# Specifying Input and Output of Visual Languages*

T. B. Dinesh

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

S. Üsküdarlı

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

**Abstract**

*We consider "context free" specification of visual syntax – with the aid of visual (i.e., 2-dimensional and user-defined) "lexicals". We argue that context-free visual syntax specification can go a long way towards specification of visual languages in the presence of certain intelligent tools. We focus on the input and output behavior of visual languages in demonstrating this point.*

*The usual notion of term construction in a editor is extended by allowing sharing of sub-terms. We demonstrate the utility of this for constructing input terms (programs) of visual languages. We also allow such sharing of sub-terms in equations editors which are used to specify language semantics. This sharing not only allows the building of appropriate (visual) terms but also provides information on the desired view of an output term. We introduce a technique called "Share-Where" which is used to preserve layout information and demonstrate its use as well as its limitations.*

*We then describe an extension to our formalism which allows dynamic input and output – interaction during term execution. We demonstrate this with a full example for the* FSA *language to animate* FSA *term evaluation.*

## 1 Introduction

Algebraic specifications are popular due to their simplicity. They can be executed by orienting equations as left-to-right rewrite rules as well as support building language specific programming environments based on the language specifications [Kli93]. Visual algebraic specifications are considered as a formalism for specifying visual languages [ÜD95], where among other tools, generic syntax directed editors for constructing visual terms can be generated [Üsk94].

*First version presented in Workshop on Theory of Visual Languages in conjunction with AVI'96. Gubbio, Italy (May 1996). This is an extended version that also describes "Share-Where" maintenance for pretty printing purposes.

In this paper, we first introduce the visual algebraic specification formalism by explaining the use of 2-dimensional lexicals for the specification of syntax and semantics with the aid of an example. Then we show how certain "Share" information that is produced by sharing certain items in an editor can be maintained and then used for pretty-printing the result term using the "Where" information. We then examine an extension of algebraic specification formalism with input and output.

Algebraic specifications, like functional languages, suffer from the lack of constructs that allow interaction with an user during execution. 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. Our extension to the visual specification formalism for input and output, not only permits interaction during execution but also for user definable specification of a language interface itself. We concentrate on 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 illustrate that the desired interaction with an user, during the execution of a visual program, can be seen as the process of editing (creating) an appropriate visual term. Some fine tuning aspects of the particular appearances of such an editor are still delegated to user-interface specification.

## 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 [Kli93] approach used for 1-dimensional (textual) languages and gives a specification of allowable visual terms that are context free. We extend the classical notion of term building by allowing sharing of similar terms on demand, in a syntax directed editor, as long as the constraints of the governing visual lexicals allow the desired placement of the term. We will demonstrate the power and limitations of this kind of sharing with aid of examples.

We then allow terms built over this syntax to define the semantics of a visual language. We introduce and illustrate *Share-Where* annotations and their utility for visual semantic specification w.r.t. the presentation of concrete terms. We also specify an interactive animator that illustrates the use of *Share-Where* for presentation of terms during input and output.

## 1.2 Syntax v/s Semantics

Typically visual languages are syntactically very complex. The sharing of terms among several terms leads one to consider context-sensitive languages. Most approaches to syntax specification have adopted a context-sensitive formalisms considering context-free specifications not useful in dealing with realistic languages. This point can be observed in [MM96] 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 aim of this work is to examine the utility of extending algebraic specifications in meeting the realistic demands of interactive tools for visual languages which can be generated from their specifications. Furthermore, we investigate how we can make use of the existing generated tools to meet these goals.

# 2 Algebraic Specification Formalisms

The algebraic specification formalism ASF+SDF [Kli93] has been successfully used to define language specific environments. Based on this, *Visual Algebraic Specification Formalism* [ÜD96] is defined to handle 2-dimensional syntax. Given the specification, a syntax directed term editor is generated, where terms over the language can be constructed. The specification is interpreted as a term rewriting system yielding executable specifications which a specifier can test with desired terms. The execution of a term is to take the initial term (constructed in the term editor) and rewrite it into its normal form. One aspect that is strongly missed is *interaction during the reduction* of a term. We address this problem, with a simple operational model of input and output which uses only the existing constructs of the formalism. The following two sections very briefly introduce the two concepts of the specification model relevant to the presentations in this paper.

## 2.1 Conditional Rewrite Rules

Conditional equations are used to specify language semantics and occur very frequently in realistic specifications. *Conditional rewrite rules* [BK86] are used to execute conditional equations.

A conditional rewrite rule takes the form $\dfrac{s_1 = t_1, \cdots, s_n = t_n}{s_0 = t_0}$ with $n \geq 0$, and $s_i, t_i$ $(0 \leq i \leq n)$ terms.

There are well-definedness constraints imposed on the variables of the conditions [Wal91, p.16] in order to ensure their definedness during the execution. We define a model for input and output that permits controlled relaxation of of some of these constraints.

## 2.2 Meta-variables

The holes in incomplete terms are represented with meta-variables or place-holders, which are represented by $\langle \mathit{SortName} \rangle$ and can be replaced with any term of the sort SortName. Meta-variables are place holders available while syntax-directed editing. They allow building the intended term via interaction by selecting permissible substitutions of the language constructs. Multiple occurrences of place-holders of the same sort are independent of each other. We make use of meta-variables in conditional equations to indicate input or output as is described in Section 6.1.

# 3 VAS specification

A VAS specification consists of three phases. The first of which is building the visual lexicals of interest using VODL [ÜD95]. The second phase consists of

| | *vod* | Description |
|---|---|---|
| 1 | $v$ | Circle containing a *vod*. |
| 2 | $v$ | a *vod* contained within double circle. |
| 3 | $\Rightarrow v$ | arrow and a *vod*. |
| 4 | $v_1 \xrightarrow{v_2} v_3$ | an arrow connected to two *vod*s. with a *vod* above. |
| 5 | $v_1\ v_2$ | a *vod* consisting of two unconstrained *vod*s – used to define FSA sort. |
| 6 | $v$ | a *vod* with a box over. |
| 7 | $\boxed{v}$ | *vod* in a box – used for variables. |

Figure 1: Some *vod*s of interest.

defining the context-free syntax of the visual language of interest, using *vod*s. This phase constructs a mapping from abstract syntax constructs to the *vod*s in the library. The third phase consists of defining the semantics of the visual language using the visual syntax defined in the previous phase.

Here we consider as an example, a finite state automata (FSA) to illustrate a VAS specification. Figure 1 shows some of the *vod*s of that are used in the next section in defining the syntax of *FSA*.

## 3.1 Syntax definition

The syntax of finite state automata is defined in the module *FSA*. An automata consists of a set of transitions along with a starting state and a collection of final states. The signature utilizes the *vod*s shown in Figure 1. The *TRAN-C* defines a collection of transitions, which is used in the definition of *FSA*. The imported module *Strings* defines upper-case characters (*UChar*) and lower-case characters (*LChar*).

**module** FSA

**imports** Strings

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

**context-free syntax**

| | | |
|---|---|---|
| *UChar* | $\rightarrow$ | *L* |
| *L* | $\rightarrow$ | *STATE* |
| $\Rightarrow$*STATE* | $\rightarrow$ | *SSTATE* |
| *LChar* | $\rightarrow$ | *INPUT* |
| $STATE \xrightarrow{INPUT} STATE$ | $\rightarrow$ | *TRAN* |
| *L* | $\rightarrow$ | *FSTATE* |
| *FSTATE* | $\rightarrow$ | *STATE* |
| $TRAN* \lhd \{\}$ | $\rightarrow$ | *TRAN-C* |
| *TRAN-C*   *SSTATE* | $\rightarrow$ | *FSA* |

**variables**

| | | |
|---|---|---|
| $\boxed{TRAN* \lhd \{\}}$ | $\rightarrow$ | *TRAN-C* |
| *Fsa* | $\rightarrow$ | *FSA* |

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 for this module can be seen in appendix A.2.

## 3.2 FSA Semantics

**module** FSA-eval

**imports** FSA

**sorts** *CSTATE RESULT INPUT-L FSA-CONF*

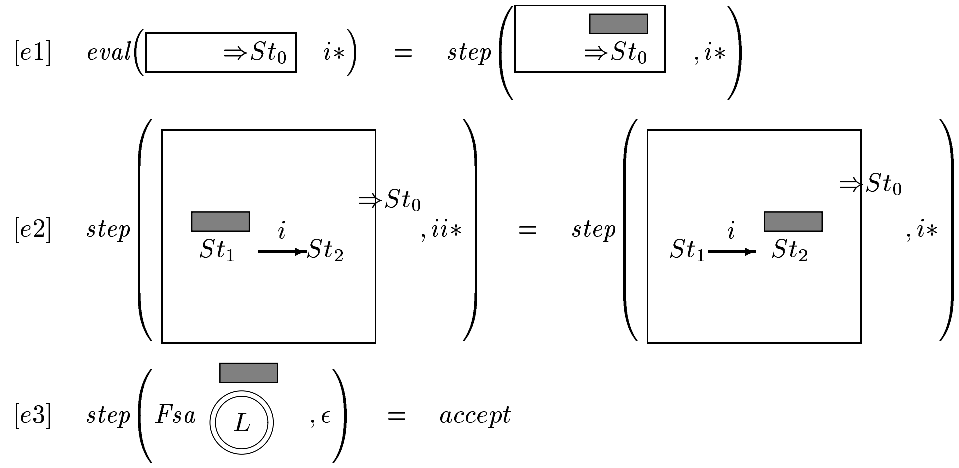
**context-free syntax**

| | | |
|---|---|---|
| *STATE* | $\rightarrow$ | *CSTATE* |
| INPUT* | $\rightarrow$ | *INPUT-L* |
| *accept* | $\rightarrow$ | *RESULT* |
| *reject* | $\rightarrow$ | *RESULT* |
| *eval* ( *FSA INPUT-L* ) | $\rightarrow$ | *RESULT* |
| $FSA^{\ CSTATE}$ , *INPUT-L* | $\rightarrow$ | *FSA-CONF* |
| *step* ( *FSA-CONF* ) | $\rightarrow$ | *RESULT* |

**variables**

| | | |
|---|---|---|
| *St* | $\rightarrow$ | *STATE* |
| *i* | $\rightarrow$ | *LChar* |
| *L* | $\rightarrow$ | *L* |
| *Cst* | $\rightarrow$ | *CSTATE* |
| *i∗* | $\rightarrow$ | *INPUT-L* |
| *FsaConf* | $\rightarrow$ | *FSA-CONF* |

**equations**




[e4] $step(FsaConf) \;=\; reject$ **otherwise**

Again, we provide the abstract syntax module (Appendix A.3) since this module also has a specification of semantics. In Section 5.1 we 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 of at least one final state and for checking that no non-final state has the same label as some final state.

# 4 FSA Term Construction

*FSA* terms are constructed by providing a collection of transitions and a start state:⟨TRAN-C⟩ ⟨SSTATE⟩. According to this syntactic definition the start state (*SSTATE*) many appear anywhere. However, we can choose to make it share the representation of a particular state. Figure 2 shows a construction sequence where such a sharing is chosen – yielding a more conventional appearance. Abstractly, of course, with or without the sharing they represent the same term.

# 5 Pretty Printing Issues

As can easily be imagined there are several criteria to adhere to when displaying. Firstly, visual representations were used in the first place to provide its user some information which presumably benefits from graphical presentation. It would make no sense to present information in manner which is incomprehensible such as overlapping and obscured bits on the screen. Similarly to this point there are many displaying principals which should be adopted to have a presentable pretty printed term[1].

The advantage of multi-dimensional syntax is the ability to use shared representation in term presentation. We have demonstrated how such terms can

[1]Graphic design issues could provide many sound guidelines for this purpose.

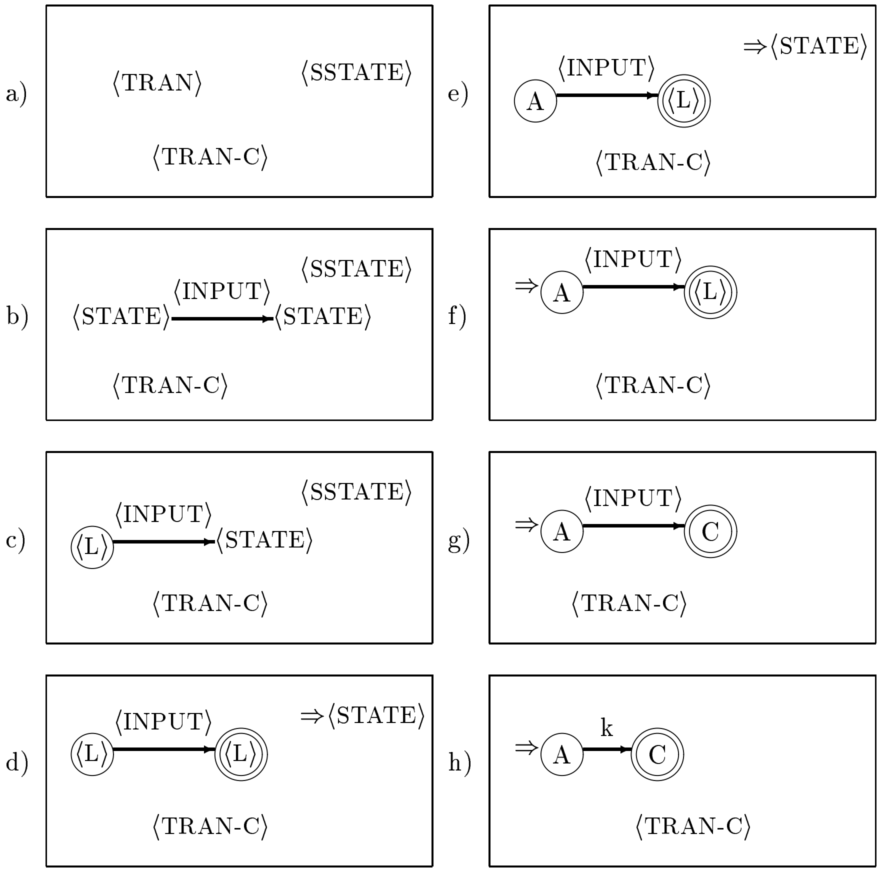
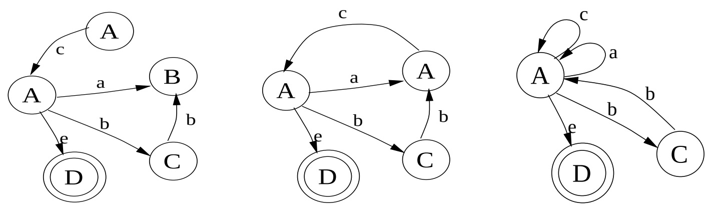



Figure 2: Initial steps of an *FSA* term construction.




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

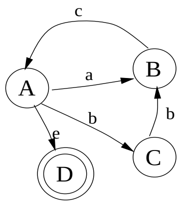
Figure 3: Various sharing possibilities

be constructed. Whenever we rewrite a term we are left with an abstract syntax term that must be graphically presented. As we have the corresponding *vod* definitions, we can find some layout consistent with these definitions. As the syntactic descriptions are under constrained corresponding to a great many pictures and as the rewriting is performed on abstract representations, we are left with highly under constrained terms to somehow display. However, it is often the case that parts of initial terms remain unchanged in the resulting term. If much is changed between these representations the user will likely loose the connection between these terms. We will concentrate on this point and propose a method whereby we maintain information from the initial term to aid in presenting the final term.

Furthermore, complications arise from the introduction of sharing which makes it difficult to determine how resulting terms should be represented. Consider 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:




The resulting term could be, among others, any of the terms shown in Figure 3. Pseudo sharing is used in work presented in this document. Real sharing is commonly found in Graph grammar approaches. Identity sharing is obtained by giving meta-level directive that all similar states should be shared. The subfigure (a) illustrates the effect of pseudo-sharing and the application of function $f$. This rewrites a transition labelled "c" and ending in some state $St_2$ to a transition labelled "c" that begins in state "A" and ends in $St_2$. Note that all "B" states is not affected but only one which matched the rule is rewritten into "A". After the rewrite it is not clear where to print the "A" state created by application of the rule.

## 5.1 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 (see also [Üsk96]). Each term $f(\vec{x})$ is annotated as $f_{[l,i:p]}(\vec{x})$ where $f$ is an abstract syntax function name, $l$ is a label, $i$ is an identifier of a term editor or an equation editor, and $p$ is a path in $i$. When a term is initially created by an user, each node gets a unique label, $i$ 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 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 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 in equations as much as possible. 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 excellent stating points or "hints" to the solver indicating where approximately they should be placed.

In this paper, we restrict ourselves to "basic" Share-Where maintenance. This consists of analyzing the equations for propagation of Share-Where information in the following three cases in order (details in [Üsk96]):

1. *Common Subterms* We look for the largest of the sub-terms common to both sides and propagate the annotation for these.

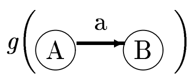
2. *Common Variables* When a variable is common to both sides of an equation, the Share-Where annotation is implicitly propagated.

3. *Common Context* When the same symbols appears at the top occurrence of the two sides of an equation (i.e., the same context under which a sub-term is modified by rewriting) the annotation is also propagated. A special case of which is the redex-contractum: when the top symbols are same on both sides.

All other symbols are considered as created by the right hand side of an equation. Sharing that is explicit on the right-hand side of an equation should also be accounted and leads to some exceptional cases for non-left-linear equations. Two of the cases are discussed in Sections 5.3 and A.4.

## 5.2 Evaluation

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




and a simple equation:

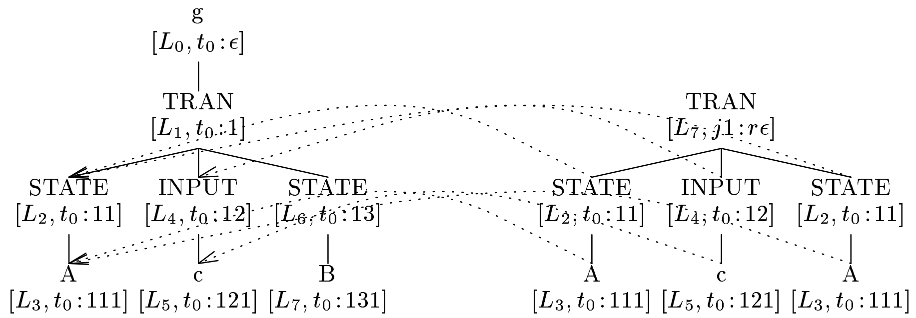



Figure 4: Share-Where annotations for a term and propagation after rewriting. The Where information "$t_0$ : 12" means that the node was created in term $t_0$ at occurrence 12 (second child of first child of top node).

$$[j1] \quad g(\, S_1 \xrightarrow{i} S_2) \quad = \quad S_1 \overset{i}{\circlearrowleft}$$

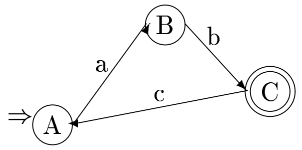
Figure 4 shows the Share-Where annotations for the initial term and after equation $[j1]$ is used to rewrite it. The dotted lines show how the shared subterms on the right-side have the same origin for their representations. The *INPUT* subterm is unchanged and thus has its Share-Where from the *INPUT* of left side (common variable rule). The "TRAN" node corresponds to the new symbol created by the equation (indicated by the Where-part $(j1 : r\epsilon)$).

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

$$[\mathrm{j1}_{sw}] \quad g_{\ell_0}(\, S_1 \xrightarrow{i} S_2\ {}_{\ell_1}) \quad = \quad S_1 \xrightarrow{i} S_1\ {}_{[uid(), j1:r\epsilon]}$$

where the variables $\ell_i$ match the Share-Where annotations. All annotations in $i$ and $S_1$ are propagated (common-variables) and an 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 $(\epsilon)$ occurrence.

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




$abcab\epsilon$

Figure 5 shows how the term is rewritten and some of the annotations of the nodes. These annotations are shown with the names of nodes. For example the first set of annotations correspond to the four states labelled 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 <u>share</u> 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 name (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

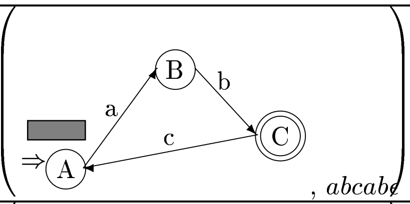
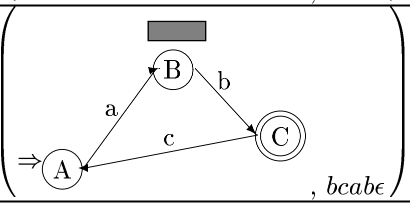
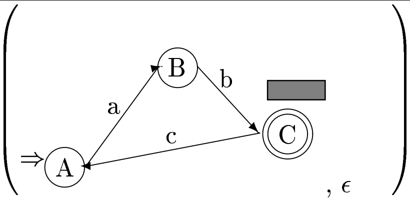
| Term | Eq | Match | Annotations |
|---|---|---|---|
| *step*  , $abcab\epsilon$ | [1] | $St_0 =$ A <br> $i\star \;= abcab\epsilon$ | $A_{ab}$:$[L_A, t_0{:}p_A]$ <br> $A_{bc}$:$[L_A, t_0{:}p_A]$ <br> $A_{ss}$:$[L_A, t_0{:}p_A]$ <br> $A_{cs}$:$[L_A, t_0{:}p_A]$ <br> $cs$:$[L_2, e1{:}p_{cs}]$ |
| *step*  , $bcab\epsilon$ | [2] | $St_0 =$ A <br> $St_1 =$ A <br> $St_2 =$ B <br> $i \;= a$ <br> $i\star \;= bcab\epsilon$ | $B_{ab}$:$[L_B, t_0{:}p_B]$ <br> $B_{bc}$:$[L_B, t_0{:}p_B]$ <br> $B_{cs}$:$[L_B, t_0{:}p_B]$ <br> $cs$:$[L_3, e1{:}p_{cs}]$ |
| *step*  , $\epsilon$ | [2]$\star$ | $St_0 =$ A <br> $St_1 =$ B <br> $St_2 =$ C <br> $i \;= b$ <br> $i\star \;= \epsilon$ | $C_{bc}$:$[L_C, t_0{:}p_C]$ <br> $C_{ca}$:$[L_C, t_0{:}p_C]$ <br> $C_{cs}$:$[L_C, t_0{:}p_C]$ <br> $cs$:$[L_8, e1{:}p_{cs}]$ |

Figure 5: Evaluation of an *FSA* term and some annotations.

Some interesting annotations 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.

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.

## 5.3 Discussion

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

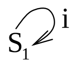
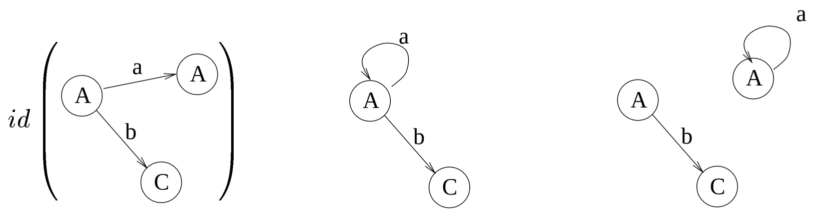
$$[j2] \quad \mathrm{id}(\mathrm{S}_1 \xrightarrow{i} \mathrm{S}_1\ ) \quad = $$ 

Figure 6(a) shows the initial term and the parts b) and c) show two possible pretty printing results depending on which Share-Where is chosen. 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(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




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

Figure 6: Pretty printing results based on two different Share-Where annotations of same term.

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 "basic" Share-Where, which only concerns itself with a simple propagation of Share-Where information for gaining the ability to pretty print terms with shared subterms.

Clearly, the Share-Where information can 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.

# 6 Input and Output Specification

In this section we define our approach to adding input and output to the algebraic specification formalism. The inclusion of input and output demands the consideration of (among others) interaction with external sources. Since we are interested not only in allowing input and output, but also in providing support in a realistic context, we have some general guidelines on how one can control the undesired back-tracking or non-determinism.

## 6.1 Input and Output functions

The basic idea for addressing IO is to define patterns in the conditions of semantic equations that identify a desire for *input* and/or *output*. These patterns are called *io-patterns*.

Each io-pattern has a meta-variable which designates that condition as io-condition. More precisely, an io-pattern is a condition of the form

$$\langle \mathit{SortName} \rangle = t_j$$

where $t_j$ is a term and $\langle \mathit{SortName} \rangle$ is a meta-variable of sort SortName. If all variables in $t_j$ are bound, then it is an output pattern, otherwise it is an input pattern where the unbound variables will get bound with external input. All io-patterns are intended to succeed (match).

Any function that uses an io-pattern in its definition is an *io-function*. Inductively any function that uses an io-function is also an io-function. In a list of conditions, in the definition of an io-function, we require that all io-functions be at the end of the list. The intention is to force every io-condition to act as a commit operation. The ordering of these conditions are significant with respect to the the order in which the input is read and output is presented. These restrictions and others are discussed in [Üsk96] and [DÜ95].

### 6.2 Interactive support for input and output

When there are unbound variables in a term $t$ of an io-pattern, then a meta-variable is displayed in a term editor, in place of each unbound variable in $t$. For now, we insist that the user complete all displayed meta-variables. The user is provided with a syntax directed editor over the sort of interest, as indicated by the meta-variable in the io-pattern. This is directly analogous to the syntax directed program construction. The term-editor is initialized to a term with meta-variables of appropriate sorts as indicated by the unknowns (unbound variables) in the term resulting in a failure free condition. One can consider this as output or as a context that helps a user when providing input. For example, this could be a prompt like "Please enter a number: $\langle NUMBER \rangle$".

## 7 Dynamic IO for FSA

Thus far we have looked at how to construct and present terms. In this section we will examine input and output during the execution of a term. We now provide a specification that interactively requests an input alphabet from the user and does an animate step.

### 7.1 A Query Language for FSA

The dynamic IO for *FSA* is defined in a module which provides a query syntax and rules specifying how the IO is handled. For the purpose of demonstrating how the IO is handled we will consider the interactive input (of sort *INPUT*[2].) to an *FSA* term. The objective is to solicit the user for a character and handle this character while executing the term as was discussed in Section 5.2. If the character provided 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 and asked for another character. This goes on until doomsday.

**module** FSA-anim

**imports** FSA-eval

**sorts** *FSA-CIO CIAO ONO FSA-Q*

**context-free syntax**

[2]Although $INPUT$ has 1-dimensional syntax, one can imagine building an $FSA$ term itself interactively, by querying the user for new transitions i.e., sort $TRAN$ with 2-dimensional syntax

$$
\begin{array}{rcl}
anim\ (\ FSA\ ) & \rightarrow & CIAO \\
astep\ (\ FSA\text{-}CIO\ ) & \rightarrow & CIAO \\
ok & \rightarrow & ONO \\
notok(INPUT) & \rightarrow & ONO \\
FSA\text{-}CONF\ ONO & \rightarrow & FSA\text{-}CIO \\
process\ (\ FSA\text{-}CONF\ INPUT) & \rightarrow & FSA\text{-}CIO
\end{array}
$$

`Choose an alphabet:` $\boxed{\boxed{INPUT}}$

$FSA\ ^{CSTATE}$ $\rightarrow$ $FSA\text{-}Q$

$\boxed{INPUT\text{-}L}$ used to reach current state

`No valid transition exists for`
"$INPUT$" `from current state!`
`Choose an alphabet:` $\boxed{\boxed{INPUT}}$ $\rightarrow$ $FSA\text{-}Q$

$FSA\ ^{CSTATE}$

$\boxed{INPUT\text{-}L}$ used to reach current state

**variables**

$FsaCio \rightarrow$ FSA-CIO

**equations**

[*i1*] $anim\left(\boxed{\Rightarrow St_0}\right) = astep\left(\boxed{\Rightarrow St_0}\ ,\epsilon\ ok\right)$

$\langle FSA\text{-}Q\rangle =$ `Choose an alphabet:` $\boxed{\boxed{i}}$

$Fsa\ \ Cst$

$\boxed{i*}$ used to reach current state

[i2]

$astep(Fsa\ \ Cst\ \ ,i*\ ok\ ) = astep(process(Fsa\ \ Cst\ \ ,i*\ \ i))$

`No valid transition exists for`
"$i_0$" `from current state!`

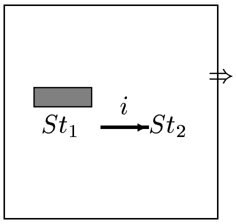
$\langle FSA\text{-}Q\rangle =$ `Choose an alphabet:` $\boxed{\boxed{i}}$

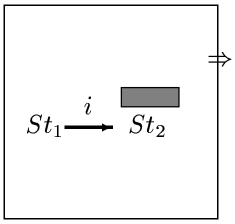
$Fsa\ \ Cst$

$\boxed{i*}$ used to reach current state

[i3]

$astep(Fsa\ \ Cst\ \ ,i*\ notok(i_0)\ ) = astep(process(Fsa\ \ Cst\ \ ,i*\ i))$

[*i4*] $process\Bigg($ 

$\Rightarrow St_0\ ,i*\quad i\Bigg) =$ 

$\Rightarrow St_0\ ,ii*\ \ ok$

[*i5*] $process(Fsa\ \ Cst\ \ ,i*\quad i) \quad = \quad Fsa\ \ Cst\ \ ,i*\quad notok(i)$ **otherwise**

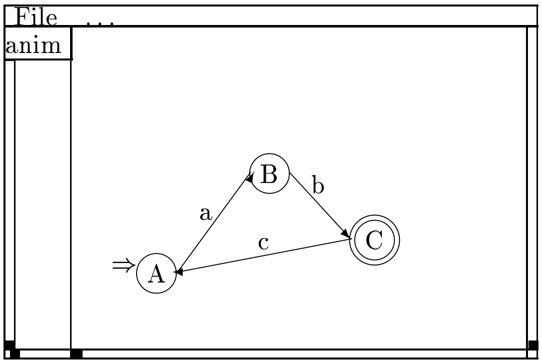



Figure 7: Term editor for the *FSA* language.

# 8 Interaction

Figure 7 shows an *FSA* instance in a term editor. After a visual program (a term) is constructed we want to execute it using the specified semantics. The "anim" button is defined[3] to apply the "anim" function defined in *FSA-anim* module.

Now let us follow a scenario to see how the equations deal with input and output during evaluation. The rest of the scenario shows the term in the editor as it is rewritten. Equation number references from Section 7.1 are used in the explanations.

Applying the *anim* function to the term, results in input request due to the io-pattern present in equation [$i2$]'s condition – which was executed by the equation [$i1$] defining the "anim" function. The resulting term is seen in Figure 8-a.

The unbound variable $i$ is replaced with $\langle INPUT\rangle$. The remainder of the query provides context and feedback during the animation. The *FSA* term provides the context by showing the *FSA* with current state as well as the valid string entered to reach this state.

The meta-variable $\langle INPUT\rangle$ demands input from the user, who can syntactically choose from a menu which presents the permitted operation or select an appropriate subterm (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 8-b). To continue, another alphabet must be entered. Consider that the character "a" 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 8-c).

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

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

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.

### 8.1 Share-Where for IO

We have already explained in Section 5.1 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 straight forward manner to handle IO. During execution, in the presence of IO, there could be many terms that are created or manipulated by an 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 subterm 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.

### 8.2 Share-Where properties for FSA animator

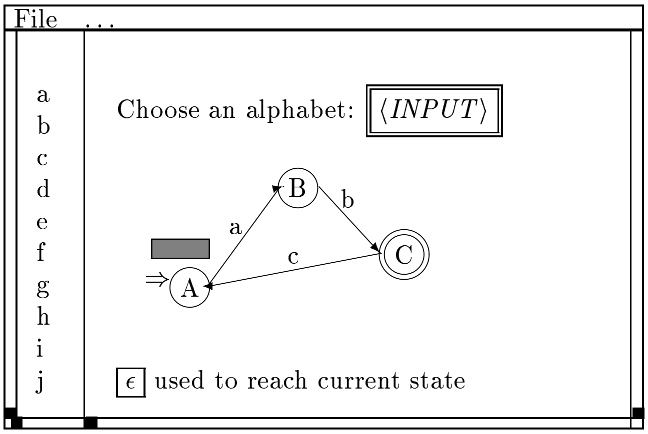
We can claim the following properties for the module FSA-anim.

- 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 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 same at the abstract level.

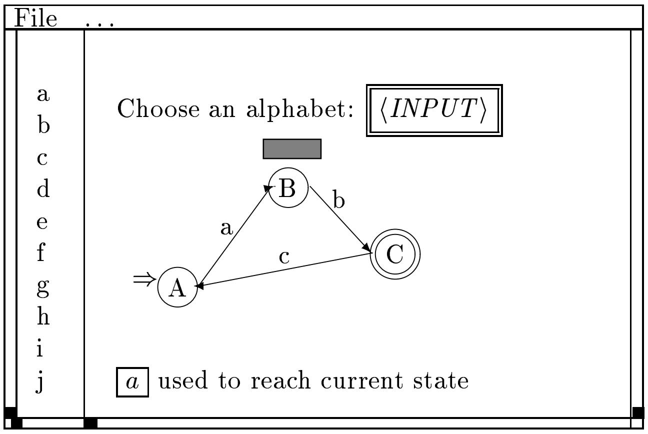
## 9 Related Work

The work of constraint multi-set grammars [MM96] and graph grammars [SZW95] 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.

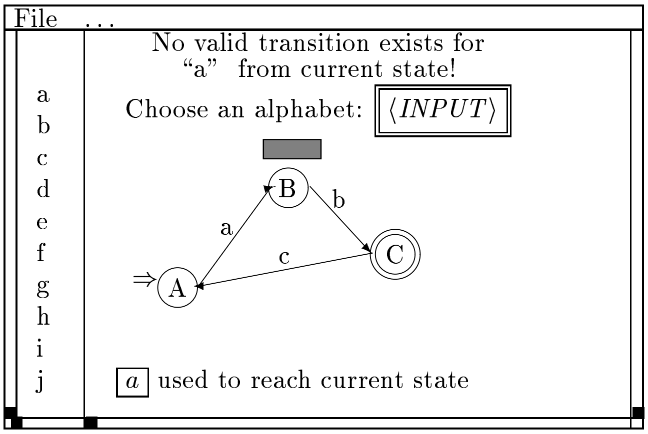
The ASF+SDF Meta-environment [Kli93] has been the source of many inspirations for our work on VAS formalism. The work on origin-tracking [DHK96,




a) After starting "anim"ation.




b) Input request after successful transition.




c) Input request after unsuccessful transition.

Figure 8: Some *FSA* term configurations during animation.

Chapters 7, 9 and 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 provide "primitive" origins in the absence of sharing. These techniques are related to residuals and descendants as can be found in [Klo92, Chapter 8] and subject-tracking defined by [Ber93].

Monads [JW93] and Unique types [PvE93] are two examples that allow extension of functional languages with input/output. Monads are not first order constructions and thus are not directly useful for us. The guidelines provided in Section 6.1 can be compared to Unique types. [WK95] have indicated how an algebraic specification can be viewed as a specification with input and output variables operationally, by transforming a specification into another that is suitable for constraint narrowing.

## 10 Conclusions

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. We have also illustrated how an interactive visual language can be specified using an extension of the traditional algebraic specification formalism to handle input and output.

**Acknowledgments** We wish to thank the organizers of the Workshop on the Theory of Visual Languages for their invitation and participants for their comments on this work. The first author is supported by NWO (The Netherlands Organization for Scientific Research) under project 612-17-4181: *Generic tools for program analysis and optimization.*

## References

[Ber93] Y. Bertot. A canonical calculus of residuals. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 140–163. Cambridge University Press, 1993.

[BK86] J. A. Bergstra and J. W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Co., 1996.

[DÜ95] T. B. Dinesh and S Üsküdarlı. Interaction ASF. preliminary draft, 1995.

[JW93] S. Peyton Jones and P. Wadler. Imperative functional programming. *20th POPL*, pages 71–84, January 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 2*, pages 1–116. Oxford University Press, 1992.

[MM96] Kim Marriott and Bernd Meyer. Towards a hierarchy of visual languages. In *International Workshop on Theory of Visual Languages*, May 1996.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[SZW95] A. Schürr, A. Zündorf, and A. Winter. Visual programming with graph rewriting systems. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[ÜD95] S Üsküdarlı and T. B. Dinesh. Towards a visual programming environment generator for algebraic specifications. In *Proc. 1995 IEEE Symposium Visual Languages*, September 1995.

[ÜD96] S Üsküdarlı and T. B. Dinesh. VAS Formalism in VASE. In *Proc. 1996 IEEE Symposium Visual Languages*, September 1996. To appear.

[Üsk94] S. Üsküdarlı. Generating visual editors for formally specified languages. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of the 1994 IEEE Symposium on Visual Languages (VL '94)*, pages 278–287. IEEE Computer Society, 1994.

[Üsk96] S. M. Üsküdarlı. *Algebraic Specification of Visual Languages.* PhD thesis, University of Amsterdam, Dec 1996. To appear.

[Wal91] H. R. Walters. *On Equal Terms — Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[WK95] H.R. Walters and J.F.Th. 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>.

# A FSA abstract syntax modules with Share-Where

## A.1 Share-Where syntax

The syntax of Share-Where annotations.
**imports** AS-Strings Aux
**exports**
  **sorts** Share Where Path TermId SW
  **lexical syntax**
    [0-9a-z]+ → Share
    [0-9]∗ → Path

[a-zA-Z]+Num → TermId
**context-free syntax**

| | |
|---|---|
| "$\epsilon$" | → Path |
| r Path | → Path |
| TermId ":" Path | → Where |
| "[" Share "," Where "]" | → SW |
| uid() | → Share |

**variables**

| | |
|---|---|
| *Share* [*0-9′*]∗ | → Share |
| *Where* [*0-9′*]∗ | → Where |
| "$\ell$"[*0-9′*]∗ | → SW |

## A.2 Abstract syntax of module FSA

## module AS-FSA

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 INPUT

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 |
| st(L) | → STATE |
| ss(STATE) | → SSTATE |
| ip(LChar) | → INPUT |
| tr(STATE, INPUT, STATE) | → TRAN |
| fs(L) | → STATE |
| coll(TRAN∗) | → TRAN-C |
| fsa(TRAN-C, SSTATE) | → FSA |

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.

**variables**

| | |
|---|---|
| *Tr* [*0-9*]∗"∗" | → TRAN∗ |
| *Fsa* | → FSA |

The Collection variable has no direct analog in ASF+SDF. Here we translate it into ASF+SDF lists.

## A.3 Abstract syntax of module FSA-eval

## module AS-FSA-eval

**imports** AS-FSA
**exports**
**sorts** CSTATE RESULT INPUT-L FSA-CONF
**context-free syntax**

| | |
|---|---|
| cs(STATE) | → CSTATE |
| list(INPUT∗) | → INPUT-L |
| acc() | → RESULT |

| | | |
|---|---|---|
| rej() | $\rightarrow$ RESULT |
| ev(FSA, INPUT-L) | $\rightarrow$ RESULT |
| fc(FSA, CSTATE, INPUT-L) | $\rightarrow$ FSA-CONF |
| sp(FSA-CONF) | $\rightarrow$ RESULT |

**variables**

| | |
|---|---|
| *St [0-9]∗* | $\rightarrow$ STATE |
| *i[01]∗* | $\rightarrow$ INPUT |
| *L* | $\rightarrow$ L |
| *Cst* | $\rightarrow$ CSTATE |
| *i"∗"* | $\rightarrow$ INPUT∗ |
| *FsaConf* | $\rightarrow$ FSA-CONF |

**equations**

$$[\mathrm{e1}_{as}]\ \mathrm{ev}(\mathrm{fsa}(\mathrm{coll}(Tr^*), \mathrm{ss}(St_0)), \mathrm{list}(i^*)) = \mathrm{sp}(\mathrm{fc}(\mathrm{fsa}(\mathrm{coll}(Tr^*), \mathrm{ss}(St_0)), \mathrm{cs}(St_0), \mathrm{list}(i^*)))$$

$$[\mathrm{e2}_{as}]\ \mathrm{sp}(\mathrm{fc}(\mathrm{fsa}(\mathrm{coll}(Tr_1^*\ \mathrm{tr}(St_1, i, St_2)\ Tr_2^*), \mathrm{ss}(St_0)), \mathrm{cs}(St_1), \mathrm{list}(i\ i^*))) = \mathrm{sp}(\mathrm{fc}(\mathrm{fsa}(\mathrm{coll}(Tr_1^*\ \mathrm{tr}(St_1, i, St_2)\ Tr_2^*), \mathrm{ss}(St_0)), \mathrm{cs}(St_2), \mathrm{list}(i^*)))$$

$$[\mathrm{e3}_{as}]\ \mathrm{sp}(\mathrm{fc}(Fsa, \mathrm{cs}(\mathrm{fs}(L)), \mathrm{list}())) = \mathrm{acc}\ (\ )$$

$$[\mathrm{e4}_{as}]\ \mathrm{sp}(FsaConf) = \mathrm{rej}\ (\ ) \quad \textbf{otherwise}$$

## A.4 Share-Where annotated FSA-eval equations

In the following, $\ell$ is declared as a variable of sort "Share-Where".

We detect which of the $\ell_i$s repeat on the right hand side by checking for subterms common between the left-hand side and right-hand side. For the new ones, we create a new Share label using function `uid()` and set the Where information to the path in the equation where they occur.

$$[\mathrm{e1}_{sw}]\ \mathrm{ev}_{\ell_0}(\mathrm{fsa}_{\ell_1}(\mathrm{coll}_{\ell_2}(Tr^*), \mathrm{ss}_{\ell_3}(St_0)), \mathrm{list}_{\ell_4}(i^*)) = \mathrm{sp}_{[\mathrm{uid}\ (\ ),\ \mathtt{e1}\ :\ \mathrm{r}\ \epsilon]}(\mathrm{fc}_{[\mathrm{uid}\ (\ ),\ \mathtt{e1}\ :\ \mathrm{r}\ 1]}(\mathrm{fsa}_{\ell_1}(\mathrm{coll}_{\ell_2}(Tr^*), \mathrm{ss}_{\ell_3}(St_0)), \mathrm{cs}_{[\mathrm{uid}\ (\ ),\ \mathtt{e1}\ :\ \mathrm{r}\ 12]}(St_0), \mathrm{list}_{\ell_4}(i^*)))$$

For variables that repeat on the left hand side in the original (abstract syntax) equation as in $St_1$ in this case, we add two distinct variables $St_{11}$ and $St_{12}$ since we do not want the Share-Where information to contribute to the matching. We then use the function *AS-eq* to compare the terms without the Share-Where information.

$$[\mathrm{e2}_{sw}]\ \frac{\mathrm{AS\text{-}eq}(St_{11}, St_{12})\ =\ \mathtt{true}}{\begin{array}{l}\mathrm{sp}_{\ell_0}(\mathrm{fc}_{\ell_1}(\mathrm{fsa}_{\ell_2}(\mathrm{coll}_{\ell_3}(Tr_1^*\ \mathrm{tr}_{\ell_4}(St_{11}, i, St_2)\ Tr_2^*), \mathrm{ss}_{\ell_5}(St_0)), \mathrm{cs}_{\ell_6}(St_{12}), \mathrm{list}_{\ell_7}(i\ i^*)))\ = \\ \mathrm{sp}_{\ell_0}(\mathrm{fc}_{\ell_1}(\mathrm{fsa}_{\ell_2}(\mathrm{coll}_{\ell_3}(Tr_1^*\ \mathrm{tr}_{\ell_4}(St_{11}, i, St_2)\ Tr_2^*), \mathrm{ss}_{\ell_5}(St_0)), \\ \mathrm{cs}_{[\mathrm{uid}\ (\ ),\ \mathtt{e2}\ :\ \mathrm{r}\ 112]}(St_2), \mathrm{list}_{[\mathrm{uid}\ (\ ),\ \mathtt{e2}\ :\ \mathrm{r}\ 3]}(i^*)))\end{array}}$$

Note also that the top node on the right-hand side of [e2] gets its Share-Where information from the top node of left-hand side since we treat redux-contractum with similar top nodes also as special for propagating the Share-Where information.

Equations [e3] and [e4] create new terms. When these are to be printed, one can use the Where information to print them (the result–*accept* or *reject*)as they appear in these equations.

$$[\mathrm{e}3_{sw}]\ \mathrm{sp}_{\ell_0}(\mathrm{fc}_{\ell_1}(Fsa,\ \mathrm{cs}_{\ell_2}(\mathrm{fs}_{\ell_3}(L)),\ \mathrm{list}_{\ell_4}())) = \mathrm{acc}_{[\mathrm{uid}\ (\ ),\ \mathtt{e3}\ :\ \mathrm{r}\ \epsilon]}(\ )$$

$$[\mathrm{e}4_{sw}]\ \mathrm{sp}_{\ell_0}(FsaConf) = \mathrm{rej}_{[\mathrm{uid}\ (\ ),\ \mathtt{e4}\ :\ \mathrm{r}\ \epsilon]}(\ ) \quad \textbf{otherwise}$$