Interactive Rule-Based Specification with an Application to Visual Language Definition

Roswitha Bardohl[1], Martin Große-Rhode[1], and Marta Simeoni[2]

[1] Institut für Softwaretechnik und Theoretische Informatik, TU Berlin,
{rosi,mgr}@cs.tu-berlin.de
[2] Dipartimento di Informatica, Università Cà Foscari di Venezia,
simeoni@dsi.unive.it

Abstract. In a rule-based approach the computation steps of a system are specified by rules that completely define how the system's state may change. For open systems a more liberal approach is required, where the state changes are only partly specified, and – interactively – other components may contribute further information on how the transformation is defined completely. In this paper we introduce a formal model for interactive rule-based specifications, where states are modelled as partial algebras and transformations are given by internal algebra rewritings and arbitrary external components. As an application we discuss how visual languages can be defined in this framework. Thereby the internal (logical) representations of visual expressions are transformed by rewriting rules, whereas their layouts are obtained interactively by external components like a constraint solver or a user working with a display and a mouse.

1 Introduction

In a traditional rule-based specification of a system a set of rules is given to describe the possible state changes of the system. Thereby the rules completely describe the relation of the initial and final states of each transformation step. For open systems, however, it is more adequate to specify the effect of a transformation step only partly by a rule. The transformation is determined completely only in interaction with the environment the system is placed in. That means, the impact of the environment on the local state changes has to be taken into account.

In a software environment for a visual language, for example, there might be an editor offering rules for creating items. Such an item would be offered when the rule is applied, but not directly placed into the figure. The actual position must be chosen (via a mouse click) by the user, or might be computed by a constraint solver that chooses one of the possible positions that satisfies the given constraints. In both cases the rule applied by the rule-based part of the system just yields the existence of the item, usually together with a predefined shape or other attributes, whereas the remaining attributes – like the concrete position and size – are set externally, by the user, the constraint solver, or another component.

M. Cerioli and G. Reggio (Eds.): WADT/CoFI 2001, LNCS 2267, pp. 1–20, 2002.

In this paper we introduce a formal algebraic approach for the precise specification of such *interactive rule-based behaviours*, where the effect of a rule is determined by two components. The internal part is modelled by the rewriting of algebras as formal models of states, as defined in [Gro99]. The basic idea for the rewriting is to present algebras by their sets of elements and their functions, where the latter are also considered as sets. Then simple set operations like subtraction, intersection and union can be used to rewrite the presentations, which induces the rewriting of the algebras. The interactive, external part of the transformation is embedded into this approach by designating parts of the algebras to be defined by the external component. That means, the signature is divided into an *internal* part that is updated by applying the algebra rewrite rules and an *external* part that is updated by the external component.

As an important application domain we investigate environments for visual languages in this framework. To treat visual languages properly it is important to distinguish the logical (grammatical) structure of an expression and its visual structure, given by its layout. Layout operations connect these two layers in that each logical item is mapped to a graphical one. This basic structure is used in [Bar00] to define a generic visual editor for the generation of visual languages and manipulation of their expressions. It is generic in the sense that it accepts language definitions as input and delivers concrete visual editors for these languages as output. The distinctive feature of logical and visual structure is that the semantics of a visual expression is completely determined by the logical structure. If, for example, the crossing of lines expresses semantic information it must be represented in the logical structure. If it occurs just due to lack of space or cannot be avoided due to topological properties, the intersection point is only present in the visual structure. Furthermore, the visual layout of an expression may change without affecting its information (logical structure). For instance, classes in a class diagram may be represented by rectangles in one style and rectangles with rounded corners in another style.

The paper is organized as follows. In the next section presentations as technical means for the rewriting of algebras are introduced. In Sect. 3 then the interactive transformation of algebras with internal algebra rewriting and external transformation by some other component is discussed. For our main application, visual languages, an example is given in Sect. 4 that illustrates the formal concepts. First an algebraic specification of class diagrams is given, containing their logical structure, a graphics domain, and layout operations. Then their interactive generation and manipulation with algebra rewrite rules for the logical part and a constraint solver or a user acting with a mouse to determine the layout is discussed. In Sect. 5 we give a conclusion, sketch further extensions of the framework, and discuss the relation with other approaches.

2 Presentations

In this section we recall from [Gro99] the basic concepts concerning presentations of partial algebras that are the technical means to define the rewriting of

algebras. The basic idea is to (re-) present partial algebras by a family of sets (their carrier sets) and, in addition, a set (re-) presenting the functions of a partial algebra via their input/output pairs. The latter are given in the form of equations $f(a_1, \ldots, a_n) = b$, where f is a function symbol and $a_1, \ldots, a_n$ and b are elements of the corresponding carrier sets. According to the terminology of universal algebra the elements of the carrier sets in such a presentation are called *generators* and the equations are called *relations*. In general arbitrary equations are allowed in a presentation, not only *function entries* $f(a_1, \ldots, a_n) = b$.

Simple set operations like subtraction, intersection, and union can be used then to transform presentations by removing or adding generators and/or relations. As shown in Prop. 1 each presentation induces a partial algebra. Thus to rewrite an algebra it is first translated into a presentation, this presentation is transformed via the set operations, and then the transformed presentation is translated into a partial algebra again which yields the result of the rewriting. (The rewriting procedure is discussed in the following section.)

Definition 1 (Presentation). *Let $\Sigma = (S, F)$ be an algebraic signature. A* Σ-presentation *$P = (P_S; P_E)$ is given by an S-indexed set $P_S = (P_s)_{s \in S}$, the* generators*, and a set $P_E \subseteq Eqns_\Sigma(P_S)$, the* relations*. A presentation is* functional *if P_E is a set of* function entries *over P_S, i.e.,*

$$P_E \subseteq \{f(a) = b \mid f : w \to v \in F,\ a \in P_w,\ b \in P_v\} \subseteq Eqns_\Sigma(P_S).$$

A morphism *of Σ-presentations $p : (P_S; P_E) \to (P'_S; P'_E)$ is an S-indexed function $p = (p_s : P_s \to P'_s)_{s \in S}$ such that $P_E[p] \subseteq P'_E$, where $\_[p]$ denotes the substitution of generators according to p. Σ-presentations and morphisms yield the category* $\mathbf{Pres}(\Sigma)$*. Finally let* $\mathbf{Pres}(\Gamma) = \mathbf{Pres}(\Sigma)$ *for each partial equational specification $\Gamma = (\Sigma, CE)$ extending Σ by conditional equations CE w.r.t. Σ.*

Presentations can be restricted to subsets of generators by deleting all relations that contain generators not contained in the subset. This will be used to model the implicit removal of function entries containing elements that have been deleted. On the other hand, presentations can be extended to larger signatures by adding empty sets of generators for the new sorts.

Definition 2 (Restriction and Extension). *Let $P = (P_S; P_E)$ be a Σ-presentation.*

1. *Given an S-indexed subset $Q_S \subseteq P_S$ the* restriction *$P|_{Q_S}$ is defined as the Σ-presentation given by*
   $$P|_{Q_S} = (Q_S; P_E \cap Eqns_\Sigma(Q_S)).$$
2. *Given a signature extension $\Sigma' \supseteq \Sigma$ with $\Sigma = (S, F)$, $\Sigma' = (S', F')$ the* extension *$P^{\Sigma'} = (P_S^{\Sigma'}; P_E^{\Sigma'})$ of P to Σ' is defined as the Σ'-presentation given by*
   $$P_{s'}^{\Sigma'} = \begin{cases} P_{s'} & \text{if } s' \in S \\ \emptyset & \text{else} \end{cases} \qquad (s' \in S')$$
   $$P_E^{\Sigma'} = P_E.$$

Consider now a partial equational specification $\Gamma = (\Sigma, CE)$ extending the signature Σ by some conditional Σ-equations CE. To each Γ-presentation $P = (P_S; P_E)$ there is a *smallest* (free generated) partial Γ-algebra A^P that contains the generators P_S and satisfies the relations (equations) P_E. If P contains only function entries $f(a) = b$ as relations and these are consistent (in the sense that $(f(a) = b) \in P_E$ and $(f(a) = b') \in P_E$ implies $b = b'$) it yields the partial Σ-algebra A^P given by

$$A^P_s = P_s \qquad (s \in S)\ ,$$
$$f^{A^P}(a) = b \ \text{ iff } \ (f(a) = b) \in P_E \qquad (f \in F)\ .$$

If furthermore A^P already satisfies the conditional equations CE this yields the partial Γ-algebra induced by P. If the equations are not satisfied, however, or if there are inconsistent function entries, further elements may be generated and some generators or generated elements may be identified.

This property is formally stated as the existence of a free functor from the category of Γ-presentations $\mathbf{Pres}(\Gamma)$ to the category of partial Γ-algebras $\mathbf{PAlg}(\Gamma)$, i.e., a left adjoint to the *presentation functor* that maps partial Γ-algebras to Γ-presentations.

Proposition 1. *Let $\Gamma = (\Sigma, CE)$ be a partial equational specification. The* presentation functor *$Pres_\Gamma : \mathbf{PAlg}(\Gamma) \to \mathbf{Pres}(\Gamma)$, given by*

$$Pres_\Gamma(A_S, A_F) = (A_S, \{f(a) = b \mid f^A(a) = b\}), \text{ and}$$
$$Pres_\Gamma(h) = h$$

has a left adjoint $PAlg_\Gamma : \mathbf{Pres}(\Gamma) \to \mathbf{PAlg}(\Gamma)$ that satisfies $PAlg_\Gamma \circ Pres_\Gamma \cong Id_{\mathbf{PAlg}(\Gamma)}$.

As mentioned above, if P is given by consistent function entries only and conditional equations are not considered (or already satisfied) the partial algebra induced by P carries exactly the same information as P itself. This is made precise in the following corollary.

Corollary 1. *Let Σ be an algebraic signature and $P = (P_S; P_E)$ be a Σ-presentation. If P is* consistently functional*, i.e., P is functional and satisfies*

$$(f(a) = b) \in P_E \ \wedge \ (f(a) = b') \in P_E \ \Rightarrow \ b = b'$$
$$\text{for all } f : w \to v \in F,\ a \in P_w,\ b, b' \in P_v,$$

then $Pres_\Sigma(PAlg_\Sigma(P)) \cong P$.

3 Interactive Transformation

In this section we describe the formal algebraic approach for the specification of interactive rule-based behaviours, where the effect of a transformation is determined by the integration of an internal mechanism and an external one. More precisely, the internal part of the transformation is modelled by the rewriting

of partial algebras, while the external part is embedded into the transformation by designating parts of the algebra to be defined by the external component. This allows the formalization of the complete framework independently of any particular mechanism to deal with the external part of the transformation.

A signature for interactive transformation reflects the distinction between the internal and external mechanisms by designating two different algebraic signatures: one designating the internal part (*internal signature*) and an extension of it including also the parts of the algebra to be defined externally (*complete signature*). Beyond these algebraic parts an interactive transformation signature also provides names of actions with parameter type lists. The actions can be used then to trigger the application of algebra rewrite rules.

Definition 3 (Interactive Transformation Signature). *An* interactive transformation signature $R\Sigma = (\Sigma_{in}, \Sigma, A)$ *is given by algebraic signatures* $\Sigma_{in} = (S_{in}, F_{in})$*, the* internal signature*, and* $\Sigma = (S, F)$*, the* complete signature*, with* $\Sigma_{in} \subseteq \Sigma$*, and a family* $A = (A_w)_{w \in S^*}$ *of sets of action names, called the* action signature*.*

The sorts and functions in $S - S_{in}$ *and* $F - F_{in}$ *respectively are called* external *sorts and functions. An action name* $a \in A_w$ *is also denoted by* $(a : w) \in A$*.*

3.1 Internal Transformation by Algebra Rewriting

Partial algebras are rewritten via rules which are essentially given by pairs of presentations. The *left hand side* of a rule specifies which elements and function entries are to be removed from the algebra representing the actual state; therefore it is required to be functional. Its *right hand side* specifies the elements and relations that are to be added. The generators that occur in the left hand side of a rule play the role of variables that are matched to the actual state. If a variable occurs in both parts of a rule the corresponding element is preserved. Such variables are used as context for the rewriting that describes how the right hand side is embedded into the remainder of the actual state after the removal of the left hand side. According to the distinction between internal and complete signature described above, only elements and function entries belonging to the internal signature can be used in the left and right hand sides of a rewrite rule. In addition to the rule body, given by its left and right hand sides as discussed just now, rules are equipped with a logical formula that specifies the positive and/or negative conditions for the rule application. Moreover, a formal action expression is added that binds the rule to the action whose behaviour is being specified and instantiates to the label of the transformation step.

Definition 4 (Rewrite Rule). *Let* $R\Sigma = (\Sigma_{in}, \Sigma, A)$ *be an interactive transformation signature. A* rewrite rule $r = (a(\bar{x}) \triangleq Cond \Rightarrow (P_l \to P_r))$ *w.r.t.* $R\Sigma$ *is given by*

- *an action name* $a : s_1 \dots s_n \in A$*,*
- *a list of variables* $\bar{x} = (x_1, \dots, x_n)$*, the formal parameters of* a*,*

(this formal action expression yields the action label*)*

- *a functional Σ_{in}-presentation $P_l = (X_l, E_l)$,*
- *an arbitrary Σ_{in}-presentation $P_r = (X_r, E_r)$,*

(the left and right hand side *of the rule)*

- *a* condition *Cond, given by a set of variables X_F and formula F with free variables in $X_F \cup X_l \cup X_r$.*

These components must satisfy the condition that

- *$x_i \in (X_F \cup X_l)_{s_i}$ for each $i \in \{1, \ldots, n\}$,*

i.e., the formal parameters of the action are contained in the sets of (free) variables of the condition and the left hand side.

Adding rewrite rules to an interactive transformation signature to specify the behaviour of its actions yields an interactive transformation specification.

Definition 5 (Interactive Transformation Specification). *An* interactive transformation specification *$R\Gamma = (\Gamma_{in}, \Gamma, A, R)$ is given by algebraic specifications $\Gamma_{in} = (\Sigma_{in}, CE_{in})$ and $\Gamma = (\Sigma, CE)$ with $\Gamma_{in} \subseteq \Gamma$, and a set R of $R\Sigma$-rewrite rules, where the interactive transformation signature $R\Sigma$ is given by $R\Sigma = (\Sigma_{in}, \Sigma, A)$.*

Consider now the application of a rewrite rule $r = (a(\bar{x}) \triangleq Cond \Rightarrow (P_l \rightarrow P_r))$ to a partial Σ-algebra A. First the free variables X_F of the condition $Cond = (X_F, F)$ and X_l of the left hand side $P_l = (X_l, E_l)$ must be instantiated in A via a mapping $m : X \rightarrow A$. Since the formal parameters x_i of the action expression $a(\bar{x}) = a(x_1, \ldots, x_n)$ are contained in $X_F \cup X_l$ this yields also a corresponding action instance $a(m(\bar{x})) = a(m(x_1), \ldots, m(x_n))$. The mapping m is a *match* of r in A if the condition and the equations of the left hand side are satisfied in A w.r.t. the instantiation m (i.e., $A, m \models Cond$ and $A, m \models E_l$). Only matches yield rewriting steps, other instantiations of the variables are not admissible.

To describe the effect of the application of r w.r.t. a match m in A the following symmetric differences and intersections are needed. They correspond to the parts that shall be deleted (X_l^0, E_l^0), retained (X_c, E_c), and added (X_r^0, E_r^0) respectively.

$$X_l^0 = X_l - X_r \qquad X_c = X_l \cap X_r \qquad X_r^0 = X_r - X_l$$
$$E_l^0 = E_l - E_r \qquad E_c = E_l \cap E_r \qquad E_r^0 = E_r - E_l$$

The internal rewriting step $r/m : A \Rightarrow_{in} iB$ rewrites then A into an intermediate state iB in three steps:

1. the subtraction (deletion) of the image of (X_l^0, E_l^0) under m in the presentation (A_S, A_E) of A,
2. the addition of (X_r^0, E_r^0), and

3. the free construction of the partial Σ-algebra from this presentation. (Since iB is only an intermediate state the conditional equations CE are not considered yet.)

The carrier sets iB_S of the intermediate result of the rewriting are thus given by $iB_S = (A_S - m(X_l^0)) \uplus X_r^0$. Its generating set of relations is given by $iB_E = (A_E - E_l^0[m]) \cup E_r^0[m]$, corresponding to the subtraction of the function entries E_l^0 and the addition of the relations E_r^0. Finally, all relations still containing variables from A_S that do not belong to iB_S must be removed in order to obtain a well defined presentation again. Thus the deletion of an element a of A has the side effect of removing all function entries containing a in any position.

Definition 6 (Internal Algebra Rewriting). *Let $R\Gamma$ be an interactive transformation specification and $r = (a(\bar{x}) \triangleq Cond \Rightarrow (P_l \to P_r))$ be a rewrite rule w.r.t. $R\Sigma$. Furthermore let A be a partial Γ-algebra with $Pres_\Sigma(A) = (A_S; A_E)$ and let $m : P_l^\Sigma \to (A_S, A_E)$ be a Σ-presentation morphism such that $A, m \models F$. (That means, m is a* match *for r in A.)*

Then the internal rewriting step *$a(m(\bar{x})) : A \Rightarrow_{in} iB$ rewrites A into the partial Σ-algebra iB defined by*

$$
\begin{aligned}
iB &= PAlg_\Sigma(iB_S, iB_E) \\
iB_S &= (A_S - m(X_l^0)) \uplus X_r^0 \\
iB_E &= ((A_E - E_l^0[m]) \cup E_r^0[m])|_{iB_S}
\end{aligned}
$$

3.2 External Transformation by Other Components

The internal algebra rewriting step only transforms (explicitly) the internal parts of a given algebra, i.e., the ones corresponding to the internal signature. (There may be side effects, however, induced by functions from the internal to the external part.) The intermediate state iB obtained by this rewriting is then further transformed by some external component. The final output of the transformation step is required to be a partial Γ-algebra, i.e., the conditional equations must hold. The external component might use all the information given in iB, but may change only its external parts, i.e., the ones corresponding to $\Sigma - \Sigma_{in}$. ¿From the most abstract point of view, the behaviour of an external component is given thus by a mapping from $\mathbf{PAlg}(\Sigma)$ to $\mathbf{PAlg}(\Gamma)$ that preserves the Σ_{in}-reducts of partial Σ-algebras. This transformation is appended to the internal rewrite step.

Definition 7 (Interactive Transformation). *Let $R\Gamma = (\Gamma_{in}, \Gamma, A, R)$ be an interactive transformation specification and $Ext : |\mathbf{PAlg}(\Sigma)| \to |\mathbf{PAlg}(\Gamma)|$ a mapping with $V_{\Sigma_{in}}(Ext(C)) = V_{\Sigma_{in}}(C)$ for all $C \in |\mathbf{PAlg}(\Sigma)|$. For each partial Γ-algebra A, $R\Sigma$-rewrite rule $r = (a(\bar{x}) \triangleq Cond \Rightarrow (P_l \to P_r))$, and match m of r in A, the* interactive transformation step *$a(m(\bar{x})) : A \Rightarrow B$ transforms A into the output state $B \in |\mathbf{PAlg}(\Gamma)|$ given by $B = Ext(iB)$, where iB is the internal output state of the internal rewriting $a(m(\bar{x})) : A \Rightarrow_{in} iB$.*

Like the internal rewriting step the behaviour of the external component may be defined in terms of presentations. That means, the presentation format can be used as an interface to connect the external component with the interactive transformation, as indicated in Fig. 1.

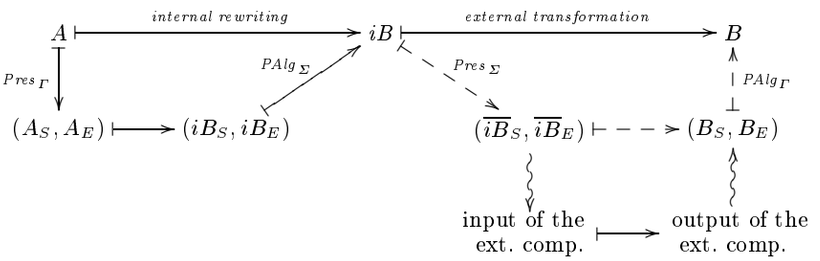

Fig. 1. Transformation step with internal algebra rewriting, external transformation by another component, and its interface to the presentation format

The decision to apply first the internal rewriting and then the external transformation mechanism seems to be non-symmetric. It reflects the idea that the transformation is guided by the rule-based component of the interactive system. On the other hand, in a sequence of transformation steps just the alternation of internal and external transformations remains, i.e. the order in which the internal and external transformations are applied does not matter. Parallel execution of internal and external steps has not been taken into the definition since it does not reflect interactivity, as desired in our applications. (Note that this does not exclude *independent* sequential internal and external transformations representing an interleavings of semantically independent or parallel steps.)

4 An Application: Specification of Visual Languages

In this section we discuss visual languages as an application domain for interactive rule based specifications. This application can be used to provide a formal basis for the construction of visual languages environments, as introduced in [Bar00].

Defining a visual expression (i.e., a sentence belonging to a visual language) means to deal with the description of two different aspects: its logical structure and its visual structure, i.e., its layout. These two aspects have to be connected by layout operations associating each logical item with a graphical one. The semantics of a visual expression is completely determined by its logical structure, the layout makes it visual.

We proceed by showing how to apply the interactive rule based specification framework to the definition of a concrete visual language, namely (a part of) the UML class diagram language (see [BRJ99]), consisting of classes and associations. We present first the algebraic specification of the language together with

some rewriting rules for the underlying logic structures, then we describe the interactive generation of class diagrams where the visual layout is determined by two possible external components.

4.1 Algebraic Specification of Class Diagrams

The specification of the class diagram language comprises the two parts discussed in section 3, the *internal* and the *complete* specification, where the latter adds the externally defined parts. In addition we distinguish a *static* part of the internal signature to denote those parts of algebras that never change. In our case this is basically given by the domain of graphic elements. ¿From the formal point of view this distinction of a static part is just a comment, because in general rewrite rules might not preserve these parts.

- The *static part* provides the graphics elements, like rectangles, arrows, etc, and operations for their construction and manipulation. Rectangles for example are created by the operation *rect* with parameters for their left upper corners (of type *Point*), width, and height (of type *Real*). This graphic domain should provide all operations to define the desired layouts of the logical elements of the language and express relationships or constraints like: the point lies on the border of the rectangle etc. When using concrete graphic tools the signature should correspond of course to the interfaces of these tools.
  The sorts *Real*, *Point*, and *Graphic* are introduced in the static part, where *Graphic* here stands for a general sort for graphical elements and *Point* indicates positions on the display. Furthermore the static part provides built-in data types like *Strings* to express attributes of the logical elements like their names.

  sorts: $Real, Point, Graphic, String, \ldots$

  funs: `(graphic operations and constructors)`
  $width : Graphic \to Real$
  $height : Graphic \to Real$
  $rect : Point,\ Real,\ Real,\ \to Graphic$
  $line : Point,\ Point \to Graphic$

  `(predicates)`
  $on\_border : Point,\ Graphic$

- The *internal part* of the signature includes the static part and extends it by a specification of the logical structure of the language. The sorts *Class* and *Assoc* are introduced, corresponding to the logical objects to be specified. Moreover, the operations allow for assigning names to logical elements and defining associations as structural relationships between classes.

sorts: $Class, Assoc$

funs: `(logical: giving names to objects)`
$c\_name : Class \to String$
$a\_name : Assoc \to String$

`(logical: building structural relationships)`
$a\_begin : Assoc \to Class$
$a\_end : Assoc \to Class$

– The *complete* specification of the class diagram language extends the internal part by an external part that specifies the layout functions. These have to be defined by an arbitrary external component.
The basic idea thereby is to associate with each logical item some *attachment points* that yield enough information to obtain the complete layout by a corresponding parameterized layout operation. For instance, classes are visualized by rectangles whose width and height is determined by the size of the graphical representation of their names. Thus it suffices to specify a point (the attachment point of the class) where the rectangle shall be positioned. For an association two attachment points are specified. Its layout is then given by a line from the first to the second point.
To formulate the constraints the predicate *on_border* is used. It allows us to specify that the association symbols are drawn in the correct way, i.e., that the lines representing them graphically start and end at the border of the corresponding class symbols.
Finally, the external part comprises the functions connecting the logical and graphical structures: each logical element is associated with a layout, i.e., a graphic element. These functions are defined in terms of conditional equations that use the graphics constructors and the attachment points of the logical elements. (In the layout definitions the definedness predicate $\downarrow$ is used. Only if the attachment points are defined the layout can be computed.)

funs: `(attachment points: to be set externally)`
$c\_pos : Class \to Point$
$a\_bpos : Assoc \to Point$
$a\_epos : Assoc \to Point$

`(graphical layout functions)`
$c\_layout : Class \to Graphic$
$a\_layout : Assoc \to Graphic$
$s\_layout : String \to Graphic$

axioms: `(constraints)`
$on\_border(a\_bpos(a), c\_layout(a\_begin(a)))$
$on\_border(a\_epos(a), c\_layout(a\_end(a)))$
$width(a\_layout(a)) \geq width(s\_layout(a\_name(a)))$

`(layout definition)`
$c\_pos(c) \downarrow \Rightarrow$

$$c\_layout(c) = rect(c\_pos(c), width(s\_layout(c\_name(c))), height(s\_layout(c\_name(c))))$$
$$a\_bpos(a)\downarrow \wedge a\_epos(a)\downarrow \Rightarrow$$
$$a\_layout(a) = line(a\_bpos(a), a\_epos(a))$$

This completes the specification of our class diagram language. Each visual expression of the language is completely specified by an algebra of the given specification.

Now we have to define the rewriting rules for the manipulation of the logical structures of visual expressions. We present here just some of them. (Layouts are considered in Sect. 4.2).

The first rule allows the insertion of a new class object. There are no preconditions to be fulfilled in order to add a new class into the logical structure: the *insert_class* rule just specifies in the right-hand side the new class element, and defines its associated name.

acts: $insert\_class : String$
rule: $insert\_class(cn) \triangleq (\emptyset; true) \Rightarrow$
$(\emptyset; \emptyset) \rightarrow (c : Class; c\_name(c) = cn)$

The *insert_assoc* rule is similar: the only precondition for its application is the existence of two classes. Since these also yield the context where to insert the new association (both logically and graphically) they are given both in the left and the right hand side of the rule. That means that they will be preserved by a rule application. The new association, its name and the connections with the source and target classes are specified in the right-hand side.

acts: $insert\_assoc : Class, Class, String$
rule: $insert\_assoc(c_1, c_2, an) \triangleq (\emptyset; true) \Rightarrow$
$(c_1, c_2 : Class; \emptyset) \rightarrow$
$(c_1, c_2 : Class, a : Assoc;$
$a\_name(a) = an, a\_begin(a) = c_1, a\_end(a) = c_2)$

Conversely, when removing an association all *relationships* to other elements, like its name, the beginning and the ending class, and its layout, are removed. (These elements themselves are of course not removed.) This is due to the restriction step in the construction of the internal algebra rewriting that removes all relations that contain removed elements (see Def. 6).

acts: $delete\_association : Assoc$
rule: $delete\_assoc(a) \triangleq (\emptyset; true) \Rightarrow (a : Assoc; \emptyset) \rightarrow (\emptyset; \emptyset)$

The deletion of a class is more complicated, since a class can be removed from a visual expression only if it is not related with any association. This is exactly the application condition specified in the rule *delete_class*. If the condition holds, the class element is deleted (by specifying it only in the left-hand side of the rule). As above, the connection with its name is deleted automatically, too.

acts: $delete\_class : Class$
rule: $delete\_class(c) \triangleq$
$(c : Class; \nexists a \in Assoc : a\_begin(a) = c \ \vee \ a\_end(a) = c) \ \Rightarrow$
$(c : Class; \emptyset) \rightarrow (\emptyset; \emptyset)$

All the above rules deal with the creation or deletion of logical objects. However, also rules for moving objects have to be provided. In our framework such rules are empty, since the logical structure of the language does not change. For instance, the *move_class* rule is defined as follows:

acts: $move\_class : Class$
rule: $move\_class(c) \triangleq (\emptyset; true) \Rightarrow (\emptyset; \emptyset) \rightarrow (\emptyset; \emptyset)$

We conclude this section by showing an example of an internal algebra rewriting given by a rule application. Consider the insertion of an association as depicted in Fig. 4.1

Fig. 2. Insertion of an association

The visual expression on the left side of the figure can be represented by the algebra defined as follows. We do not present the graphical part explicitly. The logical structure and the layout are given by:

Class	$\{x_1, x_2\}$	(there are two classes ...)
Assoc	$\emptyset$	(... and no association)
c_name	$c\_name(x_1) =$ "*Client*", $c\_name(x_2) =$ "*Order*"	(each class has a name)
a_name	–	(these functions are undefined ...)
a_begin	–	(... because there are no associations)
a_end	–	
c_pos	$c\_pos(x_1) = (0.75, 2.25)$, $c\_pos(x_2) = (3.75, 2.25)$	
a_bpos	–	
a_epos	–	
c_layout	$c\_layout(x_1) = rect((0.75, 2.25), 2.5, 1.5)$, $c\_layout(x_2) = rect((3.75, 2.25), 2.5, 1.5)$	
a_layout	–	

Applying the *insert_assoc* rule as shown in fig. 4.1 yields the following algebra as intermediate state of the transformation. A new association a is introduced, corresponding to the variable a that appears only in the right hand side of the rule. The corresponding equations connect a with its name "*make*" and the two classes x_1 and x_2 given to the *insert_assoc* action as parameters. The external parts are again defined as before. In particular, there is no attachment point and no layout for the new association yet.

Class	$\{x_1, x_2\}$	(as before)
Assoc	$\{a\}$	(the new association)
c_name	$c\_name(x_1) =$ "*Client*", $c\_name(x_2) =$ "*Order*"	(as before)
a_name	$a\_name(a) =$" *make*"	
a_begin	$a\_begin(a) = x_1$	
a_end	$a\_begin(a) = x_2$	(according to the equations in the rhs of the rule)
c_pos	$c\_pos(x_1) = (0.75, 2.25)$, $c\_pos(x_2) = (3.75, 2.25)$	
a_bpos	–	
a_epos	–	
c_layout	$c\_layout(x_1) = rect((0.75, 2.25), 2.5, 1.5)$, $c\_layout(x_2) = rect((3.75, 2.25), 2.5, 1.5)$	
a_layout	–	

4.2 Interactive Definition of Visual Layouts

After the internal rewriting of the logical structure of a diagram its layout still has to be determined. In the specification designated attachment points have been introduced whose values completely determine the layout of the logical elements via the layout functions. Thus it suffices to deliver the values for the attachment points of all objects (classes and associations) that exist in a given state. We discuss two external components here that can be used for this purpose, a graphical constraint solver and a user interacting with the system via a display and a mouse.

As mentioned in Sect. 3.2 an external component can be connected to the interactive transformation mechanism by plugging it into the interface given by the presentation format (cf. Fig. 1). That means, we must define how a Σ-presentation yields an input for the component and how its output is translated into a Σ-presentation. In the following we define this connection for the graphical constraint solver and the clicking user.

The input of a constraint solver is given by a list of typed variables and a set of constraints. In our example (and for the constraint solver ParCon used in [Bar00] for this purpose) all variables are of type *Point* or *Real*, and the constraints are predicates or equations w.r.t. the functions provided by the graphic domain. The output of the constraint solver is given by a binding of

values to the variables that respects the constraints. Usually this is given by a list of values that in comparison with the list of variables yields the association of the values to the variables.

The basic idea for the connection now is to use ground terms corresponding to the attachment points of the logical elements as names of the variables for the constraint solver. That means, the terms $c\_pos(c)$ for all $c \in iB_{Class}$ and $a\_bpos(a), a\_epos(a)$ for all $a \in iB_{Assoc}$ are used as variable names. On the other hand, the output of the constraint solver, i.e., the binding, can be translated into a set of Σ-equations $\{c\_pos(c) = t_c, \ldots\}$, where t_c is a ground term w.r.t. the graphics signature representing the value obtained by the binding for the variable named $c\_pos(c)$. (Thereby we assume that the signature is sufficiently expressive to obtain such a ground term for each value the constraint solver could possibly deliver.) Now the old function entries (equations) for the attachment points and layout operations are removed from the presentation $(\overline{iB}_S, \overline{iB}_E)$ obtained by the internal rewriting step and replaced by the equations $\{c\_pos(c) = t_c | c \in iB_{Class}\}$ and $\{a\_bpos(a) = t_a, a\_epos(a) = t'_a \mid a \in iB_{Assoc}\}$ for the attachment points obtained from the binding. In this way the missing information on the new positions of the logical elements is added. Note, however, that at this point the layout functions are completely undefined; the corresponding function entries have been removed.

The replacement of the old function entries for the attachment functions and layout operations by the new equations for the attachment points from the binding defines the behaviour of the constraint solver as a mapping on Σ-presentations, as required in Def. 7 (implicitly) and shown in Fig. 1.

The generation of the partial algebra $B = PAlg_\Gamma(B_S, B_E)$ in the interactive transformation step finally defines the layout functions according to the equations given in the specification Γ. Since the binding delivered by the constraint solver respects the constraints given in the specification, these are also satisfied by construction. Note that in this construction the layout of the whole diagram may be altered, since all attachment points are given to the constraint solver, whence all positions are determined anew. This is realistic, since the constraints are global w.r.t. the diagram. The introduction of a new association for example might require to move another class, and the renaming of a class might lead to a larger rectangle which might have side effects on its associations and the classes positioned close to it.

Consider as example again the application of the rule *insert_assoc*(x_1, x_2, "*make*") shown in Fig. 4.1. The constraint solver must move at least one of the two classes in order to display the association with its name properly in between them, as specified in the last relation in the constraints part of the specification. A possible output of the complete transformation step is given by the following algebra, with corresponding updates in the attachment points and layout functions.

Class	$\{x_1, x_2\}$
Assoc	$\{a\}$
c_name	$c\_name(x_1) =$ "*Client*", $c\_name(x_2) =$ "*Order*"

a_name	$a\_name(a) ='' make''$
a_begin	$a\_begin(a) = x_1$
a_end	$a\_begin(a) = x_2$
c_pos	$c\_pos(x_1) = (0.75, 2.25)$, $c\_pos(x_2) = (6.75, 2.25)$ (new position for x_2 !)
a_bpos	$a\_bpos(a) = (3.25, 1.5)$
a_epos	$a\_epos(a) = (6.75, 1.5)$ (attachment points for a)
c_layout	$c\_layout(x_1) = rect((0.75, 2.25), 2.5, 1.5)$, $c\_layout(x_2) = rect((6.75, 2.25), 2.5, 1.5)$ (according to the new position of x_2)
a_layout	$a\_layout(a) = line((3.25, 1.5), (6.75, 1.5))$ (the derived layout for a)

The embedding of a constraint solver into the interactive transformation can be defined more generally as follows. First a set of functions from the external signature is designated to be defined by the constraint solver. In our example these were the attachment points resp. the corresponding position functions, marked in the signature as *to be set externally*. The input variables of the constraint solver are then given by the set of all terms that consist of such a designated function symbol applied to an element of the actual state. Thereby the functions have to be compatible with the constraint solvers interface in the sense that their types are accepted (ParCon accepts real numbers and points for example). Then an appropriate set of equations is designated to obtain the constraints (marked as *constraints* in our example specification). Again, these have to be compatible with the input format of the constraint solver.

To let a user determine the positions of the logical elements is realized analogously to the connection of the constraint solver. In this case, however, we assume that the layout is changed only locally. The basic idea is again to obtain the new values of the attachment points for a logical element and replace the corresponding equations in the presentation $(\overline{iB}_S, \overline{iB}_E)$ obtained in the internal rewriting step. In this scenario an internal rewrite rule is always triggered by the user by pushing a button for one of the actions (*insert_class* etc.) and providing the parameters for this action. Then a preliminary (blinking or washy) layout appears somewhere on the screen and the user has to designate one or more points on the screen via her mouse to fix the layout. The number of points thereby depends on the specification, i.e., the number of attachment points for the concerned object (one for a class, two for an association etc.) Each mouse click is then interpreted as an equation, analogous to the translation of bindings to equations discussed above for the constraint solver.

5 Conclusion

In this paper we have introduced a formal model for interactive rule-based specifications of open systems. The states of a component are represented as partial

algebras, their transformations are obtained in two consecutive steps. The internal transformation of a state of the considered component is specified by rewriting the internal part of algebra by the application of algebra rewrite rules, the external ones are added by an external mechanism whose behaviour is only given abstractly as a mapping on algebras. In this framework we have discussed a formal model of a generic editor for visual languages that uses a graphical constraint solver as external mechanism for the computation of graphical layouts.

5.1 Extensions

As already remarked in the discussion of the example some extensions to the pure formal framework would support its applicability. In most applications it is convenient for example to distinguish – beyond internal and complete signature – further layers of the overall specification. Firstly, *static* parts may be distinguished by a subsignature $\Sigma_{static} \subseteq \Sigma_{in}$ that designates all parts of the states that are assumed to immutable, i.e., do not change when rules are applied. For instance, built-in or pervasive data types like integers, strings, or booleans are usually considered as static. In our example we used a *graphics*-algebra modelling a real plane, points, and various figures, as well as corresponding operations. Obviously, these sets and operations should always be the same. The designation of a static subsignature (or specification) yields a proof obligation for the rules since in general it cannot be assured that the rules indeed preserve the static part. Thus their consistency has to be shown.

Secondly, some functions may be derivable from other more basic, but also mutable functions. In our example the layout functions have been derived from the attachment points. This derived part can be distinguished by a superspecification $\Gamma_{der} \supseteq \Gamma$. Usually it will have the same set of sorts as Γ but introduce further functions with conditional equations that define them w.r.t. the ones in Γ. The specification extension $\Gamma \subseteq \Gamma_{der}$ induces a free functor F_{der} that yields the semantics of the derived functions. Note that in this case the semantics are always defined, since a free functor exists for each specification extension. Nevertheless, consistency and completeness of the extension are not guaranteed in general, which yields another proof obligation.

The interactive rewriting w.r.t. a *stratified transformation specification* $(\Gamma_{static} \subseteq \Gamma_{in} \subseteq \Gamma \subseteq \Gamma_{der}, A, R)$ is then defined for partial Γ_{der}-algebras and corresponding matches. It uses the interactive rewriting w.r.t. $(\Gamma_{in} \subseteq \Gamma)$ as defined above and extends it as follows. At the beginning a partial Γ_{der}-algebra $\tilde{A}$ is restricted to its Γ-part, which yields the input for the interactive transformation step, and at the end the free construction $B \mapsto F_{der}(B) = \tilde{B}$ is added to obtain the final result. That means, all derived functions are computed anew w.r.t. the new basic mutable functions, corresponding to the definitions given in the specification Γ_{der}.

5.2 Related Work

One of the main sources for the development of the interactive transformation specification framework has been the search for a complete and adequate formal model of the generic interactive visual editor environment GenGEd. The GenGEd environment proposed in [Bar00] has been developed for the visual definition of visual languages and corresponding visual editors. The definition of a visual language as well as the manipulation of visual expressions is based there on algebraic graph transformation and graphical constraint solving. Similar to formal textual languages, a visual language is defined by an alphabet and a grammar. The alphabet is represented by a graph structure signature, i.e., an algebraic signature with unary operation symbols, and a constraint satisfaction problem for the admissible layouts. Accordingly, the grammar is represented by a graph structure grammar, where the constraint satisfaction problems derived from the alphabet are satisfied for each visual expression in the grammar.

The graph structure signature for the alphabet represents both the logical and the visual part. The logical part is thereby defined by an attributed graph structure signature. This is extended by an algebraic signature for the visual part. The specification of the logical part of a visual language corresponds to the *internal part* of an interactive transformation signature, whereas the specification of the visual part corresponds to the *static part* and the *external part*. The approach using attributed graph structures and corresponding grammars, however, imposed design decisions on the formal model that turned out not to be feasible. In particular, in [Bar00] the connection between the visual part and the external component could be treated semi-formally only.

In the GenGEd environment, the graphical constraint solver ParCon [Gri96] is used to give the values (positions and sizes) for the visual part, i.e., for the layouts. Visual expressions are constructed by applying grammar rules according to the so-called *Single-Pushout* approach of graph transformation (see [LKW93]). The user of GenGEd therefore is supposed to have some knowledge about the treatment of side effects in this approach, which are not explicitly specified. In contrast to that, everything is explicitly specified in the formal approach we propose here. The only side effect is the automatic removal of relations (function entries) that contain elements that have been deleted. (In the single pushout approach also elements with identities can be removed implicitly as a side effect sometimes.)

In the literature one can find many approaches for specifying visual languages and creating editors for them ([MMW98]). These formalisms range from early approaches like array and web grammars, positional grammars, relational grammars, constraint multiset grammars, several types of graph grammars, logic-based approaches and algebraic approaches. The existence of many formalisms on the one hand gives rise to a lot of possible classification criteria [Schi98,Bur01] and, on the other hand, makes it difficult to decide about *the best* approach. Such a decision depends on the purpose of the approaches, for example, whether a visual or textual definition of a VL is in the fore, or which kind of editing mode

(freehand or syntax-directed editing) is supported in a graphical editor, or – if available – which kind of formal representation model is used.

Most tools for creating freehand editors analyze diagrams directly and avoid to create a formal representation model like a graph structure in [Bar00] or an algebra as presented in this paper. Possibly, freehand editing is desired in a graphical editor because a user can create and modify diagrams unrestrictedly; but these diagrams may contain errors that have to be recognized by a parser. In contrast, syntax-directed editing provides a set of editing commands which transforms correct diagrams into other correct diagrams; but the user is restricted to these commands.

The main aim of this paper has been the (visual) specification of visual languages which may be the basis for syntax-directed editing similar to [Bar00]. Furthermore, the approach presented here was motivated by [Bar00] in order to provide a formal approach for the logical part of a visual language and the visual part as well. Moreover, in contrast to the graph transformation formalism used in [Bar00] which has some side effects according to the formalism, in this paper all the features of a visual language are specified explicitly. A further difference between [Bar00] and our approach presented so far is given by the kind of algebras. In [Bar00] the algebras describing the logical part of a visual language are restricted to total algebras whereas we considered partial algebras.

A similar approach where expecially different kinds of graphics and graphical constraints are investigated is presented in [CLOT97,CP00]. In relation with our approach it can define an external device more concretely.

Beyond the several kinds of formalisms used for visual language specification, we have to mention the VAS (visual algebraic specification) formalism proposed in [DÜ96]. The VAS formalism is not only used for the specification of syntax but also semantics of visual languages. However, in this approach relations are not considered due to the logical part of a language and moreover, the semantics of a language is defined by evaluating terms over an algebraic specification according to conditional equations.

Concerning other formal approaches to the rule-based specification of open systems the *pull-back* approach to graph rewriting (see [EHLO98,EHLO99]) must be mentioned. Also in this approach the effect of a rule application is not completely specified. However, as opposed to the consecutive internal and external substeps in the interactive transformation specification approach only single global steps are considered, i.e., the impact of the environment is implicitly incorporated into the rule application in a parallel action. Moreover, this impact cannot be specified. Instead, each rule gives rise to (infinitely) many possible state transformations. Thus, beyond the use of graphs instead of algebras as state representations, this approach can be considered as dual w.r.t. the treatment of the other components.

The algebra rewriting formalism used for the internal rewriting steps is of course similar to the abstract state machine (ASM) approach (see [BH98]). We have chosen the former since we have been looking for a clean formal model incorporating function updates, creation and deletion of elements, and imposing

constraints directly. In fact, algebra rewriting supports arbitrary transformations of algebras in classes defined by arbitrary sets of conditional equations. the categorical approach allows us to reason about properties of transformation systems abstractly and avoid encodings of the desired structure. On the other hand, ASMs are supported by tools for the specification, analysis, and simulation. Thus to apply the interactive rule-based specification framework our abstract algebra rewriting concepts should be mapped to the ASM framework to make use of this tool support.

References

[Bar00] R. Bardohl "Visual Definition of Visual Languages based on Algebraic Graph Transformation" *Phd Thesis, Kovac Verlag* (2000)

[BH98] E. Börger and J.K. Huggins. Abstract State Machines 1988–1998. *Bull. EATCS 64*, pages 71–87, 1998. Commented ASM Bibliography.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson "The Unified Modeling Language User Guide" *Addison-Wesley* (1999)

[Bur01] M. Burnett. Visual Language Research Bibliography. URL: `http://www.cs.orst.edu/~burnett/vpl.html`.

[CLOT97] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A Framework of Syntactic Models for the Implementation of Visual Languages. In *Proc. IEEE Symp. on Visual Languages*, 1997.

[CP00] G. Costagliola, and G. Polese. Extended Positional Grammars. In *Proc. IEEE Symp. on Visual Languages*, 2000.

[DÜ96] T.B. Dinesh and S.M. Üsküdarlı. Specifying Input and Output of Visual Languages. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools.* World Scientific, 1999.

[EHLO98] H. Ehrig, R. Heckel, M Llabres, and F. Orejas. Construction and characterization of double-pullback graph transitions. In G. Engels and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Applications of Graph Transformation (TAGT'98)*, number tr-ri-98-201 in Reihe Informatik, pages 308-315. Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, 1998.

[EHLO99] H. Ehrig, R. Heckel, M Llabres, and F. Orejas. Basic properties of double-pullback graph transitions. Technical Report 99-02, Technical University of Berlin, 1999.

[ERT98] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In [EEKR99], pages 551-604.

[Gri96] P. Griebel. *ParCon – Paralleles Lösen von grafischen Constraints.* PhD thesis, Paderborn University, February 1996.

[Gro99] M. Große-Rhode "Specification of State Based Systems by Algebra Rewrite Systems and Refinements" *Tech. Report TU-Berlin 99-04* (1999)

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185-199. John Wiley & Sons Ltd, 1993.

[MM98] K. Marriott and B. Meyer (eds.). *Visual Language Theory*. Springer, 1998.
[MMW98] K. Marriott, B. Meyer, and K. Wittenburg. A Survey of Visual Language Specification and Recognition. In [MM98], pages 5-86.
[Schi98] J. Schiffer. *Visuelle Programmierung*. Addison-Wesley, 1998.