Algebra Transformation Systems and their Composition

Martin Große–Rhode [*]

Dipartimento di Informatica, Università degli Studi di Pisa,
Corso Italia, 40, I – 56125 Pisa, Italia, email: mgr@di.unipi.it

Abstract. Algebra transformation systems are introduced as formal models of components of open distributed systems. They are given by a transition graph modelling the control flow and partial algebras and method expressions modelling the data states and their transformations. According to this two–level structure they cover both labelled transition systems and rule based specification approaches, corresponding to information, computation and engineering viewpoint models. Different composition operations for algebra transformation systems are investigated. Limits and colimits model parallel and sequential composition of components, signature morphisms yield appropriate syntactical support for such compositions. The most important compositionality properties known from algebraic specification, like colimits of signatures and amalgamation of models, also hold for the framework of algebra transformation systems.

1 Introduction

Algebra transformation systems are introduced as formal models of components in open distributed systems in order to support multiple viewpoint modelling. The background for their development has been the reference model for open distributed systems RM–ODP, introduced as ISO–standard / ITU–recommendation [ODP]. One of the main features of RM–ODP is the introduction of five designated *viewpoints* as structuring means for design and specification of distributed systems. Viewpoints allow the separation of different aspects of a system, such that specifications can be restricted to the parts that are relevant for some specific use. The five viewpoints defined in RM–ODP are *enterprise*, *information*, *computation*, *engineering*, and *technology* viewpoint. With the approach of algebra transformation systems especially the information, computation, and engineering viewpoints are supported. That means the information model, the computational model, and the infrastructure needed to realize the services of the system in a distributed environment could be modelled formally as algebra transformation systems.

[*] This work has been partially supported by the EEC TMR network GETGRATS (General Theory of Graph Transformation Systems), contract number ERB-FMRX-CT960061.

As specification languages for these viewpoints RM–ODP recommends, beyond others, Z for the information viewpoint and LOTOS for computation and engineering viewpoints. In Z *static*, *invariant*, and *dynamic* schemata for the specification of designated states, state invariants, and state transformations respectively can be given, corresponding to the requirements of an information viewpoint specification language. LOTOS supports the specification of the interaction of computation objects at interfaces, corresponding to computational and engineering viewpoint resp.

The purpose of the algebra transformation system approach is to deliver a formal semantical approach that covers both specification approaches explicated by Z and LOTOS. That means, the rule based approach with internally structured states and their transformation underlying Z, as well as the approach via the temporal ordering of observable actions underlying LOTOS shall be integrated. One of the main advantages of such an integration is the possibility to embed both kinds of models into one framework to compare them and check their consistency.

An algebra transformation system is given by two levels. The first one, called the transition graph, models the reactive behaviour of the system, that is, its temporal ordering of actions. Associated with each *control state* of this level is a *data state* on the second level that models the information available in this state. The transitions of control states are accordingly labelled with sets of *method expressions* on the second level, that indicate which methods have been applied for this data state transformation and how they have been applied.

The second main purpose of this paper is to introduce composition operations for algebra transformation systems, that allow to model different kinds of compositions of components. According to their two–level structure these composition operations are inherited from composition operations for labelled transition systems on the one hand, and abstract data types on the other hand. Parallel composition of labelled transition systems for instance is modelled by limits, such as products (see [WN95]), whereas composition of abstract data types is given by colimits (see e.g. [BG77,EM85]). This duality is reflected for algebra transformation systems in the definition of the corresponding morphisms, where the two levels are mapped in opposite order. In this way the right composition operations are put together for the two levels. Beyond parallel composition, with appropriate synchronization mechanisms, sequential composition is investigated. This is particularly important for the definition of control structures for rule based specifications, which are usually encoded into the states. However, an explicit modelling of the control flow as in algebra transformation systems, independently of the information represented in the data states, is much more appropriate and supports clear and manageable specifications much better. Sequential composition is given by colimits of the transition graphs (i.e. the control structure) with identities on the data states; passing the control flow from one component to another should not change the underlying data state.

The paper is organized as follows. In the next section the category of algebra transformation systems is introduced. Then composition in the sense discussed above is introduced, i.e. limits and colimits are defined. In section 4 signature morphisms are defined and their support for composition operations is discussed. For all definitions and constructions small examples are given to show how they work and how they can be used for applications. Further examples can be found in [Gro97], where also a methodology for the presentation of transformation systems is discussed. Finally in section 5 a short summary and a comparison with other approaches are given, and further questions are discussed. Full proofs of the propositions and theorems of sections 2 and 3 of this paper can be found in [Gro97].

2 The Category of Transformation Systems

As discussed above an algebra transformation system is given by two levels, the transition graph modelling the control flow, and the data states and method expressions representing the information available in the states and transitions. The data states of an algebra transformation system are given by partial algebras to a common partial equational specification. A signature for partial algebras is a usual algebraic signature $SIG = (S, OP)$, given by a set S of sort names and a family $OP = (OP_{w,s})_{w \in S^*, s \in S}$ of operation symbols indexed by their arities. As usual an operation symbol $op \in OP_{w,s}$ is denoted $op : w \to s$. The semantical difference to total algebras is that an operation symbol $op : w \to s$ is interpreted as a *partial* function $op^A : A_{s_1} \times \cdots \times A_{s_n} \multimap A_s$ (if $w = s_1 \dots s_n$), given by a domain of definition $dom(op^A) \subseteq A_{s_1} \times \cdots \times A_{s_n}$ and a total mapping $dom(op^A) \to A_s$. Homomorphisms of partial $SPEC$–algebras are given by families of total functions that preserve the domains of definition of the operations, and operation application. An equation $t = t'$ is satisfied by a partial algebra A if both terms t and t' can be evaluated (are defined) in A, and yield the same value. Satisfaction of a conditional equation $r = r' \Rightarrow t = t'$ is defined as usual: whenever the premise $r = r'$ is satisfied, then also the conclusion $t = t'$ must be satisfied. The interpretation of equations also yields the definedness predicate for terms t, denoted $t\downarrow$, given by $t\downarrow$ iff $t = t$. A signature SIG with a set CE of conditional equations is called a partial equational specification $SPEC$. The category of partial $SPEC$–algebras and homomorphisms is denoted $PAlg(SPEC)$, its class of objects $|PAlg(SPEC)|$. (For further details see [Rei87,CGW95].)

Partial algebras have been chosen for two reasons. Firstly they comprise first order structures, since predicates can be modelled by partial functions to a singleton set. Of course they are more general than first order structures because of the partial functions. Secondly, partiality allows to model the difference between declaration and instantiation (resp. initialization) of syntactic entities in a natural way, because terms to a signature need not be interpreted in all partial algebras to this signature. Especially in the context of dynamically changing states this feature is very appropriate.

Method names are added to the data state specification, like operation symbols, as names with arities that determine the number and type of the parameters they require. However, methods do not have an output sort, they do only change the state. Data outputs have to be modelled by data type functions whose value can be determined by the actual state, according to the conditional equations in the data type specification.

Definition 1 (Transformation Signature). A *transformation signature* $T\Sigma = (SPEC, M)$ is given by a partial equational specification $SPEC = (S, OP, CE)$ and a *method signature* $M = (M_w)_{w \in S^*}$. A method name $m \in M_w$ is denoted $m : w$ for short.

Transitions are labelled by sets of method expressions that contain the names of the methods that have been applied, and corresponding lists of parameters from the actual state.

Definition 2 (Method Expression). Given a transformation signature $T\Sigma = (SPEC, M)$ the set $ME_{T\Sigma}$ of *method expressions* is defined by

$$ME_{T\Sigma} = \bigcup_{A \in |PAlg(SPEC)|} ME_{T\Sigma}(A) \ ,$$

where the components $ME_{T\Sigma}(A)$, $A \in |PAlg(SPEC)|$, are defined by

$$ME_{T\Sigma}(A) = \{m(a) \mid m \in M_w, a \in A_w\} \ .$$

As prerequisite for the definition of transformation systems let me shortly fix the formal structure of *sets of method expressions.* The powersets $\mathcal{P}(ME_{T\Sigma}(A))$ with inclusions as morphisms are categories. These are indexed by the functor

$$\mathcal{P}(ME_{T\Sigma}(\_)) : PAlg(SPEC) \to \mathbf{Cat}.$$

It is defined on a $SPEC$-homomorphism $h : A \to B$ as the direct image, i.e.

$$\mathcal{P}(ME_{T\Sigma}(h))(K) = \{m(h(a)) \mid m(a) \in K\} \subseteq ME_{T\Sigma}(B)$$

for all $K \subseteq ME_{T\Sigma}(A)$. Obviously $\mathcal{P}(ME_{T\Sigma}(h))$ and $\mathcal{P}(ME_{T\Sigma}(\_))$ are functors. This indexing induces, via the appropriate Grothendieck construction, the flat category denoted $\mathcal{P}(ME_{T\Sigma})$, whose objects are pairs (A, K), with $A \in |PAlg(SPEC)|$ and $K \subseteq ME_{T\Sigma}(A)$, and whose morphisms are pairs $(h, \subseteq) : (A, K) \to (B, L)$, where $h : A \to B$ is a $SPEC$-homomorphism such that $\mathcal{P}(ME_{T\Sigma}(h))(K) \subseteq L$. Overloading notation a bit both the functor $\mathcal{P}(ME_{T\Sigma}(h))$ and the morphism $(h, \subseteq)$ will be denoted by h in the sequel.

Beyond the data states (= partial $SPEC$-algebras) and the method expressions an algebra transformation system comprises the control flow. It is modelled by a directed graph of control states and transitions, and it may have loops and multiple edges. This graph is formally given by sets of states and transitions, and functions *src* and *tar* that assign source and target states to the transitions. The data states and sets of method expressions are then formally modelled as labels of control states and transitions.

Definition 3 (Transformation System). Let $T\Sigma = (SPEC, M)$ be a transformation signature. A $T\Sigma$-*transformation system* $\mathcal{A} = (TG_{\mathcal{A}}, lab_{\mathcal{A}})$ is given by a transition graph

$$TG_{\mathcal{A}} = (\mathcal{S}, \mathcal{T}, src, tar) \text{ with } src, tar : \mathcal{T} \to \mathcal{S} \ ,$$

and a pair of functions

$$lab_{\mathcal{A}} = (lab_{\mathcal{S}} : \mathcal{S} \to |PAlg(SPEC)|, lab_{\mathcal{T}} : \mathcal{T} \to \mathcal{P}(ME_{T\Sigma})) \ ,$$

such that

$$lab_{\mathcal{T}}(l) \subseteq ME_{T\Sigma}(lab_{\mathcal{S}}(src(l)))$$

for all $l \in \mathcal{T}$, i.e. the parameters are always taken from the actual source state.

$$\begin{tikzcd}
s \arrow[r, "l"] \arrow[d, dashed, maps to] & t \arrow[d, dashed, maps to] \\
S \arrow[r, Rightarrow, "\{m(a), \ldots\}"'] & T
\end{tikzcd}$$

The two labelling functions support the methodological separation between control flow and data transformation level. The first one is completely independent from the transformation signature, which will be used later on in the definition of the forgetful functor. Note that a transition may be labelled by the empty set, which allows to model data state transformations induced by the environment.

A transition $l \in \mathcal{T}$ with $src(l) = s$ and $tar(l) = t$ will be denoted $l : s \to t$. Moreover both $l \in \mathcal{T}$ and the triple $l : s \to t$ will be called transition. Correspondingly $\mathcal{T}$ is called the *transition relation*. The labels of states and transitions will also be indicated by capital letters, i.e. $lab_{\mathcal{S}}(s) = S$ and $lab_{\mathcal{T}}(l) = L$ for states $s \in \mathcal{S}$ and transitions $l \in \mathcal{T}$. The condition that parameters are always taken from the actual state in the definition above thus reads: $L \subseteq ME_{T\Sigma}(S)$ for all $l : s \to t \in \mathcal{T}$.

Example 4. Consider as running example the following signature of a program that increments the value of a program variable by a given positive natural number.

A specification **nat** of the natural numbers is extended by a sort *prog_var* of program variables and a partial function ! that assigns — in each state — the actual values to the program variables. Furthermore a variable name p is introduced and a method *inc* to increment the value of a variable by a given amount.

prog = **nat** +
  sorts prog_var
  opns !: prog_var $\to$ nat
    p: $\to$ prog_var
  meths inc: prog_var, nat

A **prog**–transformation system that models the expected behaviour is defined as follows. Let **prog–data** be the partial equational specification given by **prog** without the method name *inc*, and X_n for some $n \in \mathbb{N}$ be the partial **prog–data**–algebra defined by

$$X_n|_{\mathbf{nat}} = \mathbb{N},\ (X_n)_{prog\_var} = \{X\},\ p^{X_n} = X,\ !^{X_n}(X) = n,$$

i.e. X_n is the state in which X has the value n. Then a **prog**–transformation system $\mathcal{X} = (TG_{\mathcal{X}}, lab_{\mathcal{X}})$ can be defined by

control states	$\mathcal{S}_{\mathcal{X}} = \mathbb{N}$
data states	$lab_{\mathcal{S}}(n) = X_n$
transitions	$\mathcal{T}_{\mathcal{X}} = \{(k : n \rightarrow n + k) \mid k > 0\}$
method expressions	$lab_{\mathcal{T}}(k : n \rightarrow n + k) = \{inc(X, k)\}$

This model contains as labelled paths all sequences of method applications.

$$\begin{tikzcd}
n \arrow[r, "k"] \arrow[d, mapsto, dashed] & n+k \arrow[r, "k'"] \arrow[d, mapsto, dashed] & n+k+k' \arrow[r] \arrow[d, mapsto, dashed] & \cdots \\
X_n \arrow[r, Rightarrow, "\{inc(X,k)\}"'] & X_{n+k} \arrow[r, Rightarrow, "\{inc(X,k')\}"'] & X_{n+k+k'} \arrow[r, Rightarrow] & \cdots
\end{tikzcd}$$

In order to model a control flow that stops after each single step the control flow information has to be refined. E.g. the control states are extended by marks *start* and *stop*, and each transition leads from a *start*–state to a corresponding *stop*–state.

control states	$\mathcal{S} = \mathbb{N} \times \{start, stop\}$
data states	$lab_{\mathcal{S}}(n, start) = lab_{\mathcal{S}}(n, stop) = X_n$
transitions	$\mathcal{T} = \{k : (n, start) \rightarrow (n + k, stop) \mid k > 0\}$
method expressions	$lab_{\mathcal{T}}(k : (n, start) \rightarrow (n + k, stop)) = \{inc(X, k)\}$

$$\begin{tikzcd}
(n, start) \arrow[r, "k"] \arrow[d, mapsto, dashed] & (n+k, stop) \arrow[dr, mapsto, dashed] & & (n+k, start) \arrow[r, "k'"] \arrow[dl, mapsto, dashed] & (n+k+k', stop) \arrow[d, mapsto, dashed] \\
X_n \arrow[rr, Rightarrow, "\{inc(X,k)\}"'] & & X_{n+k} \arrow[rr, Rightarrow, "\{inc(X,k')\}"'] & & X_{n+k+k'}
\end{tikzcd}$$

In both examples the method *inc* induces a function, that assigns to each data state X_n and all parameters X and k in X_n a successor state X_{n+k}. In general methods need neither be total nor deterministic, that is, they correspond to relations rather than functions.

As mentioned in the introduction morphisms of transformation systems are defined in such a way that appropriate composition operations are supported. Composition of data states is modelled by colimits in the category of partial algebras. This models the superposition of data states w.r.t. some shared parts. (Thus also the communication on the data state level is given by sharing.) A coproduct of two partial *SPEC*–algebras A and B for example can roughly be described as follows. The carrier sets of $A + B$ are the unions of the (renamed) carriers of A and B, where the term generated elements are identified and the non generated parts are united disjointly. The operations of $A + B$ are given by the corresponding unions of the operations of A and B, provided they coincide on the intersections of their domains in $A+B$, and their union still satisfies the axioms. (Otherwise the carriers may collapse.) Thus coproduct is disjoint union with sharing of term generated parts.

On the other level, parallel composition of transition graphs, like parallel composition of labelled transitions systems, is given by limits. Products correspond to pure parallel composition, pullbacks correspond to parallel composition with synchronization between the parts, induced by actions that both components must perform during the same step. Since parallel composition of components should be supported for the case that both components act on different parts of a data state, possibly overlapping in basic types or the part they synchronize upon, limits of the transition graph part should be combined with colimits on the data state part. That means, the two levels of a morphism of transformation systems must have opposite direction. [1]

Definition 5 (Transformation System Morphism). Let $\mathcal{A} = (TG_{\mathcal{A}}, lab_{\mathcal{A}})$ and $\mathcal{A}' = (TG_{\mathcal{A}'}, lab_{\mathcal{A}'})$ be $T\Sigma$-transformation systems. A *$T\Sigma$-morphism* $h = (h_{TG}, (h_s)_{s \in \mathcal{S}}) : \mathcal{A} \to \mathcal{A}'$ is given by a graph homomorphism

$$h_{TG} : TG_{\mathcal{A}} \to TG_{\mathcal{A}'},$$

and a family of $SPEC$-homomorphisms

$$(h_s : S' \to S)_{s \in \mathcal{S}} \quad (\text{with } s' = h_{TG}(s)) \ ,$$

such that

$$h_s(L') \subseteq L \quad (\text{with } l' = h_{TG}(l))$$

for all $l : s \to t \in \mathcal{T}$.

$$\begin{tikzcd} s \arrow[rr, "l"] \arrow[d, "h_{TG}"'] & {} \arrow[d, "h_{TG}"] & t \arrow[d, "h_{TG}"] \\ s' \arrow[rr, "l'"'] & {} & t' \end{tikzcd} \qquad \begin{tikzcd} S \arrow[rr, Rightarrow, "L"] & {} & T \\ S' \arrow[u, "h_s"] \arrow[rr, Rightarrow, "L'"'] & {} \arrow[u, "h_s"] & T' \arrow[u, "h_t"'] \end{tikzcd}$$

Proposition 6. *$T\Sigma$-transformation systems and morphisms form a category, called* $\mathbf{TS}(T\Sigma)$*, for each transformation signature $T\Sigma$.*

[1] An analogy for these opposite directions can be seen in the duality of algebraic specification of data types and coalgebraic specification of dynamic systems. Initial algebras are the designated, typical and generated models of an algebraic specification, final coalgebras play the corresponding role for dynamic systems.

3 Composition by Limits and Colimits

A limit $\mathcal{A}$ of a diagram of transformation systems $\mathcal{A}^i$ and morphisms $h : \mathcal{A}^i \rightarrow \mathcal{A}^j$ is constructed as follows. First the limit $TG_{\mathcal{A}}$ of the transition graphs $TG_{\mathcal{A}^i}$ and graph morphisms $h_{TG} : TG_{\mathcal{A}^i} \rightarrow TG_{\mathcal{A}^j}$ is constructed. Thus a transition $l : s \rightarrow t$ in $\mathcal{A}$ is given by a family of transitions $l^i : s^i \rightarrow t^i$ in $\mathcal{A}^i$ with $h_{TG}(l^i : s^i \rightarrow t^i) = l^j : s^j \rightarrow t^j$ for all $h : \mathcal{A}^i \rightarrow \mathcal{A}^j$ in the diagram. For any state s in $TG_{\mathcal{A}}$ the data state S is then given by the colimit of all data states S^i of the control states $s^i = \pi^i_{TG}(s)$ and the data state homomorphisms $h_{s^i} : S^j \rightarrow S^i$ with $h_{TG}(s^i) = s^j$. The label L of a transition $l : s \rightarrow t$ in $TG_{\mathcal{A}}$ is obtained in the same way as a colimit in $\mathcal{P}(ME_{T\Sigma})$.

$$\begin{tikzcd}
 & t^j & \\
t \arrow[ur, mapsto, "\pi^j_{TG}"] \arrow[dr, mapsto] & s^j \arrow[u, "l^j"'] & \\
s \arrow[u, "l"] \arrow[ur, mapsto] \arrow[dr, mapsto, "\pi^i_{TG}"'] & t^i \arrow[uu, mapsto, bend right=60, "h_{TG}"'] & \\
 & s^i \arrow[u, "l^i"] \arrow[uu, mapsto, bend right=60, "h_{TG}"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & T^j \arrow[dl, "\pi^j_t"'] \arrow[dd, bend left=60, "h_{t^i}"] \\
T & S^j \arrow[u, Rightarrow, "L^j"] \arrow[dl] \arrow[dd, bend left=60, "h_{s^i}"] \\
S \arrow[u, Rightarrow, "L"] & T^i \arrow[ul] \\
 & S^i \arrow[ul, "\pi^i_s"] \arrow[u, Rightarrow, "L^i"']
\end{tikzcd}$$

Since both the categories of graphs and partial algebras have limits and colimits, the dual construction yields colimits of algebra transformation systems. Thus we obtain

Theorem 7. **TS**$(T\Sigma)$ *is complete and cocomplete.*

Example 8 (Parallel Composition). As an example for the parallel composition of transformation systems via limits consider the following extension of the transformation signature **prog** and the **prog**–transformation system $\mathcal{X}$ in example 4. First the signature is extended to a transformation signature **p,q–prog** by a second program variable $q\!: \rightarrow$ *prog_var*. Then two **p,q–prog**–transformation models $\mathcal{Y}$ and $\mathcal{Z}$ are constructed similarly to the **prog**–model $\mathcal{X}$. Each has access to one of the two program variables and increments it using the method *inc*. The corresponding data states Y_n and Z_m for $n, m \in \mathbb{N}$ are given by

$$Y_n|_{\mathbf{nat}} = Z_m|_{\mathbf{nat}} = \mathbb{N},\ (Y_n)_{prog\_var} = \{Y\}, (Z_m)_{prog\_var} = \{Z\},$$
$$p^{Y_n} = Y,\ q^{Y_n} \text{ is undefined, } !^{Y_n}(Y) = n,$$
$$p^{Z_m} \text{ is undefined, } q^{Z_m} = Z,\ !^{Z_m}(Z) = m\ .$$

The transition graphs of $\mathcal{Y}$ and $\mathcal{Z}$ are defined like the one of $\mathcal{X}$, extended by *idle* transitions $0 : n \rightarrow n$ labelled by the empty set. This allows to model *independent* transformations in the product constructed below. The other labels are defined accordingly.

	$\mathcal{Y}$	$\mathcal{Z}$
control states	$\mathbb{N}$	$\mathbb{N}$
data states	$n \mapsto Y_n$	$m \mapsto Z_m$
transitions	$\{(k : n \to n+k) \mid k \geq 0\}$	$\{(l : m \to m+l) \mid l \geq 0\}$
method exp.'s	$k \mapsto \begin{cases} \{inc(Y,k)\} & \text{if } k > 0 \\ \emptyset & \text{if } k = 0 \end{cases}$	$l \mapsto \begin{cases} \{inc(Z,l)\} & \text{if } l > 0 \\ \emptyset & \text{if } l = 0 \end{cases}$

According to the general construction of limits discussed above the product $\mathcal{Y} \times \mathcal{Z}$ is given by the products of the transition graphs of $\mathcal{Y}$ and $\mathcal{Z}$, for each control state the coproduct of the component data states, and for each transition the union of the sets of method expressions labelling the component transitions. Explicitly this means $\mathcal{Y} \times \mathcal{Z}$ is given by

$$\mathcal{S}_{\mathcal{Y}\times\mathcal{Z}} = \mathbb{N} \times \mathbb{N}$$

$$lab_{\mathcal{Y}\times\mathcal{Z}}(n,m) = Y_n + Z_m$$

$$\mathcal{T}_{\mathcal{Y}\times\mathcal{Z}} = \{(k,l) : (n,m) \to (n+k, m+l)) \mid k,l \geq 0\}$$

$$lab_{\mathcal{Y}\times\mathcal{Z}}((k,l) : (n,m) \to (n+k, m+l))) =$$

$$= \begin{cases} \{inc(Y,k), inc(Z,l)\} & \text{if } k > 0, l > 0 \\ \{inc(Y,k)\} & \text{if } k > 0, l = 0 \\ \{inc(Z,l)\} & \text{if } k = 0, l > 0 \\ \emptyset & \text{if } k = 0, l = 0 \end{cases}$$

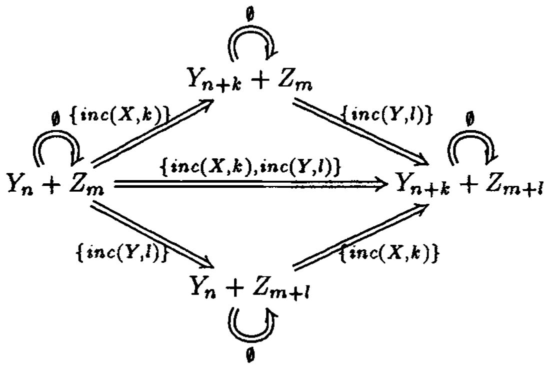
where the partial **prog–data**–algebras $Y_n + Z_m$ are given by

$$Y_n + Z_m|_{\mathbf{nat}} = \mathbb{N},\ (Y_n + Z_m)_{prog\_var} = \{X, Y\},$$

$$p^{Y_n+Z_m} = Y, q^{Y_n+Z_m} = Z,$$

$$!^{Y_n+Z_m}(Y) = n, !^{Y_n+Z_m}(Z) = m,$$

i.e. $Y_n + Z_m$ is the state in which Y has the value n and Z has the value m .



$$\begin{tikzcd}
 & Y_{n+k} + Z_m \arrow[loop above, Rightarrow, "\emptyset"] \arrow[dr, Rightarrow, "\{inc(Y,l)\}"] & \\
Y_n + Z_m \arrow[loop above, Rightarrow, "\emptyset"] \arrow[ur, Rightarrow, "\{inc(X,k)\}"] \arrow[rr, Rightarrow, "{\{inc(X,k),inc(Y,l)\}}"] \arrow[dr, Rightarrow, "\{inc(Y,l)\}"'] & & Y_{n+k} + Z_{m+l} \arrow[loop above, Rightarrow, "\emptyset"] \\
 & Y_n + Z_{m+l} \arrow[loop below, Rightarrow, "\emptyset"] \arrow[ur, Rightarrow, "\{inc(X,k)\}"'] &
\end{tikzcd}$$

Example 9 (Sequential Composition). In this example colimits are used for the sequential composition of transformation systems. A mutable list of data items with list cursor on a given set of data shall be modelled, and a method to delete the n'th element of this list. This *delete*–method is put together by a *move*–method that moves the cursor to the right place and an *erase*–method that erases the actual element of the list.

The list is defined in each state by a partial function *next:data* $\rightarrow$ *data*, the list cursor by a pointer to natural numbers. The position n of the element that shall be deleted is given to the system as parameter of the *move* and *delete* methods.

```
list = nat + data +
    opns   next: data → data
           cursor: → nat
    meths  erase:
           move, delete: nat
```

The data states are represented by two lists corresponding to the parts of the list before the cursor and the remainder. These lists $l, r \in A^*$ shall satisfy the side condition that their concatenation lr has no repetitions in order to define a partial function *next*. Each such pair $(l, r) \in A^* \times A^*$ then defines a partial **list–data** algebra $S(l, r)$ by

$$S(l,r)|_{\mathbf{nat}} = \mathbb{N},\ S(l,r)_{\mathbf{data}} = A,$$
$$next^{S(l,r)}(a_i) = a_{i+1} \text{ if } lr = a_1 \ldots a_m$$
$$cursor^{S(l,r)} = length(l) + 1\ .$$

Now three **list**–**transformation** systems are defined on top of these data states, one for each method. The same signature can be used for all three, because not all method names need to occur in a model.

Move : The pairs of lists (l, r) are used as control states for the *move* method, together with a natural number n that records the target position. If the target is reached this is represented by the tag $st(n)$; the position is *stable.*

$$\mathcal{S}^{move} = A^* \times A^* \times (\mathbb{N} \cup \{st(n) \mid n \in \mathbb{N}\})$$
$$labs_{\mathcal{S}}(l, r, \eta) = S(l, r)$$

$\mathcal{T}^{move}$ is defined by

$$\begin{array}{llll} (l, xr', n) & \rightarrow (lx, r', n) & \text{iff} & length(l) + 1 < n \leq length(lxr') \\ (l, r, n) & \rightarrow (\epsilon, lr, n) & \text{iff} & n \leq length(l) \\ (l, r, n) & \rightarrow (l, r, st(n)) & \text{iff} & length(l) + 1 = n \end{array}$$

All these transitions are labelled $\{move(n)\}$, i.e. the index n in the abstract state representation is the parameter of the method until it reaches its target.

Erase : Since the erasion of the actual list element requires exactly one step the marks *start* and *stop* are used again in the state representation.

$$\mathcal{S}^{erase} = A^* \times A^* \times \{start, stop\}$$

$$lab_{\mathcal{S}^{erase}}(l, r, s) = S(l, r)$$

The transition relation is given by the transitions

$$(l, xr', start) \longrightarrow (l, r', stop)$$

all labelled $\{erase\}$.

Delete : Now *move* and *erase* are put together to implement the *delete* method. For that purpose define the connecting **list**–transformation model C that contains all pairs of control states of *move* and *erase* that are to be connected, and no transitions. Note that the data states of connected control states are identical.

$$\mathcal{S}_C = \{((l, r, st(n)), (l, r, start)) \mid l, r \in A^*, n \in \mathbb{N}\}$$

$$lab((l, r, st(n)), (l, r, start)) = S(l, r)$$

$$\mathcal{T}_C = \emptyset$$

The transformation system morphisms from C to the *move* and *erase* models are given by the projections of $\mathcal{S}_C$ to $\mathcal{S}^{move}$ and $\mathcal{S}^{erase}$ resp., and the identity data state homomorphisms. Due to the pushout construction of graphs the pushout of this diagram of transformation systems identifies all pairs of control states $(l, r, st(n)), (l, r, start)$. That means, when *move* has reached its target, *erase* starts, and performs one step in which it erases the actual (= n'th) list element.

4 Composition by Signature Morphisms

The two program models in example 8 that update different variables could be defined w.r.t. a common signature, because partial algebras make it possible to leave parts of the signature uninterpreted. In general a signature can always be chosen large enough to be suitable for a given set of partial algebras in this way. The same holds for method names, because they need not necessarily occur in any transition label set. Thus formally local signatures and signature morphisms would not be necessary. However, restricting the signatures to the parts that are actually accessed yields a much better overall structure of the specification. It documents independence of methods and supports local modelling. Moreover signature morphisms can be used to rename components and put them together in different ways. Instead of equality of names, that might always be considered as accidental, categorical composition techniques always require morphisms to make explicit the connections between the components. All items from different components that are not explicitly related are considered as being different.

Definition 10 (Transformation Signature Morphism). Let $T\Sigma = (SPEC, M)$ and $T\Sigma' = (SPEC', M')$ be transformation signatures. A *transformation signature morphism* $\sigma = (\sigma_{SPEC}, \sigma_M) : T\Sigma \longrightarrow T\Sigma'$ is given by a specification morphism $\sigma_{SPEC} : SPEC \rightarrow SPEC'$ and a family of functions $\sigma_M = (\sigma_{M,w} : M_w \rightarrow M'_{\sigma_{SPEC}(w)})_{w \in S^*}$, where S is the set of sorts of $SPEC$.

Transformation signatures and transformation signature morphisms define the category **TransSig** .

Since signatures and signature morphisms for algebra transformation systems formally coincide with signatures and signature morphisms for partial algebras we immediately obtain the following constructions, that are the basis for the subsequent compositionality results.

Proposition 11. *The category TransSig is cocomplete.*

The forgetful functor of transformation systems induced by a transformation signature morphism leaves the transition graphs unchanged. (Recall that the transition graphs are independent of the transformation signatures.) The data states are replaced by their restrictions according to the forgetful functor of partial algebras, and the sets of method expressions are replaced by the (renamed) subsets of method names corresponding to the smaller set of method names.

Definition 12 (Forgetful Functor). Given a transformation signature morphism $\sigma = (\sigma_{SPEC}, \sigma_M) : T\Sigma \rightarrow T\Sigma'$ the *forgetful functor* $V_\sigma : \mathbf{TS}(T\Sigma') \rightarrow \mathbf{TS}(T\Sigma)$ is defined as follows.

1. Let $\mathcal{A}' = (TG_{\mathcal{A}'}, lab_{\mathcal{A}'}) \in \mathbf{TS}(T\Sigma')$.
   Then $V_\sigma(\mathcal{A}') =: \mathcal{A} = (TG_{\mathcal{A}}, lab_{\mathcal{A}}) \in \mathbf{TS}(T\Sigma)$ is defined by
   - $TG_{\mathcal{A}} = TG_{\mathcal{A}'}$
   - $lab_S(s) = V_{\sigma_{SPEC}}(lab_{S'}(s))$
     where $V_{\sigma_{SPEC}} : PAlg(SPEC') \rightarrow PAlg(SPEC)$ is the forgetful functor induced by $\sigma_{SPEC} : SPEC \rightarrow SPEC'$,
   - $lab_T(l : s \rightarrow t) = \{m(a) \in ME_{T\Sigma} \mid \sigma_M(m)(a) \in lab_{T'}(l : s \rightarrow t)\}$
2. Let $h' = (h'_{TG}, (h'_{s'})_{s' \in S_{\mathcal{A}'}}) : \mathcal{A}' \rightarrow \mathcal{B}'$ in $\mathbf{TS}(T\Sigma')$.
   Then $V_\sigma(h') =: h = (h_{TG}, (h_s)_{s \in S_{\mathcal{A}}}) : V_\sigma(\mathcal{A}') \rightarrow V_\sigma(\mathcal{B}')$ is defined by
   - $h_{TG} = h'_{TG}$
   - $h_s : lab_{V_\sigma(\mathcal{B}')}(h_{TG}(s)) \rightarrow lab_{V_\sigma(\mathcal{A}')}(s) =$
     $= V_{\sigma_{SPEC}}(h_s : lab_{\mathcal{B}'}(h'_{TG}(s)) \rightarrow lab_{\mathcal{A}'}(s))$

It is easily checked that V_σ is well defined.

Definition 13 (Model Functor). The *model functor* Mod : **TransSig** $\rightarrow$ $\mathbf{Cat}^{op}$ is defined by $Mod(T\Sigma) = \mathbf{TS}(T\Sigma)$ and $Mod(\sigma) = V_\sigma$.

Since the forgetful functor for algebra transformation systems is given by the identity on transition graphs and algebraic forgetful functors on data states, the *amalgamation property*, known from algebraic specification, carries over to algebra transformation systems. That means, given a pushout of transformation signatures its model category is the pullback of the model categories of the components.

Theorem 14 (Amalgamation). *Mod preserves pushouts.*

This property guarantees the existence and uniqueness of amalgamated sum of algebra transformation systems and morphisms. If $\sigma_1 : T\Sigma_0 \to T\Sigma_1$ and $\sigma_2 : T\Sigma_0 \to T\Sigma_2$ are transformation signature morphisms and $\bar{\sigma}_1 : T\Sigma_1 \to T\Sigma_3$ and $\bar{\sigma}_2 : T\Sigma_2 \to T\Sigma_3$ is their pushout, then for each pair of transformation systems $\mathcal{A}_1 \in \mathbf{TS}(T\Sigma_1)$ and $\mathcal{A}_2 \in \mathbf{TS}(T\Sigma_2)$ with $V_{\sigma_1}(\mathcal{A}_1) = V_{\sigma_2}(\mathcal{A}_2)$ there is a $T\Sigma_3$–transformation system $\mathcal{A}_3 = \mathcal{A}_1 +_{\mathcal{A}_0} \mathcal{A}_2 \in \mathbf{TS}(T\Sigma_3)$, where $\mathcal{A}_0 = V_{\sigma_1}(\mathcal{A}_1)$, such that $V_{\bar{\sigma}_1}(\mathcal{A}_3) = \mathcal{A}_1$ and $V_{\bar{\sigma}_2}(\mathcal{A}_3) = \mathcal{A}_2$. Moreover $\mathcal{A}_3$ is uniquely determined by this property. The same amalgamation property holds for morphisms. The model $\mathcal{A}_3 = \mathcal{A}_1 +_{\mathcal{A}_0} \mathcal{A}_2$ is called the *amalgamated sum* of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Let's finally combine the signature morphisms for the data transformation level with morphisms of transition graphs in order to obtain the appropriate composition operations for both levels together. For that purpose the appropriate Grothendieck category has to be taken. (Cf. the construction of sets of method expressions above.) In this case it is given by

objects	$(T\Sigma, \mathcal{A})$, with $\mathcal{A} \in \lvert\mathbf{TS}(T\Sigma)\rvert$
morphisms	$m = (\sigma, h) : (T\Sigma, \mathcal{A}) \to (T\Sigma', \mathcal{A}')$
	with $\sigma : T\Sigma' \to T\Sigma$ in **TransSig**
	and $h : V_\sigma(\mathcal{A}) \to \mathcal{A}'$ in $\mathbf{TS}(T\Sigma')$

Note how the opposite direction of transition graph morphisms and data state morphisms is reflected in the direction of the algebraic specification morphisms.

Pullbacks in this category are given by pullbacks of transition graphs, and for each control state s the amalgamated data state $S = S_2 +_{S_0} S_1$, where the $s_i = \pi_i(s), i = 0, 1, 2$, are the projections of s in the components. The method expressions are the corresponding amalgamations (= unions of renamed sets of method expression).

Example 15. With signature morphisms and the Grothendieck morphisms, the construction of the parallel composition $\mathcal{Y} \times \mathcal{Z}$ in example 8 can be reformulated in such a way, that the independence of the two incrementation processes is documented already on the syntactical level. That means, the models $\mathcal{Y}$ and $\mathcal{Z}$ can be reconstructed as models to a signature that contains only one program variable — the variable the model has access to — and their product can be reconstructed as a corresponding pullback.

According to the definition of Grothendieck morphisms first an appropriate pushout diagram of transformation signatures has to be given:

$$\begin{tikzcd}
\mathbf{prog\_0} \arrow[r] \arrow[d] & \mathbf{prog} \arrow[d] \\
\mathbf{prog} \arrow[r] & \mathbf{p,q-prog}
\end{tikzcd}$$

Here **prog** is defined as in example 4, introducing the only program variable and the incrementation method. **prog_0** is **prog** without the program variable p, and the morphisms are inclusions. Then the transformation signature **p,q–prog** of example 8, together with the inclusion morphisms, is a pushout of this diagram. There are two copies of the program variable, because it is not shared in **prog_0**.

Now let $\mathcal{X}'$ be given by the **prog**–transformation system $\mathcal{X}$ defined in example 4, extended by idle transitions as in example 8. Obviously $\mathcal{X}'$ coincides with the restrictions to **prog** of $\mathcal{Y}$ and $\mathcal{Z}$. The image $V(\mathcal{X}')$ under the forgetful functor V induced by the inclusion **prog_0** $\rightarrow$ **prog** is the same as $\mathcal{X}'$, except that in the data states $V(X_n)$ the program variable X is no longer designated by the constant p.

In order to obtain the product of the transition graph of $\mathcal{X}'$ with itself as a pullback, it must be connected by graph morphisms to the terminal graph, that consists of one node and one edge.

Finally the sharing of the data states must be expressed by appropriate amalgamations. The natural numbers shall be shared, whereas the sets of program variables and the label sets of method expressions shall be united disjointly. All this is obtained by the **prog_0**–transformation system $\mathbb{1}$, given by the one node–one edge–transition graph, data state $\mathbb{N}$ with empty set of program variables, and empty set of method expressions for the transition. Then there is a Grothendieck morphism $\mathcal{X}' \rightarrow \mathbb{1}$ given by the transition signature inclusion **prog_0** $\rightarrow$ **prog** , the unique morphism of transition graphs $TG_{\mathcal{X}'} \rightarrow TG_{\mathbb{1}}$, data state injections $(\mathbb{N}, \emptyset) \rightarrow V(X_n)$ for all $n \in \mathbb{N}$, and inclusions of sets of method expressions $\emptyset \rightarrow \{inc(X,k)\}$.

Since $TG_{\mathbb{1}}$ is a terminal graph the pullback of $TG_{\mathcal{X}'} \rightarrow TG_{\mathbb{1}}$ with itself is the product $TG_{\mathcal{X}'} \times TG_{\mathcal{X}'}$. Data states are the amalgamations $X_n X'_m :=$ $X_n +_{(\mathbb{N},\emptyset)} X_m$, that contain two copies of X, i.e. $p^{X_n X'_m} = X$ and $q^{X_n X'_m} = X'$, with values $!^{X_n X'_m}(X) = n$ and $!^{X_n X'_m}(X') = m$ respectively. The sets of method expressions are given by the disjoint unions of the corresponding sets of method expressions of the components. Thus the pullback of $\mathcal{X}' \rightarrow \mathbb{1}$ with itself is the desired parallel composition of $\mathcal{X}'$ with a copy of itself, that manipulates the copy q of p .

5 Conclusion

In this paper I have introduced the two layered structures of algebra transformation systems, their morphisms and their composition by limits, colimits, and signature morphisms. This framework belongs to the *algebras–as–states* approach to the specification of dynamic systems, whose foremost representative are the abstract state machines, formerly called evolving algebras (see [Gur94]). A formalisation of evolving algebras has been presented in [DG94], where however algebraic specifications are considered as algebraic programs, and consistency conditions become part of the definition of the semantics. A very general abstract mathematical model within this approach, D–oids, has been presented in [AZ95]. It introduces a model theory for dynamic systems, parameterized by the underlying static framework for values and state algebras. Specification means, i.e. sentences and satisfaction for D–oids are introduced in [Zuc96]. There, however, methods are total functions, which is problematic for the modelling of non–deterministic systems, and identities are modelled by a tracking map, which might be in conflict with the data signature. However, there are no composition operations for D–oids, and the technique developed here cannot be applied directly, because signatures are used in a very different way. The idea to use signature morphisms to compose specifications of concurrent systems has been adapted from [FM92]. There temporal logic theories are introduced as specification units and specification morphisms as interconnections. Due to the temporal logic approach one temporal structure for all models had to be fixed, in this case discrete linear time, as opposed to the arbitrary transition graphs of algebra transformation systems.

Labelled transition systems can be embedded into the framework of algebra transformation systems, taking the transition system as transition graph, empty data states (over the empty signature), and the labels as method expressions. Models of Z–specifications and graph transformation systems can be embedded as the other extreme case, where the control states do not contain additional information. I.e. the transition graph is the graph of all reachable data states (= data models in Z, supposed they are (first order) partial algebras, graphs in graph transformation systems). In Z the signature of the data states is given explicitly, signatures for graphs would introduce sorts for nodes and edges, and functions *src* and *tar* for sources and targets of edges. More elaborated graph structures can be defined accordingly.

Partial algebras have been chosen as specification framework for the data states for the reasons discussed above. However, it is easy to see that the approach is (rather) institution independent concerning the data state models. The only requirement used in this paper has been that data state models have carrier sets from which the parameters can be chosen, i.e. the model theory is *concrete* (see [BT96]), and that the model categories have limits and colimits. In this way *institution independent transformation systems* can be defined.

What is left open in this paper are the development of a syntax to represent or specify the transition graphs, and axioms for the description of transformation systems, i.e. the logical part of the institution. First results concerning such axioms are presented in [Gro96], where the *descriptive* and the *constructive* meanings of *replacement rules* for (a class of) partial algebras are investigated. The constructive interpretation of a rule describes how a successor state can be constructed from a given state and parameters, its descriptive meaning is a pre/post condition for a method.

The first point is left open because process languages can be used to present the transition graphs, or regular expressions for instance. A more detailed investigation however would have to take into account also the possible mutual relationships between control states and data states. There should be means for instance to state that a method can (cannot) be applied if the data state satisfies a certain condition, like being *stable* for instance. (A state is stable if all admissible method applications yield isomorphic states.) Furthermore a *diagram language* should be developed that allows to specify diagrams of connected components. Ideally such a language should also support dynamic evolution of diagrams, i.e. creation, deletion, and reconfiguration of components. Some ideas concerning static diagram languages have been presented in [Fia97].

References

[AZ95] E. Astesiano and E. Zucca. D-oids: A model for dynamic data types. *Math. Struct. in Comp. Sci.*, 5(2):257–282, 1995.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. Int. Conf. Artificial Intelligence*, 1977.

[BT96] M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. In *Proc. CAAP'96*, Springer LNCS 1059. 1996.

[CGW95] I. Claßen, M. Große-Rhode, and U. Wolter. Categorical concepts for parameterized partial specifications. *Math. Struct. in Comp. Science*, 5(2):153–188, 1995.

[DG94] P. Dauchy and M.C. Gaudel. Algebraic specifications with implicit states. *Tech. Report, Univ. Paris Sud*, 1994.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[FM92] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specifications. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[Fia97] J.L. Fiadeiro. Algebraic semantics of coordination. Talk given at the 12th Workshop on Algebraic Development Techniques, Tarquinia, Italy, 1997.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journals of the ACM*, 39(1):95–146, January 1992.

[Gro96] M. Große-Rhode. First steps towards an institution of algebra replacement systems. Technical Report 96-44, Technische Universität Berlin, 1996. Also available under http://tfs.cs.tu-berlin.de/˜ mgr.

[Gro97] M. Große-Rhode. Sequential and parallel algebra transformation systems and their composition. Technical Report 97–07, Università di Roma *La Sapienza*, Dip. Scienze dell'Informazione, 1997. Also available under http://tfs.cs.tu-berlin.de/˜ mgr.

[Gur94] Y. Gurevich. Evolving algebra 1993. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[ODP] ISO/IEC International Standard 10746, ITU–T Recommendation X.901–X.904: Reference model of open distributed processing – Parts 1–4.

[Rei87] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford University Press, Oxford, 1987.

[WN95] G. Winskel and M. Nielson. Models for concurrency. In *Handbook of Logic in Computer Science*. Oxford University Press, 1995.

[Zuc96] E. Zucca. From static to dynamic abstract data–types. In W.Penczek and A. Szalas, editors, *Mathematical Foundations of Computer Science 1996*, volume 1113 of *Lecture Notes in Computer Science*, pages 579–590. Springer Verlag, 1996.