SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

Hierarchical structures and dynamic parameterization without parameters

Christoph Beierle, Angelika Voß

August 1986 SEKI-REPORT SR-86-13

Hierarchical structures and dynamic parameterization without parameters

Christoph Beierle, Angelika Voß
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 6750 Kaiserslautern, West Germany
UUCP: ..!mcvax!unido!uklirb!beierle

Abstract

Hierarchical structuring and parameterization concepts are investigated. It is argued that such concepts can be studied independently of a particular domain of application and orthogonally to the design of the `flat' objects. A dynamic parameterization concept which disposes of the static declaration of formal parameters is proposed and realized in a hierarchy definition language. The methods suggested are illustrated by applying them to algebraic specifications, and it is shown how the approach extends the notion of an institution by a general structuring and parameterization concept.

Contents

0. Introduction

The advantages of a structured and parameterizable design are undoubted ever since the invention of structured programming: It is perspicicuous, reusable, economic, flexible, easy to modify and maintain. Hierarchical structures are the most general acyclic structures and therefore can be interpreted without any fixpoint semantics and in unordered domains. Hierarchical structures are wide spread. They arise wherever some kind of subpart relation prevails: in modules and submodules, in types and subtypes, in objects and components, in specifications and subspecifications, in problems and subproblems, etc..

Since a hierarchical structure naturally identifies its subhierarchies, it suggests a particular, highly flexible parameterization concept where every subhierarchy may be regarded as a formal parameter to be identified dynamically at instantiation time. It may be called dynamic parameterization, because the static declaration of formal parameters is disposed of.

The idea of dynamic parameterization also occurs in the algebraic specification language Look [ZLT 82], where it is called adaption. In the specification language Ordinary [Go 81], the concept is implicitly available by the "combine ... and ... over ..." construct. The high-level logic based programming language OBJ [GMP 82] provides a limited form of dynamic parameterization but without a formal semantics.

In this paper it is argued that structuring and parameterization mechanisms can be studied orthogonally to the `flat' object domain they are applied to. We investigate these concepts independently of a particular domain and demonstrate its applicability to different areas. Since in particular all institutions in the semse of [GB 83] define suitable `flat' domains we show that our approach extends the notion of an institution by a structuring and parameterization concept. Py using e.g. the methods of [ST 84] structured and parameterized specification over arbitrary institutions may be built.

In our examples we use an institution of loose algebraic specifications which is briefly sketched in Section 1. In Section 2 we introduce our dynamic parameterization concept which renders superfluous any static distinction between parameter, parameterized, and non-parameterized objects. In Section 3 we describe the concept of hierarchical structures as documentation of construction processes. In Section 4 we show how hierarchical structures and dynamic parameterization can be neatly combined. It is demonstrated that the resulting parameterization-by-use concept for hierarchical structures avoids some problems

encountered otherwise in structured specification languages. Section 5 summarizes the essential ideas of the formal development which is carried out in detail in [BV 85]. In Section 6 we describe a general hierarchy definition language supporting dynamic parameterization, and draw some conclusions.

1. Institutions

The concept of an institution was first introduced by Burstall and Goguen in [BG 80] as a "language" and was studied in detail in their paper [GB 83]; a brief introduction is given in e.g. [ST 84].

In the examples of this paper we will assume an institution defining a category of theories denoted by SPEC. The objects of SPEC are (loose) algebraic specifications which are connected by specifications morphisms. The signature of SPEC objects are usual equational signatures (pairs of sorts and operation symbols), but we do not make any specific assumptions about the types of sentences. However, since we have loose specifications there will be some kind of constraint mechanism to exclude unreachable elements (e.g. initial [HKR 80], data [BG 80], hierarchy [SW 82], or algorithmic constraints [BV 85]). We assume to have specifications BOOL and NAT for the booleans and the natural numbers, a loose specification ELEM denoting arbitrary carriers, and specifications LIST and LIMITED-STACK for standard lists and stacks over ELEM, the size of the stacks being limited by some constant but arbitrary natural number introduced in the specification LIMIT.

2. Dynamic parameterization

In programming languages as well as in specification languages the parameterization concepts usually involve two basic steps:

- Declaring formal parameters when defining a parameterized object.
- Giving a correspondence between formal and actual parameters when instantiating a parameterized object.

In general, the following points must be observed and may be regarded as drawbacks in some applications:

1. When defining a parameterized object P one has to determine the complete set F of P's formal parameters. If one realizes later on that some additional parts of P could be regarded as formal parameters and one wants to substitute them by other objects this is impossible without rewriting P and extending its parameter declaration.

2. To instantiate P an actual parameter must be supplied for each $f \in F$. Even if one actually wants a partial instantiation of P where some formal parameters $F' \subset F$ are kept unchanged one still has to give a dummy actual parameter for every $f' \in F'$.

3. The distinction between non-parameterized, parameterized, and parameter objects sometimes appears to be artificial, e.g. regarding a parameterized object as non-parameterized object or vice versa is not possible.

In loose specifications there is in principle no need to distinguish formal parameters from ordinary specifications since both are interpreted loosely denoting all their models. Hence, in a loose specification language the conventional parameterization concept sketched above consists of

- declaring a specification SP as parameterized by
- identifying some subspecification(s) SPf as formal parameter(s), and
- giving some specification(s) SPa as actual parameter(s) by
- supplying specification morphism(s)

  ρ: SPf → SPa,

  usually called fitting morphisms or views, in order to describe the replacements.
- The resulting instance is usually defined by a pushout construction (e.g. [Ehc 82]) of the diagram

However, the replacement of SPf may take place in the same way if SP had not been declared as parameterized specification and SPf as its formal parameter. This is the essential idea of dynamic parameterization which solves the problems described above:

> Instead of statically declaring any objects as parameterized or as formal parameters, the object to be instantiated (SP) and the subobjects to be replaced (SPf) are dynamically identified at instantiation time together with the replacing objects (SPa) and the replacement prescription (ρ).

This concept is realized in the loose specification language Look in a non-hierarchical framework ([ZLT 82]) and in ASPIK ([BV 83], [BV 85]) which supports the definition of hierarchical objects. Other specification languages like Clear ([BG 80]), Ordinary ([Go

81]), CIP-L ([CIP 85]), and ASL ([SW 83]) provide conventional parameterization concepts in the sense that parameterized specifications and formal parameters must be declared statically.

The possible identification of parameter and non-parameter specifications in a loose approach was suggested for Clear in [BG 80]. This identification lead to a subtle error as pointed out in [Sa 81]: an actual parameter still had to contain its formal parameter. But the solution suggested in [Sa 81] introduces again a distinction between the two types of specifications such that a parameter (meta theory) differs from an ordinary specification (theory) only in the keyword 'meta'.

Note that the idea of dynamic parameterization is not applicable in fixed specification languages like ACT-ONE ([EFH 83]) where formal parameters must be interpreted loosely in contrast to parameterized and non-parameterized specifications which are interpreted in a fixed way.

Example 2.1 [instances of LIMITED-STACK]

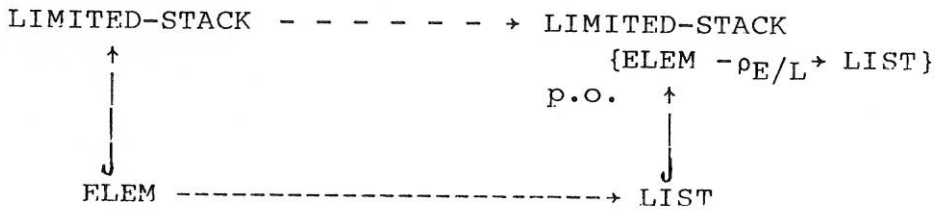
LIMITED-STACK denotes all algebras of stacks over an arbitrary element carrier where the depth of the stacks does not exceed a fixed but arbitrary limit. Like all specifications we indicated in Section 1, LIMITED-STACK has no parts declared as formal parameters. However, the subspecification ELEM could be dynamically replaced e.g. by the specification LIST via the specification morphism

$$\rho_{E/L}: \text{ELEM} \to \text{LIST}$$

mapping sort elem to sort list. The result is a specification of limited stacks over lists over arbitrary elements. It is denoted by

$$\text{LIMITED-STACK}\{\text{ELEM} \xrightarrow{\rho_{E/L}} \text{LIST}\}$$

and defined by the pushout diagram:



Alternative instances of LIMITED-STACK can be obtained by replacing the limit constant in LIMIT e.g. by the natural number 100, or by refining the subspecification NAT by a more elaborate version providing e.g. additional operations.

3. Hierarchies

Hierarchical structures arise in programming languages (e.g. structured programming), specification languages (e.g. hierarchical specifications), and many other areas (e.g. classification schemes, devisions in a company). In such hierarchical structures two aspects may be distinguished: the objects to be structured and the structure to be imposed on the objects.

The objects to be structured are usually provided with some kind of subpart relationship such that there is at most one subpart relation between any two objects, although the two objects may be connected in various other ways.

Such a situation is characterized in general terms by an appropriate category $(C,\vec{C})$: C is an arbitrary category with a subcatgory $\vec{C}$ where C and $\vec{C}$ have the same objects but there is at most one morphism between any two objects in $\vec{C}$. Thus, the morphisms in $\vec{C}$ represent an abstract form of subpart relationship, and the morphisms in C describe arbitrary other relationships.

Usually, such objects can be constructed stepwise by starting with some initial object and adding finitely many subparts in each step. The structure resulting from such a construction process is described by an appropriate order $AO = (O,<,\bot)$: $(O,<)$ is a well-founded, irreflexive, partially ordered set of nodes with minimum $\bot$ such that every node has finitely many predecessors.

AO expresses the structure of a hierarchical object in $(C,\vec{C})$ as follows: the minimum node is labelled with the initial subobject and the other nodes are labelled with the remaining subobjects such that the subpart relationship agrees with the ordering of the nodes. Since AO may be viewed as a category, namely the induced order category, we obtain a concise definition of a hierarchy H over $(C,\vec{C})$ as a functor

$$H: AO \to \vec{C}$$

over some appropriate order AO. Thus, a hierarchy describes a structured environment where the elements of AO serve as names for objects in C.

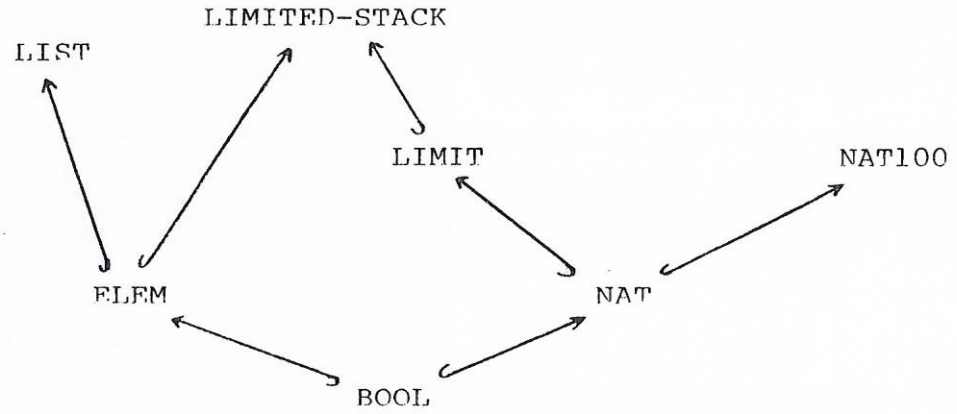
Example 3.1 [a SPEC hierarchy]

The construction of the specifications LIMITED-STACK, LIST, and an extension NAT100 of NAT by the constant number 100 can be documented by a hierarchy

$$H: AO \to \vec{SPEC}$$

where $\vec{SPEC}$ is the subcategory of SPEC with inclusions as

morphisms. AO is defined by the following graph:



H labels this graph by associating every node n with the specification denoted by n, e.g.

H(BOOL) = <<{bool}, {true, false, ...}>,
{not(true) = false, ...}>

4. Parameterization-by-use in hierarchical structures

The concepts of hierarchical structuring and dynamic parameterization can be smoothely combined according to the following principles:

- When defining hierarchical objects no distinction is made between formal parameters, parameterized objects, and non-parameterized objects.
- Every subobject of a hierarchical object is a potential formal parameter.
- Actual parameters are also hierarchical objects.
- The structure of the formal parameters must be respected by their actual parameters.
- The result of the instantiation is again a hierarchical object reflecting the structures of the instantiated object and of the actual parameters.

We say that a hierarchical object X uses all its subobjects. Parameters of X are identified only when some other object uses an instantiation of X. For this reason we call the concept designed according to the principles above parameterization-by-use.

Since hierarchies introduce object names, the objects to be instantiated as well as formal and actual parameters are identified by their names. Let n ε AO be the name of an object in

a hierarchy H: AO → $\vec{C}$. The parameterization-by-use concept requires the following data for every instance of n:

(1) A set M $\subseteq$ AO of nodes used by n that are to be replaced.

(2) A mapping f: M → AO associating every formal parameter node by its actual parameter node.

(3) A mapping h: M → /C/ associating every formal parameter with a fitting morphism to its actual parameter, i.e. for m ε M we have:
h(m): H(m) → H(f(m)) ε /C/.

The data in (1) - (3) are supplied in an application term

$$n\{m \xrightarrow{h(m)} f(m) \mid m \in M\}$$

which must satisy two requirements: f must respect the ordering of AO and h must be compatible with f, i.e. for every m < m´

$$\begin{array}{ccc} H(m') & \xrightarrow{h(m')} & H(f(m')) \\ \uparrow & & \uparrow \\ | & & | \\ | & & | \\ H(m) & \xrightarrow{h(m)} & H(f(m)) \end{array}$$

must commute in C.

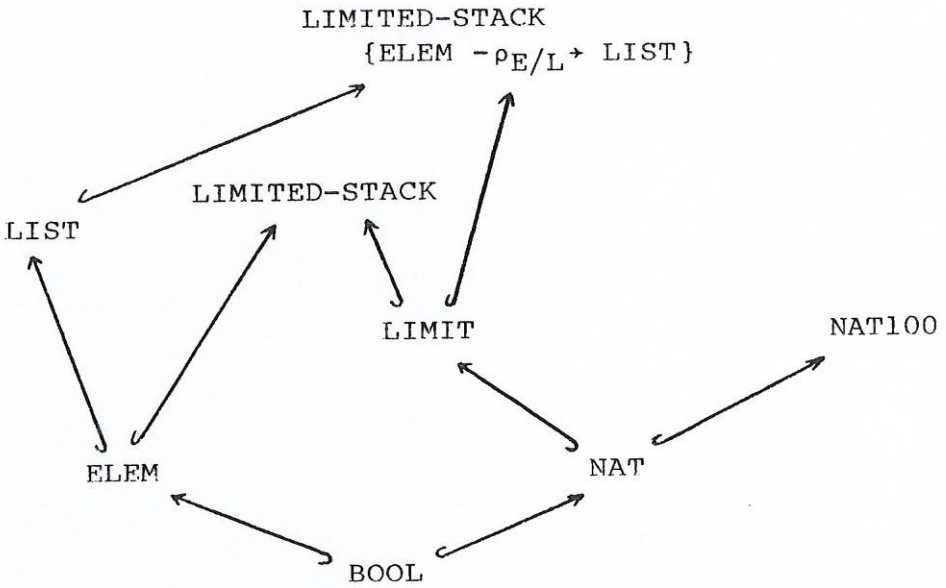
The hierarchical object denoted by an application term still uses all subobjects that were not affected by the actualization. Moreover, there are also subpart relations between the actual parameter objects and the new object, which is defined by a colomit construction generalizing the pushouts of the dynamic parameterization concept.

Example 4.1 [SPEC hierarchies with instantiations]

Consider the SPEC hierarchy from Example 3.1 and the specification morphism $\rho_{E/L}$ from Example 2.1. Taking M = {ELEM} as formal paramter for the node n = LIMITED-STACK, f(ELEM) = LIST as actual parameter, and h(ELEM) = $\rho_{E/L}$ as fitting morphism we get the application term

(1) LIMITED-STACK{ELEM $\xrightarrow{\rho_{E/L}}$ LIST}.

Viewed as a flat specification the object denoted by (1) is identical to the pushout object from Example 2.1. However, as a structured object it determines the hierarchy



which results from the hierarchy in Example 3.1 by adding the application term (1) as a new node. This structure expresses for instance that there is only one copy of the specifications LIMIT and NAT which are subparts of both LIMITED-STACK and LIMITED-STACK{ELEM $-\rho_{E/L}\rightarrow$ LIST} (c.f. the "based objects" used to express shared substructures in Clear [BG 80]).

Similarly we could take M = {ELEM, LIMIT} as formal parameter set for LIMITED-STACK, obtaining e.g. the application term

(2) LIMITED-STACK{ELEM $-\rho_{E/L}\rightarrow$ LIST, LIMIT $-\rho_{L/N}\rightarrow$ NAT100}

where $\rho_{L/N}$ replaces the constant limit in LIMIT by the natural number 100.

An application term defining lists of natural numbers is

(3) LIST{ELEM $-\rho_{E/N}\rightarrow$ NAT}

where $\rho_{E/N}$ maps sort elem to sort nat.

The instances denoted by (1) and (3) suggest to generate also a specification of limited stacks over lists of natural numbers, which can be achieved in two ways: Either by taking the stacks over lists in (1) and replacing the elements of the lists by the natural numbers

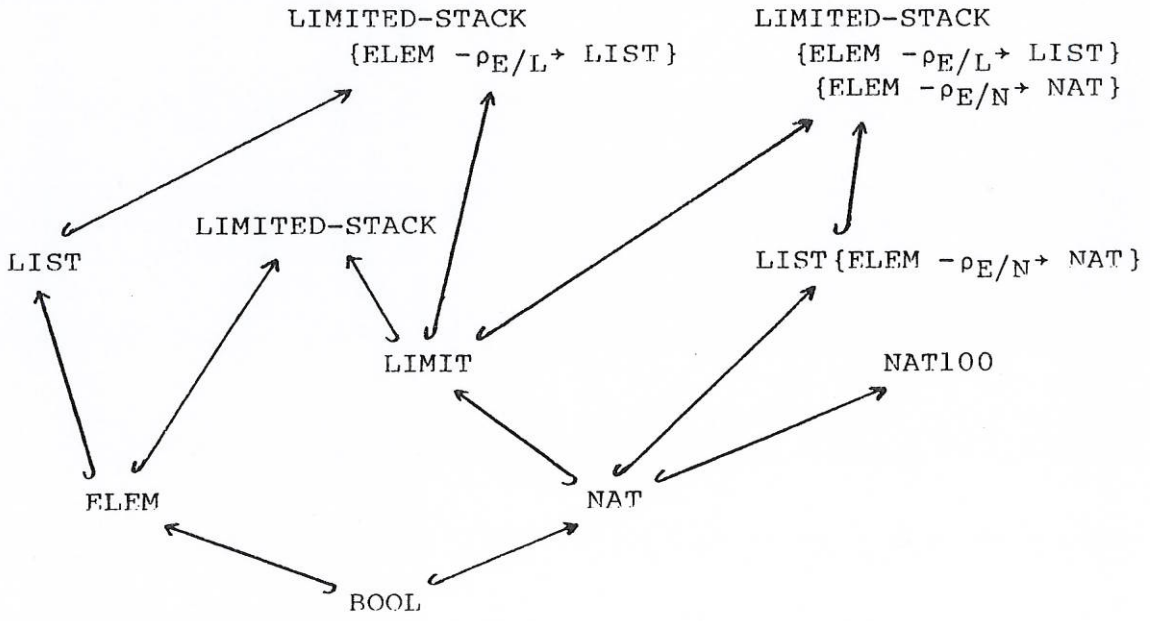
(4) LIMITED-STACK{ELEM $-\rho_{E/L}\rightarrow$ LIST} {ELEM $-\rho_{E/N}\rightarrow$ NAT}

or by replacing the elements of the stacks by the lists of natural numbers in (3)

(5) LIMITED-STACK{ELEM $-\rho_{E/LN}\rightarrow$ LIST{ELEM $-\rho_{E/N}\rightarrow$ NAT}}

where $\rho_{E/LN}$ maps sort elem to sort list of the specification denoted by (3).

Obviously, the different replacement sequences in (4) and (5) should not matter and both application terms should denote the same flat specification. Moreover, (4) and (5) should also denote the same hierarchical object which should use the intermediate instantiation denoted by (3). This situation is depicted in the further extended hierarchy



Since the node LIST lies between the node LIMITED-STACK{ELEM $-\rho_{E/L}\rightarrow$ LIST} to be instantiated and the formal parameter ELEM, the application term (4) is called an indirect application term.

From the observations in Fxample 4.1 we can describe some problems encountered in the context of structured parameterized specifications: When writing the equivalents of application terms like

(6) LIMITED-STACK{ELEM $-\rho_{E/N}\rightarrow$ NAT}

(7) LIMITED-STACK{ELEM $-\rho_{E/B}\rightarrow$ BOOL}

in ACT-ONE ([EFH 83]) or ASL ([SW 83]) in the context of the same specification, one gets the probably undesired effect of having one stack with both booleans and natural numbers as elements instead of two different stack types.

In contrast, the approach taken in Clear ([BG 80]) generates a new instance every time a term like (6) or (7) is used. This, however, leads to what has been termed Clear's proliferation problem ([BG 81], [Sa 81]): Although a term like (6) should always denote the same structured specification of stacks over natural numbers, the semantics of Clear generates as many copies as there are occurrences of the term.

The solution to Clear's proliferation problem suggested in [Sa 81] is based on a set-theoretic semantics of Clear: Different occurrences of a term like (6) will always yield the same specification and still a different one than (7). But terms like

(8) LIMITED-STACK{FLEM $-\rho_{E/N}\rightarrow$ NAT}{LIMIT $-\rho_{L/N}\rightarrow$ NAT100}

(9) LIMITED-STACK{LIMIT $-\rho_{L/N}\rightarrow$ NAT100}{FLEM $-\rho_{E/N}\rightarrow$ NAT}

(10) LIMITED-STACK{FLEM $-\rho_{E/N}\rightarrow$ NAT, LIMIT $-\rho_{L/N}\rightarrow$ NAT100}

would yield three different copies of actually the same instantiation object.

Another remark concerns multiple instantiations as in the application terms (4) and (5) which must denote the same specification if instantiation is associative. Associativity results are given in e.g. [Ehc 82], [EKTWW 80], and [Ga 83] in a non-hierarchical framework, but so far there seems to be no approach providing a hierarchical semantics where (4) and (5) denote the same hierarchical object which is based on (3).

5. Closed hierarchies

The evaluation of application terms to hierarchical objects in a hierarchy H: AO $\rightarrow \vec{C}$ as outlined in the previous section is described by an evaluation function

$$eval_H: \{t \mid t \text{ is application term in } H\} \rightarrow AO$$

taking application terms to nodes in AO. If such an evaluation function exists and is totally defined we say that H is closed under applications. Moreover, all the problems mentioned at the end of the previous section can then be expressed in terms of the evaluation function: Obviously, the same application term T will always yield the same node $eval_H(T)$, and two application terms T, T' with $eval_H(T) = eval_H(T')$ represent not only the same flat object but also the same hierarchical object.

For the stepwise construction of a hierarchy we introduce an

inductively defined canonical closure construction which turns a hierarchy into a closed hierarchy and supplies an evaluation function for it each time a new node has been added explicitly. The idea is to determine a normal form for application terms such that all normal form terms represent different objects and all other terms can be transformed into their normal forms. The normal form terms are added implicitly to the hierarchy as new nodes and are labelled by the respective instantiation objects. Roughly speaking, a normal form term does not contain any sequential parameter replacements but only parallel ones.

Looking at the examples in Section 4, we observe for instance that (10) is the normal form for both (8) and (9), and that (5) is the normal form for (4). In particular, the normalization process transforms any indirect application term like (4) to a direct one by implicitly supplying a derived actual parameter for every node between an explicit formal parameter and the node to be instantiated. For instance, (4) is transformed into

(11) LIMITED-STACK{ELEM $-\rho_{E/L}\rightarrow$ LIST}
{ELEM $-\rho_{E/N}\rightarrow$ NAT,
LIST $\rightarrow$ LIST{ELEM $-\rho_{E/N}\rightarrow$ NAT}}

where $\rightarrow$ denotes the obvious morphism from LIST to the instance of LIST. (11) in turn is normalized to (5) by composing the parameter replacements.

The instantiation objects labelling normal form nodes are particular colimit objects with a subpart relation to all used objects. Two general conditions guarantee the existence of such colimits in our closure construction:

- $(C,\vec{C})$ must have mixed pushouts, i.e. the pushout of a $\vec{C}$-morphism with an arbitrary C-morphism can always be chosen such that one of the pushout injections is a $\vec{C}$-morphism, too:

- There must be a so-called prefix function for $(C,\vec{C})$ that guarantees that for any finite hierarchy with prefixed objects there is a colimit in C that is a colimit in $\vec{C}$, too.

In our example, (SPEC, $\vec{SPEC}$) obviously has mixed pushouts since a morphism in $\vec{SPEC}$ is the component-wise (on sorts, operations, and sentences) set-theoretic inclusion. A prefix function is obtained by implicitly prefixing all sorts and operation names by the node

name where they are introduced, e.g. LIST.list and LIST{ELEM - $\rho_{E/N}$→ NAT}.list distinguish arbitrary lists from lists of natural numbers.

Thus, the closure construction is defined in such a way that the resulting evaluation function avoids the problems sketched in Section 4: It avoids Clear's proliferation problem and guarantees associativity of hierarchical instantiation.

6. A hierarchy definition language

With the notions introduced in the previous sections we can now devise a general hierarchy definition language suitable for the stepwise definition of hierarchies in an arbitrary appropriate category $(C,\vec{C})$ with mixed pushouts and a prefix function. All we need to know about the underlying category are the following basic notions:

- A syntactic domain 'something' describing the construction of a new object.

- A corresponding object constructing function
  $$\text{build-object: } 2^{|C|} \times \text{something} \to |C|$$
  that takes a set B of objects and a syntactic description sth and composes them to a new object which has all objects in B as subobjects, i.e.
  $$\forall\, b \in B \;.\; b \to \text{build-object}(B,\text{sth}) \in /\vec{C}/.$$

Based on these notions a hierarchy definition consists of a list of object definitions, one for the initial object with an empty use clause, and the others for based objects with non-empty use clauses:

```
hierarchy-definition ::= initial-object  based-object ...

initial-object       ::= object node
                                 sth
                         endobject

based-object         ::= object node
                                 use (application term | node) ...
                                 sth
                         endobject
```

The semantics of this language yields a hierarchy where each node explicitly defined by an object definition is labelled with the object obtained by the build-object function. The set of subobjects needed as argument for that function is determined by the nodes and application terms in the use clause, which can

always be evaluated in the hierarchy since it is implicitly closed after every explicit object definition.

For the appropriate category (SPEC, $\vec{SPEC}$) the syntactic description sth something will contain the new sorts, operations, and sentences to be added to the union of the used specifications. Assuming sorts, ops, and axioms clauses as syntactic descriptions of these parts we obtain:

```
sth ::= sorts ...
        ops ...
        axioms ...
```

The function

build-object: $2^{|SPEC|}$ x something → |SPEC|

now takes a set of flat subspecifications and the sorts, ops, and axioms clauses and combines them into a new flat specification:

```
build-object({SP1, ...,SPn},
             sorts S, ops Op, axioms E)
            :=
               SP1 u ... u SPn u
               <<S,Op>, E>
```

We have used an instance of this hierarchy definition language in the formal semantics of the algebraic specification development ASPIK which has been implemented in Interlisp as a central part of an integrated software development and verification system ([BV 85]). The system maintains a database for specification objects and relations between them and fully supports the parameterization-by-use concepts so that e.g. arbitrary application terms are available to the user.

As another example we elaborate in [BV 83] a hierarchy definition language for equational signatures with (SIG, $\vec{SIG}$) as appropriate category. We show explicitly why the preconditions of mixed pushouts and a prefix function are satisfied and point out how these preconditions easily carry over to the appropriate category (SPEC, $\vec{SPEC}$) of specifications with signatures in SIG.

This observation can be generalized to an arbitrary institution I with signature category SIG_I and theory category Th_I:

1. The notion of subpart relationship only has to be defined for the signatures SIG_I. The resulting appropriate category (SIG_I, $\vec{SIG}_I$) immediately gives rise to an appropriate category (Th_I, $\vec{Th}_I$) since every theory morphism is in particular a signature morphism.

2. It suffices to ensure that (SIG_I, $\vec{SIG}_I$) has mixed pushouts

and a prefix function: Since mixed pushouts and colimits of theories can be computed from the mixed pushouts and colimits of their signatures (because the forgetful functor taking theories to their signatures reflects colimits [GB 83]) we conclude that $(Th_I, \vec{Th}_I)$ has mixed and colimits as well.

This connection confirms that the concept of institution for the definition of flat objects on the one hand and our complementary structuring and parameterization concepts on the other hand are compatible and orthogonal to each other.

References

[BG 80] Burstall, R.M., Goguen, J.A.: The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. LNCS Vol.86, pp. 292-332.

[BG 81] Burstall, R.M., Goguen, J.A.: An informal introduction to specifications using Clear. in: The Correctness problem in Computer Science (Eds. R.S. Boyer, J.S. Moore). Academic Press 1981.

[BV 83] Beierle, C., Voß, A.: Canonical term functors and parameterization-by-use for the specification of abstract data types. SEKI-Projekt, Memo SEKI-83-07, Interner Bericht 77/83, Universität Kaiserslautern, FB Informatik, May 1983.

[BV 85] Beierle, C., Voß, A.: Algebraic Specifications and Implementations in an Integrated Software Development and Verification System. Memo SEKI-85-12, FB Informatik, Univ. Kaiserslautern, (joint SEKI-Memo containing the Ph.D. thesis by Ch. Beierle and the Ph.D. thesis by A. Voß), Dec. 1985.

[CIP 85] CIP Language Group: The Munich Project CIP, Vol. I: The Wide Spectrum Language CIP-L. LNCS, Vol. 183, 1985.

[EFH 83] Ehrig, H., Fey, W., Hansen, H.: ACT ONE: An algebraic specification language with two levels of semantics. TU Berlin, Fachbereich Informatik, Bericht Nr. 83-03, 1983.

[Ehc 82] Ehrich, H.-D.: On the theory of specification, Implementation and Parametrization of Abstract Data Types. JACM Vol. 29, No. 1, Jan. 1982, pp. 206-227.

[Ehg 81] Ehrig, H.: Algebraic Theory of Parameterized Specifications with Requirements, Proc. 6th Colloquium on Trees in Algebra and Programming (E. Astesiano, C. Böhm, eds.), LNCS 112, pp. 1-24, 1981.

[EKTWW 80] Ehrig, M., Kreowski, H.-J., Thatcher, J., Wagner, E., Wright, J.: Parameter Passing in Algebraic Specification languages. Draft version, TU Berlin, March 1980, see also 7th ICALP, LNCS Vol. 85.

[Ga 83] Ganzinger, H.: Parameterized Specifications: Parameter Passing and Implementation with respect to Observability. ACM TOPLAS Vol. 5, No.3, July 1983, pp. 318-354.

[GB 83] Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Program Specification. Draft version. SRI International and University of Edinburgh, January 1983.

[GMP 82] Goguen, J.A., Meseguer, J., Plaisted, D.: Programming with Parameterized Abstract Objects in OBJ. SRI International and University of Illinois, 1982.

[Go 81] Goguen, J.A.: Two Ordinary Specifications. Technical Report CSL-128, SRI International, October 1981.

[HKR 80] Hupbach, U.L., Kaphengst, H., Reichel, H.: Initial algebraic specifications of data types, parameterized data types, and algorithms. VEB Robotron, Zentrum für Forschung und Technik, Dresden, 1980.

[Sa 81] Sannella, D.T.: A new semantics for Clear. Report CSR -79-81, Dept. of Computer Science, Univ. of Edinburgh, 1981.

[SB 83] Sannella, D.T., Burstall, R.M.: Structured theories in LCF. 8th Colloquium on Trees in Algebra and Programming, 1983.

[ST 84] Sannella, D., Tarlecki, A.: Building specifications in an arbitrary institution. Symp. Semantics of Data Types, Sophia-Antipolis. LNCS Vol. 173, 1984, pp. 337-356.

[SW 82] Sannella, D.T., Wirsing, M.: Implementation of parameterized specifications, Proc. 9th ICALP 1982, LNCS Vol. 140, pp 473 - 488.

[SW 83] Sannella, D., Wirsing, M.: A kernel language for algebraic specification and implementation. Proc. Intl. Conf. on Foundations of Computing Theory, Bergholm, LNCS Vol. 158,1983.

[ZLT 82] Zilles, S.N., Lucas, P., Thatcher, J.W.: A Look at Algebraic Specifications. RJ 3568 (41985), IBM Research Division Yorktown Heights, New York, 1982.