V. Claus, H. Ehrig, and G. Rozenberg, editors, Proc. Int. Colloquium on Graph Grammars, LNCS 73, pages 180–191, Berlin, 1978. Springer–Verlag

CONSTRUCTING SPECIFICATIONS OF ABSTRACT DATA TYPES BY REPLACEMENTS

H.-D. Ehrich / V.G. Lohberger

Abteilung Informatik, Universität Dortmund
Postfach 5oo5oo, 46oo Dortmund 5o, West Germany

Abstract - Categories of specifications, equational specifications, and partially labelled partial specifications of abstract data types are shown to have pushouts. These results allow us to carry over the machinery of graph replacement to specifications. We give some examples. Parametrization is considered as an important special case of replacement.

1. Introduction

In the development of programs and program systems, the initial specification phase is of increasing importance. It is essential to have a clean, unique, complete, and implementation independent description of what the system is intended to do in order to cope with many problems of reliability. Such a specification is not only necessary as a documentation and communication basis for the programmer team. Also, logical errors can be detected and debugged in an early state, seperate from implementation errors occuring later in the implementation process. Moreover, implementations or implementation steps can be matched against the requirements of the specification, thus controlling the correctness of the program development. The role of specification and the related concept of modularity has been studied by Parnas [19] , Liskov and Zilles [16] and others.

There is a great need for formal methods to support these tasks. Program modules have been modelled by abstract data types [16, 13] , i.e. by sets of operations on various domains interrelated in a certain way. Mathematically speaking, abstract data types are abstract algebraic structures. To specify the desired properties of a program module means to specify a class of abstract data types, and this means to give a presentation of a class of algebraic structures.

There are well approved methods in algebra and logic to give presentations and investigate the structure of their models. Equational presentations and corresponding classes of algebras are especially well understood [1, 2, 4, 5, 6, 9, 11, 12, 18] . Specifications consist of a set of sorts, a set of operation symbols with information about their domains and codomains, a set of predicates with information about their domains, and a collection of conditions or axioms.

We give some introductory examples that may serve to illustrate the basic ideas. In our examples, we use a somewhat ad hoc notation which can be viewed to be an informal algebraic specification language.

Example 1.1: The natural numbers with their ordering relation can be specified as follows.

Sorts:	nat	Preds:	$\leq$: nat $\times$ nat
Ops:	0: $\longrightarrow$ nat	Conds:	$x \leq$ succ (x)
	succ: nat $\longrightarrow$ nat		$x \leq x$
			$x \leq y \wedge y \leq z \Rightarrow x \leq z$
			$x \leq y \wedge y \leq x \Rightarrow x = y$
			$x \leq y \vee y \leq x$

Example 1.2: The interval 1 to 10 of natural numbers will be used in the next example.

Sorts: $[1:10]$

Ops: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10: \to [1:10]$

$\quad$ suc : $[1:10] \to [1:10]$

Preds: $\lesssim$: $[1:10] \times [1:10]$

Conds: $1 \lesssim 2,\ 2 \lesssim 3, \ldots, 9 \lesssim 10,$ $\quad x \lesssim y \wedge y \lesssim x \Longrightarrow x = y$

$\quad x \lesssim x$ $\quad x \lesssim y \vee y \lesssim x$

$\quad x \lesssim y \wedge y \lesssim z \Longrightarrow x \lesssim z$ $\quad$ suc $(1)=2, \ldots,$ suc$(9) = 10$

Example 1.3: We give a specification of an array with the components consisting of the sorts, ops, preds, and conds of the previous examples plus the following:

Sorts: array

Ops: new: $\longrightarrow$ array

$\quad .[.] :=. \ :$ array $\times [1:10] \times$ nat $\longrightarrow$ array

$\quad .[.] \ :$ array $\times [1:10] \longrightarrow$ nat

Conds: new $[i] = 0 / (a\ [i] := n)\ [j] =$ if i eq j then n else a $[j]$ fi

Intuitively speaking, a $[i]$:= n assigns value n to the i-th component of array a, while a[i] denotes the value stored in that component.

Example 1.4.: We extend the previous example by an operation sorting a given array. The following operations and conditions are added to those of the previous example.

Ops: sort: array $\longrightarrow$ array

Conds: $i \lesssim j \Rightarrow \mathrm{sort}(a)\ [i] \leq \mathrm{sort}(a)\ [j]$

$\quad (\exists P)\ [(\forall i)\ a[P(i)] = \mathrm{sort}(a)\ [i]$

$\quad\quad \wedge (P(i) = P(j) \Rightarrow i = j) \wedge (\forall i)\ (\exists j)\ P(j) = i]$

The first condition expresses what it means for an array to be sorted, and the second condition expresses that the contents of the array may not be changed but only permuted. If we would like to restrict ourselves to equational specifications where the conditions are just sets of equations, we could do so by viewing predicates π : x as operations π : x $\longrightarrow$ bool, where bool has two constants, true and false, equipped with appropriate boolean operations. The first three examples are easily rewritten in equational form, but there seems to be no way to express the idea of sorting by equations as conveniently as example 1.4 does.

One possivility is to use an auxiliary operation

$\quad$ sort1 : array $\times [1:10] \longrightarrow$ array

and express the algorithm of bubble sort by the following equations:

```
sort(a) = sort1(a,1)
sort1(a,i) = if i =10 then a else
          if a [i] ≤ a[suc(i)] then sort1(a,suc(i)) else sort1(a',1) fi fi
```

where a' = (a[suc(i)] := a[i]) [i] := a[suc(i)].

We feel that example 1.4 gives a more adequate and easier to understand description of what sorting means.

Although the expressive power of equational specification is principally sufficient for all practical cases [2, 17], convenience requires more comfortable specification language. In the present paper, we therefore use second-order predicate calculus.

With increasing complexitiy of program systems, the design process of specifications must be given more and more attention. A structured and modular approach to specification design requires means to manipulate pieces of specifications, put them together, and consistently replace parts of them. For example, it is very convenient to give specifications with formal parts, socalled parametric specifications, where the formal parts can be replaced by different actual specifications [6, 7]. Other types of replacement operations occur when specifications are to be modified, e.g. to remove errors or to adapt the system to changed user needs [7].

In the field of graph grammars, methods and tools for handling replacement operations on graphs have been successfully developed and applied to various situations. The purpose of this paper is to demonstrate that these ideas can be carried over to tackle some problems of specification design.

In the categorical approach to graph grammars, the mechanisms of graph rewriting have been formalized by means of pushouts in the category graph of graphs [10, 20, 21]. It has been realized that these mechanisms can be applied to any structures forming a category with pushouts [8]. Therefore, we investigate the existence of pushouts in various related categories of specifications. The usefulness of rewritings on these specifications is demonstrated by examples. The techniques and results carry over to partially labelled partial specifications, which suit better to applications. Here, we refer to [21].

In a certain sense, specifications can be viewed as graphs enriched by conditions Conversely, if we forget about the conditions, we get the socalled syntax graph of a specification. Thus, there are forgetful functors from specification categories to corresponding graph categories. It is shown that these functors respect pushouts. Therefore, specification rewritings effect corresponding graph rewritings on the syntax graphs.

Parametric specifications can be considered as special cases of rewriting rules, giving the rules how the formal parameter is substituted by the actual parameter. We illustrate by an example how parametric specifications and corresponding specification productions can be used in the editing process of specifications.

2. Categories of specifications and their graphs

Let S be a set of sorts. S^* and S^+ denote the sets of words resp. nonempty words over S. Elements $x \in S^*$ will be called sorts, too. A signature over S is a mapping $\Omega : \bar{\Omega} \to S$. Mappings into a sort set S are called S-sorted sets or simply sorted sets, if the sort set is clear from the context. Thus, a signature Ω is an S^+-sorted set, and its elements are called operations. If $\omega \mapsto xs$ is in Ω, $x \in S^*$, $s \in S$, we call x the domain sort and s the codomain sort of $\omega \mapsto xs$ (or just of ω). A codomain sort always has word length 1. Ω determines two mappings: ${}^\circ\Omega : \omega \mapsto x$ and $\Omega^\circ : \omega \mapsto s$ for each $\omega \longmapsto xs$ in Ω, the domain resp. codomain mappings.

We assume that the reader is familiar with the category graph of graphs [10, 20, 21]. A signature Ω will sometimes be considered to be that graph with nodes S^*, edges $\bar{\Omega}$, source function ${}^\circ\Omega$, and target function Ω°. Signature morphisms $f : \Omega_1 \to \Omega_2$ are those graph morphismus $f = (h,g)$ $h : S_1^* \to S_2^*$, $g : \bar{\Omega}_1 \to \bar{\Omega}_2$, where h is a length preserving string homomorphism. Thus, h is completely determined by its restriction to S_1 and S_2, also denoted by $h : S_1 \to S_2$.
If $\Omega_i : \bar{\Omega}_i \longrightarrow S_i^+$, $i = 1,2$, the morphism condition is equivalent to $\Omega_1 h = g\Omega_2$ (we write function composition and application from left to right, e.g. xf and xfg instead of the more conventional f(x) and gf(x).) Let sign be the subcategory of graph consisting of all signatures and its morphisms. The following result carries over form graph.

Theorem 2.1: sign has pushouts.

Proof: We do not give the complete proof since it parallels that in the graph case [10, 20, 21]. For later reference, we give the pushout construction in sign.

Let $f_1: \Omega_1 \longrightarrow \Omega_2$, $f_2: \Omega_1 \longrightarrow \Omega_3$ be morphisms in sign. We construct Ω_4, $f_3: \Omega_2 \rightarrow \Omega_4$, $f_4: \Omega_3 \rightarrow \Omega_4$ such that fig.1(1) is a pushout: let $\Omega_i: \bar{\Omega}_i \rightarrow S_i^+$ and $f_i=(h_i, g_i)$ for

$$\begin{tikzcd}
\Omega_1 \arrow[r, "f_1"] \arrow[d, "f_2"'] \arrow[dr, phantom, "(1)"] & \Omega_2 \arrow[d, "f_3"] & S_1 \arrow[r, "h_1"] \arrow[d, "h_2"'] \arrow[dr, phantom, "(2)"] & S_2 \arrow[d, "h_3"] & \bar{\Omega}_1 \arrow[r, "g_1"] \arrow[d, "g_2"'] \arrow[dr, phantom, "(3)"] & \bar{\Omega}_2 \arrow[d, "g_3"] \\
\Omega_3 \arrow[r, "f_4"] & \Omega_4 & S_3 \arrow[r, "h_4"] & S_4 & \bar{\Omega}_3 \arrow[r, "g_4"] & \bar{\Omega}_4
\end{tikzcd}$$

figure 1

i=1, 2, 3, 4, and let h_3, h_4 and g_3, g_4 be given such that the diagrams in fig.1(2) and (3) are pushouts in the category set of sets. Due to the definition of morphisms in sign, the diagram in fig.2 is commutative. The upper quadrangle coincides with fig1(3), i.e. it is a pushout. Thus, there is a unique mapping $\Omega_4: \bar{\Omega}_4 \dashrightarrow S_4^+$ making the whole diagram in fig.2 commutative. ///

$$\begin{tikzcd}
 & \bar{\Omega}_2 \arrow[rr, "g_3"] \arrow[dd, "\Omega_2"' near start] & & \bar{\Omega}_4 \arrow[dd, dashed, bend left=40, "\Omega_4"] \\
\bar{\Omega}_1 \arrow[ur, "g_1"] \arrow[rr, "g_2" near end] \arrow[dd, "\Omega_1"'] & & \bar{\Omega}_3 \arrow[ur, "g_4"'] \arrow[dd, "\Omega_3"' near start] & \\
 & S_2^+ \arrow[rr, "h_3" near start] & & S_4^+ \\
S_1^+ \arrow[ur, "h_1"] \arrow[rr, "h_2"] & & S_3^+ \arrow[ur, "h_4"'] &
\end{tikzcd}$$

figure 2

Now we continue to develop our specification language. Given a sort set S, let $\Pi: \bar{\Pi} \rightarrow S^*$ be an S^*-sorted set of predicates. If $\pi \mapsto x$ is in Π, x is called the domain of the predicate. We suppose that an S-sorted set $V_i: \bar{V}_i \rightarrow S$ of individual variables is given, together with an S^*-sorted set $V_\pi: \bar{V}_\pi \longrightarrow S^*$ of predicate variables and an S^+-sorted set $V_\omega: \bar{V}_\omega \longrightarrow S^+$ of operation variables. Let $V=V_i \cup V_\pi \cup V_\omega$.

Let a sort set S, a signature Ω, predicates Π and variables V_i, V_π, V_ω over S be given, and let all these sets be disjoint. Terms and formulae are now constructed as usual in sorted second-order predicate calculus with equality [14]. We shortly introduce our notation.

The S^*-sorted set T of terms (including term n-tuples) is given as follows:

Definition 2.2: $v \mapsto s \in V_i \succ \langle v \rangle \mapsto s \in T$

$\langle \tau_1 \rangle \mapsto x\ ,\ \langle \tau_2 \rangle \mapsto y \in T \succ \langle \tau_1, \tau_2 \rangle \mapsto xy \in T$

$\omega \mapsto xs \in \Omega\ ,\ \langle \tau \rangle \mapsto x \in T \succ \langle [\langle \tau \rangle \mapsto x]\omega \rangle \mapsto s \in T$

$F \mapsto xs \in V_\omega\ ,\ \langle \tau \rangle \mapsto x \in T \succ \langle [\langle \tau \rangle \mapsto x] F \rangle \mapsto s \in T$

$\varphi \in \Lambda, \langle \tau_1 \rangle \mapsto x,\ \langle \tau_2 \rangle \mapsto x \in T \succ$

$\langle \text{ if } \varphi \text{ then } \langle \tau_1 \rangle \mapsto x \text{ else } \langle \tau_2 \rangle \mapsto x \text{ fi } \rangle \mapsto x \in T$

Here, Λ is the set of formulae defined as follows:

Definition 2.3: true, false $\in \Lambda$

$\langle \tau_1 \rangle \mapsto x,\ \langle \tau_2 \rangle \mapsto x \in T \succ \tau_1 = \tau_2 \in \Lambda$

$p \mapsto x \in \Pi, \langle \tau \rangle \mapsto x \in T \succ p(\tau) \in \Lambda$

$P \mapsto x \in V_\pi\ , \langle \tau \rangle \mapsto x \in T \succ P(\tau) \in \Lambda$

$\varphi_1, \varphi_2, \varphi_3 \in \Lambda \succ$ IF φ_1 THEN φ_2 ELSE φ_3 FI $\in \Lambda$

$\varphi \in \Lambda,\ (v \mapsto x) \in V \succ ((\forall v)\varphi) \in \Lambda$

As usual we will use some deviations from the strong syntactical rules of notation in order to increase readability. Especially, we will omit the sort part $\mapsto x$ whenever it is clear from the context. Furthermore, we will use the notational abbreviations $\neg\varphi$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, etc. for the conventional boolean operations that are easily expressible by IF-THEN-ELSE-FI. Another conventional notation is $((\exists v)\varphi)$ for $\neg((\forall v)\neg\varphi)$.

The set Λ of formulae is called the (object) language of S, Ω , Π . We write Λ(S, Ω , Π) in order to express the underlying items explicitely. The variable sets are assumed to be fixed in the sequel. If we take only the first two lines of definition 2.3, we get a subset of Λ, called the equational language of S,Ω and denoted by H(S, Ω). We will not explain the semantics of these languages in detail but adopt the usual conventions (see, e.g.[14] or any text book on logic).

Now we have the tools to give a precise definition of specification.

Definition 2.4: A specification is a quadruple $D = \langle S, \Omega, \Pi, C \rangle$ where S is a set of sorts, Ω is a signature over S, Π is an S^*-sorted set of predicates, and $C \subset \Lambda(S, \Omega, \Pi)$ is a set of conditions. The specification is called equational iff $C \subset H(S, \Omega)$.

In equational specifications, Π is empty. Therefore we write $D = \langle S, \Omega, C \rangle$ if we have an equational specification. The signature Ω - viewed as a graph - is called the graph of the specification.

Definition 2.5: A model (or interpretation) of D is given by

(1) an S^*-sorted set A: $\bar{A} \to S^*$ with the properties

(1.1) $xA^{-1} \neq \emptyset$

(1.2) $(xy)A^{-1} = xA^{-1} \times yA^{-1}$

} for all $x,y \in S^*$

(2) an assignment of a function $\bar{\omega}: xA^{-1} \to sA^{-1}$ to each operation $\omega \mapsto xs \in \Omega$

(3) an assignment of a relation $\bar{\pi} \subset xA^{-1}$ to each predicate $\pi \mapsto x \in \Pi$.
Equality must be interpreted by a congruence relation on A.

(4) C is satisfied (i.e. each formula in C evaluates to true when assigning

arbitrary values of appropriate sorts to the free variables).

Definition 2.6: Let $D=\langle S,\Omega,\Pi,C\rangle$ be a specification, and let B_1, $B_2 \subset \Lambda := \Lambda(S,\Omega,\Pi)$.
$B_1 \models_\Lambda B_2$ means that B_2 is a logical consequence of B_1, i.e. B_2 is satisfied in each model of $\langle\Lambda, B_1\rangle$. We shortly write $D \models B$ instead of $C \models_\Lambda B$.

Specifications can be related by specification morphisms in a natural way. Let $D_i=\langle S_i, \Omega_i, \Pi_i, C_i\rangle$, i=1, 2, be specifications, $f'=(h, g_\omega): \Omega_1 \rightarrow \Omega_2$ be a signature morphism, and $g_\pi : \overline{\overline{\Pi}}_1 \rightarrow \overline{\overline{\Pi}}_2$ be a mapping such that $\Pi_1 h = g_\pi \Pi_2$. The triple $f=(h,g_\omega,g_\pi)$ can then be extended to a mapping

$$\hat{f}: \Lambda_1 \rightarrow \Lambda_2 ,$$

where $\Lambda_i=\Lambda(S_i,\Omega_i,\Pi_i)$, i=1, 2, by replacing each occurrence of a sort x by xh, of an operation symbol ω by ωg_ω, and of a predicate symbol π by πg_π. (Without restricting generality, we assume that there are variables $(v \mapsto xh)$ for each variable $(v \mapsto x)$.)

Definition 2.7: A specification morphism $f:D_1 \rightarrow D_2$ is a triple $f=(h,g_\omega,g_\pi)$ with the properties
(1) $f'=(h,g_\omega)$ is a signature morphism, $f': \Omega_1 \rightarrow \Omega_2$
(2) $g_\pi : \overline{\overline{\Pi}}_1 \rightarrow \overline{\overline{\Pi}}_2$ satisfies $g_\pi \Pi_2 = \Pi_1 h$
(3) for each formula $B \in \Lambda_1$, if $D_1 \models B$, then $D_2 \models B\hat{f}$.

Composition of specification morphisms is defined by composition of the constituent mappings, separately for each component. It is easily checked that the criteria of a category are satisfied.

Definition 2.8: The category of specifications is denoted by spec. By $\text{spec}^=$ we denote the full subcategory of equational specifications.

We are now ready to prove our main result: spec and $\text{spec}^=$ inherit the existence of pushouts from their signatures sign. Let Γ, $\Gamma^=$ be the forgetful functors Γ:spec$\rightarrow$sign resp. $\Gamma^=$: $\text{spec}^= \rightarrow$sign sending each specification to its signature (considered as a graph).

Theorem 2.9: spec and $\text{spec}^=$ have pushouts. Γ and $\Gamma^=$ respect pushouts.
Proof: Let $f_1:D_1 \rightarrow D_2$ and $f_2:D_1 \rightarrow D_3$ be specification morphisms in spec. We construct D_4, $f_3:D_2 \rightarrow D_4$ and $f_4:D_3 \rightarrow D_4$ as follows (cf. figures 1 and 2).

Let $f_i=(h_i,g_i^\omega, g_i^\pi)$, $f_i' =(h_i,g_i^\omega)$, $D_i=\langle S_i, \Omega_i, \Pi_i, C_i\rangle$, i=1, 2, 3, 4.
We define Ω_4, f_3', f_4' to be the pushout of f_1' and f_2' in sign (cf. theorem 2.1) and Π_4', g_3^π, g_4^π to be the pushout of g_1^π, g_2^π in set. $\Pi_4: \Pi_4' \rightarrow S_4^*$ is then obtained in the same unique way as Ω_4 in the proof of theorem 2.1 (cf. fig.2). Conditions C_4 are defined as follows: $C_4=\{B\hat{f}_3 \mid D_2 \models B\} \cup \{B\hat{f}_4 \mid D_3 \models B\}$ (or any set of formulae that is logically equivalent).

We claim that D_4, f_3, f_4 constructed this way form a pushout of f_1, f_2 in spec. In order to prove this, let D_5, $f_5:D_2 \rightarrow D_5$, $f_6:D_3 \rightarrow D_5$ be such that $f_1f_5=f_2f_6$. It follows that $f_1'f_5'=f_2'f_6'$ and $g_1^\pi g_5^\pi=g_2^\pi g_6^\pi$, and thus there is exactly one $f_7': \Omega_4 \rightarrow \Omega_5$

and exactly one $g_7^\pi : \bar{\Pi}_4 \longrightarrow \bar{\Pi}_7$ such that $f_3'f_7'=f_5'$, $f_4'f_7'=f_6'$ resp. $g_3^\pi g_7^\pi = g_5^\pi$, $g_4^\pi g_7^\pi = g_6^\pi$. Therefore there is at most one morphism $f_7 : D_4 \longrightarrow D_5$ in <u>spec</u> satisfying $f_3f_7=f_5$ and $f_4f_7=f_6$, namely $f_7=(f_7', g_7^\pi)$. That f_7 is in fact a morphism in <u>spec</u> follows from the definition of C_4: if $D_4 \models B$, we have $B=B'\hat{f}_3$ and $D_2 \models B'$ or $B=B'\hat{f}_4$ and $D_3 \models B'$. In the first case, we have $D_5 \models B'\hat{f}_5$ since f_5 is a morphism. Obviously, $B'\hat{f}_5 = B'(\hat{f}_3\hat{f}_7) = (B'\hat{f}_3)\hat{f}_7 = B\hat{f}_7$. The second case is proven in the same way.

Pushouts in $\underline{\text{spec}}^{=}$ are constructed in the same way. That Γ and $\Gamma^{=}$ respect pushouts is clear from the definition. ///

If we consider other languages of logic, e.g. first-order predicate calculus, propositional calculus etc., similar results hold and can be proven in the same way. The main drawback of the above construction is the rather clumsy set of conditions C_4. It can be shown, however, that simplifications are possible: condition (3) of definition 2.7 can be replaced by $D_2 \models C_1\hat{f}$, and C_4 can then be defined in the above proof to be simply $C_4 = C_2\hat{f}_3 \cup C_3\hat{f}_4$. The verification of these propositions requires a little bit heavier machinery, and we cannot give the details here.

3. Labelled and partial specifications

To use specifications and specification productions in practical situations, as [3] do with graphs, it is desirable to generalize the tools developed so far, to include labels and partiality.

The symbols that we used for sorts and operations in specifications, until now, are only a notational means to identify the different items in the specifications. We see that the morphisms do not care about these notations. On the other hand we feel that these names, although sometimes called "syntactical sugar", play an essential part as a guide through large specifications, and we should provide for taking these names to be part of the structure and to be respected by morphisms. In conjunction with partiality, we can have the possibility to alter these names, but these alterations must be made explicit and are formally contained in the calculus and not mere arbitrary notation, as we shall learn from the examples. For these purposes we need a globally defined set of labels.

[21] give a motivation for the use of partial graphs which we accept also for specifications: if we want to describe syntactic operations by these formalisms, operations that replace only parts of graphs, it would in general not be a natural thing to add unnecessary information, e.g. context to make the occurring partial graphs or specifications total, and to increase at the same time the number of replacement rules, since there are many possibilities to make these graphs or specifications total. Instead, we should be able to express handling of partial specifications directly.

To give an example, let us think of the specification of our array with natural numbers as entries and a finite subset of natural numbers as keys. Imagine that the above array of nat specification is supplied with labels "array", "nat", "new", etc., such that we have a labelled specification with explicit labels. Now, we want to alter the specification by deleting nat and inserting int for nat. The names (labels) of all items not directly affected by this substitution shall be maintained. This can be expressed by the production of figure 3. It is obvious how 'p and p' shall map ('p: key$\mapsto$nat, s$\mapsto$succ, etc.; p' analogously). The labelling is expressed by the symbol $\underline{l}$.

Here, the gluing specification has no labels. That is why there are no labels to be respected by the morphisms 'p and p' and so the production can alter the labels as required. It is obvious how the specification of example 1.3 has to be supplemented by labels. On the other hand, the application of the production must care for saving the labels of the old specification which are not affected.

We need not emphasize the fact that there are possibilities to handle labelling implicitely when writing down a specification, thus simplifying the work of writing specifications in a practical situation. Here, the explicit handling suits to the formal treatment.

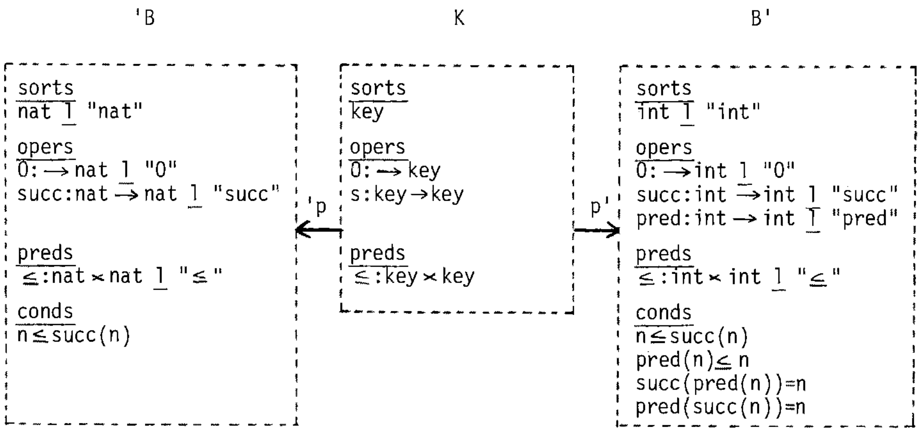

Figure 3: production to substitute nat by int

Guided by the definitions of total specifications in section 2, by our intuition (example), and definitions of partially labelled partial graphs [21], we give the following precise formulation:

Definition 3.1: The category spec-, called the category of partially (L_S, L_Ω, L_Π) -labelled specifications, has objects $D=\langle S, \Omega, \Pi, C, \lambda\rangle$, where S, Ω, Π, C are items as in the definition of spec, except of the requirement that $°\Omega$, $\Omega°$, need only be partial pappings, but those operations and predicates occurring in C must be total; and $\lambda=(\lambda_S, \lambda_\Omega, \lambda_\Pi)$ is a triple of three partial mappings $\lambda_S: S^*\to L_S$, $\lambda_\Omega:\Omega\to L_\Omega$, $\lambda_\Pi:\Pi\to L_\Pi$, and this category has morphisms $f:D_1\to D_2$, $f=(h, g_\omega, g_\pi)$ which obey the same laws as in spec but additionally the labelling of sorts, operations, and predicates is transported by morphisms if it is defined.

In [21], the respective morphisms for graphs are called "weak morphisms".

Theorem 3.2: spec- has pushouts.

proof: The argument is as with spec, although we must be careful with some of the arrows because of the partiality of the involved mappings. ///

In analogy to the discussion of the last section, the same result holds for the subcategories of spec- determined by a special calculus, as e.g. equational calculus or first order predicate calculus. A pushout diagram in such a specification category implies a pushout diagram in the underlying graph-category, i.e. we have a forgetful functor from the specification catagory to the graph category respecting pushouts.

We hint further that it is useful for practical situations to be sure that applying a production to a total specification yields a total specification, even if this production consists of partial specifications. [21] give sufficient criteria for this problem in the graph case, which carry over to the spec case.

We give an example in figure 4 to demonstrate that it is conveniently possible to use productions for relabelling of operations. Here, the partiality of operations provides the economy of specifying not more than required. This production can be applied to any specification where we find an analysis pushout. Obviously, all other labels of the analysed specification are saved via the pushout-complement.

$$\begin{tikzcd}
{'B} & K & B' \\
\boxed{\begin{array}{l}\underline{\text{ops}}\\ \text{succ: } . \longrightarrow . \ \underline{1}\ \text{"succ"}\end{array}} & \boxed{\begin{array}{l}\underline{\text{ops}}\\ \text{s: } . \longrightarrow .\end{array}} \arrow[l, "'p"'] \arrow[r, "p'"] & \boxed{\begin{array}{l}\underline{\text{ops}}\\ \text{N: } . \longrightarrow . \ \underline{1}\ \text{"N"}\end{array}}
\end{tikzcd}$$

Figure 4: production to relabel the successor operation

4. Parametric specifications

Parametric specifications are a very convenient means for the specification process. We do not care about semantical problems of parametric specifications cf.[15], but discuss a more syntactically oriented treatment following [7]. To give an example, the concept of an array is rather independent of the type of its entries and keys. The only requirement is that there should be an "eq"-relation on keys and a constant entry serving as the value of new(i). Thus we would like to give specifications as follows.

Example 4.1: The parametric specification of array(key, entry) is:

```
params  key, entry, eq, 0

sorts   array, key, entry

ops     0 : ⟶ entry
        new: ⟶ array
        .[.]:=. : array × key × entry ⟶ array
        .[.] : array × key ⟶ entry
        if.then.else.fi : bool × entry×entry ⟶ entry

preds   eq : key × key

conds   new[i] = 0
        (a[i]:=n)[j] = if i eq j then n else a[j] fi
```

We denote the formal parameter part by the params symbol. In spec-, an embedding will serve for this purpose. A parameter assignment should map formal parameters to actual parameters which suit in type and behaviour. So,. assignments are well modelled by morphisms. To apply a parametric specification to an actual parameter is intended to substitute the formal parameter specification by the actual parameter specification and this is done by the pushout construction.

Definition 4.2: A parametric specification p:F⟶P is an embedding in spec-. A parameter assignment for p:F⟶P is a morphism f:F⟶A. The result of the application of the parametric specification p:F⟶P to the actual parameter A via f:F⟶A is the pushout-object of the pushout of p and f.

In our examples with arrays, 1.3 and 4.1, we have:

$$\begin{tikzcd}
(\text{key, entry}) \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{p.o.}"] & \text{array (key, entry)} \arrow[d] \\
([1{:}10], \text{nat}) \arrow[r] & \text{array } ([1{:}10], \text{nat})
\end{tikzcd}$$

Not that the actual parameter A can itself be a parametric specification $q : E \longrightarrow A$ with a formal parameter part E. Thus, sequences of applications of parametric specifications can be constructed. The following result for spec = [6,7] carries over to spec and spec -:

Theorem 4.3: The application of parametric specifications to parametric specifications is associative.

When we have a parametric specification, we can use it as well as a production by duplicating the parameter. The idea is demonstrated by figures 5 und 6.

params	$d, \leq$
sorts	d
preds	$\leq : d \times d$
conds	$x \leq x$
	$x \leq y \wedge y \leq z \Longrightarrow x \leq z$
	$x \leq y \wedge y \leq x \Longrightarrow x = y$
	$x \leq y \vee y \leq x$

Figure 5: Parametric specification of a total ordering

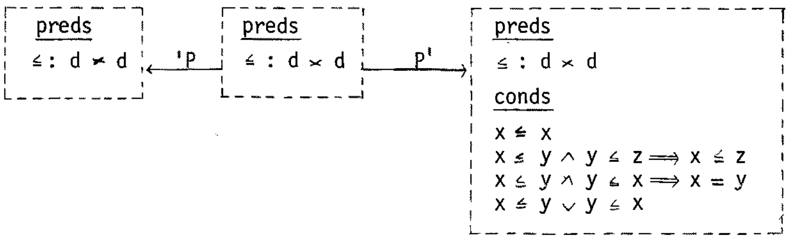

Figure 6: Production to generate total orderings

Taking parametric concepts, supplementing them to productions, and storing these productions in a library is a promising tool for editing specifications. Using the production of figure 6, when we want to specify any situation where a total ordering is involved, we need only give the few axioms that determine the specific ordering. For example, for natural numbers we need only give

sorts:	nat
ops:	$0 : \longrightarrow$ nat
	succ: nat $\longrightarrow$ nat
preds:	$\leq$: nat $\times$ nat
conds:	$x \leq$ succ(x)

Applying the production of figure 6 to this specification, the ordering axioms are inserted, and we get the specification of example 1.1.

We can imagine that for more complex concepts there may be a considerable economy in using parametric specifications as replacement rules or productions.

5. References

1. ADJ (Goguen, J.A.-Thatcher, J.W.-Wagner, E.G.):
   An initial algebra approach to the specification, correct ness, and implementation of abstract data types. Current Trends in Programming Methodology IV, ed. by R. Yeh, Prentice Hall, New Jersey 1977

2. ADJ (Thatcher, J.W.-Wagner, E.G.-Wright, J.B.):
   Data type specification, parametrization, and the power of specification technigues. Proc-Sigact Annual Symp. Theory Comp., 1978

3. Brendel, W.-Bunke, H.-Nagl, M.:
   Syntaxgesteurte Programmierung und inkrementelle Compilation. Proc. GI-7. Jahrestagung, Informatik Fachberichte 10, 57-73, Springer Berlin 1977

4. Burstall, R.M.-Goguen, J.A.:
   Putting theories together to make specifications. Proc. 5th IJCAI 77, MIT, Cambridge, Mass. 1977

5. Ehrich, H.-D.:
   Extensions and implementations of abstract data type specifications. Proc. MFCS'78, ed. by J. Winkowski, Lecture Notes in Computer Science, Vol. 64, Springer-Verlag, Berlin 1978, 155-164

6. Ehrich, H.-D.:
   On the theory of specification, implementation , and parametrization of abstract data types. To be published

7. Ehrich, H.-D.-Lohberger, V.G.:
   Parametric specification of abstract data types, parameter substitution, and graph replacements. Proc. Workshop "Graphentheoretische Konzepte in der Informatik", Applied Comp. SC., Carl Hanser Verl., Muenchen-Wien 1978

8. Ehrig, H.-Kreowski, H.-J.-Maggiolo-Schettini, A.-Rosen, B.K.- Winkowski, J.:
   Deriving structures from structures. Proc. MFCS 1978, ed. by J. Winkowski, Lecture Notes in Computer Science, Vol. 64. Springer-Verlag, Berlin 1978, 177-19o

9. Ehrig, H.-Kreowski, H.-J.-Padawitz, P.:
   Stepwise specification and implementation of abstract data types. Proc. 5th Intern Colloq. on Automata, Languages, and Programming, Venice 1978

1o. Ehrig, H.-Pfender, M.-Schneider, H.-J.:
   Graph-Grammars-an algebraic approach. Proc. Conf. Switch. Automata Theory 1973, 167-18o

11. Goguen, J.A.:
   Correctness and eqivalence of data types. In: Proc. Conf. on Alg. Syst. Th., Udine, Lecture Notes In Comp. SC., Springer-Verl., Berlin 1975

12. Goguen, J.A.:
   Abstract errors for abstract data types. In: Proc. Conf. on Formal Description of Programming Languages, Ed. by E.J. Neuhold, North-Holland Publ. Company, Amsterdam 1976

13. Guttag, J.V.:
The specification and application to programming of abstract data types. Techn. Report CSRG-59, Univ. of Toronto 1975

14. Kreisel, G.-Krivine, J.L.:
Modelltheorie, Springer, Berlin 1972

15. Lehmann , D.-J.-Smyth, M.B.:
Data types. Proc. 18th IEEE Symp. on Foundations of Computing. Providence R.I., 1977, 7-12

16. Liskov, B.-H.-Zilles, S.N.:
Specification Tehcniques for data abstractions. IEEE Transact. Softw.-Eng., Vol. SE-1 (1975), 7-19

17. Majster, M-E.:
Data types, abstract data types and their specification problem. Report TUM-INFO-774o, Techn. Univ. Muenchen, 1977

18. Manes, E.G.:
Algebraic Theories. Springer-Verl., New York 1976

19. Parnas, D.L.:
A technique for module specification with examples. Comm. ACM 15 (1972) 33o-336

2o. Rosen, B.K.:
Deriving Graphs from Graphs by applying a production. Acta Informatica 4, 337-357 (1975)

21. Schneider, H.J.-Ehrig, H.:
Grammars on partial graphs. Acta Inf. 6, 297-316 (1976)