretty print the resulting terms. The first problem is that the information regarding the precise concrete properties of the visual representation are missing in the abstract syntax tree. The visual definition of the constructs is given in the language syntax specification, so we can always construct some correct visual sentence for presentation. However, there are typically numerous syntactically correct representations possible.

How should we select among permitted visual representations of terms? Furthermore, even if we could choose some arbitrary representation by some means, would that be acceptable? What if there are sub-sentences both in the initial and final sentences that represent identical terms? If the visual representations significantly differ there is a considerable risk that the observer will not be able to make the connection between them. Should the appearance of a picture change (even not so) significantly, it can lead the user to perceive identical sentences as different. This would defeat the use of visual notation as it is the recognition of visual patterns and relationships which we try to capitalize on in using such notation. In order to convey the same information we must maintain a certain stability in visual representations. For this reason we must try to maintain as much as possible of the visual appearances of sub-sentences that represent the same terms.

The second problem is related to sharing in terms. In order to represent two-dimensional syntax with conventional trees we extended the usual notion of structured editors by allowing sharing of sub-sentences (Section 3.6.2). Such sharing is reflected in the abstract syntax tree as copies. While this solves the problem of two dimensional sentence construction, it does not address how to present terms that have been rewritten so that the sharing present in the input sentence is reflected in the resulting sentence. Simply pretty printing such a term will result in each copy having its own separate visual representation rather than being shared. Both problems of pretty printing are most relevant when there are common subterms in initial and final terms so as to maintain the presentation of similar parts. However, the technique we will introduce also applies to terms with no common terms by providing a default visual presentation for newly constructed terms which are created by means of equations.

We will provide an example of a specification of the syntax and semantics of a visual language and its abstract syntax. Then we show how certain "Share" information that is produced by sharing certain sub-terms in an editor can be maintained and then used for pretty-printing the resulting term using the "Where" information.

6.1.1 Approach and Aim

We consider "context-free" specification of visual syntax – with the aid of visual (i.e., 2-dimensional and user-defined) "lexicals". This style is analogous to the classical BNF and SDF [42] approach used for 1-dimensional (textual) languages and gives a specification of allowable visual terms that are context-free. We have extended the classical notion of term building by allowing sharing of terms on user demand, in a syntax directed editor, so long as the constraints of the governing visual lexicals are not violated. We will demonstrate the power and limitations of this kind of sharing with the aid of examples.

Our basic approach to the pretty printing problem is to annotate the abstract syntax tree with information which allows the editor to construct output sentences using the information present in either or both of the input sentence and the semantic equations. In Figure 6.1 this process in shown with a finite state automata and character string as an input sentence and the result of its evaluation as an output sentence. We will consider this example in detail in Section 6.2. In the figure, A_{input} is the abstract term corresponding to the input sentence and A_{ouput}

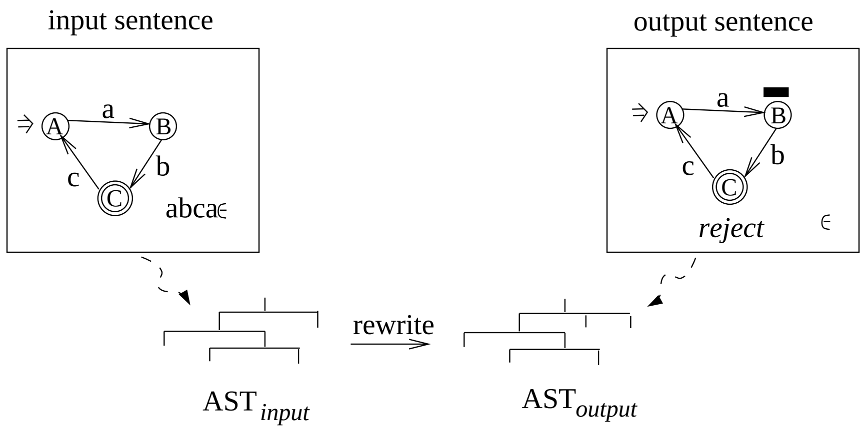


Figure 6.1: An input and output term with sharing.

is the abstract syntax of the rewritten term. The output sentence contains all of the sub-sentences of the input term plus a current state marker and a "reject" string.

Constraints present in the syntax definition govern the valid representations, which are a great many. In order to choose from these representations we will use the information that is provided by the user when the initial sentence was constructed. In the cases where sub-sentences are created by virtue of an equation during rewriting, we will use the representations from the equation which created it. Recall that the equations are also created in visual editors just like input terms. The information obtained from these sources will be used as a preferred appearance and location of the constructs and will be used by the constraint solver in attempts to reduce their search space.

For the purpose of maintaining the sharing in a term we introduce a method called *Share-Where* maintenance which annotates the abstract syntax tree with information regarding which terms are shared and where the shared term resides. The Share-Where annotations present both in terms constructed in end-user visual editors as well as in equations.

The annotations described in this chapter are used only for the purposes of presentation and have not been exploited for other purposes, such as to affect the rewriting. Other possible uses of Share-Where annotations will be mentioned in Chapter 9. Our aim is to examine how far we can push the utility of term representations. We have aimed at leaving the abstract representation unchanged as far as the term rewriting machinery is concerned and have annotated the tree for the sole purpose of presentation of terms. This is done by maintaining the sharing information that is created in the editor in terms of annotations which are only used by the editors when presenting and have no further impact on the formalism.

6.2 VAS specification

A VAS specification consists of three phases. The first is building the visual lexicals of interest using VODL which was presented in Chapter 4. The second uses these lexicals to define the context-free syntax of the visual language of interest. This phase constructs a mapping from abstract syntax constructs to the *vod*s. The third phase consists of defining the properties of interest, such as dynamic semantics, of the visual language using the visual syntax defined in the previous phase. Here we consider an example of a VAS specification, a deterministic finite state automata (*FSA*).

6.2.1 Visual lexicals

Figure 6.2 shows some of the *vod*s we use for defining the syntax of our *FSA* language. These *vod*s are all parameterized composite *vod*s, where a parameter is represented with a possibly subscripted v. The first five *vod*s will be used to represent some syntactic construct of an *FSA*: 1) normal state; 2) final state; 3) start state; 4) transition; 5) transition set and a start state. The fifth *vod* in fact defines a *vod* consisting of two *vod*s with no constraints between them, thus it is used only for relating them together. The last two *vod*s are used in the definition of the dynamic semantics of the language: 6) used in defining *vod* 7 and defining the output configuration of the fsa (*FSA-OUT-CONF* in Figure 6.5); 7) current state; 8) variable definition. These *vod*s are used in the next two sections where sorts serve as the parameters for these *vod*s.

6.2.2 Syntax definition

The syntax of finite state automata is defined in the module *FSA* which is shown in Figure 6.3. An automaton consists of a set of transitions along with a starting state and a collection of final states. The imported module *Strings* defines upper-case characters (*UChar*) and lower-case characters (*LChar*). Upper case letters are used as state labels and lower case letters are used for input characters. The sort L is a parameter for *vod*s 1 and 2 (see Figure 6.2). The sort *STATE* is a parameter for *vod*s 3 and 4 (twice for 4). The sort *ALPHA* is a parameter for *vod* 4. The *TRAN-C* defines a collection of transitions, which is used in the definition of *FSA* along with a *SSTATE*. Note that these representations follow from the constraint definitions which, themselves, have no physical appearance. For example, the fifth *vod* is shown as two vods that are horizontally next to each other which is purely coincidental and results from making some arbitrary choice in order to present.

	vod	Description
1	$\textcircled{v}$	a *vod* contained within a circle.
2	$\textcircled{\textcircled{v}}$	a *vod* contained within double circle.
3	$\Rightarrow v$	arrow followed by a *vod*.
4	$v_1 \xrightarrow{v_2} v_3$	an arrow connected to two *vod*s with a *vod* above.
5	$v_1\ v_2$	two unconstrained *vod*s
6	$\begin{matrix} v_1 \\ v_2 \end{matrix}$	one *vod* above another *vod*.
7	$\overset{\rule{2em}{0.6em}}{v}$	a *vod* with a rectangle above it.
8	$\boxed{v}$	*vod* in a rectangle.

Figure 6.2: Some of the *vod*s used in defining the FSA language.

module FSA

imports Strings

sorts *L FSA STATE SSTATE FSTATE TRAN TRAN-C ALPHA*

functions

$UChar$	$\rightarrow$	L
$LChar$	$\rightarrow$	$ALPHA$
L	$\rightarrow$	$STATE$
L	$\rightarrow$	$FSTATE$
$\Rightarrow STATE$	$\rightarrow$	$SSTATE$
$STATE \xrightarrow{ALPHA} STATE$	$\rightarrow$	$TRAN$
$FSTATE$	$\rightarrow$	$STATE$
$TRAN* \lhd \{\}$	$\rightarrow$	$TRAN\text{-}C$
$TRAN\text{-}C \quad SSTATE$	$\rightarrow$	FSA

Figure 6.3: The syntax specification for *FSA* language.

Such representations may lead one to assume there is some constraint that leads to such a presentation, whereas there are none.

In this module we find the syntactic description of the *FSA* language. This syntax will be used in term construction as well as semantic specification in the next section. Each such definition has a corresponding abstract syntax with an association between the two, permitting moving from one level to the other. All term rewriting is done at the abstract level. The abstract syntax is shown in Figure 6.4.

Abstract function names represent the concrete visual syntax used in module *FSA*. Here we use mnemonic names such as “st” to represent the circle with a label L and “ss” to mean start state. The functions are listed in the same order as in the module FSA. The Collection variable has no direct analog in ASF+SDF. Here we translate it into ASF+SDF lists. Such a direct translation is possible since these collections have no *COPS* constraining the elements in the collection (see Section 6.2.3).

The abstract syntax that would be produced by the visual syntax is given here.
imports AS-Strings
exports
sorts L FSA STATE SSTATE TRAN TRAN-C ALPHA
The abstract syntax names are arbitrarily choosen except for "list" and "coll" names that represent a pre-defined data type that the underlying rewriting machine is aware of.
context-free syntax

lab(UChar)	→ L
ip(LChar)	→ ALPHA
st(L)	→ STATE
fs(L)	→ FSTATE
ss(STATE)	→ SSTATE
tr(STATE, ALPHA, STATE)	→ TRAN
i(FSTATE)	→ STATE
coll(TRAN∗)	→ TRAN-C
fsa(TRAN-C, SSTATE)	→ FSA

Figure 6.4: The abstract syntax of the *FSA* module.

6.2.3 FSA Evaluation

To specify the dynamic semantics of *FSA* we first specify a syntax for the evaluation functions. Figure 6.5 shows the chosen syntax for *FSA* and defines a set of variables that may be used in the equations. The current state is represented by a rectangle above a state. The input stream consists of zero or more *ALPHA*s. The *eval* function takes an *FSA* and an input list (*ALPHA-L*) and steps through the automaton returning either *accept* or *reject* depending on whether it succeeds to end in a final state. The function *evit* iteratively processes the input string. This function uses a sort *FSA-CONF* as an *FSA* configuration which consists of an *FSA*, a *CSTATE*, and an *ALPHA-L* which are all needed to perform a single step.

Figure 6.6 shows the semantic equations for evaluating an *FSA* which takes an *FSA* and a character list and steps through it according to the transitions. The current state is represented by a black box above it. The semantics are defined by three functions that process the input string until there is no possible move left or the input string is consumed. After that, they examinesif that configuration is an *accept* of *reject* state.

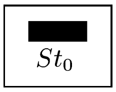
The declarations involving *TRAN*∗ $\lhd$ {} are about using the collection data type as defined in Section 4.2.2. *TRAN-C* is a collection of zero or more transitions (*TRAN*∗) with no constraining operations between the items in the collection ($\lhd$ {}). A collection variable matches a collection. Here for example, $\boxed{\text{Tr}}$ matches a collection with at least one transition Tr, where $\boxed{}$ matches the rest of the collection other than Tr.

The *eval* function starts the input string processing by replacing the start state by the current state. It then applies the *evit* function, the result of which is combined with the start state again. The reason for putting back the start state is to present it back in the result of evaluating the *FSA*. Otherwise it is not necessary. The *accept*? function decides whether the *FSA* configuration is in an acceptable state. The resulting term consists of the *FSA* configuration along with the string *accept* or *reject* positioned below the *FSA*. The intention is that seeing the state of the *FSA* is useful to observe, especially when the input string is rejected.

In equation [*st*] the evaluation of the *FSA* begins by setting up an *FSA-CONF* by making the start state a current state with a current state marker that will be used to step through the *FSA* according to the provided input string. The resulting term is constructed in a manner to convey some useful visual information. Instead of only indicating a result as *accept* or *reject* a visual term consisting of the *FSA*[1], the current state, the remaining input string, and the resulting status is constructed.

The equations [*e*1] and [*e*2] step through the input string moving the current state marker when a transition occurs. *FSA-CONF* does not have a *SSTATE* as it is not needed for the evaluation, furthermore it allows the demonstration of the Share-Where maintenance when $\Rightarrow St_0$ is put back in the resulting term.

In [*e*2] and [*a*2] we have used the variable *FsaCon*, we could have alternatively used a visual term like:



or equivalently


where the collection $\boxed{}$ matches any *TRAN-C* collection. Or we could have created a more visual variable representation for the variable instead of relying on a textual name. We chose this represenation here to provide some abstraction in terms of using a variable in the specification and avoid a visual variable representation for lack of finding an appropriate representation that is distinguishable from the terms. That is not to suggest that such represenations would not be appropriate. This is simply a choice of the language specifier.

Again, we provide the abstract syntax module (Figure 6.7) which Section 6.3.1 isi used to explain how this is further transformed into a specification that aids in displaying the result.

The *FSA* specification should be accompanied by another module that specifies what are type-correct *FSA*s. In our case, one can imagine checking for existence

[1] Actually the *FSA* here is represented with the *FSA-CONF* and the *SSTATE* is added.

module FSA-eval

imports FSA

sorts *CSTATE RESULT ALPHA-L FSA-CONF*

functions

[redacted]		
STATE	$\rightarrow$	*CSTATE*
ALPHA∗*	$\rightarrow$	*ALPHA-L*
accept	$\rightarrow$	*RESULT*
reject	$\rightarrow$	*RESULT*
TRAN-C CSTATE ALPHA-L	$\rightarrow$	*FSA-CONF*
SSTATE FSA-CONF RESULT	$\rightarrow$	*FSA-OUT-CONF*
eval (*FSA ALPHA-L*)	$\rightarrow$	*FSA-OUT-CONF*
evit (*FSA-CONF*)	$\rightarrow$	*FSA-CONF*
accept? (*FSA-CONF*)	$\rightarrow$	*RESULT*

variables

St	$\rightarrow$	*STATE*
i	$\rightarrow$	*LChar*
L	$\rightarrow$	*L*
Cst	$\rightarrow$	*CSTATE*
i∗	$\rightarrow$	*ALPHA-L*
$\boxed{TRAN\ast \lhd \{\}}$	$\rightarrow$	*TRAN-C*
Fsa	$\rightarrow$	*FSA*
FsaCon	$\rightarrow$	*FSA-CONF*

Figure 6.5: The specification of the syntax for the evaluation semantics of a deterministic *FSA*.

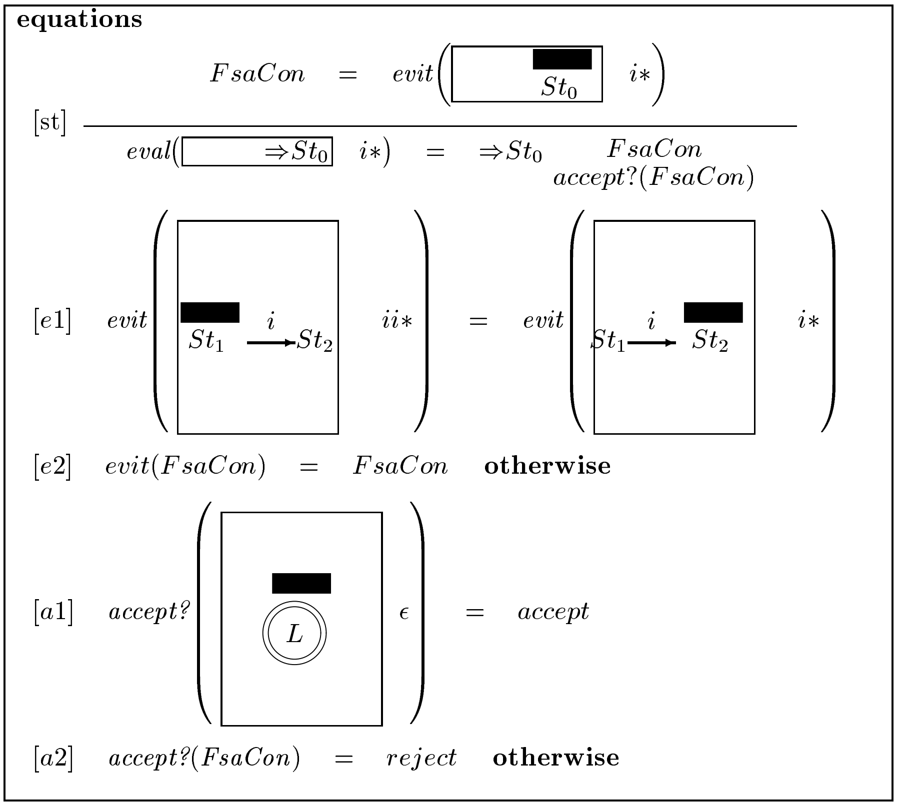


Figure 6.6: The specification of the evalution semantics of *FSA*.

module AS-FSA-eval
imports AS-FSA
exports
  sorts CSTATE RESULT ALPHA-L FSA-CONF FSA-OUT-CONF
  context-free syntax

cs(STATE)	$\rightarrow$ CSTATE
list(ALPHA$*$)	$\rightarrow$ ALPHA-L
acc()	$\rightarrow$ RESULT
rej()	$\rightarrow$ RESULT
fc(TRAN-C, CSTATE, ALPHA-L)	$\rightarrow$ FSA-CONF
foc(SSTATE, FSA-CONF, RESULT)	$\rightarrow$ FSA-OUT-CONF
ev(FSA, ALPHA-L)	$\rightarrow$ FSA-OUT-CONF
evi(FSA-CONF)	$\rightarrow$ FSA-CONF
ac(FSA-CONF)	$\rightarrow$ RESULT

  variables

"*St*"[*012*]	$\rightarrow$ STATE
"*i*"	$\rightarrow$ ALPHA
"*L*"	$\rightarrow$ L
"*Cst*"	$\rightarrow$ CSTATE
"*i$*$*"	$\rightarrow$ ALPHA$*$
"*Tr$*$*"[*012*]	$\rightarrow$ TRAN$*$
"*Fsa*"	$\rightarrow$ FSA
"*FsaCon*"	$\rightarrow$ FSA-CONF

equations

$$\frac{FsaCon = \mathrm{evi}(\mathrm{fc}(\mathrm{coll}(Tr_0^*),\ \mathrm{cs}(St_0),\ \mathrm{list}(i^*)))}{\begin{array}{l}\mathrm{ev}(\mathrm{fsa}(\mathrm{coll}(Tr_0^*),\ \mathrm{ss}(St_0)),\ \mathrm{list}(i^*))\ =\\ \mathrm{foc}(\mathrm{ss}(St_0),\ FsaCon,\ \mathrm{ac}(FsaCon))\end{array}} \quad \text{[st1]}$$

$$\begin{array}{l}\mathrm{evi}(\mathrm{fc}(\mathrm{coll}(Tr_1^*\ \mathrm{tr}(St_1,\ i,\ St_2)\ Tr_2^*),\ \mathrm{cs}(St_1),\ \mathrm{list}(i\ i^*)))\\ =\ \mathrm{evi}(\mathrm{fc}(\mathrm{coll}(Tr_1^*\ \mathrm{tr}(St_1,\ i,\ St_2)\ Tr_2^*),\ \mathrm{cs}(St_2),\ \mathrm{list}(i^*)))\end{array} \quad \text{[e1]}$$

$$\mathrm{evi}(FsaCon) = FsaCon \quad \textbf{otherwise} \quad \text{[e2]}$$

$$\mathrm{ac}(\mathrm{fc}(\mathrm{coll}(Tr_0^*),\ \mathrm{cs}(\mathrm{inj}(\mathrm{fs}(L))),\ \mathrm{list}())) = \mathrm{acc}\ (\ ) \quad \text{[a1]}$$

$$\mathrm{ac}(FsaCon) = \mathrm{rej}\ (\ ) \quad \textbf{otherwise} \quad \text{[a2]}$$

Figure 6.7: The abstract syntax of the *FSA-eval* module.

of at least one final state and for checking that no non-final state has the same label as some final state.

6.2.4 FSA Term Construction

FSA terms are constructed by providing a collection of transitions and a start state:⟨TRAN-C⟩ ⟨SSTATE⟩. According to the syntax definition the start state, *SSTATE*, may appear anywhere since it is unconstrained. We can, however, choose to make it share the representation of a particular state in the *FSA*. Figure 6.8 shows a construction sequence where such a sharing is chosen. The sharing choice is made in step f. Alternatively, ⟨SSTATE⟩ could have been replaced with a new state. Provided that the state was A, they would abstractly represent the same term. We do not allow a subterm to share its ancestor (no vertical sharing). In fact a type-checker would be needed to check whether the start state is one of the states of *FSA*. The type-checking issues are out of the scope of this paper.

6.3 Pretty Printing Issues

There are several general criteria to adhere to when displaying terms. We assume that visual representations were used in the first place to provide its user some information which benefits from graphical presentation. It would make no sense to present information in a manner which is incomprehensible such as say by obscuring the presented picture by overlapping sub-sentences. Similarly to this point there are many displaying principals which should be adopted to have a presentable pretty printed term. We are not addressing these general principles in this work. Graphic design principles should also provide many sound guidelines regarding these criteria.

One of the main advantages of multi-dimensional syntax is the ability to use shared representation in term presentation in depicting multiple relations that exist on a (sub)term. We have already demonstrated how such terms can be constructed. When we rewrite terms we use the abstract syntax of the term constructed in the editor. After rewriting the term we are left with an abstract syntax term that must be pretty printed for presenting back to the user.

The syntax definition of the language provides us with the information we need to obtain a visual term corresponding to the abstract term. However, there are some problems. First of all the *vod* definitions are typically under constrained and thus correspond to a great many pictures. Even if we choose some criteria to select among these pictures, it is likely that what we present will bear little similarity to the initial term. In fact this is sometimes fine since the resulting term has nothing common with the initial term. However, it is often the case that parts of initial terms remain in the resulting term. If the appearances of these common terms are significantly changed the observer will likely loose the connection between these

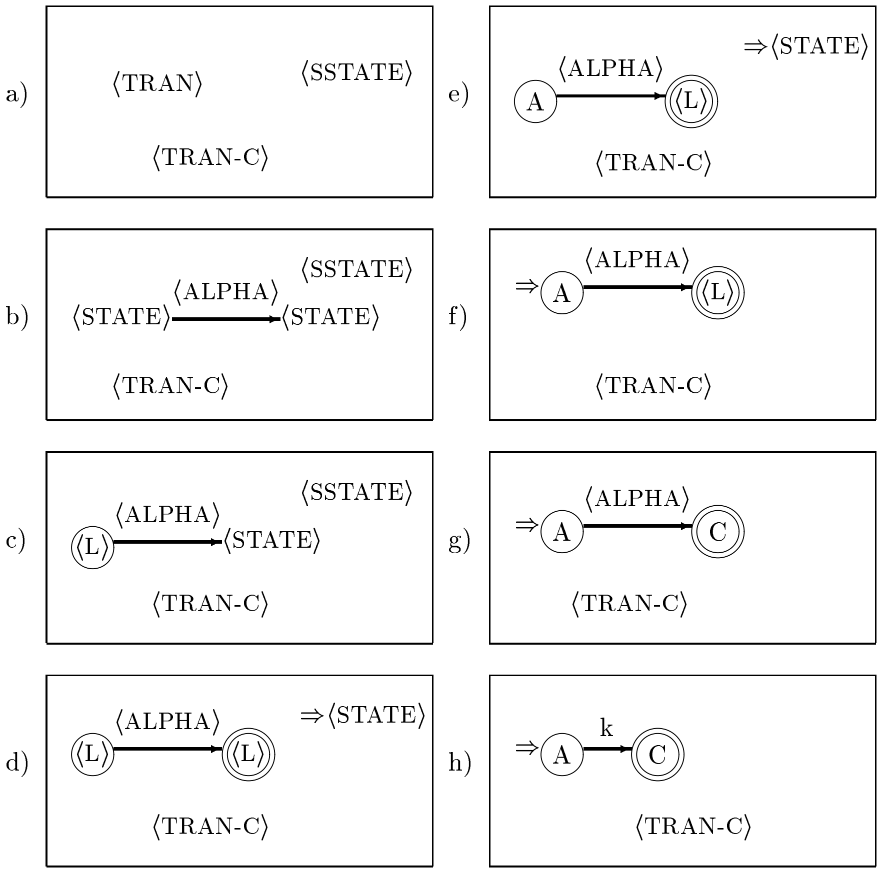


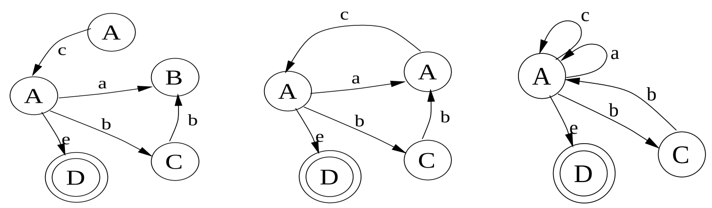
Figure 6.8: Initial steps of the construction of an *FSA* sentence. Note that in step f) the sharing of the initial ⟨STATE⟩ is established.

terms – which is unacceptable. We will focus on this aspect and propose a method whereby we maintain information from the the initial term and instance at which they were created during rewriting, to aid in the presentation of the final term.

The effect of pseudo sharing can be illustrated by the following equation:

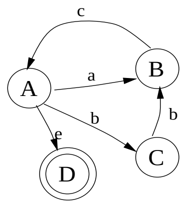
$$[p1] \quad f\left(\boxed{St_1 \xrightarrow{c} St_2}\right) \quad = \quad \boxed{A \xrightarrow{c} St_2}$$

and the following term represented in a visual form that emphasizes sharing:



(a) Pseudo (b) Real (c) Identity

Figure 6.9: Various sharing possibilities resulting from the application of equation [$p1$].



Applying the equation to this term results in a term where B is replaced by A. Depending on the underlying representation and interpretation there are choices to be made as how to present the term. Should A replace B? Should states be considered unique and thus have a single representation? The resulting term could be, among others, any of the terms shown in Figure 6.9. In this work we only consider pseudo sharing which represents each shared construct with a copy of it in the tree. Graph grammars [71] use real sharing which use graphs as underlying representations and shared presentations also are shared in the underlying representation. Identity sharing, where some criteria (say state label) determines the uniqueness of constructs, and its presentation is shared by giving a meta-level directive that all similar states should be shared. The sub-figure (a) illustrates the effect of pseudo-sharing and the application of function f. This rewrites a transition labeled "c" and ending in some state St_2 to a transition labeled "c" that begins in state A and ends in St_2. Note that all B states are not affected but only the one which matched the rule is rewritten into A. After the rewrite it is not clear where to print the A state created by the application of the rule. Using the approach we discuss below, it would be possible to print it close to B with an appropriate constraint solver.

6.3.1 Share-Where maintenance

This method annotates terms and propagates created annotations as a result of applying rewrite rules to the term. The underlying rewriting semantics is itself not influenced, since the annotations are not used for determining when a term matches another except for the shared variable case. Each term $f(\vec{x})$ is annotated as $f_{[l,t_i:p]}(\vec{x})$ where f is an abstract syntax function name, l is a label, t_i is an identifier of a term editor or an equation editor, and p is a path in t_i. When a term is initially created by a user, each node gets a unique label, t_i is the name of the term editor that created the term and p is the path to the node. However, when certain sub-terms are shared the share annotation of the sharer is copied to the sharee. The Share-Where annotation uniquely identifies the sub-terms modulo sharing. This allows the (re)creation of shared representation when displaying the term.

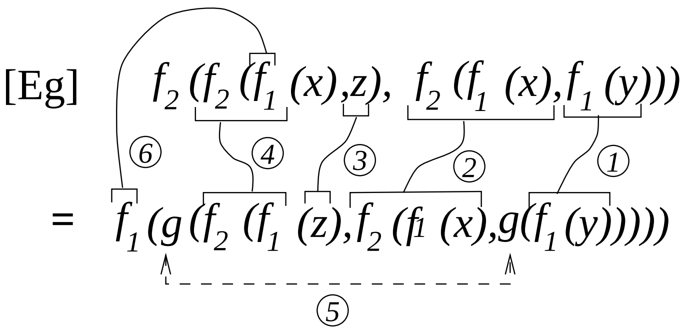
The idea is to maintain information regarding how terms are created and where the creator resides. Having this information allows us to have a starting point for presenting terms as close as possible to how they were constructed. One of the major problems in pretty printing is how to treat newly created sub-terms. This is addressed by preserving references to the equations that create the terms. Thus, newly introduced terms will appear as they appeared as much as possible in the equations that introduced them. Such information provides good starting points for presenting the term. We say starting point, since clearly, the chosen representations may result in constraint violations in a larger term. However, they still give good starting points or "hints" to the solver indicating where approximately they should be placed.

Initially, we will introduce "primitive" Share-Where maintenance. This consists of analyzing the equations for propagation of Share-Where information in the following four cases in order of preference (Figure 6.10 may be used as a reference example):

1. *Common Sub-term in maximal common sub-context:* Relate corresponding nodes of the common sub-terms, i.e. terms common to both left hand side and right hand side of an equation. If a sub-term on the right hand side can be related to more than one of the left hand side (i.e. $f_1(x)$) then look for the maximal sub-context in which they appear, in order to arbitrate.

2. *Common sub-context (maximal):* Look for sub-contexts on the right hand side that correspond to similar sub-contexts on the left and relate them by giving priority to larger matches.

3. *Introduced symbols:* The new symbols introduced on the right hand side get new share annotations. If these are explicitly shared in the editor then they get the same annotation (i.e. the function g).



① ③ : common sub-term
② : common sub-term in a common sub-context
④ : common sub-context (maximal)
⑤ : introduced symbols
⑥ : arbitrary choice (common sub-context)

Figure 6.10: Various Share-Where relations in an equation.

4. *Trivial arbitration:* These are common variables and function symbols that remain unrelated by the above rules. These could then be arbitrarily related to one of the choices on the left[2].

In the above we have used the some terminology that should be explained:

- sub-term: is a term that is a part of another term.
- common sub-term: is a sub-term that is both on the left and the right hand side of an equation.
- context: is a term with one or more 'holes'.
- sub-context: is a context that yields a sub-term when its 'holes' are filled.
- common sub-context: is the case when a sub-context is common to both the left and right hand sides of an equation.
- maximal common sub-context: are the largest possible sub-context that is common to both the left and the right hand side of an equation.

[2]For function symbols (the f_1 case in figure) this could lead to same share information between a parent (ancestor) and a child, however this does not imply pseudo sharing. These cases could also be better handled by dependence labeling.

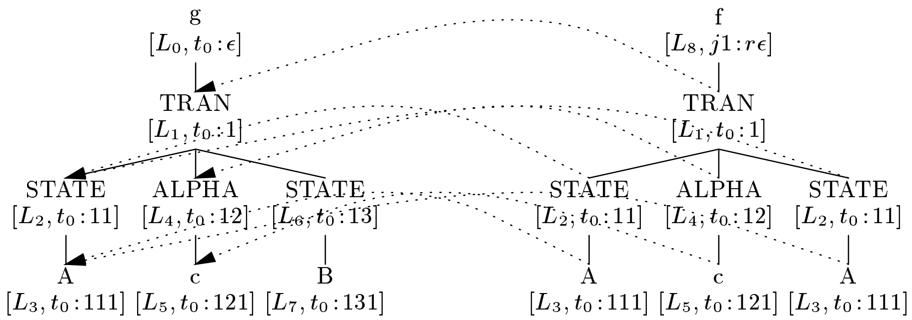


Figure 6.11: Share-Where annotations for a term and propagation after rewriting. The Where information "t_0 : 12" means that the node was created in term t_0 at occurrence 12 (second child of first child of top node).

6.3.2 Share-Where maintenance during evaluation

To see how the Share-Where information is determined and propagated we consider the simple term:

$$g\left(\text{A} \xrightarrow{\text{a}} \text{B} \right)$$

and a simple equation:

$$[j1] \quad g\left(S_1 \xrightarrow{i} S_2 \right) \quad = \quad f\left(S_1 \circlearrowleft^{i} \right)$$

Figure 6.11 shows the Share-Where annotations for the initial term and equation $[j1]$ is used to rewrite it. The dotted lines show how the shared sub-terms on the right-side have the same origin for their representations. The Where information "t_0 : 12" means that the node was created in term t_0 at occurrence 12 (second child of first child of top node). The *ALPHA* and *STATE* sub-terms are unchanged and thus have their Share-Where annotations from the left side (common variable rule). The "TRAN" node also has the same annotation due to common sub-context rule. The symbol f is a new symbol created by the equation. The new annotation $[new(), j1 : r\epsilon]$ says: a unique label[3] is generated (by function uid) for the Share part and the Where part tells that this f was created in equation $[j1]$ at the right-hand side (r) top (ϵ) occurrence. Note that if instead of the shared S_1s on the right, they were introduced constants, they would both share the same $new()$s every time the rule is applied.

The annotations are determined by the translation of equation $[j1]$ to one that is appropriately decorated with Share-Where annotations as follows:

[3]New Share annotations need an unique identifier, otherwise too many symbols would be shared while rewriting. However, unique identifiers cause too many symbols being non-shared. See Section 6.3.3.

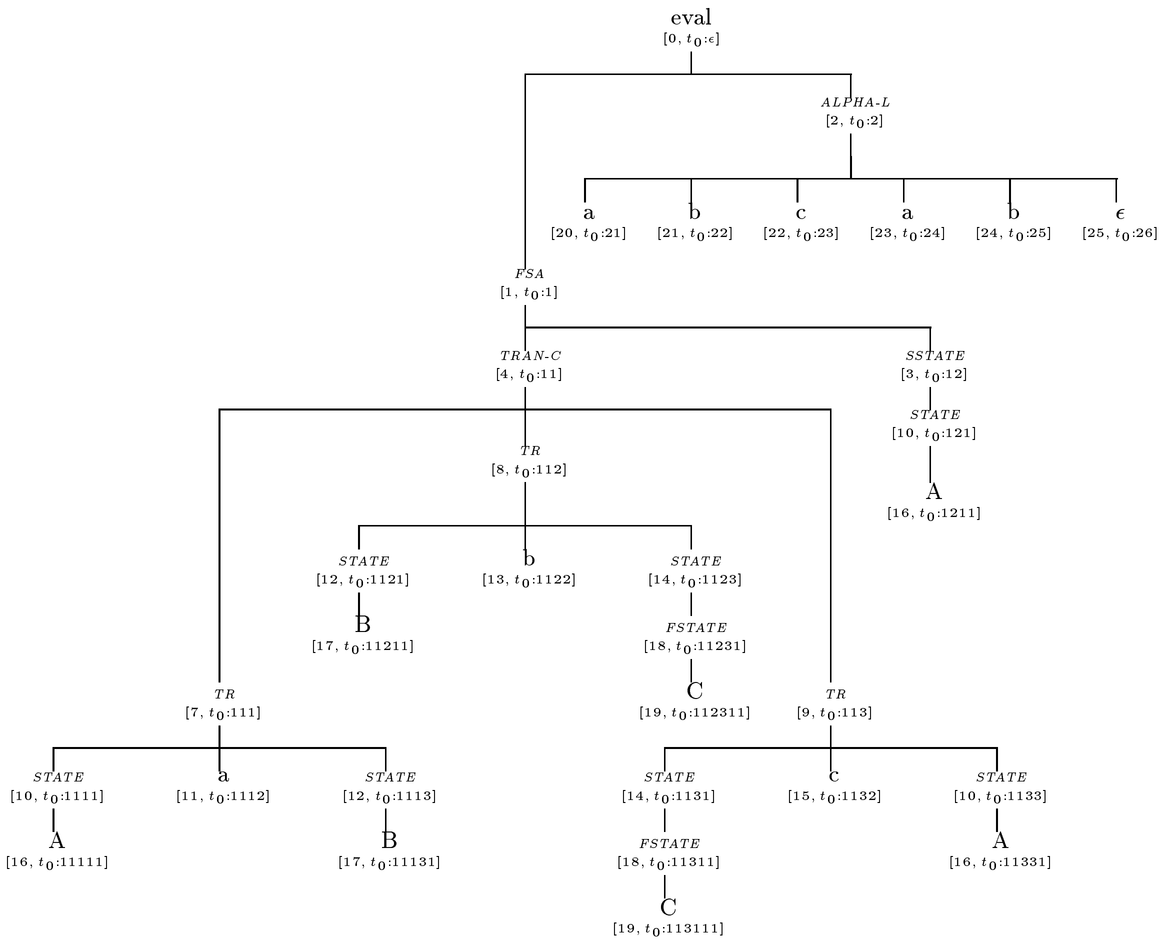


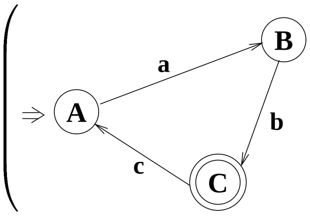
Figure 6.12: Annotated abstract syntax tree of the function eval applied to an *FSA* term.

$$[\mathrm{j1}_{sw}] \quad g_{\ell_0}(\ \mathrm{S}_1 \xrightarrow{i} \mathrm{S}_2\ \ _{\ell_1}) \quad = \quad f_{[uid(),j1:r\epsilon]}(\ \mathrm{S}_1 \xrightarrow{i} \mathrm{S}_1\ \ _{\ell_1})$$

where the variables ℓ_i match the Share-Where annotations. All annotations in i and S_1 are propagated (common-variables) and a new annotation is created for the transition created on the right-hand side. The new annotation $[uid(), j1 : r\epsilon]$ says: a unique label is generated (by function uid) for the Share part and the Where part tells that this transition was created in equation $[j1]$ at the right-hand side (r) top (ϵ) occurrence.

Now we can look at an *FSA-OUT-CONF* term. Consider the following *FSA-CONF-OUT* term:

$eval$ (



$abcab\epsilon$)

The abstract syntax tree representation of this term is shown in Figure 6.12.

Figure 6.13 shows how the term is rewritten and some of the annotations of the nodes resulting from the rewriting. These annotations are shown with the names of nodes. For example, the first set of annotations corresponds to the four states labeled A (two under transitions, one under start state, and one under current state) and the *CSTATE* node. In all annotations for the state A the labels are identical denoting that they all share the same representation. All nodes in the term are created in the term editor except for *CSTATE* which is created by equation $[e1]$ of the *FSA-eval* module. The annotation could use full names (including the module information) in practice, but here we use the short names. Note that the paths seen here are also identical which may lead one to think that the path information alone is sufficient to indicate sharing. This is, however, not the case as application of the same rule always results in the same path in an annotation and does not imply the presence of any sharing.

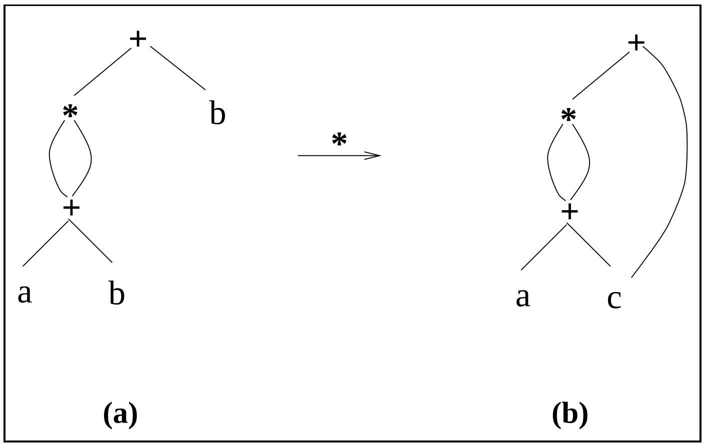
6.3.3 Dependence labeling

"Primitive" *Share* maintenance does not accommodate sharing of the symbols introduced in the equations (unless they are shared in the equations editor). This often results in undesirable effects in the output picture. For example, Figure 6.14-b shows what happens when the equation

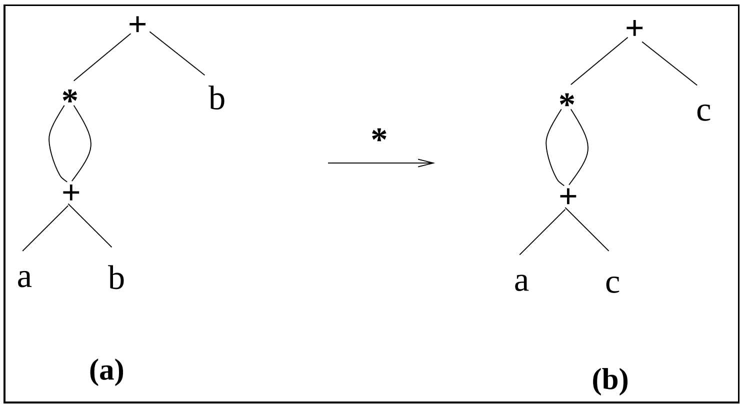
$$[eg1] \quad \mathrm{b} \quad = \quad \mathrm{c}$$

is applied to the sentence in Figure 6.14-a. Whereas one would desire that the generated "c" symbols are also shared in the output picture as in Figure 6.14-c.

On the other hand if the newly introduced symbols get the same share labeling (instead of a new label on the fly) there would likely be unintended symbols shared. For example, observe what happens when the equation $[eg1]$ is applied to (a):



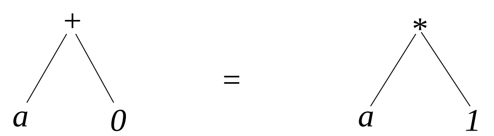
To address this problem, we introduce dependence labeling, where the share labeling in the introduced symbols depend on certain labels of some symbols on the left-hand side. In $[eg1]$ the labeling of c could be made to depend on the label of b. Instead of 'c' getting a new() share annotation, a unique identifier, the share annotation for 'c' could be generated using the share annotation of 'b' and the where annotation of 'c' – e.g. $new(share(b), share(c))$. Thus generating identical annotations for the desired 'c's. Then the following (a) term, could after rewriting be presented as (b):



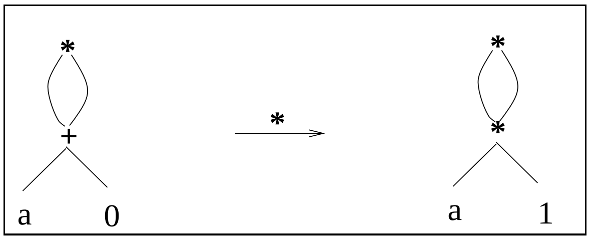
Dependence labeling appropriately replaces the "introduced symbols" rule above with two new cases:

- *redex-contractum case:* the labeling of the introduced symbols depend on the labeling of the top symbol of the redex and a label associated with the occurrence of the symbol. If an occurrence is shared, then the associated labels are identical. In

[$eg3$]

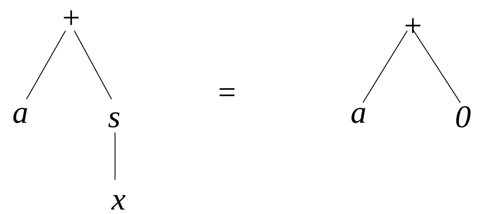


the labels of $*$ and 1 depend on the label of $+$.

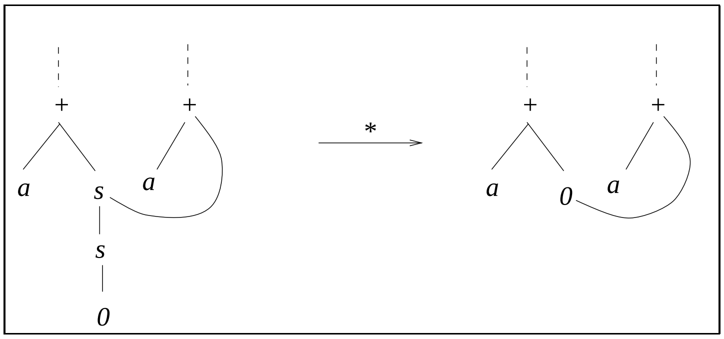


- If the common sub-context can narrow down the dependency to a more specific sub-term then the labeling of the introduced symbol depends on that sub-term. For example, in the equation

[$eg4$]



the label of 0 depends on the label of s. Thus:



Dependency labeling assures that when the term-rewriting system is non-overlapping, shared parents implies shared children.

- Shuffling a picture is okay. When no new symbols exist then no sharing is destroyed.
- New nodes are okay. Share annotations in new nodes depend on the annotations of the nodes that are responsible for their creation.

6.3.4 Discussion

Now let's consider a case which introduces complications. Let an equation $j2$ be defined as follows:

$$[j2] \quad id\left(\boxed{S_1 \xrightarrow{i} S_1}\right) \quad = \quad \boxed{S_1 \circlearrowleft \boldsymbol{i}}$$

Figure 6.15(a) shows the initial term and the parts (b) and (c) show two possible pretty printing results depending on which S_1 from left-hand side passes the Share-Where annotations to the S_1 on the right. Although these two terms look considerably different their abstract syntax is equivalent and thus share the same semantic behavior. We may consider the representation in Figure 6.15(c) visually undesirable as it gives the appearance of a disconnected *FSA* although its behavior is correct. The problem here comes from determination of which A from the left side is to be chosen as being the representation corresponding to the A on the right side.

One approach to deal with this could be to collect a set of labels rather than choosing only one. Examining the subset relation could allow the determination of "identity" sort sharing. However, one would still need to figure out which one to chose among the set as the representation and different choices would result in different looking pictures. We consider this approach as an extension to the "primitive" Share-Where, which only concerns itself with a simple propagation of Share-Where information for gaining the ability to pretty print terms with shared sub-terms.

Clearly, the Share-Where information could be used more extensively (i.e. during matching) and more sophisticated Share-Where information can be collected for better pretty printing results. One of the immediate applications of this information is their use in pretty printing rules which could be specified by the language designer as a separate module and could be used to pretty print the terms in a language-specific preference. The pretty printing issues addressed in this paper would then be considered as default pretty printing. Such a pretty printer would be able to track colored (marked) input constructs and help in debugging specifications by indicating where a certain part of the output term originated from.

6.4 Implementation Ideas

The implementation of Share-Where maintenance can be done by transforming the original TRS (R) to another TRS (R_{sw}) that reduces a Share-Where annotated term. Such a transformation would preserve the essential properties of the original TRS R. If a term t reduces to t' by R the term t with Share-Where annotations t_{sw} would be reduced to t'_{sw} by R_{sw} and removing the Share-Where annotations would yield t':

$$\begin{tikzcd} t \arrow[r, "add\ sw"] \arrow[d, "R"'] & t_{sw} \arrow[d, "R_{sw}"] \\ t' & t'_{sw} \arrow[l, "remove\ sw"'] \end{tikzcd}$$

In [79] van Deursen specifies a library of functions necessary for such transformations of TRSs. He uses these functions to implement origin tracking. We need some additional functionality to detect maximal sub-contexts (see Section 6.3.1) to implement Share-Where maintenance proposed in this chapter. Such a transformer would transform the abstract syntax shown in Figure 6.7 to the transformed module. This work transforms a term rewriting system into another one with annotations.

In our case the origin relation would be defined for the Share-Where annotations. For the abstract syntax shown in Figure 6.7 the transformed module would have the annotations shown in Figure 6.16.

6.5 Related Work

Typically visual languages are syntactically very complex. The sharing of terms among several terms leads one to consider context-sensitive grammars for specifying visual languages. Most approaches to syntax specification have adopted context-sensitive formalisms considering context-free specifications not useful in dealing with realistic languages. This point can be observed in [50] which details formal approaches to visual language specification. These approaches tend to define other language aspects such as type-checking and semantics along with the syntactic definition. Contrary to this approach we wish to examine language specification style which separates these two aspects clearly and pushes the definition of the semantics, type-checking and so on to the domain of semantic specification.

The work of constraint multi-set grammars [50] and graph grammars [71] are alternative approaches to the algebraic specification approach we have considered. Our approach is an exercise in using the traditional algebraic specification formalisms to define visual languages by allowing two-dimensional lexicals where as both the constraint multi-set grammars and graph grammars attempt to define context-sensitive language formalisms for the purposes of defining visual syntax.

The ASF+SDF Meta-environment [42] has been the source of many inspirations for our work on VAS formalism. The work on origin-tracking [18, Chapters 7 and 9] and error reporter generation [18, Chapter 4] in the ASF+SDF context has further inspired the maintenance of Share-Where information. Our Share-Where information is in some ways more general than origin-tracking due to the presence of the Where information; as the labels used in Share annotations could help provide "primitive" origins in the absence of sharing. These techniques are related to residuals and descendants as can be found in [43, Chapter 8] and subject-tracking defined by [10].

Our dependence labeling is simpler than the dependence tracking [22], since we are only interested in the dependence of labels on other labels and not on the function symbols.

6.6 Summary

In this chapter, we have, by means of an example, elaborated on the utility of visual algebraic specifications for defining visual languages. We have introduced Share-Where, with which we can use the information about the initial term construction, in the style of "programming by example", to present the result of computation back to the user.

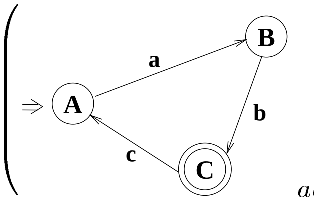
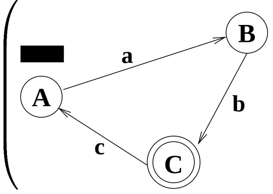
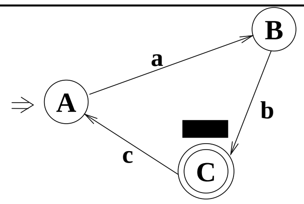
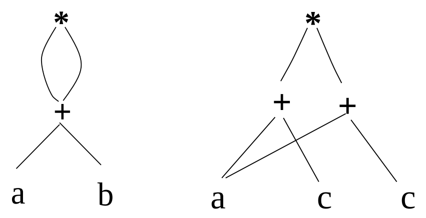
Term	Eq	Match	Annotations
eval ( $abcab\epsilon$)	$[st]$	$St_0 =$ (A) $i* \;= abcab\epsilon$	A_{AB}:$[10, t_0$:1111] A_{CA}:$[10, t_0$:1133] A_{ss}:$[10, t_0$:121] A_{cs}:$[10, t_0$:1111] cs:$[L_1$, *e1:c1:r*12] ss:$[3, t_0$:12] $list$:$[2, t_0$:2]
evit ( $abcab\epsilon$)	$[e1]$	$St_1 =$ (A) $St_2 =$ (B) $i \;= a$ $i* \;= bcab\epsilon$	A_{AB}:$[10, t_0$:1111] B_{AB}:$[12, t_0$:1113] B_{BC}:$[12, t_0$:1121] B_{cs}:$[12, t_0$:1113] cs:$[L_1$, *e1:c1:r*12]
	⋮		
 *accept* ϵ			C_{BC}:$[14, t_0$:1123] C_{CA}:$[14, t_0$:1131] C_{cs}:$[14, t_0$:1123] cs:$[L_1$, *e1:c1:r*12] ss:$[3, t_0$:12]

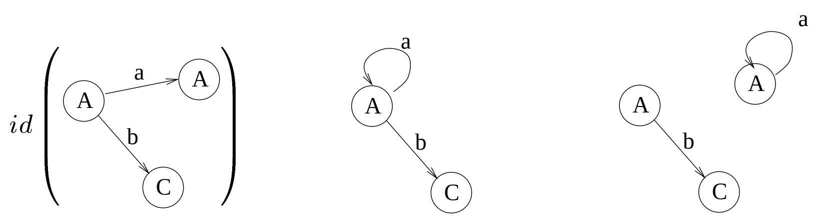
Figure 6.13: Some annotations during the evaluation of an *FSA* term

S_{xy} stands either for the state (S) in the transition from state x to state y or the start state or the current state it belongs to. Some interesting annotations (in reduct) are listed in the "Annotations" column. Note that all As, Bs and Cs maintain their input-term annotations. The annotations of current state marker (indicated by "cs") is provided by the right-hand side of equation $[e2]$, but the annotation of its STATE sub-term (indicated by subscript cs) is obtained from the match information. Also see Figure 6.7. L_1 is a label generated during the application of equation $[st]$.



(a) initial. (b) resulting. (c) desired.

Figure 6.14: Terms with shared parents and children.



(a) initial term. (b) left prefer. (c) right prefer.

Figure 6.15: Pretty printing results based on two different Share-Where annotations of same term.

equations

These are the transformed equations of AS-FSA-eval to contain Share-Where information.

$$\frac{\begin{array}{c}\ell_5 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{st1} : \texttt{cr1} : \text{r } 1],\\ \ell_6 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{st1} : \texttt{cr1} : \text{r } 11],\\ \ell_7 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{st1} : \texttt{cr1} : \text{r } 12],\\ \ell_8 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{st1} : \text{r } 0],\\ \ell_9 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{st1} : \text{r } 12],\\ \mathit{FsaCon} = \text{evi}_{\ell_5}(\text{fc}_{\ell_6}(\text{coll}_{\ell_2}(\mathit{Tr}^*), \text{cs}_{\ell_7}(\mathit{St}_0), \text{list}_{\ell_4}(i^*)))\end{array}}{\begin{array}{l}\text{ev}_{\ell_0}(\text{fsa}_{\ell_1}(\text{coll}_{\ell_2}(\mathit{Tr}^*), \text{ss}_{\ell_3}(\mathit{St}_0)), \text{list}_{\ell_4}(i^*)) \ = \\ \text{foc}_{\ell_8}(\text{ss}_{\ell_3}(\mathit{St}_0),\ \mathit{FsaCon},\ \text{ac}_{\ell_9}(\mathit{FsaCon}))\end{array}} \quad \text{[st1]}$$

In the AS-FSA-eval [e1], $St1$ and i are repeated on the left hand side in order to transform this in a way that Share-Where does not affect matching, we duplicate the variables referring to $St1$ (and similarly i) and check that their abstract syntax without Share-Where matches using the function AS-eq.

$$\frac{\begin{array}{c}\mathit{TranC}_0 = \text{coll}_{\ell_2}(\mathit{Tr}_1^*\ \text{tr}_{\ell_3}(\mathit{St}_{11},\ i_{11},\ \mathit{St}_2)\ \mathit{Tr}_2^*),\\ \text{AS-eq}(\mathit{St}_{11},\ \mathit{St}_{12}) = \texttt{true},\\ \text{AS-eq}(i_{11},\ i_{12}) = \texttt{true}\end{array}}{\begin{array}{l}\text{evi}_{\ell_0}(\text{fc}_{\ell_1}(\mathit{TranC}_0, \text{cs}_{\ell_4}(\mathit{St}_{12}), \text{list}_{\ell_5}(i_{12}\ i^*))) \ = \\ \text{evi}_{\ell_0}(\text{fc}_{\ell_1}(\mathit{TranC}_0, \text{cs}_{\ell_4}(\mathit{St}_2), \text{list}_{\ell_5}(i^*)))\end{array}} \quad \text{[e1]}$$

$$\text{evi}_{\ell_0}(\mathit{FsaCon}) = \mathit{FsaCon} \quad \textbf{otherwise} \qquad \text{[e2]}$$

$$\frac{\ell_6 = [\text{new}\ (\ ), \texttt{Fsa-eval} : \texttt{a1} : \text{r } 1]}{\text{ac}_{\ell_0}(\text{fc}_{\ell_1}(\text{coll}_{\ell_2}(\mathit{Tr}^*), \text{cs}_{\ell_3}(\text{fs}_{\ell_4}(L)), \text{list}_{\ell_5}())) \ = \ \text{acc}_{\ell_6}(\ )} \quad \text{[a1]}$$

$$\frac{\ell_1 = [\text{new}\ (\ ), \texttt{FSA-eval} : \texttt{a2} : \text{r } 1]}{\text{ac}_{\ell_0}(\mathit{FsaCon}) \ = \ \text{rej}_{\ell_1}(\ )} \quad \textbf{otherwise} \qquad \text{[a2]}$$

Figure 6.16: Equations of the module *SW-FSA-eval* which is a transformation of module *FSA-eval* to Share-Where annotated abstract syntax. Some conditions used here are only to make the equations fit in the page. Also, here module names are indicated in the annotations.

Chapter 7

Interaction Specification

7.1 Introduction

The term rewriting described so far lacks the possibility of interaction. This is a serious drawback for specifying languages that are interactive by nature, which includes most practical languages in general and most visual languages in particular. The usual manner of executing a sentence, which is constructed in an editor, is to take its corresponding term and to rewrite it until it reaches a normal form. The resulting term is pretty printed in order to present it back to the user. One aspect that is strongly missed is *interaction during the rewriting* of a term. We address this problem, with a simple operational model of interaction. Note that we will only describe our interaction model as an extension of term rewriting and that we don't attempt to give an algebraic interpretation to it.

The aim of the work presented in this chapter is hence to examine the utility of extending term rewriting in a way to support interaction. The extension of the visual specification formalism for input and output not only permits interaction during execution, but als makes a user definable specification of a language interface itself possible. We examine the utility of a such an extension in the presence of the generated visual term editors by using only the generic features of these term editors.

We argue that the input part of interaction with a user is nothing but constructing terms in editors. Thus, during execution, the user will be presented with terms whose construction must be completed. The particular appearance of such editors is delegated to a separate user-interface specification.

We will first describe the extension with interaction by means of a toy specification. Section 7.2 provides an algebraic specification highlighting two of the concepts used in our extension. Section 7.3 describes the extension for interaction. Then we will demonstrate its utility with two examples. In Section 7.4, we give a detailed example of a simple calculator that demonstrates our approach. In Section 7.5 we revisit the *FSA* language, which was discussed in Chapter 6, by specifying an interactive animator that illustrates the use of *Share-Where* for presentation of terms during input and output. We conclude this chapter with related work in Section 7.7.

7.2 Rewrite Systems

Consider the following signature that describes a language:

sort B
functions

$$\begin{array}{rcl} zero & \rightarrow & B \\ one & \rightarrow & B \\ B \diamond B & \rightarrow & B \end{array}$$

One can think of these as BNF rules (reading right to left) where B denotes a non-terminal and *zero*, *one* and $\diamond$ denote terminals. With the above signature we can construct *terms* of the form[1] $zero$, one, $(zero \diamond one) \diamond zero$, $\cdots$.

7.2.1 Conditional Rewrite Rules

A conditional equation has the form

$$\frac{s_1 = t_1, \cdots, s_n = t_n}{s_0 = t_0}$$

with $n \geq 0$, and s_i, t_i $(0 \leq i \leq n)$ terms. Such an equation can also be read as a conditional rewrite rule by considering that s_0 rewrites to t_0 when the conditions are satisfied, i.e., when s_i and t_i can rewrite to the same (normal) form $(1 \leq i \leq n)$. Usually, some well-definedness constraints are imposed on the variables of the conditions in order to ensure their definedness during the execution.

For example, the following oriented (unconditional) rules describe the semantics of the language B.

$$zero \diamond x = zero$$

$$one \diamond x = x$$

where x is a variable over the sort B, which we sometimes write as: $x \rightarrow B$.

[1]The parentheses are not a part of the syntax but are used to indicate the underlying structure.

7.2.2 Meta-variables

Meta-variables are placeholders available during syntax-directed editing. They represent the holes in incomplete terms. A hole of sort S is represented by $\langle S \rangle$ and can be replaced with any term of the sort S. They allow interactively building the intended term by choosing among permissible substitutions of the language constructs. In a syntax directed editor, multiple occurrences of place-holders of the same sort are independent of each other. E.g., in an editor for building a B term, $\langle B \rangle \diamond \langle B \rangle$ is a B term under construction where the two $\langle B \rangle$s represent separate (unrelated) place-holders. In our explanation of interaction below, we use constants that look like meta-variables, but any (introduced) constant would suffice.

7.3 Interaction

We describe how an algebraic specification formalism, interpreted as a term rewriting system, can be extended to accommodate interaction. Briefly,

1. we introduce "χ-terms" for the purpose of interaction,
2. terms are rewritten using multi-stage rewriting, where each stage uses ordinary rewriting, and
3. between stages, an external process helps remove the "χ-terms" by filling-in some holes — thus modeling interaction.

7.3.1 Motivation

The situation and the extension are illustrated by a toy example. Consider the following set R of oriented (conditional) equations (i.e., rules):

$$\mathit{zero} \diamond \mathit{one} = \mathit{zero}$$

$$\mathit{zero} \diamond \mathit{zero} = \mathit{zero} \diamond y$$

where y is a variable over the sort B.

This is not a term rewriting system in the usual sense as the second rule introduces y, a new variable on the right-hand side. However, as we will show, this extension (in some form or other) of the notion of term rewriting systems is essential in our case. If we start reducing a term $\mathit{zero} \diamond \mathit{zero}$, it matches the left-hand side of the second rule and this term will be rewritten to say $\mathit{zero} \diamond y'$, where y' is a renaming of variable y — different from other existing variables. Now, for further reduction of this term, it has to match one of the left-hand sides again[2]. The unbound variable y' matches neither *one* (in the first rule) nor *zero* in the second rule. Therefore, the term $zero \diamond zero$ reduces to $zero \diamond y'$ and the reduction stops. It can be restarted by concretizing y' to any valid term (any term of sort B). This narrowing substitution that happens external to the rewriting essentially models input.

In this case, rewriting continues as long as *zero* is entered interactively and stops as soon as *one* is entered, terminating the interactive reduction process. We can denote such a situation by:

$$zero \diamond zero \quad \xrightarrow[zero*\ one]{*\quad R} \quad zero$$

where $zero \diamond zero$ is the initial term, $\xrightarrow{*\,R}$ denotes the multi-step reduction relation over R, $zero*\ one$ is a regular expression describing the input sequence, and *zero* is the resulting normal form.

7.3.2 Input

We will use a notation that helps capture the need for input explicitly (instead of the unbound y discussed earlier). A term of form $\chi(\langle B\rangle)$ is used in place of y. This is defined by *additional* syntax[3]:

functions

$$\begin{array}{ccc} \text{“}\langle B\rangle\text{”} & \to & B \\ \chi(B) & \to & B \end{array}$$

The "χ" captures a desire that only an "external" process can narrow its contents – which one could interpret as what happens in an interaction.

We write the above rules as follows:

$$zero \diamond one = zero$$

$$zero \diamond zero = zero \diamond \chi(\langle B\rangle)$$

Note the use of $\chi(\langle B\rangle)$ in place of y. Since y was declared to be of sort B, $\langle B\rangle$ indicates a place holder which needs to be filled in. The semantics of term

[2] Since y' is an unbound variable, the occurance of y' in term $zero \diamond y'$ could unify with either *one* in the first rule or with *zero* in the second rule. Taking into account both these possibilities is the subject of narrowing based term rewriting systems. However, in a typical term rewriting system only matching and no narrowing is present.

[3] $\langle B\rangle$ is a constant in the domain of specification, but a meta-variable in the domain of editors.

rewriting is unchanged resulting in a normal form of the term with occurrences of $\chi(\langle B\rangle)$ in it.

The function χ projects the term that was used as a replacement for the place-holder in its argument, after an interaction. Thus the narrowing substitution rule

$$\chi(\langle B\rangle) = zero$$

would result when the place-holder $\langle B\rangle$ is filled by *zero*. In the example above, after the interaction, the rule would rewrite all occurrences of $\chi(\langle B\rangle)$ to *zero* and the process of reducing the resulting term continues using the rules R.

Given a term t and a set of rules R, let $reduce(R, t)$ denote the normal form of the term t obtained by rewriting the term t using the rules R until no further reduction is possible. An interactive reduction of a term t, given a set of rules R is then defined as:

$$interact(R, t) = reduce(R, t) \textbf{ when } \chi\text{-free}(reduce(R, t)) = true$$

$$interact(R, t) = interact(R, reduce(I_i, reduce(R, t))) \textbf{ otherwise}$$

where I_i denotes the ith interaction and the predicate χ-free(t) checks if the term t has any χ-terms.

Note that according to the above definition, regardless of the point in which a request for interaction arises, all similar χ-terms are replaced at once. For example, consider the reduction of the term $(zero \diamond zero) \diamond (zero \diamond zero)$. We comment on why this by itself is not very limiting in the next subsection. Also, we do not consider it as a problem in this thesis and expect that the specification could be written differently, possibly using auxiliary functions or by imposing some conditions on the terms that are to be rewritten interactively.

7.3.3 Output

Until now, we have considered input but what is interactive output in such a rewriting environment? We can put additional constraints on the nature of values expected from an external process. For instance, we can require that the replacements for the place-holder matches certain patterns. In the above, instead of allowing all B values, one could restrict the possible substitutions for $\langle B\rangle$ to values that are of the form $one \diamond \cdots$. Consider the alternate set R_1 of rules:

$$zero \diamond one = zero$$

$$zero \diamond zero = zero \diamond \chi(one \diamond \langle B\rangle)$$

Here the constraint on the input is that it should not only be of the sort B, but should also have the form $one \diamond \cdots$. Thus a user can provide a term that narrows the contents of χ and then the projection of the term that replaced the place-holder would be the value of the χ term. Thus the rule

$$\chi(one \diamond \langle B\rangle) = zero \diamond one$$

results as the effect of an interaction that provides a term $one \diamond \boxed{zero \diamond one}$. A term $zero \diamond zero$ reduces to a term of the form $zero \diamond \chi(one \diamond \langle B\rangle)$ which could further reduce to $zero \diamond e$ (for some B term e) when a user provides the term $one \diamond e$. For our purposes, we allow in every χ term only one place holder – the sort of which is the sort of the term it projects to. In an interactive sense, this means that a user is constrained to provide a term of the form $one \diamond \cdots$ for the reduction to proceed further. The "$one \diamond$ " provides to the user context information while inputing a value for $\langle B\rangle$. The context information can in turn be perceived as output. Note that, this results in requesting specific patterns. Thus:

$$zero \diamond zero \qquad \xrightarrow[(one \diamond zero) * \ (one \diamond one)]{* \qquad\qquad R_1} \qquad zero$$

Alternatively, $\chi(one \diamond \langle B\rangle)$ in the second rule could be $\chi(enter\ value : \langle B\rangle)$ where "$enter\ value : \langle B\rangle$" is a valid term over some sort. One could also give better context like "$enter\ value\ i : \langle B\rangle$", which depending on the value of i could result in queries of the form: "$enter\ value\ 1 : \langle B\rangle$", "$enter\ value\ 2 : \langle B\rangle$", $\cdots$, thereby avoiding some of the problems that seem to appear as a result of not distinguishing the source of an "interaction request".

7.3.4 In Practice

The rewriting starts as usual and when a normal form is obtained, the term is checked for existence of χ-terms. If there are no χ-terms, we are done and the result is presented using its visual syntax. Otherwise, a χ-term, which does not have an occurrence of another χ-term, is chosen arbitrarily and the contents[4] of the χ-term are presented to a user in a (visual) editor. The constant "$\langle B\rangle$" is treated as a meta-variable that should be filled in with an allowed substitution in the editor window. When done, this results in substituting in the normal form the value entered by the user for all occurrences of the same χ-term and the rewriting starts again depending on the value substituted.

We have ignored the issues of fairness here. One could, alternately choose to eliminate all χ-terms via interaction before restarting the rewriting process. Also, we could set some guidelines on the use of χ-terms in a specification, such that it

[4] The term under the χ.

adheres to an intuitive notions of interaction. One such guideline could be that the occurrence of χ-terms in conditions is limited to (or should be equivalent to) them appearing only on the right-hand side of the equations. This guideline helps one not to worry about backtracking in conditions. Also one could use Share-Where like annotations to identify where two independent occurrences of an interaction might coincide. This would distinguish identically looking χ-terms, depending on when and where they were created.

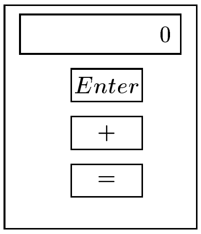
7.4 Calculator Example

In order to demonstrate how we address interaction we provide a very simple example of a calculator. Albeit small, this example describes a graphical language with semantics requiring human interaction. This example is not fine tuned for any specific interaction style and thus only the default interaction behavior is discussed.

Building a user interface is done in two phases. The first phase is to specify the look of the interface. This is similar to many common user interface builders available today. The additional flexibility we provide is that certain syntactic constructs can be grouped together by using a small constraint language for layout of the interface look. The details of the visual syntax specification of the calculator is not provided here. But rather, the focus is on aspects related to input and output and their specification. However, the following sort definitions are needed to follow the forthcoming specifications.

$$\begin{array}{lcl}
\boxed{\mathit{Enter}} & \rightarrow & \mathrm{OP} \\
\boxed{+} & \rightarrow & \mathrm{OP} \\
\boxed{=} & \rightarrow & \mathrm{OP} \\
\boxed{\mathit{NUM}} & \rightarrow & \mathrm{DISPLAY} \\
\begin{array}{l}\mathrm{OP}\\ \mathrm{OP}\\ \mathrm{OP}\end{array} & \rightarrow & \mathrm{OPS} \\
\boxed{\begin{array}{c}\mathrm{DISPLAY}\\ \mathrm{OPS}\end{array}} & \rightarrow & \mathrm{CALC}
\end{array}$$

The *Calculator* language defines three operations: $\boxed{\mathit{Enter}}$ for resetting the current value of the calculator; $\boxed{+}$ for adding a new value to the running total of the calculator; and $\boxed{=}$ for displaying the total. The sort *OPS* describes that three *OP*s are placed in vertical alignment and the sort *CALC* describes that a *DISPLAY* and *OPS* are laid out such that the *OPS* is centered and below the *DISPLAY*. This is specified by constraining these appropriately (Chapter 4). A visual term of the calculator could be, depending on the nature of the constraints specified:



7.4.1 Calculator Query Syntax

The calculator requires two input functions: one for retrieving an "OP" selection and one for retrieving values. The values in question are numbers, the definition of which is imported (predefined).

In order to interact we need to define syntax for queries, which would generally be an extension to the Calculator syntax itself. Queries can be as simple or complicated as desired. The simplest queries are just meta-variables appearing in a term window without indicating any context. In essence the goal of a query is to fetch some value and the manner in which the input is retrieved only bears on interface aspects. The point is that the language designer can simply define input prompts (which are in fact the outputs) in a uniform and convenient manner. The utility of these definitions can be seen in Section 7.4.2. The following defines a query syntax for the Calculator language:

$$
\begin{array}{llll}
\text{CALC} & \text{NUM} & \rightarrow & \text{CALC-Q} \\
\text{CALC} & \text{OP} & \rightarrow & \text{CALC-Q}
\end{array}
$$

Terms of sort CALC-Q will be used to present the current state of the calculator as well as demand input from a user who must respond by "building" an input term. In this case, depending on the context, the input requested will either of the sort NUM or the sort OP.

7.4.2 Evaluation Semantics

After the first stage which is specifying the desired user interface components and the layout, the second stage involves specifying the semantic component of the user interface. This is done using equations. Note that one might need to specify additional syntax during this stage that need not be part of the user interface itself. For the syntax of the calculator evaluation we use the additional syntax of the *eval* and *eval-op* functions and define their functionality.

$$
\begin{array}{lll}
\textit{eval}(CALC,NUM) & \rightarrow & NUM \\
\textit{eval-op}(CALC,OP,NUM) & \rightarrow & NUM
\end{array}
$$

The semantics is defined using conditional equations as explained in Section 1.4.3. These equations make use of the following variables:

$$
\begin{array}{lcl}
Calc & \rightarrow & CALC \\
Ops & \rightarrow & OPS \\
TheOp & \rightarrow & OP \\
Store, Num, Num' & \rightarrow & NUM
\end{array}
$$

For example, *Calc* could be bound to any calculator (visual) term that can be composed from the above syntax specification for the sort *CALC*. Furthermore, in this example, we use the notation $\boxed{\blacksquare\mathrm{term}}$ instead of the $\chi(term)$ used in Section 7.3. We provide a brief explanation for each equation below.

equations

Evaluating a CALC term with a given store, is to query for an operation and then evaluate the term using the result of this query. The variable *TheOp* represents the result of interaction that would be obtained after interactively binding the variable to an operation. Note that the current *Calc* contents are displayed to the user in order to provide the context for interaction. The right hand side of the equation gets the user desired operation which is bound to the variable *TheOp* which guides the interface to the next interaction caused by *eval-op*.

$$
[1] \quad \frac{TheOp \quad = \quad \boxed{\blacksquare Calc\ \langle \mathrm{OP} \rangle}}{\mathit{eval}(Calc, Store) \quad = \quad \mathit{eval\text{-}op}(Calc, TheOp, Store)}
$$

Evaluating a *Calc* when the operation is $\boxed{Enter}$ is to query for a new number which will be displayed in the calculator.

$$
\begin{array}{ll}
[2] & \mathit{eval\text{-}op}\left(\boxed{\begin{array}{c} \boxed{\qquad Num} \\ Ops \end{array}}, \boxed{Enter}, Store \right) \quad = \\
 & \mathit{eval}\left(\boxed{\begin{array}{c} \boxed{\qquad \boxed{\blacksquare\langle NUM \rangle}} \\ Ops \end{array}}, 0 \right)
\end{array}
$$

Evaluating the operation $\boxed{=}$ amounts to displaying the current value in the *Store*.

$$
[3] \quad \mathit{eval\text{-}op}\left(\boxed{\begin{array}{c} \boxed{\quad\quad \mathit{Num}} \\ \mathit{Ops} \end{array}}, \boxed{=}, \mathit{Store} \right) \quad = \quad \mathit{eval}\left(\boxed{\begin{array}{c} \boxed{\quad\quad \mathit{Store}} \\ \mathit{Ops} \end{array}}, \mathit{Store} \right)
$$

Evaluating the operation $\boxed{+}$ means to query for a number and display the result of the query, as well as storing the sum of the new number and the old store as the new store. Note that one interaction results in spite of Num' appearing twice on the right hand side.

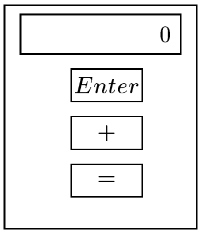
$$
[4] \quad \frac{\mathit{Num}' \quad = \quad \boxed{\blacksquare\langle \mathit{NUM} \rangle}}{\mathit{eval\text{-}op}\left(\boxed{\begin{array}{c} \boxed{\quad\quad \mathit{Num}} \\ \mathit{Ops} \end{array}}, \boxed{+}, \mathit{Store} \right) \quad = \quad \mathit{eval}\left(\boxed{\begin{array}{c} \boxed{\quad\quad \mathit{Num}'} \\ \mathit{Ops} \end{array}}, \mathit{Store} + \mathit{Num}' \right)}
$$

7.4.3 Interaction issues

Thus far, we have touched upon the syntax and semantic aspects of the calculator. In this section, we discuss how all these specifications can be brought together to yield a practically useful set of tools for an end user environment for this language.

The term editor

The term editor, which is generated from the syntax specification of a language, allows the creation of terms of that language. For the *Calculator* language, the *Calculator Term Editor* allows the creation, for example, of the following term:



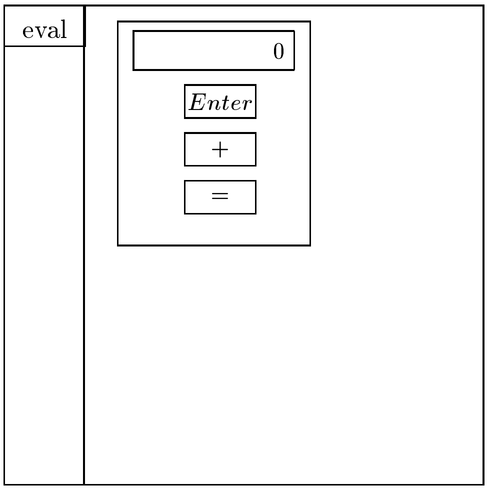
Input and output representation

When an input request is presented the user can replace the meta-variable with a permissible replacement as dictated by the language syntax, which is always type correct and represented just as the syntax is defined (graphical input). Thus, the variable which was unbound becomes bound after the user interaction. In the case of human interaction we may very well prefer to present the input request in a more user-friendly manner. For example, we may prefer to have: “Please enter an operation: ⟨OP⟩”.

Term reduction

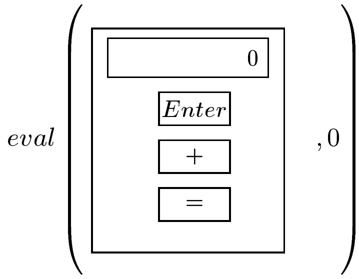
After a visual program (a term) is constructed we want to execute it using the semantics. To do so, we apply the *eval* function defined in Section 7.4.2.

To start the scenario, first a calculator term must be created. This is done in a term editor over CALC:



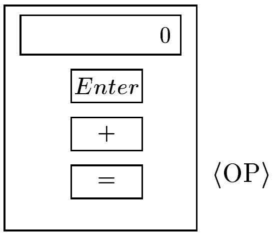
In this editor, the eval button is defined to apply the “eval” function to the

CALC term constructed in it[5]:



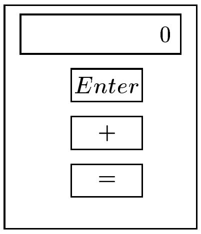
Now let us follow a scenario to see how the equations deal with input and output during evaluation. Note that at the time of evaluation the exact appearance of the calculator is determined. The actual ordering of the operations is determined when the calculator term is constructed. The syntax, in fact, permits any ordering or even repeated occurrences of the operations as long as there are three operations. After the evaluation is requested, this calculator term is continually rewritten driven by the input received.

The rest of the scenario shows the term in the editor as it is rewritten. The equation number references are from Section 7.4.2. Applying the *eval* function to the term, invokes an external-match due to the (i.e., χ term) present in equation [1], which presents the term to a user:



The meta-variable demands input from the user, who can syntactically choose from a menu which presents the permitted operation or select an appropriate subterm from the existing term. The latter choice means that the user can select any operation from the calculator term. If the user selects $\boxed{Enter}$ then the term becomes the one on the left below, using which the *eval-op* function matches equation [2] which invokes yet another I/O (the right term):

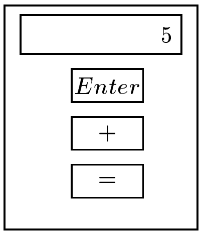
[5] The term editor supports the binding of a function from a language specification to a button.



$\boxed{Enter}$ $\langle$NUM$\rangle$

Now, in order to continue, a number must be provided. Considering that the number 5 is entered, the rewrite of equation [2] can be completed, which is another *eval* function matching equation [1] again.

5



$\langle$OP$\rangle$

Notice that the terms driven by input and output are presented in a window. In Chapter 6 we investigated how information could be maintained (called Share-Where maintenance) so that the initial look of the calculator is preserved through the interaction. We have not discussed the issue of how certain window control information can be incorporated.

7.5 Specification of FSA Animation

In this section we will examine input and output during the execution of an *FSA* sentence, which was introduced in Section 6.2. Here, we provide a specification that interactively requests an input alphabet from the user and does an animation step. We also show how Share-Where is used in displaying animation steps.

First, we need to define a new syntax for an animation language. Figures 7.1 and 7.2 show the syntax and semantic equations of *FSA-anim* which defines a query syntax and the semantic rules to be used for interactive animation.

The interaction is defined in terms of soliciting a character from the user as part of the input string while presenting the state of the *FSA* configuration (*FSA-Q*) as feedback. In fact, it is created with an *FSA-CONF* and an *SSTATE* visual term. This way, we get both the current state as well as the start state in the presented picture – due to the influence of Share-Where during pretty printing. Recall that $\square$ projects the value of the meta-variable, which in this case is $\langle ALPHA \rangle$. The conditions in each equation result in retrieving a character from the user.

The query is presented with the *FSA*, a prompt for input, and the alphabet sequence that has been provided to reach this state. The *evit* function defined in Section 6.2.3 for the *FSA-eval* is used to process the character retrieved from the user according to the state of the sentence. If the character retrieved allows a transition the current state is updated appropriately and the user is asked for another character. If the character does not allow a transition the user is informed of this condition and is asked for another character.

Equation $[e1]$ creates an *FSA-CONF* visual term which is used by the *evit* function. Unlike in the *FSA-eval* language, where the entire input string is given to start with, here the input string is constructed according to the user input. Accordingly, the *evit* function is given an *FSA-CONF* visual term consisting of an *ALPHA-L* with only a single character. If the transition is allowed this character is consumed, otherwise it remains. Thus, we can differentiate among the cases when a query should inform the user that the transition was not valid with the given character and request for another choice. Also, the input string is constructed only with the characters that result in transitions. This goes on until doomsday (the sort *CIAO* has no constructors). We could have alternatively chosen to stop it with an empty character and give the resulting state of *accept* or *reject* as in Section 6.2.

7.6 Interaction

Figure 7.3 shows an *FSA* instance in a term editor. After such a visual program is constructed we want to execute it using the specified semantics. The “anim” button is defined as to apply the “anim” function which is defined in the *FSA-anim* module.

Now, let us follow part of a scenario to see how the equations result in input and output during evaluation. The equation labels used in the explanations correspond to the labels of the equations in the module *FSA-anim*. As a difference from the specification of *FSA-eval* which did not use the start state during evaluation (*evit* function), during animation, for displaying intermediate steps, we need to put back the start state. If we did not the *FSA* picture would appear different to the observer. During the animation, we will display the start state, the current state, the input string retrieved so far from the user and the input request query.

Applying the *anim* function to the term, results in input request due to equation $[i2]$ – which gets called by the equation $[i1]$ which defines the “anim” function. The resulting term is seen in Figure 7.4. The unbound variable i is replaced with $\langle ALPHA \rangle$. The remainder of the query provides context and feedback during the animation. The *FSA* term provides the context by showing the *FSA* with current

module FSA-anim

imports FSA-eval

sorts *FSA-CIO CIAO FSA-Q*

functions

anim (*FSA*)	$\rightarrow$	*CIAO*
astep (*FSA-CIO*)	$\rightarrow$	*CIAO*
FSA-CONF , *ALPHA-L* , *SSTATE*	$\rightarrow$	*FSA-CIO*
Choose an character: *ALPHA* *FSA-CONF SSTATE* $\boxed{ALPHA\text{-}L}$ used to reach current state	$\rightarrow$	*FSA-Q*
No valid transition exists for: "*ALPHA*" from current state! Choose another character: *ALPHA* *FSA-CONF SSTATE* $\boxed{ALPHA\text{-}L}$ used to reach current state	$\rightarrow$	*FSA-Q*

variables

i	$\rightarrow$	*ALPHA*
i'	$\rightarrow$	*ALPHA*
Ss	$\rightarrow$	*SSTATE*

Figure 7.1: The syntax specification for a query language for *FSA-anim* language.

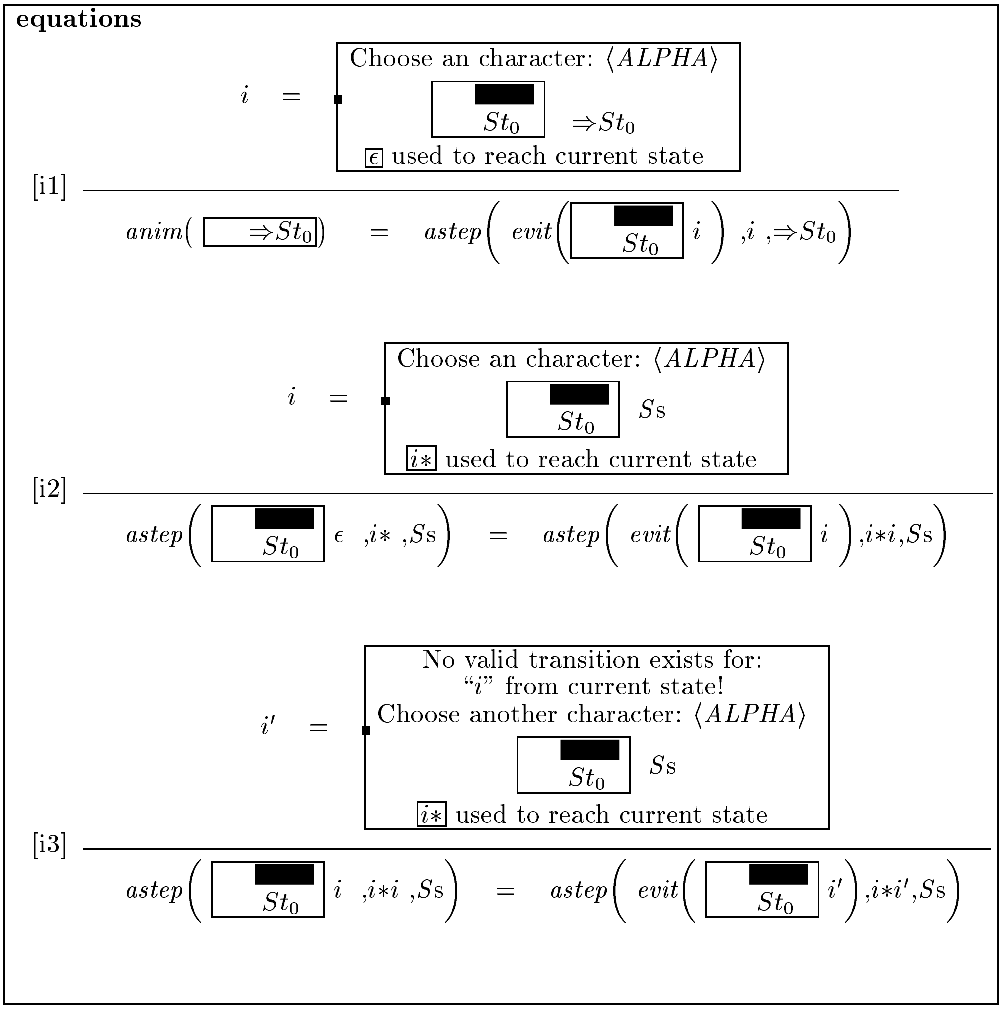


Figure 7.2: The specification of *FSA-anim* semantic rules for the animation.

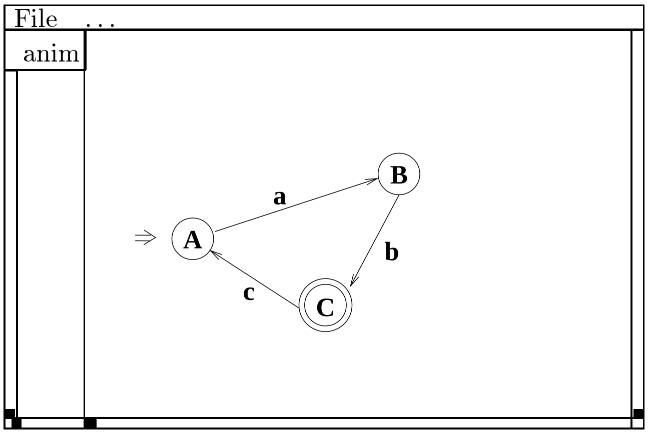


Figure 7.3: Term editor for the *FSA* language.

state as well as the valid string entered to reach this state.

The meta-variable $\langle ALPHA \rangle$ demands input from the user, who can syntactically choose from a menu which presents the permitted operations or select an appropriate sub-term (transition labels) from the existing term. If the user selects "a" then the term matches equation $[i2]$ again which updates the current state and the input sequence and combines these in the next query for input (Figure 7.5-a). To continue, another character must be entered. Consider that the character "a", say, is selected again. This time the equation $[i3]$ matches which outputs a message that such a transition is invalid and solicits for a new character (Figure 7.5-b).

Notice that the terms seen in the editor are the ones resulting from input and output. No other intermediate terms are presented. It is not the case that each rewrite changes the term in the editor. This model extends it to present intermediate terms identified by input and output. Clearly, several user interface issues need to be investigated to address the various ways that intermediate terms as well as the input and output could be presented. Among these choices are to replace terms in place, present new terms in separate windows or in separate places of the same window, or hiding parts of the term that are not relevant to the user etc.

7.6.1 Share-Where for IO

We have already explained in Section 6.3.1 how Share-Where information is used to construct desirable output terms that correspond, to some extent, to the input terms or the terms responsible for the production of the output term. This technique can be used in a straightforward manner to handle IO. During execution, in the presence of IO, there could be many terms that are created or manipulated by

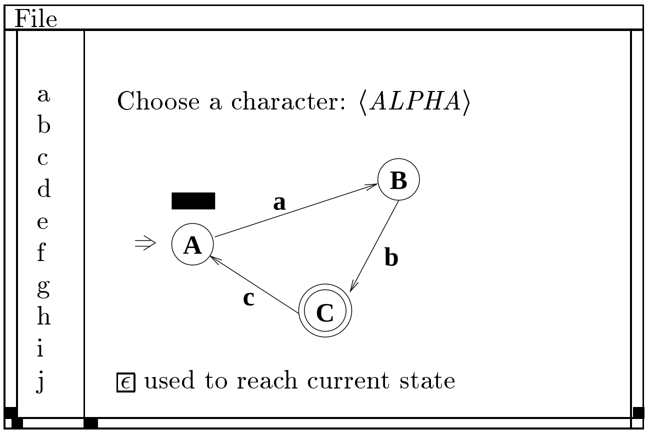


Figure 7.4: First interaction after "anim" is pressed.

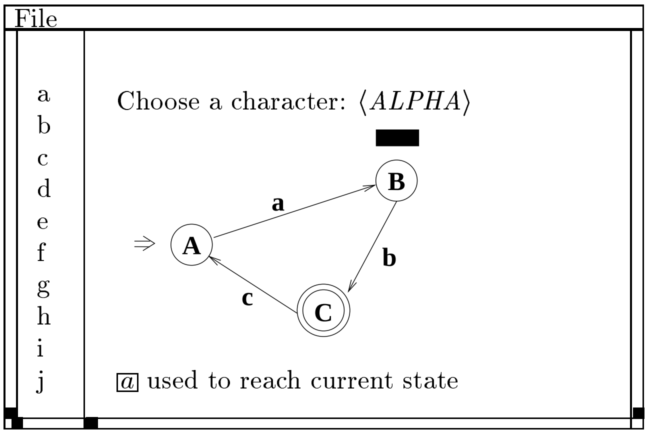
a user which have a bearing on the final term. The following observations help in proper implementation of Share-Where in the presence of IO.

- The Where information necessitates that all creations be available for lookup if necessary. This means that all versions of IO windows which created any term should be accessible when presenting a new term.
- When a term is displayed in an IO window it preserves the Share-Where information from before. The Share-Where information originates from an IO window for input (sub)terms only.
- Any sub-term that is created in an IO window gets new Share-Where annotations, unless it is shared with an output (sub)term. If it is shared, it gets the annotation of the sharer.

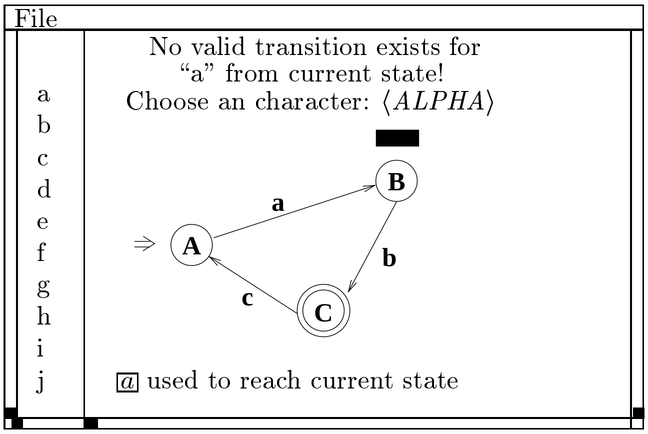
7.6.2 Share-Where properties for FSA animator

We can claim the following properties for the module FSA-anim.

- The FSA animator preserves the look of the FSA input term.
- The FSA animator moves the current state marker without surprises, when in the FSA term each state has a unique presentation (i.e., no state looks repeated). Should they not have a unique presentation, a transition could appear to "fire" from a current state with no transition that matches the input. The FSA-anim semantics specifies that all states that appear the same are considered the same at the abstract level.



(a) Input request after successful transition.



(b) Input request after unsuccessful transition.

Figure 7.5: Some *FSA* term configurations during animation.

7.7 Related Work

Monads [39] and *unique types* [59] are two examples that allow extension of functional languages with input/output. Monads are higher order constructions and thus are not directly useful for us. Also monads are basically used to more or less transparently thread different input and output streams through a program. We on the other hand support interaction as a notion independent of input and output streams. *Unique types* are used to type-check a program so that non-determinism is limited when one reads or writes from a stream. We do not have the same problem, but we could also use such a typing for warning a specifier of potentially different "interaction requests" that may collapse into one interaction phase.

Koorn [45] has developed a language called SEAL that provides the necessary interaction and window management constructs in a context of algebraically specified rewriting. SEAL is a separate language used for coordinating the different terms in various windows that are involved in interaction. One cannot use the same (visual) syntax like one does in an equations editor, instead one is forced to work with focus movement operations. SEAL also provides for other operations that manipulate a user interface which could also be captured by extending our notion of interaction appropriately.

Walters and Kamperman [82] have indicated how a term rewriting system can be viewed as a specification with input and output variables, by transforming a specification into another one that is suitable for constraint narrowing. Their intention is to determine when part of a term can be output (say written to a file) and thus need not be carried around by the rewriting machinery until the whole term is computed. The motivation for their work is to be able to make a term rewriting system deal with huge terms efficiently—when space is a problem, although it is not very clear how one can use their approach to provide interaction. Here we consider the dual problem of allowing explicit input and output in a term rewriting system so that interaction with a user is possible.

The language Prograph [58] (not a functional language) is a popular visual language but it provides only limited support for user definable visual data types that could be used for input and output. In our setting, a user can define arbitrary visual terms can could be used during interaction.

7.8 Summary

In this chapter we have introduced a simple model for supporting interaction during term rewriting. The nature of visual languages demands highly interactive environments. Having support for interaction addresses this need and presents the possibility of animation and debugging driven by users. The model presented is very basic and could be further developed in regards to both the graphical user interface and the sophistication at which input is demanded. Right now it demands one input per query request.

Chapter 8

Implementation Experiments

This chapter reports on experimental implementations conducted to test the ideas discussed in this thesis as well as describe a proposed implementation for the new ASF+SDF Meta-Environment. In Section 8.1 we provide some context and history of the implementation. In Section 8.2 we describe some of the early implementation efforts related to Chapter 3. In Section 8.3 we describe the new architecture of the ASF+SDF Meta-Environment and two tools that were built for this new architecture. Finally, in Section 8.4 we conclude with describing the implementation required to implement VASE as discussed in Chapter 5.

8.1 History of the Implementation Experiments

The aim of the work on VASE was to eventually integrate it within the ASF+SDF Meta-Environment. At the time that this work was initiated the ASF+SDF Meta-Environment was a system whose user-interface consisted of simple textual editors. It was a monolithic application consisting of a variety of components handling various aspects of language specification and term execution. The monolithic nature of the system made it difficult to extend it with graphical editors.

Our initial efforts were geared towards finding public-domain software[1] for constructing graphical editors and for solving numerical constraints. After some investigation and testing we settled upon Garnet [54] as it supported both these needs and was reasonably mature software. Using Garnet we experimented with ideas described in Chapter 3. These experiments are described in Section 8.2.

[1] We are constrained to using public-domain software not only from our own financial constraints but also from the point of view of distributing any resulting software.

We gained considerable insight from these initial efforts. However, we abandoned this path for two main reasons. The first and most significant reason was motivated by the major redesign of the ASF+SDF Meta-Environment which is switching from a monolithic architecture to one consisting of a dynamically configured set of cooperating tools based on ToolBus [9]. We, thus, wanted to decouple the constraint solver and the graphical editor to have two separate tools which could be used independently. The second reason was to move to a graphical platform that was smaller, more stable, and better supported. The architecture of the proposed implementation for VASE is discussed in Section 8.4.

We proceeded building separate tools for our work. We focused on the VODL editor and the constraint solver. As for a new graphical platform we initially experimented with Tcl/Tk [57] which was also the graphical interface-builder for the new Meta-Environment. We were quickly frustrated with the lack of structuring facilities of Tcl. This prompted us to switch to Python/Tk [66] which is an object-oriented interpreted language offering excellent structuring facilities. We found this essential as we wanted to deal with composite graphical objects which we generate from *vod*s. Without appropriate structuring mechanisms the maintenance and comprehensibility of any resulting code would be very difficult. In Tcl/Tk all graphical objects exist at the same level in a canvas (a Tk component where graphical objects are drawn) where imposing and maintaining structure over these to get the effect of composite graphical objects was quite unpleasant. The VODL editor developed using Python is described in Section 8.3.1.

As for the constraint solver we constructed tools based on the solvers DeltaBlue [69] and then on SkyBlue [68]. The tool created using DeltaBlue is briefly described in Section 8.3.2. We have not really found a satisfying solution for our constraint solving problem yet. The software we used, in fact, maintains (not necessarily solves) constraints via value propagation. This is suitable for cases where we are interested in maintaining graphical relationships for well defined pictures since the constraints and defaults can be specialized for these cases. However, in our case, we have a different problem, where we are interested in instantiating graphical objects based on *vod*s which are continuously being defined. This would require that values have to be automatically generated for attributes of the *vod*s that are not defined explicitly. But DeltaBlue does not deal with undefined variables. So in our experiments we had to define default values for attributes. For the graphical editor case, from the user-interface point of view, we need to select appropriate values from the generated set of solutions. For example, we would want the picture to remain as stable as possible avoiding unnecessary movements or resizing.

We would ideally want to have a solver that produces appropriate values according to the definitions. We have been particularly interested in using the constraint solver being developed by the visual languages and CLP(R) research group at Monash University. The best description of the nature of this solver can be found in [37].

8.2 Generating Visual Editors

In this section we discuss some early work regarding the generation of editors as described in Chapter 3. This work consists of two parts: (a) the specification of languages, their mapping to visual representations and editor generation; and (b) the basis of the structured visual editor (see Figure 3.1).

8.2.1 Editor Shell

Using Garnet we created a very simple shell for the editor as explained in Section 3.5.4. Basically, this editor consists of:

- A replacement panel: which contains the valid replacements for the currently selected placeholder. A list of valid replacements for each sort of the language is constructed into a menu, where only one of the items in the menu can be selected at a time. The menu is constructed based on the generated replacement behavior as described in Section 3.5.1. The menu present in the replacement panel depends on the sort of the most recently selected item in the visual term, containing the replacement menu for that sort. In the initial configuration the replacement menu consists of a list of all sorts of the language.
- A work space: which allows the creation of visual terms of the language. The user can interactively select and create syntactically correct visual terms by selecting items from the replacement panel. Figure 8.1 shows an instance of such an editor for binary trees where the leaves are boxed numbers. Placeholders are presented here by "$\langle$*sort-name*$\rangle$".

8.2.2 Definition of *vod*s

The definition of the *vod*s as well a translation to garnet code is specified using the ASF+SDF Meta-Environment. Figures 8.2 and 8.3 show the syntax of the translation and some of the equations transforming the *vod* definitions to garnet-*vod*s respectively. In Figure 8.3 equation [rvod] creates a rectangle graphical object and equations [comp-vod1] and [comp-vod2] create a composite graphical object referred to as aggregadgets in Garnet by first creating an aggregate object and then adding the sub-*vod*s to the aggregate. For example, the following *vod* definition:

garnet-vod(text() $\oplus$ [name = `"Hello!"`])

yields the following garnet code (which we refer to as a garnet-*vod*):

```
( create-instance ' TEXT-VOD opal:cursor-text
    ( :known-as :text-label )
    ( :right ( o-formula ( + ( gvl :left ) ( gvl :width ) ) ) )
```

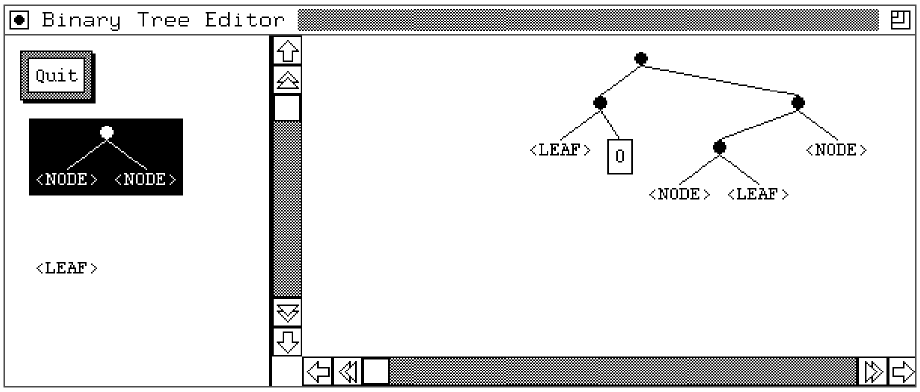


Figure 8.1: Snapshot of a binary tree editor, where Garnet is the graphical platform.

Garnet-VOD **Module**
imports Lisp VOD Garnet-Constraints
exports
  sorts GARNET-VOD GARNET-VOD-LIST
  context-free syntax

garnet-vod(VOD)	→ GARNET-VOD-LIST
LISP-C	→ GARNET-VOD
GARNET-VOD-LIST "&" GARNET-VOD-LIST	→ GARNET-VOD-LIST
GARNET-VOD+	→ GARNET-VOD-LIST
make-agg(VODNAME, GARNET-VOD-LIST)	→ GARNET-VOD-LIST
vl2cl(GARNET-VOD+)	→ LISP-C
vodname2lisp(VODNAME)	→ LISP-C

hiddens
  variables

Name [*0-9*]∗	→ STRING
Garnet-Vod [*0-9*]∗	→ GARNET-VOD
Garnet-Vod [*0-9*]∗"∗"	→ GARNET-VOD∗
Garnet-Vod [*0-9*]∗"+"	→ GARNET-VOD+
Garnet-Vod-list [*0-9*]∗	→ GARNET-VOD-LIST
CType [*0-9′*]∗	→ CTYPE
v[*0-9*]∗"∗"	→ CHAR∗

Figure 8.2: The syntax definition of the module Garnet-VOD.

equations

`default-height` = (`:height` 40) [ht-def]

$$\frac{[AttrP_1^+] = \text{prep-comp}([AttrP^+]),\quad [AttrP_2^+] = \text{prep2}([AttrP_1^+])}{\text{garnet-vod}(\text{rectangle}() \oplus [AttrP^+]) = }$$

(`create-instance` ' str-to-lisp2("`RECT-VOD`")
`opal:rectangle`
(`:right` (`o-formula` (+ (`gvl :left`) (`gvl :width`))))
(`:bottom` (`o-formula` (+ (`gvl :top`) (`gvl :height`))))
(`:known-as :rectangleref`) garnet-constrs($AttrP_2^+$)) [rvod]

$$\frac{Str_1 = \texttt{":"} \parallel Str,\quad [AttrP^+] = \text{prep2}([AttrP_1^*, \text{name} = Str, AttrP_2^*])}{\text{garnet-vod}(\text{text}() \oplus [AttrP_1^*, \text{name} = Str, AttrP_2^*]) = }$$

(`create-instance` ' str-to-lisp2("`TEXT-VOD`") `opal:cursor-text`
(`:known-as :text-label`)
(`:right` (`o-formula` (+ (`gvl :left`) (`gvl :width`))))
(`:bottom` (`o-formula` (+ (`gvl :top`) (`gvl :height`))))
garnet-constrs($AttrP^+$)) [tvod]

$$\frac{\textit{Garnet-Vod-list} = \text{garnet-vod}(Vod)\ \&\ \text{garnet-vod}(\text{defv}\ VodName(X^*)\ \{VodDec^*\} \oplus [AttrP^+])}{\text{garnet-vod}(\text{defv}\ VodName(X^*)\ \{VRef: Vod, VodDec^*\} \oplus [AttrP^+]) = \text{make-agg}(VodName, \textit{Garnet-Vod-list})}$$ [comp-v1]

$$\frac{\text{vl2cl}(\textit{Garnet-Vod}^+) = \textit{lisp-code},\quad \textit{lisp-code}_2 = \text{vodname2lisp}(VodName)}{\text{make-agg}(VodName, \textit{Garnet-Vod}^+) = }$$

(`create-instance` ' $\textit{lisp-code}_2$ `opal:aggregadget`)
(`opal:add-components` $\textit{lisp-code}_2$ *lisp-code*) [comp-v2]

vodname2lisp(vodname(v^*)) = lisp-c(v^*) [v2str]

Figure 8.3: Some of the equations transforming *vod*s into garnet-*vod*s.

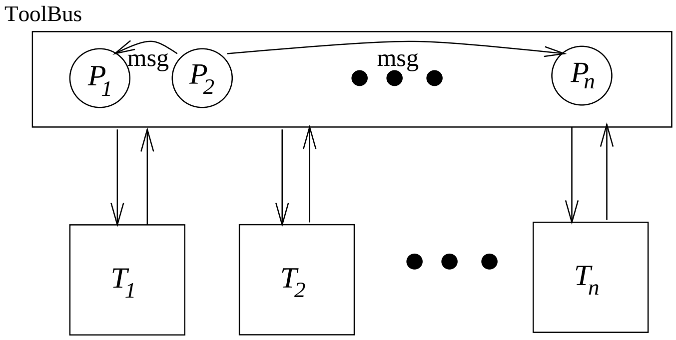


Figure 8.4: The ToolBus architecture. $P_1, ..., P_n$ are processes and $T_1, ..., T_n$ are tools which communicate via the ToolBus.

```
( :bottom ( o-formula ( + ( gvl :top ) ( gvl :height ) ) ) )
( :left ( o-formula ( gvl :parent :left ) ) )
( :top ( o-formula ( gvl :parent :top ) ) )
( :string "Hello!" ) )
```

The values for the `:left` and `:top` slots result from the *vod* shown above being pre-processed prior to begin translated to a garnet-*vod*. This is necessary since the presence of some slots is required for garnet to render as well as deal with formulas[2]. Thus prior to translating any *vod* to a garnet-*vod* we pre-process them to assure they have all the required values. The translation definitions for constraints are defined in the module (imported) Garnet-Constraints.

8.3 The new ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment is near completion and with its new architecture it will be much easier to incorporate our tools. It is based on the ToolBus [9] architecture which supports the interconnection of distributed, heterogeneous, tools. The asynchronous communication among the components is carried out with messages that pass data called 'terms'. Adapters are required to translate from the data types of the ToolBus and the data types of the tools. Figure 8.4 shows the architecture of the ToolBus.

[2]Formulas are slot values that are defined as constraints.

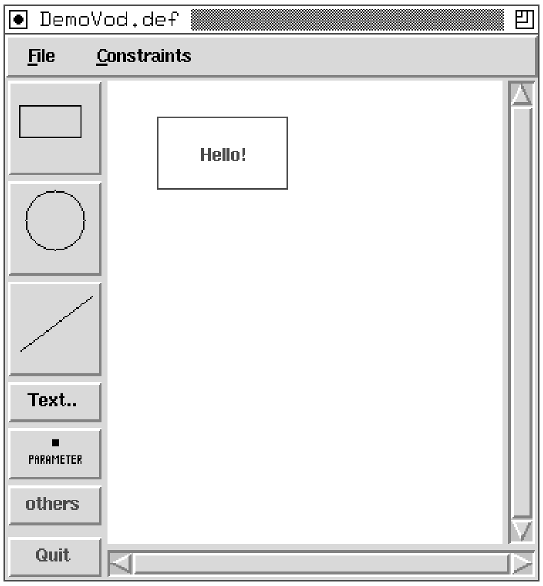


Figure 8.5: Snapshot of a the *vod* editor editor. Here the graphical platform is Python/Tk.

8.3.1 VODL Editor

The VODL editor is currently being implemented[3] with Python/Tk as the graphical platform. Figure 8.5 shows a snapshot of the editor which defines a composite *vod* consisting of a rectangle containing the text "Hello!". It provides a set of primitive *vods* to construct composite *vods*.

The definition consists of interactively selecting from a set of primitive and predefined *vods* (accessed with the **others** button) which identify the sub-*vods* and a set of constraints governing the sub-*vods*. This editor is an interactive editor where the sub-*vods* are drawn on the panel. For each *vod* that is being created, a label has to be entered (via a dialog box) that will serve as a *vod*-reference. This is to keep the correspondence with the *vod* syntax presented in Chapter 4. These labels could alternatively be generated automatically. However, it is rather useful to have user defined labels for sub-*vods* which often correspond to something meaningful to the creator. Since sub-*vods* may be referenced in other *vods*, labels that make sense to the user are also preferable.

The definition that is saved for this *vod* is:

[3]The implementation work for the VODL editor is being carried out by Harold Breebart as part of his masters thesis project. The full description of this editor will be available in his upcoming masters thesis.

```
defv DemoVod()
{ hi: text() _wa [ name = ''Hello!''],
  border: rectangle()}
{ border contains hi }
{ hi: 67.0, 47.0
  border: 37.0, 27.0, 128.0, 77.0}
```

The first and second group are the sub-*vod*s and constraints sections that we are familiar with. The third group are values that are used to present the *vod* when retrieved into the VODL editor.

8.3.2 Constraint Solving

A constraint solver tool [74] based on DeltaBlue [69] was developed for the purposes of attaching it to the ToolBus. It has already been interfaced with the ToolBus. The DeltaBlue constraint solver was specialized for our needs of graphical constraints. Then an interface language for the solver was developed which allowed the constraints to be expressed as simple equations such as:

$$\begin{aligned}
&r1.x := 50,\\
&r1.x + 100 = r2.x,\\
&r1.y = r2.y,\\
&r1.y := 50
\end{aligned}$$

where $r1.x, r2.x, r1.y$, and $r1.y$ are variables. The operation := is used for a temporary assignment and = as the equality constraint. The coordinates of *vod*s can easily be defined where "." serves as a separator between *vod* names and attributes.

We are not very satisfied with the solver as it often gets into loops due to cyclic references which are commonly defined for graphical objects. Also, with DeltaBlue, the constraints can only be defined with equalities. Having inequalities, ranges and domains would be most useful. We simulated results using only equations, but this is quite limiting. In order to get values at all we were obliged to define many default values. However, for prototyping purposes this was still satisfactory. We are presently pursuing other options for a new constraint tool.

8.4 Future implementation directions

Both our specification environments (VASE), and language-specific, generated, visual environments can re-use various tools and components of the new ASF+SDF Meta-Environment. We briefly discuss their implementation in the following two sections.

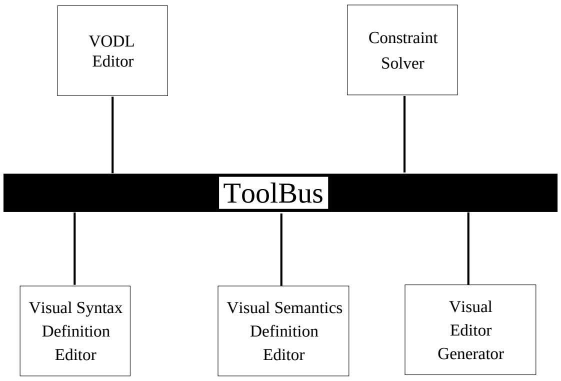


Figure 8.6: Tools in the visual language specification environment.

8.4.1 A VPE for VASE

Some of the tools comprising VASE as described in Chapter 5 are shown in Figure 8.6. The tools seen above the ToolBus handle the definition and presentation of the visual lexicals. The VODL editor (Section 8.3.1) is used to define *vods* and the constraint solver assures that the lexicals abide by the constraints present in their definitions.

The VODL editor and the constraint solver tools were already described in Section 8.3.1 and 8.3.2. To facilitate structured editing we need to implement a graphical editor shell using Python/Tk similar to the one described in Section 8.2.1. The next step would be to define visual mappings from the syntax definition and for the equations definition languages providing visual representations for these formalisms. The editor generation process applied to these languages provides us with the corresponding visual editor for defining the syntax and semantics of visual languages. Recall also that the equation editor is parameterized by the syntax definition of the language for which it is specifying the semantics. So it is affected both by the syntax defined for general equations as well as the syntax of a particular language. Thus, a language definition consisting of both these syntax definitions needs to be constructed before generating an editor for the equation editor. The concepts related to these editors were described in Section 5.3.

By applying the visual mapping and editor generation we can obtain the editors needed to specify visual languages. Once these editors are present the syntax and semantics of the visual languages are defined visually. In these editors the visual mappings must, of course, be obtained implicitly and not defined by the user. The language syntax is defined by the user who defines the lexicals using the VODL editor and the language syntax with the syntax definition editor. In order to maintain the visual presentation of the syntactic constructs we need to save the *vods* used to construct the signature of the language. This information is present in the editor and must be maintained as part of the signature since it is necessary to access this information for pretty printing and any future retrievals.

In order to describe where such information will be retained we need to know how the abstract syntax is represented in the new ASF+SDF Meta-Environment. The abstract syntax is represented using a language called Asfix [78] which uses prefix notation and unique names for each context-free function of the language. It also allows arbitrary annotations to be included along with each function definition. The visual presentation of each function is maintained as an annotation.

The annotation is nothing but the *vod* instantiation used to construct the left hand side of the function. Such a *vod* could simply be a predefined *vod* representing a constant function. It would more likely be a parameterized *vod* allowing sorts to be part of the signature. Also, recall that *vod*s can be created on the fly during syntax definition since other *vod*s can serve as arguments to parameterized *vod*s as was discussed in Section 5.3.1. The sorts present in the function definition represent the arguments to the function.

To support the Share-Where maintenance discussed in Chapter 6, whose implementation was discussed in particular in Section 6.4, each specification would need to be transformed into one with Share-Where annotations. Thus, after the language is specified it would be transformed into another specification having Share-Where annotations and then its abstract syntax would be annotated with *vod* descriptions to maintain its visual presentation.

8.4.2 A VPE for the specified language

Some of the relevant tools for a visual programming environment for a specified language are shown in Figure 8.7. The structured visual editor is again generated from the syntax description of the language and allows the construction of syntactically correct sentences of the language. Share-Where maintenance is used for tracking relevant information for presentation.

The term rewriter executes (reduces) the constructed program as given by the semantic definition of the language. The abstract syntax processor maintains the abstract syntax of the constructed term which is used by the term rewriter and pretty printer. Results of rewriting (execution of a program) are pretty printed and displayed in a result window.

8.5 Concluding remarks

There has been quite some work and thought exercised regarding the implementation aspects of VASE. However, the implementation remains far from completion. It would have indeed been very interesting to put these ideas to a practical test as to the author's knowledge there is no such approach tested for visual languages.

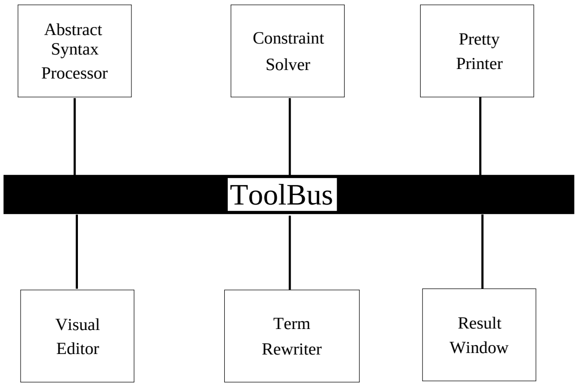


Figure 8.7: Tools in a visual programming environment for a specific language.

Chapter 9

Conclusions

In this chapter we make our final remarks by expressing what we feel are the contributions of this work as well as discussing the limitations of the proposed approach. We also comment upon possible future directions.

9.1 Contributions

This work was born out of the desire to extend a textual language specification formalism (ASF+SDF) and its supporting environment (ASF+SDF Meta-Environment) for dealing with visual languages. We conducted our explorations within the framework of context-free specifications with term representations and examined what we could achieve within these boundaries. With this as our starting point, we began investigating how we could specify visual languages and generate visual editors from languages specified with this formalism such that desired sentences could be constructed.

Our approach separates the definition of syntax and semantics. Many other approaches combine these definitions into a single definition like in the attribute grammars of [5, 34, 29, 48]. We believe that our approach allows the distribution of concerns in language definition and improves the comprehension of these different aspects of languages.

In order to build visual terms, we needed a language to specify the graphical language constructs. For this we introduced the picture definition language VODL (Chapter 4). Using graphical notation defined with this language we describe how we could define visual languages. Initially, we considered language definition with explicit mappings of visual notation to language constructs as discussed in Chapter 3. Then, we presented a visual specification formalism that rendered the mapping implicit and used visual notation in the specification itself (Chapter 5). In Chapter 4 we described how a structure-oriented editor could be generated for constructing visual terms. In order to accommodate multi-dimensional syntax common in visual languages, we extended the usual notion of editing by allowing sharing of similar sub-sentences (of appropriate sorts). In order to preserve the appearances after rewriting, we introduced Share-Where maintenance which annotates terms with information that is used in constructing output terms (Chapter 6). Finally, in Chapter 7 we introduced an extension of the term rewriting system model to support interaction.

Constructing visual terms with the structure-oriented editor, would in fact, be quite cumbersome. Such editors have proven to be unsuccessful in the case of textual languages. With visual languages they may be somewhat more acceptable due to the complexity of visual syntax and the lack of a standard set of representations like a character set as in textual languages. Nevertheless, we have not been so concerned with the efficiency of how visual terms could be constructed in as much as that they could somehow be constructed. Our focus was on how visual languages could be specified and what could be done with constructed visual terms, such as rewriting and pretty printing. It is on these points that this work has attempted to make some contributions. This having been said, we feel that the most interesting aspects of this work can be summarized as:

- VODL: a language for defining parameterized visual objects.
- A visual specification formalism for visual languages.
- Use of domain specific syntax in language specification.
- Pretty printing of rewritten terms with Share-Where.
- Specification of interaction.

We have striven to advocate the use of visual methods during the specification process as well as in end-user environments in the spirit of [46, 24, 51]. Note that most of the above points can be considered orthogonal to the underlying choices of language formalism. They are methods that could also be employed by other formalisms.

9.2 Discussion

In this section we present some observations regarding our formalism and approach and outline some limitations.

9.2.1 Limitations

Our approach relies on handling context-sensitive aspects of languages, such as type-checking, with equations as part of the language semantics. Dealing with

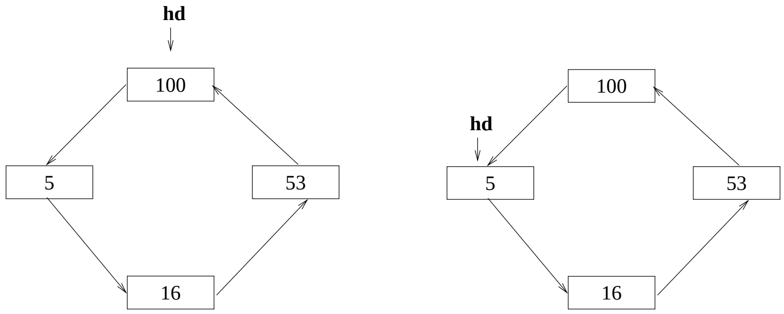


Figure 9.1: Desired visual representation for circular lists.

context-sensitive syntax in a similar manner is not necessarily ideal. We will try to explain this using an example of circular lists. Figure 9.1 shows a possible desired visual representation for circular lists. The graphical notation $\overset{\text{hd}}{\downarrow}$ is used to show the head of the list marker. The list on the left shows a configuration of the circular list and the list on the right shows the effect of a 'rotate right' function applied to the list. The intended meaning is clear in these pictures. But, how can we i) construct such a visual representation, ii) keep the head marker properly adjusted over the head of the circular list and iii) define the desired layout for circular lists.

In order to examine these problems, let us first consider the module in Figure 9.2, which defines a straight forward specification of a circular-list data-type. While we can define syntax that allows the creation of circular lists, we can not easily define the needed checking that assures that the head pointer is spatially appropriately related to the list. Furthermore, we also run into problems when pretty printing circular lists as we shall explain.

The Figure 9.3 shows a constructed circular list term using this syntax. Figure 9.4 shows the result of applying the $\gg$ on the circular list shown in Figure 9.3. The resulting presentations can be different depending on how the editor chooses to use the Share-Where information. In the figure, the different presentations result from the choice of attempting to maintain the *head* marker or the arrow properties stable. The circular list on the left is certainly misleading.

We will now specify an alternate syntax (shown in Figure 9.5) in a way that the empty-list marker ("tail-end of list") denoted by $\overset{\text{hd}}{\downarrow}$ can share the *head* ("beginning of the list"). With this specification we will force the head marker to be linked and move along with an item that belongs to the circular list. Again, no particular layout for the circular list considered in the specification. Then we appropriately enhance the semantics of such a circular list so that some head-adjustment can be done before pretty-printing the picture. We intend to make the tail-end marker

imports Integers
sorts *ITEM CL*
functions

$$
\begin{array}{rcl}
\boxed{Int} & \rightarrow & ITEM \\
\overset{\text{hd}}{\downarrow} & \rightarrow & CL \\
ITEM \longrightarrow CL & \rightarrow & CL \\
head(CL) & \rightarrow & ITEM \\
\gg(CL) & \rightarrow & CL
\end{array}
$$

variables

$$
\begin{array}{rcl}
I,\, I_0,\, I_1,\, I_2 & \rightarrow & ITEM \\
C & \rightarrow & CL
\end{array}
$$

equations

$$
\begin{array}{lrcl}
[\text{c1}] & \gg(\overset{\text{hd}}{\downarrow}) & = & \overset{\text{hd}}{\downarrow} \\
[\text{c2}] & \gg(I \longrightarrow \overset{\text{hd}}{\downarrow}) & = & I \longrightarrow \overset{\text{hd}}{\downarrow} \\
[\text{c3}] & \gg(I_1 \longrightarrow I_2 \longrightarrow C) & = & I_2 \longrightarrow \gg(I_1 \longrightarrow C) \\
[\text{c4}] & head(I \longrightarrow C) & = & I
\end{array}
$$

Figure 9.2: The specification for *CList* language.

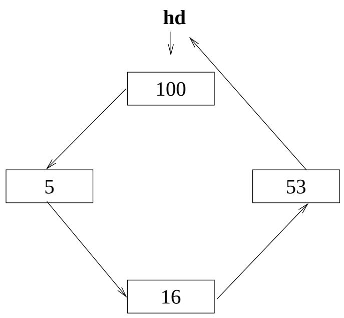


Figure 9.3: A circular list constructed with syntax shown in Figure 9.2.

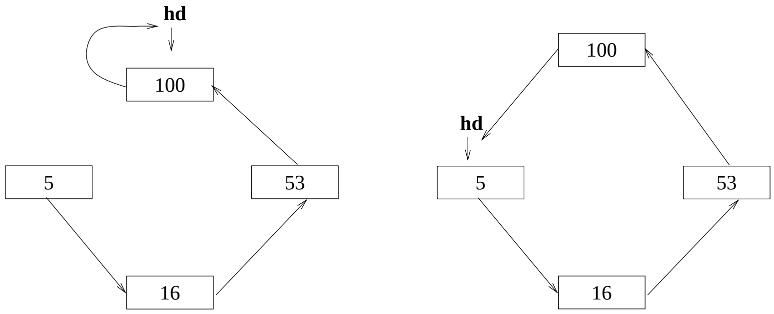


Figure 9.4: Two of the possible pretty printed terms after applying $\gg$ to the circular list shown in Figure 9.3.

share an item of the list. The semantics will ensure that the sharing will only share the head of the list if any. An empty circular list is denoted by $\overset{\text{hd}}{\underset{\boxed{\otimes}}{\downarrow}}$.

The rules [c1′]-[c4′] are very similar to the rules [c1]-[c4]. The additional rules specify the semantics of head-adjustment function *headadj*. These equations are used to re-adjust the circular list to maintain the head adjustment. The equation [c5′] assures that if there is only one item in the list that it is the empty item (a circular list has at least two items one of which is the empty item). Equation [c6′] causes the first item and the last item to be the one and the same using the function *sharehd*. The function *sharehd* is defined in two equations, one considering the case of traversing to the end of the list ([c8′]) and the other defining the head marker to be over the last item (also the first item).

This specification allows pictures like the desired ones shown in Figure 9.1 to be presented. The assumption is that the head-adjustment function *headadj* will be used before presenting a modified circular list. Such an approach forces the specifier to consider the presentation problems and define a method that results in the appropriate picture. Note that some such specification is essential in any formalism to cause the head marker to move over another item as a result of the application of an equation and thus the sharing of the tail-end with head-end as desired.

Another problem is that the given syntax definitions permit other various sentences with strange appearances to be constructed, since the arrows are, in fact, connecting an item to another circular list. This could lead to highly confusing pictures as it would be possible to attach the arrow to any place that is considered as part of the list. This problem could be alleviated using collections instead of the list constructed in these specifications. Another approach could be to extend the VODL language by denoting a sub-*vod* as a *reference vod* to indicate attachment *vod*s (see first point of future work in Section 9.3). In this example,

imports Integers
sorts $ITEM'$ CL' Int'
functions

$$
\begin{array}{rcl}
Int & \rightarrow & Int' \\
\otimes & \rightarrow & Int' \\
\boxed{Int'} & \rightarrow & ITEM' \\
\begin{array}{c}\text{hd}\\ \downarrow \\ ITEM'\end{array} & \rightarrow & CL' \\
ITEM' \longrightarrow CL' & \rightarrow & CL' \\
head(CL') & \rightarrow & ITEM' \\
\gg(CL') & \rightarrow & CL' \\
headadj(CL') & \rightarrow & CL' \\
sharehd(CL', ITEM') & \rightarrow & CL'
\end{array}
$$

variables

$$
\begin{array}{rcl}
I,\, I_0,\, I_1,\, I_2 & \rightarrow & ITEM' \\
C & \rightarrow & CL'
\end{array}
$$

Figure 9.5: The syntax specification for $CList'$ language.

equations

$$
\begin{array}{lrcl}
[\mathrm{c1}'] & \gg(\begin{array}{c}\text{hd}\\ \downarrow\\ I_0\end{array}) & = & \begin{array}{c}\text{hd}\\ \downarrow\\ I_0\end{array} \\[1em]
[\mathrm{c2}'] & \gg(I \longrightarrow \begin{array}{c}\text{hd}\\ \downarrow\\ I_0\end{array}) & = & I \longrightarrow \begin{array}{c}\text{hd}\\ \downarrow\\ I_0\end{array} \\[1em]
[\mathrm{c3}'] & \gg(I_1 \longrightarrow I_2 \longrightarrow C) & = & I_2 \longrightarrow \gg(I_1 \longrightarrow C) \\[1em]
[\mathrm{c4}'] & head(I \longrightarrow C) & = & I \\[2em]
[\mathrm{c5}'] & headadj(\begin{array}{c}\text{hd}\\ \downarrow\\ I\end{array}) & = & \begin{array}{c}\text{hd}\\ \downarrow\\ \boxed{\otimes}\end{array} \\[1em]
[\mathrm{c6}'] & headadj(I \longrightarrow C) & = & sharehd(I \longrightarrow C, I) \\[1em]
[\mathrm{c7}'] & sharehd(I_1 \longrightarrow \begin{array}{c}\text{hd}\\ \downarrow\\ I\end{array}, I_0) & = & I_1 \longrightarrow \begin{array}{c}\text{hd}\\ \downarrow\\ I_0\end{array} \\[1em]
[\mathrm{c8}'] & sharehd(I_1 \longrightarrow I_2 \longrightarrow C, I_0) & = & I_1 \longrightarrow sharehd(I_2 \longrightarrow C, I_0)
\end{array}
$$

Figure 9.6: The semantics specification for *CList′* language.

we have not used collections to avoid taking advantage of the special properties of the collection primitive in order to fully demonstrate the difficulties presented regarding limitations under consideration.

9.2.2 Use of visual notation in specification

The use of visual syntax leaves room for misinterpretation by the reader. Some people find it rather difficult to read *visual* visual specifications. Frequently, but certainly not exclusively, these people are more formally oriented and feel the need to understand the precise correspondence between the chosen notation of the specification and the formalism. The problem stems from inferring unintended relations from a picture.

Typically, there are many visual representations for the same specification or program. It could be difficult for a reader to distinguish among the physical aspects that are semantically relevant from those that are incidental. Often arbitrary decisions are needed to be made for graphical attributes for the sole purpose of rendering – such as the use of a color. On the other hand, in some cases, the use of a color could be quite intentional and semantically relevant. Such a relevancy is part of the definition of the language syntax and is reflected in the abstract syntax. So, even if there is no problem regarding the formalism, it may very well pose a problem for the user's perception.

The motivation for having domain specific (in this case visual) syntax in semantic specification is for providing a connection between the definitions and what they are defining by using familiar notation. This allows an immediate recognition and connection between our mental representation and how the specification looks. But clearly caution needs to be exercised when choosing representations to avoid undue confusion.

One of the big challenges in using visual syntax is when we need to represent abstractions. For example, in our specifications we use variables to represent some arbitrary visual terms of a given sort. The question of how to represent these variables arises. Our approach allows any arbitrary representation, however, we have usually chosen to rely upon textual variable names to reduce the overhead of more visual notation. When we use visual notation we run a greater risk that the reader would not distinguish this as a variable. There could be some appropriate visual notation that could be used as a convention. Due to the clear conflict between the concreteness of visual languages and abstraction, such representations must be chosen carefully. This is an issue of further investigation.

The choice of which visual notation and what kind of spatial relations to use is very important. While visual syntax can be very powerful in relaying concepts or objects and their relations effectively, when the chosen notation is confusing it can do quite the opposite. Wang, in her thesis [83] has some nice discussions and suggestions regarding some criteria to adhere to.

The use of visual notation in computer languages is still rather new although rapidly gaining speed. With time we shall see if getting accustomed to visual notation will bring about useful standards and styles leading to good design and use of visual notation. There are some attempts to harvest knowledge from graphics design such as [47] and cognitive aspects such as in [31, 83, 84, 32] of perception to meet this end. We certainly claim no expertise in this area and are interested in tools we may be able to offer such experts.

9.2.3 Application of Share-Where maintenance

We developed a technique called Share-Where maintenance to be used in displaying terms (Chapter 6). This technique maintains information regarding shared sub-terms created by the user during editing and where these visual terms were created. Share-Where is used to get the needed information regarding the original visual term responsible for each (sub)term's creation, which can either be during initial visual term creation in an end-user editor or in an equation editor. The latter comes into play during term rewriting. The goal was to get pictures that are similar in the appearance to the initial picture when the same sub-pictures are present in both. In the case that the sub-picture is created during rewriting then the appearance in the equation that creates it is used since that is the only possible preference that exists in the system. This technique is used in order to produce fairly similar pictures when this is applicable. Ignoring this would likely lead to arbitrary representations of visual terms where the connections are lost.

Share-Where annotations are employed only in pretty printing terms. They could also be exploited for other purposes such as the use of the *Share* information in matching to recognize cases where sharing occurs in a picture graph. This would enable one to specify when a picture graph is considered acceptable. This kind of use would be for simulating graph rewriting as found in graph grammars. See Figure 9.7 for a specification which checks whether a circular list is acceptable using a predefined **share** predicate, by checking if the head and the tail-end items are shared. As this was not our aim we have not considered its use in this work so as not to disturb the underlying formalism.

For example, Share-Where could be extended with another case called *Shared Variables/sub-terms*. When variables (non-left linear case) are shared on the left hand side or sub-terms are shared on the left hand side then the rule is transformed to a conditional rule that checks for the sharing of these. For the left hand side we can check for sharing and for the right hand side introduce sharing. For example, consider the following function definitions for f and g:

$$
\begin{array}{llcl}
[S1] & f(I) & = & \dots \\
[S2] & \dots & = & g(I)
\end{array}
$$

where I is a variable of sort INT. In the equation $[S1]$, we may want to check

imports CList′
sorts *CLOkay*
functions

$$\begin{array}{rcl} check(CL') & \rightarrow & CLOkay \\ okay & \rightarrow & CLOkay \\ tailitem(CL') & \rightarrow & ITEM' \end{array}$$

variables

$$\begin{array}{rcl} I & \rightarrow & ITEM' \\ C & \rightarrow & CL' \end{array}$$

equations

$$\begin{array}{llrcl}
[\mathrm{ch1}] & & check\left(\begin{array}{c}\mathrm{hd}\\ \downarrow \\ \boxed{\otimes}\end{array}\right) & = & okay \\
[\mathrm{ch2}] & & check(I \longrightarrow C) & = & okay \\
 & \textbf{when} & \mathbf{share}(I, tailitem(C)) & = & true \\
[\mathrm{ch3}] & & tailitem(I \longrightarrow C) & = & tailitem(C) \\
[\mathrm{ch4}] & & tailitem\left(\begin{array}{c}\mathrm{hd}\\ \downarrow \\ I\end{array}\right) & = & I
\end{array}$$

Figure 9.7: A specification that does share-checking for *CList* language.

for sharing using the **share** predicate so that the equation will match a term only when there is sharing. And in the equation $[S2]$ sharing could be introduced giving the same share annotations for both Is. *Share* annotations could be further exploited and created in such cases. Such extension, however, would be changing the rewriting model.

9.3 Future directions

There are numerous future directions for this work. Along with the completion of a prototype implementation, there are also other extensions and features to be considered. In this section we will mention some of these considerations.

- Considering the *vod* language, in order to be able to have more general definitions it would be useful to have ranges of values. The *vod*s considered so far are two-dimensional. Two and a half and three dimensional *vod*s should be examined as well. Also, when there is a spatial relation defined between sub-*vod*s which themselves are composite, it may be useful to denote a *vod* within a sub-*vod* as a reference *vod* to which the relation will be applied. For example, in the following *vod* definition:

$$\begin{array}{l}
\textbf{defv}\ \textit{Vod1()} \\
\{\ \ \mathrm{a} : \textit{rectangle}()\ \{\textbf{ref-vod}\}\textbf{,} \\
\quad \mathrm{b} :\ \ \textit{oval}() \oplus [\mathrm{width} = 20] \\
\} \lhd \{ \\
\quad \mathrm{a}\ \textbf{followed-by}\ \mathrm{b} \quad \}
\end{array}$$

the sub-*vod* labeled 'a' is marked as the refence *vod* with the *ref-vod* marker. When the *vod Vod1* is used in another definition such as:

$$\begin{array}{l}
\textbf{defv}\ \textit{Vod2()} \\
\{\ \ \mathrm{v1} :\ \textit{Vod1}(), \\
\quad \mathrm{v2} :\ \textit{oval}() \\
\} \lhd \{ \\
\quad \mathrm{v1\ touches\ v2} \quad \}
\end{array}$$

the touches relation would be applied to the *rectangle* sub-*vod* of v1 rather than the entire composite *vod*. This would allow denoting a general property of the *vod* as an attachement place. Of course, if we wanted to define it more specifically, we could have defined the touches relationship with the rectangle in the constraints.

Finally, regarding VODL, the specification of constraints needs to be analised further both for the necessary restrictions in reality and for its limited expressive ability (e.g., there is no disjunction or negation operation in it). The use of spatial graphs (see Section 1.4.2) of graph grammars instead of VODL is worth investigating.

- Alternate editors for specifications that reflect the structure and spatial relationships of the syntax. Although our visual editors can build graph-like structures (as shown in figure 3.10), our choice of ASF for the meta level leads to an underlying tree representation and tree processing. Although effectively working with trees could limit the extent to which such an environment could be useful, the approach itself is independent of a tree based underlying formalism and could be used in a similar manner with a graph based underlying formalism.

- We have focused on the interactive construction of programs and examined how we can preserve the presentations that come about due to the user's choices. Another consideration is pretty printing the programs according to some criteria or some predefined description of preferred layout for that language, such as in [13].

- There are many graphical tools that would be useful in programming environments, whose generation would be very attractive: debuggers, animators, editors for constructing graphical constructs, browsers, viewing hierarchical structures such as import graphs and class structures in object-oriented systems, searching for visual constructs in a document, etc.

- It is difficult to define spatial relations among items of collections in a manner that governs the overall look of a collection. This kind of pretty printing is in need of context-sensitive information. For example, with the circular lists, it would be desirable to present the items in a circular fashion, where depending on the number of items in the list the placement of individual items would be determined. It may be useful to define separate pretty printing specifications that could be applied to terms and print them according to the defined preferences, such as in [14, 15].

- For the maintenance of specifications, the design and development of a debugger and an animator for such an environment generator are also interesting. A debugger would be needed when developing specifications. Also, tools that help maintain the specification in the face of changes are needed. There is a need to handle the effects of any changes in the visual syntax of a given language in such a way that minimal modification of the semantics specification is necessary. For example, should a syntax rule be modified then the corresponding equations that use that syntax must also be modified to match the new syntax. Such equations should automatically be identified so that they can be updated. This problem would not be so significant if a parser generator was available that would generate visual parsers. Availability of a parser allows both the free form or hybrid construction of programs. Work in this area that could be influential is, among others, [61, 48] for context sensitive languages and [86] for context-free languages.

- The Share-Where maintenance introduced in Chapter 6 was used only in relation to presentation. The utility and implications of using Share-Where more extensively should be examined. One possibility was already mentioned in Section 9.2.3.

- Finally, an assessment of the appropriateness of the VAS formalism is needed. As we have discussed earlier, visual representations are highly concrete and powerful in affecting a user's perception. This can be positive or negative depending on what this implies for the perception of a user and this may very well vary from user to user. An assessment study for determining the usability of the VAS formalism would be very interesting. Such an assessment should at least cover the understandability of domain specific languages defined using VAS.

Bibliography

[1] A. Aho, B. Kernighan, and P. Weinberger. *The Awk Programming language.* Addison-Wesley, 1988.

[2] M. Andries. *Graph Rewrite Systems and Visual Database Languages.* PhD thesis, Leiden University, February 1996.

[3] M. Andries, G. Engels, and J. Rekers. How to represent a visual program? In *Workshop on Theory of Visual Languages*, May 1996.

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

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

[6] J. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[7] J. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[8] J. Bergstra and J. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[9] J. A. Bergstra and P. Klint. The Discrete Time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, 1996.

[10] Y. Bertot. A canonical calculus of residuals. In G. Huet and G. Plotkin, editors, *Logical Environments.* Cambridge University Press, 1993.

[11] P. Borenstein and J. Mattson. *THINK C User's Manual*, 1989.

[12] A. Borning. Graphically defining new building blocks in ThingLab. *Human Computer Interaction*, 2:269–295, 1986.

[13] M. G. J. v. d. Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996. http://www.acm.org/pubs/toc/TOC/1049-331X/Vol5.html.

[14] M. v. d. Brand. Prettyprinting without losing comments. Technical Report P9315, Programming Research Group, University of Amsterdam, 1993. http://www.fwi.uva.nl/research/prog/reports/P9315.ps.Z.

[15] 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. http://www.fwi.uva.nl/research/prog/reports/1994/P9420.ps.Z.

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

[17] A. v. Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[18] A. v. Deursen, J. Heering, and P. Klint. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST series in Computing*. World Scientific, 1996.

[19] T. B. Dinesh and S. Üsküdarlı. Guiding user-interfaces equationally. In *Proceedings of the ERCIM Workshop on User Interfaces for All*, pages 120–132, November 1996.

[20] T. B. Dinesh and S. M. Üsküdarlı. Visual Object Definition Language. In R. C. Veltkamp and E. H. Blake, editors, *Proceedings of the fifth Eurographics workshop on Programming Paradigms in Graphics*, Eurographics'95, pages 109–124, Amsterdam, September 1995. CWI. Held in Maastricht, The Netherlands.

[21] T. B. Dinesh and S. M. Üsküdarlı. Specifying input and output of visual languages. In *Workshop on Theory of Visual Languages*, Gubbio, Italy, May 1996.

[22] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 415–431. Springer-Verlag, 1994.

[23] Frame Technology Corp. FrameMaker. Software product of Frame Technology Corp, `e-mail: custserv@fram.com`.

[24] G. W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of the CHI '91*, pages 71–78, April 1991.

[25] D. Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Programming*. PhD thesis, SUNY Stony Brook, Department of Computer Science, 1983.

[26] E. P. Glinert, editor. *Visual Programming Environments: Applications and Issues*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[27] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading MA, 1984.

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

[29] E. J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.

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

[31] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In *Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*, 1992.

[32] T. R. G. Green and M. Petre. Usability analysis of visual programming environments. *J. Visual Languages and Computing*, 7:131–174, 1996.

[33] C. Gurr. On the isomorphism (or otherwise) of representations. In *Workshop on Theory of Visual Languages*, May 1996.

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

[35] J. Heering and P. Klint. The syntax definition formalism SDF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 283–297. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 6.

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

[37] R. Helm, K. Marriott, T. Huynh, and J. Vlissides. An object-oriented architecture for constraint-based graphical editing. In C. Laffra, E. Blake, V. de Mey, and X. Pintad, editors, *Object-Oriented Programming for Graphics*, pages 217–238. Springer-Verlag, 1995.

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

[39] S. P. Jones and P. Wadler. Imperative functional programming. *Proceedings of 20th ACM Symposium on Principles of Programming Languages*, pages 71–84, Jan. 1993.

[40] K. Kahn and V. Saraswat. Complete visualizations of concurrent programs and their executions. In *IEEE 1990 Workshop on Visual Languages*, pages 7–14, 1990. See also Technical Report SSL-90-38 [P90-00099], Xerox Palo Alto Research Center.

[41] B. W. Kernighan. *A Graphics Language for Typesetting: User Manual*, May 1991.

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

[43] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[44] 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.

[45] J. W. C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments*. PhD thesis, University of Amsterdam, 1994.

[46] F. Lakin. Visual grammars for visual languages. In *Proceedings of the 6th Nat. Conf on Artificial Intelligence*, pages 683–688, Seattle, WA, 1987.

[47] H. Lieberman. The visual language of experts in graphic design. In *Proceedings of the 1995 IEEE Symposium Visual Languages*, pages 5–12, September 1995.

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

[49] K. Marriott and B. Meyer. Toward a Hierarchy of Visual Languages. In *Proceedings of the 1996 IEEE symposium on visual languages*, pages 196–203, Boulder, Colorado, USA, 1996.

[50] K. Marriott and B. Meyer. Towards a hierarchy of visual languages. In *Workshop on Theory of Visual Languages*, May 1996.

[51] D. W. McIntyre. *A Visual Method for Generating Iconic Programming Environments.* PhD thesis, Rensselaer Polytechnic Institute, Troy, N.Y., 1992.

[52] D. W. McIntyre and E. P. Glinert. Visual tools for generating iconic programming environments. In *Proceedings of the 1992 IEEE Workshop Visual Languages*, pages 162–168, Seattle, WA., Sept. 1992.

[53] Microsoft Corporation. Microsoft visual basic white paper. Web page, 1996. http://www.microsoft.com/VBASIC/vbwhite/vbwhite.htm.

[54] 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.

[55] M. A. Najork. *Programming in Three Dimensions.* PhD thesis, Dept. of Computer Science, Univ. of Illinois, 1993.

[56] M. A. Najork and S. M. Kaplan. Specifying visual languages with conditional set rewrite system. In *Proceedings of the 1993 IEEE Symposium Visual Languages*, pages 12–18, 1993.

[57] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, USA, 1994.

[58] Pictorius Incorporated. Welcome to Pictorius. Web page, 1996. http://www.pictorius.com/pi/welcome/welcome.html.

[59] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[60] S. P. Reiss. Integration mechanisms in the field environment. Technical report, Brown University, 1988.

[61] J. Rekers and A. Schürr. A parsing algorithm for context-sensitive graph grammars – short version. In *Proceedings of the 1995 IEEE Symposium Visual Languages*, September 1995.

[62] A. Repenning and W. Citrin. Agentsheets: Applying grid-based spatial reasoning. In *IEEE 1993 Workshop on Visual Languages*, pages 77–82, 1993.

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

[64] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[65] R. Robinson, D. Cook, and S. Tanimoto. Programming agents with visual rules. In *1995 IEEE Symposium on Visual Languages*, pages 13–20. IEEE Computer Society Press, September 5-9 1995.

[66] G. v. Rossum. Python Language Home Page. Web page. http://www.python.org/.

[67] 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.

[68] M. Sannella. The skyblue constraint solver and its applications. In *Informal Proceedings of Principles and Practice of Constraint Programming*, pages 258–268, 1993. http://joinus.comeng.chungnam.ac.kr/ dolphin/db/indices/a-tree/s/Sannella:Michael.html.

[69] 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.

[70] A. Schürr. PROGRES, a visual language and environment for programming with graph rewrite systems. Technical report, RWTH Aachen, 1994.

[71] A. Schürr, A. Zndorf, and A. Winter. Visual programming with graph rewriting systems. In *Proceedings of the 1995 IEEE Symposium Visual Languages*, September 1995.

[72] D. C. Smith, A. Cypher, and J. Spohrer. Kidsim: Programming agents without a programming language. In *CACM*, volume 37, pages 54–68, 1994.

[73] S. Sutanthavibul. Xfig. Software available by *ftp* from `ftp.x.org:contrib/applications/drawing_tools/xfig`.

[74] M. Taiafat. Constraint solving in visual editors. Technical report, University of Amsterdam, April 1996. Masters thesis.

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

[76] S. M. Üsküdarlı and T. B. Dinesh. Towards a visual programming environment generator for algebraic specifications. In *Proceedings of the 1995 IEEE Symposium Visual Languages*, September 1995.

[77] S. M. Üsküdarlı and T. B. Dinesh. VAS Formalism in VASE. In *Proceedings of the 1996 IEEE Symposium Visual Languages*, pages 140–147, September 1996.

[78] M. G. J. van den Brand, P. Klint, P. Olivier, and E. Visser. Aterms: representing structured data for exchange between heterogeneous tools. Technical report, University of Amsterdam, 1997. to appear.

[79] A. van Deursen. Origin tracking for system renovation. In A. van Deursen, P. Klint, and G. Wijers, editors, *Program Analysis for System Renovation*, Resolver Release 1, chapter 15, pages 15–1 – 15–20. CWI, Amsterdam, January 1997.

[80] J. van Wijk and R. van Liere. An environment for computational steering. Technical Report CS-R9448, Center for Mathematics and Computer Science (CWI), 1994. Presented at the Dagstuhl Seminar on Scientific Visualization, 23-27 May 1994, Germany, proceedings to be published.

[81] H. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. ftp://ftp.cwi.nl/pub/gipe/reports/Wal91.ps.Z.

[82] H. Walters and J. Kamperman. A model for I/O in equational languages with don't care non-determinism. Technical Report CS-R9572, CWI, December 1995. Available also at URL: http://www.cwi.nl/cwi/publications/index.html#AP.

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

[84] D. Wang and H. Zeevat. A syntax directed approach to picture semantics. In *Workshop on Theory of Visual Languages*, May 1996.

[85] D. H. D. Warren. Logic programming and compiler writing. *Software-Practice and Experience*, 10(2):97–125, 1980.

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

The proceedings for the *Workshop of the Theory of Visual Languages* does not include global page numbers and hence no page numbers are indicated for citations from this source.

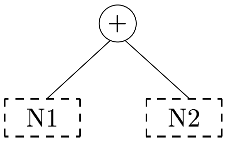
Samenvatting

Steeds meer programmeertalen worden ondersteund door grafische programmeeromgevingen, ook wel visuele programmeeromgevingen genoemd, die hulp bieden bij het schrijven, wijzigen en uitvoeren van programma's in deze talen. Bij het ontwikkelen van nieuwe omgevingen wil men zoveel mogelijk voortbouwen op eerder verworven expertise bij het bouwen van vergelijkbare omgevingen. Een manier om dit te bereiken is om een dergelijke omgeving automatisch te genereren uit een specificatie van een taal. Een taal kan gespecificeerd worden met behulp van een formalisme waarin syntax en semantiek kunnen worden uitgedrukt.

Dit proefschrift behandelt de generatie van programmeeromgevingen voor visuele talen uit specificaties van deze talen. Uitgangspunt is dat het wenselijk is deze specificaties te schrijven in een formalisme dat (a) het gebruik van visuele notatie toestaat; en (b) zelf ondersteund wordt door een visuele omgeving. Dit proefschrift bespreekt een nieuw formalisme dat aan deze eisen voldoet: De visuele specificatietaal VAS (Visual Algebraic Specification), dat ondersteund wordt door de visuele specificatie-omgeving VASE (VAS Environment). De omgeving VASE bestaat enerzijds uit editors om de syntax en semantiek van visuele talen te definiëren, en anderzijds uit een editorgenerator. Met behulp van deze generator worden, gegeven de specificatie van de taal, taal-specifieke editors gegenereerd, waarmee een eindgebruiker termen over deze taal kan invoeren en evalueren. In de paragrafen die volgen bespreken we hoe dit in zijn werk gaat, aan de hand van een voorbeeldspecificatie van zogenaamde rechtslineaire binaire bomen. Dat zijn bomen waarvan de linkerkinderen uitsluitend uit bladeren bestaan.

Het definiëren van syntax

Om de syntax van een taal te definiëren moeten we eerst beschrijven hoe de taalconstructies er uit zien. In VAS doen we dit door aan te geven wat de grafische lexicale tekens zijn, die we *vods* noemen (*Visual Object Definitions*). Dit kunnen primitieve grafische objecten zijn zoals rechthoeken en cirkels. Ook kunnen zij opgebouwd worden uit andere sub-*vods*, waarbij grafische voorwaarden aangegeven kunnen worden, zoals "is bevat in" en "sluit aan op". Een *vod* wordt beschreven in de taal VODL (Visual Object Definition Language). De volgende figuur is een geparameteriseerde *vod* die gedefinieerd is met behulp van een VODL-editor. De parameters zijn aangegeven middels de gestippelde rechthoeken met daarin de naam van de parameter.



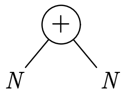
De syntax voor binaire bomen, die gebruik maakt van deze *vod* definitie, kan met behulp van het VAS formalisme alsvolgt beschreven worden:

module RLinear-BTree

imports Integers

sorts L N

functions

INT	$\rightarrow$	L
L	$\rightarrow$	N
	$\rightarrow$	N
$\curlywedge\ (N)$	$\rightarrow$	N

variables

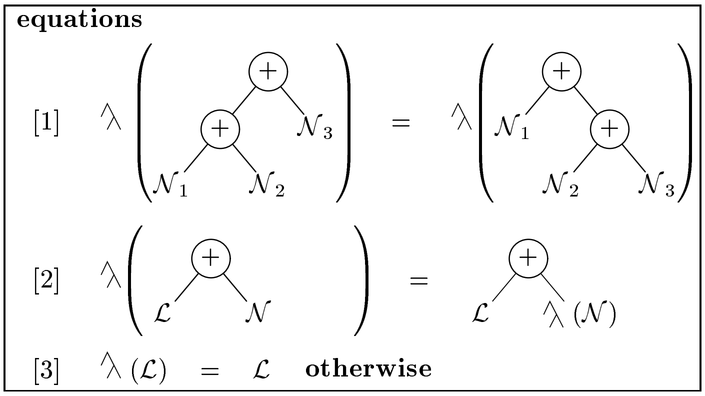
$\mathcal{L}$	$\rightarrow$	L
$\mathcal{N}$	$\rightarrow$	N

Deze regels definiëren zowel de soorten van de taal als hun presentatie. De derde functiedefinitie gebruikt de eerder gedefinieerde *vod*-definitie, waarbij beide parameters nu vervangen zijn door de soort N. De functie $\curlywedge(N)$ beeldt een argument van soort N af op een resultaat dat ook van soort N is.

Module RLinear-BTree begint met het importeren van module "Integers" waarin eenvoudige rekenkundige operaties zijn gedefinieerd. Daarna worden twee soorten genaamd L en N, respectievelijk voor bladeren en knopen, geïntroduceerd. De twee variabelen $\mathcal{L}$ en $\mathcal{N}$ die over deze soorten worden gedefinieerd worden gebruikt in de semantische vergelijkingen.

Het definiëren van semantiek

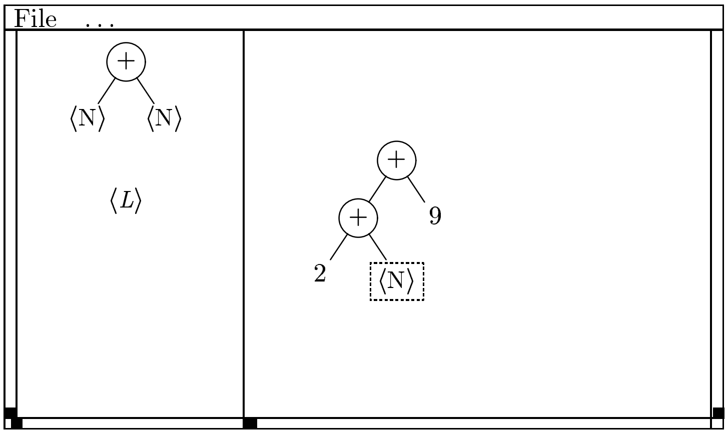
De volgende figuur geeft de definitie van de semantiek voor rechtslineaire binaire bomen.



De vergelijkingen gebruiken de visuele notatie van de gedefinieerde taal. Zij definiëren de functie $\lambda(N)$ zodanig dat deze een willekeurige binaire boom omzet in een rechtslineaire boom. Dankzij de gebruikte visuele notatie is duidelijk te zien hoe de verschillende vergelijkingen de boomstructuur manipuleren zodat deze rechtslineair wordt.

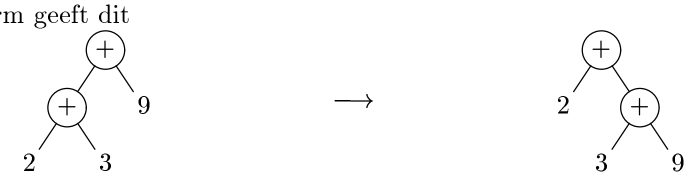
Gegenereerde omgevingen

Gegeven de specificatie van een taal wordt een editor gegenereerd waarmee de eindgebruiker programma's kan schrijven. Deze programma's kunnen worden geëvalueerd (herschreven) met behulp van een termherschrijfsysteem. De volgende figuur toont een instantiatie van een *BTree term editor*:



Bij het construeren van termen wordt de eindgebruiker geleid door de syntax, gebruikmakend van een *focus* (aangegeven door een gestippeld vierkant). De focus kan vervangen worden door een van de toegestane expansies, zoals te zien is in het *selectiepaneel* in de linkerhelft van de editor.

De op deze manier geconstrueerde term kan vervolgens gereduceerd worden door de vergelijkingen uit te voeren als herschrijfregels. Voor de hierboven ingevoerde term geeft dit



In dit proefschrift gaan we dieper in op de problemen die komen kijken bij het visueel definiëren van talen, en bij het afleiden van visuele gereedschappen uit dergelijke definities. De diverse hoofdstukken bespreken de voordelen van het gebruik van visuele talen en de technieken die visueel editen mogelijk maken. Er worden twee nieuwe formalismes voorgesteld: VODL om elementaire plaatjes op te bouwen, en VAS om visuele talen in te definiëren, ondersteund door de omgeving VASE. In de latere hoofdstukken worden twee moeilijke problemen behandeld: hoe kan de sharing die in visuele notatie zit behouden blijven tijdens het herschrijven van termen volgens de gespecificeerde semantische vergelijkingen, en hoe kunnen vergelijkingen gebruikt worden om interactie met de eindgebruiker te definieren. Het proefschrift wordt afgesloten met een beschrijving van de belangrijkste problemen — en mogelijke oplossingen — die een implementatie van de voorgestelde technieken met zich meebrengt.