Formal Problem Specification on an Algebraic Basis

H.A. Partsch

Fakultät für Informatik
Universität Ulm
D-89069 Ulm
Germany

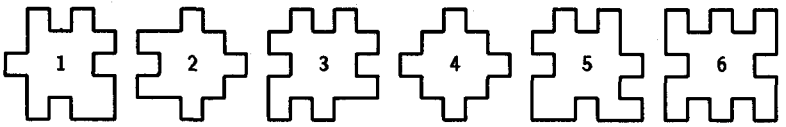
Abstract

When aiming at developing correct software, formal problem specification is nowadays considered an important intermediate stage in the software development process. An algebraically based formalism for problem specification is introduced with the emphasis both on how to use such a formalism for the specification of concrete problems and on the methodological aspects of formalization. The formalism used is essentially the one developed in the project CIP which may be considered as a representative of the state-of-the-art in algebraic specification.

1 Introduction: Why Formal Specification?

The major difficulty in software development is caused by the fact that an original problem description usually consists of an unstructured bunch of half-baked wishes which are neither precise, detailed, nor even complete, whereas a program, by nature, has to be precisely defined and fully detailed up to each single instruction. It is obvious that an attempt to bridge the huge gap between these extreme positions in one large step is doomed to fail, i.e., the resulting software will most likely not work as expected.

Example
A simple, though typical example to illustrate this situation is given by the request to write a program to solve the "cube problem". This problem deals with a commercially available puzzle consisting of 6 pieces of (maybe) different shape. The puzzle's goal is to build, if possible, a cube with the 6 pieces as its faces. The problem to be solved is immediately clear when seeing the pieces of the puzzle. However, it is not as easy to give a verbal description of the problem.
Each of the pieces of the puzzle is "roughly quadratic" and has a thickness of 1 unit and a fixed "kernel" of size 3×3 square units. Additionally, each of the sides of a piece may have upto 5 quadratic "teeth" of size 1 square unit. The following are typical pieces:



There are two basic reasons why software does not work as intended:

- Either the original problem was misunderstood or misinterpreted, and thus the program solves a "different" problem; or
- The problem was well understood, but the program does not solve this very problem.

In order to cope with the first source of trouble, it is widely accepted today that the process of software development should be broken into smaller, manageable steps. A minimum requirement is a decomposition into two steps (frequently called 'requirements engineering' and 'program construction') with a precise, preferably formal statement of the problem as an intermediate stage (cf. also [Balzer et al. 83], [Agresti 86], [Bauer et al. 89]). How to formally describe the above-mentioned "cube problem" will be dealt with in section 5.2.

In order to cope with the second source of trouble, various approaches have been suggested on how to construct an efficient program that satisfies a given formal specification. All these approaches have correctness as their central goal, irrespective of whether they are based on program transformations (for overviews, see [Feather 86], [Goldberg 86], or [Partsch 90]), on assertional techniques (e.g., [Dijkstra 86], [Gries 81], [Backhouse 86]), or another conceptual basis.

A *problem specification* is a description of the problem to be solved. Ideally, it should describe *what* the problem is without giving the solution or even the details about its implementation. In such a case, a specification is said to be *descriptive*. Otherwise, i.e. if the specification describes *how* to solve the problem, we call it *operational*.

In software engineering, a problem specification is called a *requirements specification* and there are numerous traditional approaches (for overviews, see, e.g., [IEEE 77], [IEEE 85], [Partsch 91]) to providing suitable formalisms. All of them use formal concepts only to an extent that is still manageable by a non-expert user, and provide but simple linguistic means for formulating requirements specifications, mainly relying on an intuitive understanding of the semantics. Additionally, some of them are even backed by methodological principles to ensure a systematic conversion of an informal problem statement into the respective formalism. Nearly all of them, however, leave open how to obtain programs that solve the specified problem, and, furthermore, how to verify that these programs indeed meet the specification.

In order to overcome the fundamental deficiencies of the traditional approaches, viz. semantic imprecision and lack of an integrated methodology, there are various new approaches that focus on formal problem specification.

A specification is called a **formal specification** if it is formulated in a formal language, i.e. a language whose syntax and semantics are explicitly established prior to its use; otherwise, we call a specification **informal**. Thus, in particular, specifications in natural language are informal.

All of the new approaches assume a rigorous formal basis for an initial problem specification which is, e.g.,

- relational (e.g., Gist [Balzer 81], EREA [Dubois et al. 88]),
- functional (e.g.[Henderson 80], [Bird, Wadler 88], VDM [Bjørner 82], [Jones 86]),
- predicative (e.g., [Hehner et al. 86], [Broy 87]),
- set theoretical (e.g., Z [Spivey 88]),
- assertional (e.g., [Dijkstra 76], [Gries 81], [Backhouse 86]), or
- algebraic (e.g. ACT TWO [Fey 86], ASF [Bergstra et al. 89], ASL [Wirsing 83], CLEAR [Burstall, Goguen 80], COLD [Feijs et al. 87], LARCH [Guttag, Horning 83], OBJ [Futatsugi et al. 85], PLUSS [Gaudel 85]).

Additionally, all these new approaches also aim at an integrated methodical support for (formally) constructing programs from a given formal specification. For solving the problem of building a specification, however, in all these formal approaches, essentially the same difficulties as in traditional requirements engineering have to be faced. Therefore, we suggest an approach to formalization which basically builds on experiences gained there, but also takes our envisaged specification formalism into account. This will be the topic of the following section.

2 The Process of Formalization

Formalization deals with the problem of how to proceed in order to build a formal specification in a systematic way. A rough guideline is given in [Rzepka, Ohno 85] where at least three essential sub-activities are identified, viz.

- identification of the problem,
- formal description of the problem, and
- analysis of the formal problem description.

2.1 Problem Identification

Problem identification means finding out what the problem is. The difficulties here mainly originate in the ambiguities and sources of misunderstanding inherent to the communication of different people by means of natural language. Usually, the person who gives the problem is not the one who is to describe it formally; additionally, due to different educational and professional backgrounds, they usually do not speak the same (technical) language. Therefore, problem identification involves a "translation" between universes of discourse, and the essential part of problem identification has to concentrate on finding this translation.

Usually a problem statement (implicitly) assumes basic knowledge about the context of the problem, frequently called the *problem domain*. For truly identifying the problem it is essential to make these implicit assumptions explicit, i.e. to first identify the respective problem domain. Having done so, further steps in finding the above-mentioned translation are

- choosing a concept of the problem domain,
- representing the concept, and
- associating the constituents of the problem with the representation of the concept.

Following [Webster 74], we use the notion *concept* for "*an idea or thought, especially a generalized idea of a class of objects; abstract notion*". Hence, a concept of a (given) problem domain is an abstract view of the problem domain, free from irrelevant details, but suited to reflect its essential characteristics. As we are concentrating on software systems, rather than on more general ones, we can further rule out arbitrary technical concepts and focus our attention onto concepts from mathematics.

Example
In order to illustrate our notion of a (mathematical) concept, we consider the problem of building software for a traffic control system for a particular city. The problem domain here comprises, among others, the topology of the respective city, i.e., a street map, which has to be reflected as part of a concept of our sample problem domain. In a simplified view, a street map is a structure consisting of streets and intersections, and one straightforward concept for modelling a street map is a finite graph. ■

Further examples of mathematical concepts are

- sets, relations, mappings, functions, orderings and lattice structures,
- algebraic structures (e.g., groups, rings, fields, sequences, bags, trees),
- relational structures (e.g., graphs, Petri nets),
- formal systems (e.g., equational systems, grammars, automata, rewrite systems, deduction systems, systems of concurrent processes).

The choice of a suitable concept already entails a tremendous gain with respect to precision, as potential sources of misunderstanding are ruled out. Frequently, in addition, the choice of a concept even amounts to a solution of the problem, as certain tasks for certain concepts are already formalized and solved in generality. Examples of this kind are

- minima and maxima in orderings;
- construction and modification of particular algebraic structures;
- paths, cycles, or closures in relational structures;
- languages generated by grammars or accepted by automata; or
- deadlock or starvation in systems of concurrent processes.

There is a lot of freedom in *choosing a concept* for a particular problem domain. Only in rare cases a suitable concept is obvious or straightforward because of concrete hints that can be found in the informal problem description.

However, generally no such hints are available. Therefore, the choice of an adequate concept requires *decisions*. These decisions have far-reaching consequences, since they not only affect the level of abstraction and the complexity of the formalization of the problem, but also available solutions to the problem. Consequently, choosing an adequate concept requires intuition and experience, and should be done very carefully.

In general, a concept consists of

- objects associated with certain object classes;
- operations on the object classes; and
- relations between objects and/or object classes.

A concept is either primitive or composed of other concepts. *Representing a (non-primitive) concept* has to deal with the refinement and the detailing of its constituents and their description on the basis of simpler concepts. As there may be several representations of the same concept, again, a lot of freedom is available here which involves further decisions.

Example
The concept "finite directed graph", which we used in connection with our sample problem admits several (equivalent) descriptions, e.g.,

a. a set of nodes and a set of edges (represented by pairs of nodes); (2.1.1)
b. a set of nodes and a pair of incidence functions which associate with each node the set of its predecessors and successors; (2.1.2)
c. an adjacency matrix where component (i, j) has the value 1, if there is an edge from i to j, and 0 otherwise. (2.1.3)

■

Having decided on a concept of the problem domain and a suitable representation of the chosen concept, it remains to *associate the constituents of the problem with the representation of the concept*, which, again, entails decision making.

Example
If, for our city map, we, for example, decided on description (2.1.1), we still would have to decide on the association of streets and intersections with nodes and edges. One obvious possibility is to associate intersections with nodes, and streets with edges. However, we also might associate streets with nodes that are connected by an edge if they intersect. ■

Which of several possible associations to choose, of course depends on further details of the problem to be solved. Thus, e.g., in the first association (i.e., intersections as nodes, streets as edges), it is easy to check how many streets are involved in an intersection, but more difficult to trace an entire street. The second association, on the other hand, gives easy access to individual streets, but, for example, expressing that a street admits only one-way traffic is more difficult.

2.2 Problem Description

If a problem has been identified properly, its (formal) description amounts to translating the result of the identification process into constructs available in the formal specification language. In particular, this means

- mapping the (representation of the) concept of the problem domain onto available constructs; and

- "glueing" together the constituents of the problem by giving an expression in the formal specification language that describes the task to be fulfilled in terms of the image of the representation of the concept.

Similarly to other sub-activities of formalization, decisions are necessary here, too, depending on the particular specification language. Whereas translation of the representation of the concept into available language constructs in most cases will be straightforward, the formulation of the problem proper as an expression in the specification language usually again leaves a lot of freedom.

None of the decisions to be taken during the formalization process is unique, as we tried to illustrate by the simple examples above. Therefore a prime concern of any formalism for formal specification of problems is the provision of as much flexibility as possible in order to allow the adequate formulation of all possible representations of a variety of different concepts.

At least, however, any formalism for the formal specification of some task has to offer constructs that allow the representation of the constituents of a concept, i.e., objects and object classes, operations, and relations, and the formulation of expressions that reflect that task.

2.3 Analysis of the Problem Description

Since the problem specification is the basis for a subsequent program development, it is important that it is "correct". Analysis of a problem specification should comprise checks on

- syntactic aspects;
- semantic properties; and
- the relationship to the originally given problem.

Obviously, formal specifications entail the usual problems to be encountered in using a formal language, viz. correctness with respect to syntax and context conditions that have to be checked.

The "meaning" of a formal specification is defined by the semantics of the specification language used. Usually this is a partial mapping from syntactic constructs to (sets of) semantic values. On this basis additional practically important semantic properties of formal specifications can be introduced such as

- A formal specification is called *defined* (also *consistent* or *satisfiable*) if it has a "non-empty meaning", i.e., if there is at least one semantic value associated with the specified problem; otherwise it is called *undefined* (or *inconsistent*).
- A formal specification is called *determinate* if there exists at most one semantic value associated with the specified problem; otherwise it is called *ambiguous*.
- A formal specification is called *redundant* if there exists a semantically equivalent specification which is "simpler".

Except for redundancy, these properties can be formally checked on the basis of the semantics of the specification language. There are, however, additional properties that

are not formally verifiable. These properties characterize the relationship between the meaning of the formal specification and the originally intended problem. Examples of such properties are

- A formal specification is called *adequate*, if its meaning coincides exactly with the original problem.
- A formal specification is *overspecified*, if its meaning comprises not all of the solutions to the original problem.
- A formal specification is *underspecified*, if its meaning comprises all solutions to the original problem, but also additional ones.

Obviously, these properties are not independent of each other: an adequate specification is neither over- nor underspecified, but inadequacy does not necessarily imply over- or underspecification.

Analysis of a formal specification is an essential part of the formalization process. The process of formalizing a problem may be considered finished only, when the formal specification is syntactically correct, and its adequacy with respect to the originally given problem is ensured. For practical reasons, an analysis with respect to redundancy seems worthwhile, too.

Before dealing with adequacy itself, however, a semantic analysis with respect to the semantic properties seems worthwhile, because it gives valuable information. Thus, for example, recognizing a formal specification to be undefined usually indicates a defect in the formalization process rather than unsolvability of the originally given problem. Likewise, an ambiguous formal specification of a problem which is known to have a unique solution implies inadequacy. Also, an examination of the specification with respect to overspecification and underspecification provides insight with respect to adequacy. Very often, underspecification can be removed by simply adding further conditions. Similarly, overspecification frequently can be eliminated by weakening certain restrictions. However, checking these properties is not sufficient. Further considerations with respect to adequacy are necessary, which may lead to redoing (parts of) the formalization process. Examples of such considerations are (formal) derivations of logical consequences of the formal specification to be validated against the original problem, or derivations of acceptable answers to questions concerning certain scenarios.

3 Algebraic Types

In section 2.2, we came to the conclusion that any formalism for the formal specification of problems should at least provide constructs that allow the representation of the constituents of a concept, i.e. objects and object classes, operations, and relations. Moreover, we found that the formalism should provide as much flexibility as possible. In section 2.3 we pinpointed the importance of analyzing a formal problem description which in turn implies that these kinds of analysis activities should be supported by a suitable specification formalism. A formalism that satisfies all these requirements is given by algebraic types.

An algebraic type provides a rather powerful formalism for defining objects, object kinds, and their characteristic operations in an abstract, implementation-independent,

and thus non-operational way. Object classes can be defined by (systems of) algebraic type declarations. Objects are characterized implicitly by their construction, modification, and access operations, rather than explicitly by exhibiting their internal structure; operations are defined by properties describing their mutual relationship.

In the following subsections we are going to introduce systems of type declarations in a step-by-step fashion by starting with a nearly trivial case and extending the formalism by gradually adding new concepts.

3.1 Signatures and Terms

To start with, we consider a simple example:

type NAT_0
  sorts **nat**;
  functs 0: → **nat**,
    succ: **nat** → **nat**
endoftype

This **type declaration** (marked by the keywords **type - endoftype**) introduces

- a sort symbol (here **nat**); and
- two function symbols 0 and *succ* together with their functionality. The functionality of 0 is → **nat**, i.e., 0 is a symbol for a constant. The functionality of *succ* is **nat** → **nat**.

As to the notation of typed symbols, we use the convention introduced by Pascal where a (possibly singleton) list of identifiers or symbols (separated by ",") is followed by ":" and their type.

The pair $\Sigma = (S, F)$, where S and F denote the sets of all sort and operation symbols (inclusive of their respective functionalities) defined in a type, is also referred to as the **signature** of that type.

Example
In case of NAT_0, we have

$$\Sigma_{NAT_0} = (\{\mathbf{nat}\}, \{0: \rightarrow \mathbf{nat}, \mathit{succ}: \mathbf{nat} \rightarrow \mathbf{nat}\})$$

Thus, in this particular example, type definition and signature coincide. In general, of course, this will not be the case. ■

The signature $\Sigma = (S, F)$ of a type defines the **well-formed terms** of sort **s**, for each $\mathbf{s} \in S$, with free variables from an **s**-sorted family $\{X_\mathbf{s}\}_{\mathbf{s} \in S}$ (inductively) as follows:

- every variable of sort **s** is a well-formed term of sort **s**;
- if $t_1, \ldots, t_n$ are well-formed terms of sorts $\mathbf{s}_1, \ldots, \mathbf{s}_n$ and f is an operation symbol with functionality $(\mathbf{s}_1 \times \ldots \times \mathbf{s}_n) \rightarrow \mathbf{s}$, then $f(t_1, \ldots, t_n)$ is a well-formed term of sort **s**. Hence, as a special case, nullary functions are well-formed terms;
- there are no other well-formed terms of sort **s**.

For a type T with signature Σ, any well-formed term (of sort $s \in S$) is called a **Σ-term** (of sort **s**). A Σ-term that is built from operation symbols only, is called a **ground term.** Ground terms may be used to denote objects of some sort.

Example
According to the above definitions, obviously, $succ(succ(x))$ is a well-formed term of sort **nat** provided x is a variable of sort **nat.** An example of a ground term of sort **nat** is $succ(succ(succ(0)))$. ■

3.2 Axioms and Semantics

The example NAT_0 given above is but a special case of a type, since it coincides with its signature. In general, a type T is completely characterized by a pair (Σ, E) where Σ denotes a signature, and E is a collection of axioms. An **axiom** (or **law**) is an arbitrary closed, well-formed first-order formula over equations (with symbol: "$\equiv$") and inequalities (with symbol: "$\not\equiv$") between Σ-terms.

Due to this (syntactic) definition of axioms, parentheses are only necessary for formulating complex terms or for disambiguating first-order formulas over equations and inequalities.

The simplest form of a law is an equation or an inequality between Σ-terms of the same sort, which is preceded by a universal quantification of its variables. In order to avoid notational overhead, a quantification (following the keyword **axioms**) may extend over several formulas separated by the symbol ",", which then means logical conjunction.

A simple example of an algebraic type with axioms is the following one (see also [Bauer et al. 85]) that defines the truth values **true** and **false**, as well as the operations $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction):

type BOOL
  sorts **bool**;
  functs **true**, **false**: $\rightarrow$ **bool**;
    $.\neg.$: **bool** $\rightarrow$ **bool**, (*negation*)
    $.\wedge.$, $.\vee.$: (**bool** $\times$ **bool**) $\rightarrow$ **bool**; (*conjunction, disjunction*)
  axioms $\forall$ **bool** x, y:
    true $\not\equiv$ **false**, (1)
    $\neg$ **true** $\equiv$ **false**, (2)
    $\neg$ **false** $\equiv$ **true**, (3)
    true $\wedge$ $x \equiv x$, (4)
    false $\wedge$ $x \equiv$ **false**, (5)
    $x \vee y \equiv \neg(\neg x \wedge \neg y)$ (6)
endoftype

The dots next to operation symbols such as $\neg$, $\wedge$, or $\vee$ indicate the positions of their arguments, the number of dots reflects the number of arguments. Thus $\neg.$ is a monadic prefix operator and $.\wedge.$ and $.\vee.$ are dyadic infix operators. The parenthesized texts to the right of some of the lines in the above definition (such as (*negation*) or (1)) are

comments, being irrelevant for the type definition. They will be used to convey some intuition with operation symbols or to be able to refer to individual axioms. The symbol "||" as in "$\forall\, x, y$: **bool** ||" is used to separate the quantified variables from the equations and inqualities.

The axioms of a type are not required to be minimal. In fact, adding properties, that are provable (cf. below) from the axioms, as additional laws, often helps in understanding a type definition. Thus, we could have added the usual properties of conjunction and disjunction, such as commutativity, associativity, and distributivity, which are provable from the given axioms. Note, however, that the law

true $\not\equiv$ **false**,

which guarantees that these symbols denote different constants, is necessary here, since it cannot be proved from the other axioms.

The well-formed terms of a type denote abstract objects. The axioms "equate" certain terms, i.e. they define a **quotient structure** on the set of well-formed terms.

In order to define the semantics of an algebraic type T = (Σ, E) with signature Σ = (S, F) and laws E, we first introduce the notion of a (partial) Σ-algebra:

A (partial) **Σ-algebra** A = $((\mathbf{s}^A)_{\mathbf{s}\in S}, (f^A)_{f\in F})$ consists of

- a family $(\mathbf{s}^A)_{\mathbf{s}\in S}$ of carrier sets (one for each sort);
- a family $(f^A)_{f\in F}$ of (partial) functions f^A: $(\mathbf{s}_1{}^A \times \ldots \times \mathbf{s}_n{}^A) \rightarrow \mathbf{s}^A$, if the symbol f has the functionality $(\mathbf{s}_1 \times \ldots \times \mathbf{s}_n) \rightarrow \mathbf{s}$.

As is known, the result of applying a partial operation may be undefined. In order to propagate undefinedness, we require that every operation f^A of an algebra A is **strict**, meaning that its application is undefined whenever one of its arguments is undefined.

Example

As an example of a Σ-algebra we consider the algebra

$$\text{FSET} = (\mathcal{P}(\mathbb{1}), \{\mathbb{1}, \emptyset, \mathbf{C}, \cap, \cup\}), \tag{3.2.1}$$

where $\mathbb{1}$ denotes an arbitrary singleton set, $\mathcal{P}(\mathbb{1})$ denotes the set of all subsets of $\mathbb{1}$, and **C**, $\cap, \cup$ denote complement, union, and intersection, respectively. FSET is a Σ_{BOOL}-algebra with

$$\mathbf{bool}^{FSET} = \mathcal{P}(\mathbb{1}),\ \mathbf{true}^{FSET} = \mathbb{1},\ \mathbf{false}^{FSET} = \emptyset,$$
$$\neg^{FSET} = \mathbf{C},\ \wedge^{FSET} = \cap,\ \vee^{FSET} = \cup.$$

Another Σ_{BOOL}-algebra is

$$\text{RNAT} = (\{0, 1\}, \{0, 1, (.+1)\ \mathbf{mod}\ 2, .\times., ((.+.)+(.\times.))\ \mathbf{mod}\ 2\}) \tag{3.2.2}$$

where 0 and 1 are supposed to be natural numbers and where +, $\times$, and **mod** denote the usual operations on natural numbers. ■

In order to be able to relate different Σ-algebras, we introduce the notion of Σ-homomorphisms, i.e. structure-preserving mappings between Σ-algebras:

A **weak** (resp. **strong**) **Σ-homomorphism** Φ: A $\rightarrow$ B from a Σ-algebra A to a Σ-algebra B is a family of partial (resp. total) functions $(\Phi_{\mathbf{s}}: \mathbf{s}^A \rightarrow \mathbf{s}^B)_{\mathbf{s} \in S}$ such that for all f: $\mathbf{s}_1 \times \mathbf{s}_2 \times \ldots \times \mathbf{s}_n \rightarrow \mathbf{s}$ in F and all $a_i \in \mathbf{s}_i^A$ (for i = 1, ..., n)

$$\Phi_{\mathbf{s}}(f^A(a_1, \ldots, a_n)) \equiv f^B(\Phi_{\mathbf{s}_1}(a_1), \ldots, \Phi_{\mathbf{s}_n}(a_n)).$$

A bijective Σ-homomorphism is called a **Σ-isomorphism**.

Example

As an example we consider the Σ_{BOOL}-algebras FSET and RNAT from (3.2.1) and (3.2.2). Obviously,

$$\Phi_{\mathbf{bool}} : \mathcal{P}(\mathbb{1}) \rightarrow \{0,1\} \text{ and } \Psi_{\mathbf{bool}} : \{0,1\} \rightarrow \mathcal{P}(\mathbb{1}),$$

defined by

$$\Phi_{\mathbf{bool}}(\emptyset) =_{\text{def}} 0,\ \Phi_{\mathbf{bool}}(\mathbb{1}) =_{\text{def}} 1, \text{ and } \Psi_{\mathbf{bool}}(0) =_{\text{def}} \emptyset,\ \Psi_{\mathbf{bool}}(1) =_{\text{def}} \mathbb{1},$$

are (strong) Σ_{BOOL}-homomorphisms. Since both are bijective, they are also Σ_{BOOL}-isomorphisms. ■

For every signature Σ there exists a special Σ-algebra, the **term algebra** (or **word algebra**), denoted by W(Σ), which consists of all ground Σ-terms as carrier sets and term-forming operations according to the respective operation symbols. Any other Σ-algebra A can be related to W(Σ) by a particular weak Σ-homomorphism i:W(Σ) $\rightarrow$ A. This Σ-homomorphism i, called the **interpretation** of W(Σ) in A, is defined by

$$i(f(a_1, \ldots, a_n)) =_{\text{def}} f^A(i(a_1), \ldots, i(a_n))$$

for any term $f(a_1, \ldots, a_n)$ from W(Σ). The interpretation of an arbitrary term t (in A) will be abbreviated by t^A.

An equation $t_1 \equiv t_2$ between closed Σ-terms t_1 and t_2 (of the same sort) is **valid** in a Σ-algebra A iff their interpretations t_1^A and t_2^A are both undefined, or both defined and equal ("**strong equality**"). The validity of laws that are arbitrary first-order formulas is then defined as usual.

Example

For FSET as defined in (3.2.1) we have, e.g.,

$$i(\mathbf{false}^{FSET} \wedge^{FSET} x) =_{\text{def}} (\emptyset \cap x) \equiv \emptyset =_{\text{def}} i(\mathbf{false}^{FSET}),$$

and, hence, (**false** $\wedge$ x) $\equiv$ **false** is valid in FSET. In the same way it can be shown that the other laws of BOOL are valid in FSET, too. ■

A Σ-algebra A is called **term-generated**, if every element of any of the carrier sets $\mathbf{s}^A$ can be obtained by finitely many applications of functions f^A.

Example
Obviously, the Σ-algebra FSET from (3.2.1) is term-generated. However, it would not be term-generated, if 1l was an arbitrary (non-empty) set instead of a singleton set. ■

A Σ-algebra A is called a **model** of a type $T = (\Sigma, E)$ if it is term-generated and all laws of T are valid in A. A type T is called **consistent** (or **satisfiable**), if it has at least one model; otherwise it is called **inconsistent**. A type is called **monomorphic** if all its models are isomorphic.

Example
According to the latter definition, FSET is a model of BOOL. Another model for BOOL is provided by RNAT. Both models are obviously isomorphic. In fact, BOOL, as defined above, can be proved to be monomorphic. Moreover, obviously, BOOL is consistent. ■

The **semantics** of a type T is defined to be the family of all isomorphism classes of models of T. In the following the semantics of T will be denoted by GEN(T).

There are various other approaches where the semantics of a type is defined as a distinguished model (e.g., an 'initial' or 'terminal' one, cf. [Wirsing et al. 83]). A comparison of these different semantic definitions can be found in [Wirsing 89]. We prefer the above definition, as it is closer to our intuitive understanding of a formal specification: each of these models of a type definition characterizes a possible solution of the task that is formally specified by the type definition. Because of the restriction of the semantics of a type to term-generated models, proofs by *term induction* and *structural induction* are possible (for details, cf. [Wirsing et al. 83]).

3.3 Hierarchical Types

In order to be able to define object classes in a structured way, we extend our formalism for types and allow to build up (hierarchical) **systems of type declarations** where types may use other types as primitives. If a type T uses a type T' as primitive, this is indicated by

based on T'

in the definition of T; T' is then called a **primitive type** of T. If T' is primitive to T, all entities defined by T' may be used in the specification of T. As an example of a hierarchical type we consider a simple definition of natural numbers which uses BOOL as primitive type:

type NAT_1
  based on BOOL;
  sorts **nat**;
  functs 0: → **nat**, (*zero*)
  succ: **nat** → **nat**, (*successor*)
  .=0: **nat** → **bool**; (*is-zero*)
  axioms ∀ *x*: **nat** ||
  0 =0 ≡ **true**,
  succ(*x*) =0 ≡ **false**
endoftype

The signature of a hierarchical type is simply defined as the union of the newly introduced sort and function symbols with the signatures of the primitive types.

So far, we have tacitly assumed that a hierarchical type may use anything that is in the signature of its primitives. For practical reasons, however, it is important that certain entities defined by some type are hidden to the "outside world". Also, in particular with hierarchical types, we probably do not want to make available everything which is in the signature of its primitives. Therefore, we introduce the (syntactic) notion of (visible) **constituents** of a type as a list of entities in its signature that are made available to other types. These constituents are marked by the keyword **exports**.

Thus, for the example NAT_1, e.g., we might prefer to write

type NAT_1
  exports **nat**, 0, *succ*, .=0;
  based on BOOL;
  ⋮
endoftype

in order to state that any type that uses NAT_1 may only refer to the sort **nat** and the operations 0, *succ*, .=0, although the signature of NAT_1 comprises the signature of BOOL.

Rather than just referring to a primitive type by its name, one also could be more specific by listing explicitly those sorts, constants, and operations that are used from the constituents of the primitive type (cf. [Bauer et al. 87]). Additionally, (partial) **renaming** of the constituents of the primitive type is possible.

Through the notions of primitive type and constituents a relation between the types of a type system is defined. This relation is obviously not reflexive and symmetric. Moreover, it is also not transitive: if T is based on T' and T' is based on T", the constituents of T" may be used in T only if they are included in the constituents of T' or if T" is also indicated as primitive in T. Of course, within a system of type declarations, primitives leading to a cyclic relationship make no sense and, therefore, are forbidden.

A (hierarchical) type T is characterized by a tuple $(\Sigma, E, P_1, \ldots, P_n)$ where, for $1 \leq i \leq n$, the $P_i = (\Sigma_i, E_i)$ denote the primitive types of T with $\Sigma_i \subseteq \Sigma$ and $E_i \subseteq E$. As to the semantics of a hierarchical type, we do not simply take GEN(T), but rather would like to have "hierarchical models", i.e., models in which the hierarchical structure of the type definition is reflected. To this end further properties are needed.

For a hierarchical type T we require that it is **hierarchy-preserving**, i.e. that for all its models A and for every primitive type P_i of T the restriction of A to the signature of P_i is a model of P_i, and thus, in particular, is generated by the operations of A that correspond to the operation symbols of P_i.

Another important property for a hierarchical type T is **persistency** which means that all combinations of models of the primitive types P_i can be extended to a model of T. This guarantees that types may be implemented independent of other types which are based on them. If the primitive types are monomorphic then the type T is either persistent or inconsistent.

The **semantics** of a hierarchy-preserving and persistent (hierarchical) type T then is the family of all isomorphism classes of models of T. According to this definition, "soundness" of a hierarchical type requires proofs on hierarchy-preservation and persistency. Fortunately, however, these semantic properties are implied by the (simpler) syntactic properties of "sufficient completeness" (cf. section 3.4) and "hierarchy-conservativity", resp. For a comprehensive and more elaborate discussion on this topic, cf. [Wirsing 89].

3.4 Instantiation

Looking again at the definition of NAT_1, we realize that this type contains all of the type NAT_0. This purely syntactic relationship can be made explicit by the mechanism of instantiation. **Instantiation** is a syntactic means (differently from hierarchical basing which is defined semantically) for structuring that may appear in a type definition. It is indicated by the keyword **include** and defined by textual substitution (similar to 'macro expansion') of the type body without the list of visible constituents. Using this mechanism in the definition of NAT_1, would result in the definition:

type NAT_2
  exports **nat**, 0, *succ*, .=0;
  based on BOOL;
  include NAT_0;
  functs .=0: **nat** → **bool**;
  axioms $\forall\ x$: **nat** ||
    $0 = 0 \equiv$ **true**, (1)
    $succ(x) = 0 \equiv$ **false** (2)
endoftype

As for hierarchical basing, instantiation may be coupled with renaming, too. Rather than keeping with the names as defined by NAT_0, we also could have used

include NAT_0 **as** (**natural**, *zero*, .+1),

in order to rename all occurences of **nat**, 0, *succ* within NAT_2 into **natural**, *zero*, .+1, respectively. For renaming coupled with instantiation, we also allow abbreviations, e.g.,

include NAT_0 **as** (**natural**, ...)

for renaming just the sort identifier.

3.5 Partial Functions

So far, all functions within our sample type definitions were total. Under certain circumstances, however, one might wish to make clear that certain terms denote erroneous situations or are simply not defined. A simple example is given by the following type

type NAT_3
  exports **nat**, 0, *succ*, *pred*, .=0;
  include NAT_2;
  functs *pred*: **nat** → **nat**; (*predecessor*)
  axioms ∀ x: **nat** ||
    defined(*pred*(x)) ⇒ (x =0) ≡ **false**, (3)
    pred(*succ*(x)) ≡ x (4)
endoftype

which introduces natural numbers with a predecessor operation *pred*. In this example, *pred* is a partial operation which can only have a defined value for arguments x which are not equal to 0. In the type definition this is formalized by means of a special (semantic) predicate **defined** in law (3).

Note, that allowing partial operations in our type definitions does not involve any changes in the semantic definition, since we already introduced the concept of partial Σ-algebras.

When using partial operations in type definitions, some care has to be taken in order not to introduce inconsistencies. For example, adding the axiom

$$succ(pred(x)) \equiv x \tag{3.5.1}$$

to the definition of NAT_3 would result in an inconsistency. On the one hand, according to axiom (3.5.1), we would have $succ(pred(0)) \equiv 0$. On the other hand, due to the definedness axiom on *pred*, $pred(0)$ is undefined and thus, due to the strictness of operations, $succ(pred(0))$ is undefined, too. In order to avoid inconsistencies in connection with a partial operation f, it has to be ensured that either f is applied only to arguments that fulfil the definedness axiom, as in the example NAT_3 above, or applications of f are safe-guarded by means of **conditional axioms**. Thus, for example, adding

$$(x = 0 \equiv \textbf{false}) \Rightarrow (succ(pred(x)) \equiv x) \tag{3.5.2}$$

to NAT_3 above would do no harm, in particular, as it is a property that is provable for NAT_3. As an alternative, semantically equivalent notation for conditional axioms such as (3.5.2) we allow to write

$$succ(pred(x)) \equiv x \textbf{ provided } \neg(x = 0). \tag{3.5.3}$$

In a hierarchical type T = (Σ, E, P) with primitive type P = (Σ_P, E_P) terms may be distinguished with respect to their sort: A Σ-term t is **of primitive sort**, if it is of a sort from Σ_P; otherwise, it is **of non-primitive sort**. Thus, for NAT_3, the term $succ(succ(0))=0$ is of primitive sort, whereas $succ(succ(0))$ is of non-primitive sort.

A hierarchical type T = (Σ, E, P) with P = (Σ_P, E_P) is called **sufficiently complete**, if for every ground term $t \in W(\Sigma)$ of primitive sort either $\neg$ **defined**(t) or $t \equiv p$ for some term $p \in W(\Sigma_P)$ is provable in T. A sufficiently complete type is hierarchy-preserving [Wirsing et al. 83]. A (syntactic) criterion that guarantees sufficient completeness is given in [Guttag, Horning 78].

For a sufficiently complete type the axioms may be used for "evaluating" ground terms of primitive sort. Thus, any question about the behaviour of some specification can be expressed by an appropriate (possibly large) ground term which is then reduced to a primitive term (the "answer" to the question) by a term-rewriting process using the axioms of the type as rewrite rules. In this sense algebraic types may be used for "rapid prototyping" (for details, cf., e.g., [Geser, Hußmann 86]).

Example

Obviously, the example NAT_3 is sufficiently complete: all (visible) ground terms t of primitive sort (here: **bool**) are of the form x =0 where x is a ground term of non-primitive sort; if x is 0 or of the form *succ*(...), t may be reduced to the term **true**, respectively **false**, according to axioms (1) and (2) of NAT_3; if x is of the form *pred*(...), then either x is undefined (according to axiom (3)) and so is t, or x may be reduced to a term of the form 0 or *succ*(...), using the axiom (4) of NAT_3. ■

The use of partial operations (semantically captured by partial Σ-algebras) is but one possibility of coping with the problem of terms denoting erroneous situations. Another possibility is, e.g., error algebras. An elaborate discussion on these various possibilities and their mutual relationships can be found in [Wirsing 89].

3.6 Type Schemes

Types, as introduced above, can be used not only for defining elementary object kinds such as numbers, truth values, characters, but can be used also for specifying composite object kinds.

As an example we consider sequences of natural numbers. A *sequence* (sometimes also called a *list*) is a sequential structure consisting of an arbitrary number of elements (which are natural numbers). A sequence containing no elements is called empty; otherwise, it is called non-empty. A sequence can be extended by adding elements to it. The elements in a non-empty sequence can be accessed sequentially: an operation **first** yields the first element of the sequence; an operation **rest** yields the sequence without the first element.

On closer inspection of this informal description we realize that, for an axiomatic definition, only the sort symbol of the element type has to be known. This means that specifications of sequences of objects other than natural numbers will follow the same pattern. Since a similar phenomenon can be observed with other composite object kinds, it seems appropriate to extend our type mechanism further to allow parametrized types.

Parametrized types are also called **type schemes**. For indicating the parameters in the definition of a parametrized type the keyword **params** is used. In contrast to types, type schemes allow us to express certain structural principles for composite object kinds.

Thus, a type scheme SEQU which gives the essential properties of the sequential composition of objects into a new (composed) object can be formally defined as follows:

type SEQU(**m**)
  params **m: sort;**
  exports **sequ**, <>, . =<>, .≠<>, **first.**, **rest.**, .+.;
  based on BOOL;
  sorts **sequ;**
  functs <>: → **sequ;** (*empty sequence*)
  .=<>, .≠<>: **sequ** → **bool,** (*test on empty sequence*)
  first.: **sequ** → **m,** (*first element*)
  rest.: **sequ** → **sequ,** (*remainder*)
  .+.: (**m** × **sequ**) → **sequ;** (*addition of an element*)
  axioms ∀ x: **m**; s: **sequ** ||
  <> =<> ≡ **true,**
  ($x + s$) =<> ≡ **false,**
  s ≠<> ≡ ¬(s =<>),
  defined(**first** s) ⇒ (s ≠<>) ≡ **true,**
  first($x + s$) ≡ x,
  defined(**rest** s) ⇒ (s ≠<>) ≡ **true,**
  rest($x + s$) ≡ s
endoftype

The (formal) parameter **m** of SEQU is just a sort symbol. In general, arbitrary collections of constituents, i.e. sorts and operations, are allowed. Additionally, these may be constrained by predicates or appropriate degenerate types (cf. below).

As discussed earlier, type schemes may be used in the definition of other types through instantiation. Of course, in the case of a parametrized type, actual parameters have to be supplied which consistently replace the formal ones within an instantiation. Using instantiation, e.g., sequences of natural numbers can be defined by

type NATSEQU
  exports **natsequ**, <>, .=<>, .≠<>, **first.**, **rest.**, .+.;
  based on NAT_3;
  include SEQU(**nat**) **as** (**natsequ**, ...)
endoftype

Type schemes do not have an independent semantics. Using a type scheme in the definition of another type, however, is well-defined through the instantiation mechanism.

3.7 "Degenerate" Types

In general, types (and type schemes) define new object kinds and operations. However, there are also "degenerate" forms of types which define just operations (and no new object kind). Of course, such a type has to be based on some other type T (via **based on**, instantiation, or parametrization). Therefore, it is called an **extension** of T. A simple example of an extension was already provided with the definition of NAT_2.

When using an extension, one might want to clearly indicate which constituents are added through the extension and which are simply taken over from the included type. For this purpose we allow to use (within a list of constituents) **exports** T as an abbreviation of the list of constituents of T. Using this abbreviation, an extension of the type NAT_3 can be given as

type NAT_4
  exports NAT_3, .=., .<.;
  include NAT_3;
  functs .=., . < .: (**nat** × **nat**) → **bool**;
  axioms ∀ x, y: **nat** ||
    $x = 0 \equiv x = 0$,
    $x = y \equiv y = x$,
    $succ(x) = succ(y) \equiv x = y$,
    $x < 0 \equiv$ **false**,
    $0 < succ(x) \equiv$ **true**,
    $succ(x) < succ(y) \equiv x < y$
endoftype

In this way, any operation over some type T can be defined via an appropriate extension of T.

Another instance of degeneration is given by types which define neither new object kinds nor new operations, but only additional properties. A typical example of such a type is the type scheme [Bauer et al. 85]:

type EQUIV(**m**, *eq*)
  params **m**: **sort**,
    eq: (**m** × **m**) → **bool**;
  based on BOOL;
  axioms ∀ x, y, z: **m** ||
    $eq(x,y) \equiv$ **true** ∨ $eq(x,y) \equiv$ **false**, (*totality*)
    $eq(x,x) \equiv$ **true**, (*reflexivity*)
    $eq(x,y) \equiv eq(y,x)$, (*symmetry*)
    $(eq(x,y) \equiv$ **true** ∧ $eq(y,z) \equiv$ **true**)
      ⇒ $(eq(x,z) \equiv$ **true**) (*transitivity*)
endoftype

which states that a total binary predicate *eq* on **m** is an equivalence relation. For **nat** and "=" defined by NAT_4, obviously EQUIV(**nat**, =) is provable.

Since type definitions of this degenerate kind can be viewed as abbreviations for collections of axioms, they can conveniently be used for restrictions on the parameters of some type or for a compact formulation of laws.

4 More on Algebraic Types

So far we have introduced all the basic concepts of algebraic types. For the formal specification of problems, however, it is convenient to have more basic types and also some additional syntactic sugar. This will be the topic of this section.

4.1 Further Examples of Basic Algebraic Types

Like sequences, other composite structures can be defined by algebraic types. Examples are *finite sets, bags,* and *finite mappings.* Further examples can be found in the literature (e.g. [Bauer et al. 81, 85], [Partsch 90]).

Finite sets. Many problems can be specified in a straightforward way using the concept of finite sets. *Finite sets* differ from sequences in two respects. They do not have multiple occurrences of elements and the ordering of elements is irrelevant. Finite sets can be specified by the (monomorphic) type scheme

type SET(**m**, *eq*)
  params **m**: **sort**,
    eq: (**m** × **m**) → **bool**
      constrained by EQUIV(**m**, *eq*);
  exports **set**, $\emptyset$, {.}, .∪., .−., .∈., .∉.;
  based on BOOL;
  sorts **set**;
  functs $\emptyset$: → **set**, (*empty set*)
    {.}: **m** → **set**, (*singleton set former*)
    .∪.: (**set** × **set**) → **set**, (*set union*)
    .−.: (**set** × **m**) → **bool**, (*deletion of an element*)
    .∈., .∉.: (**m** × **set**) → **bool**; (*membership, non-membership*)
  axioms ∀ x, y: **m**; s, t, u: **set** ||
    $s \cup \emptyset \equiv s$, (*neutrality of* $\emptyset$ *w.r.t.* ∪)
    $s \cup t \equiv t \cup s$, (*commutativity of* ∪)
    $(s \cup t) \cup u \equiv s \cup (t \cup u)$, (*associativity of* ∪)
    $s \cup s \equiv s$, (*idempotency*)
    $y \in \emptyset \equiv$ **false**,
    $y \in \{x\} \equiv eq(x, y)$,
    $y \in (s \cup t) \equiv (y \in s) \vee (y \in t)$,
    $x \notin s \equiv \neg(x \in s)$,
    $(\emptyset - x) \equiv \emptyset$,
    $x - y \equiv$ **if** $eq(x, y)$ **then** $\emptyset$ **else** $\{x\}$ **fi**
    $(s \cup t) - y \equiv (s - y) \cup (t - y)$
endoftype

Parameters are here an object kind **m** and a binary predicate *eq* on **m**, which is constrained to have the properties of an equivalence relation.

A parameter constraint (indicated by the keyword **constrained by**) is a closed, well-formed first-order formula over the parameter symbols and/or instantiations of degenerate types abbreviating collections of axioms.

Also, some "syntactic sugar" is used in the definition of SET. For example, an axiom such as

$\{x\} - y \equiv$ **if** $eq(x, y)$ **then** $\emptyset$ **else**{x} **fi**

is just shorthand for the pair of axioms

$eq(x, y) \equiv$ **true** $\Rightarrow \{x\} - y \equiv \emptyset$,
$eq(x, y) \equiv$ **false** $\Rightarrow \{x\} - y \equiv \{x\}$.

Bags. Many problems conveniently can be specified on the basis of bags. *Bags*, sometimes also called 'multisets', can be defined by the type scheme

type BAG(**m**, *eq*)
params **m**: **sort**,
eq: (**m** × **m**) → **bool**
constrained by EQUIV(**m**, *eq*);
exports **bag**, $\emptyset$, .+., .∈., . ∉., .−., ♯*occs*;
based on BOOL, NAT_4;
sorts **bag**;
functs $\emptyset$: → **bag**, (*empty bag*)
.+., .−.: (**bag** × **m**) → **bag**, (*addition, deletion of elements*)
.∈., .∉.: (**m** × **bag**) → **bool**, (*membership, non-membership*)
♯*occs*: (**m** × **bag**) → **nat**; (*number of occurrences*)
axioms ∀ *x*, *y*: **m**; *b*: **bag** ||
$y \in \emptyset \equiv$ **false**,
$y \in (b + x) \equiv$ **if** $eq(x, y)$ **then true else** $y \in b$ **fi**,
$x \notin b \equiv \neg (x \in b)$,
$\emptyset - x \equiv \emptyset$,
$(b + x) - y \equiv$ **if** $eq(x, y)$ **then** b **else** $(b - y) + x$ **fi**,
♯$occs(x, \emptyset) \equiv 0$,
♯$occs(y, b + x) \equiv$ **if** $eq(x, y)$ **then** $succ($♯$occs(y, b))$ **else** ♯$occs(y, b)$ **fi**
endoftype

An alternative definition of bags with the operations $\emptyset$, {.}, and .∪. analogous to the definition of SET is obvious and left as an exercise to the interested reader.

Note, however, that in contrast to SET, BAG is not monomorphic. If a definition is wanted where for all models addition of an element to a bag is commutative and/or associative, appropriate laws have to be added.

Finite Mappings. *Finite mappings* associate finitely many elements of an index set with values. They can be defined by the following type scheme:

type MAP(**ind**, **m**, *eq*);
  params **ind**, **m**: **sort**,
    eq: (**ind** × **ind**) → **bool**
      constrained by EQUIV(**ind**, *eq*);
  exports **map**, [], .[.]←., *isdef*, .[.];
  based on BOOL;
  sorts **map**;
  functs []: → **map**, (*empty map*)
    .[.]←.: (**map** × **ind** × **m**) → **map**, (*"updating"*)
    isdef: (**map** × **ind**) → **bool**, (*definedness-test*)
    .[.]: (**map** × **ind**) → **m**; (*indexed access*)
  axioms ∀ *i*, *j*: **ind**; *m*: **map**; *x*: **m** ||
    isdef[], *i*) ≡ **false**,
    isdef[*m*[*i*]←*x*, *j*) ≡ **if** *eq*(*i*, *j*) **then true else** *isdef*[*m*, *j*) **fi**,
    defined(*m*[*i*]) ⇒ *isdef*[*m*, *i*) ≡ **true**,
    (*m*[*j*]←*x*)[*i*] ≡ **if** *eq*(*i*, *j*) **then** *x* **else** *m*[*i*] **fi**
endoftype

Parameters are here two object kinds **ind** and **m**, as well as a binary predicate *eq* on **ind**, having the properties of an equivalence relation.

Rather than checking definedness explicitly, as is done by the operation *isdef* in the type MAP, it is sometimes convenient to have "total maps", which are finite maps with a pre-defined value for each index. Further variants for finite mappings, e.g. with more than one index, are obvious. As an example, we give a definition of "total square matrices":

type MATRIX(**ind**, **m**, *eq*)
  params **ind**, **m**: **sort**,
    eq: (**ind** × **ind**) → **bool**
      constrained by EQUIV(**ind**, *eq*);
  exports **matrix**, *init*, .[., .]←., .[., .];
  based on BOOL;
  sorts **matrix**;
  functs *init*: **m** → **matrix**, (*initialization*)
    .[., .]←.: (**matrix** × **ind** × **ind** × **m**) → **matrix**, (*"updating"*)
    .[., .]: (**matrix** × **ind** × **ind**) → **m**; (*indexed access*)
  axioms ∀ *i*, *j*, *k*, *l*: **ind**; *m*: **matrix**; *x*: **m** ||
    init(*x*)[*i*, *j*] ≡ *x*,
    (*m*[*i*, *j*]←*x*)[*k*, *l*] ≡ **if** *eq*(*i*, *k*) ∧ *eq*(*j*, *l*) **then** *x* **else** *m*[*k*, *l*] **fi**
endoftype

4.2 Extensions of Basic Types

The types as defined in the examples above contain only a few operations. For practical purposes, however, a richer set of operations often allows a much more flexible and adequate formalization. Such a richer set simply may be defined as an extension (cf. section 3.7) of an existing basic type using the instantiation mechanism.

A typical example is the following definition of indexed sequences as an extension of the type SEQU:

type INDSEQU(**m**)
  params **m: sort**;
  exports SEQU, |.|, .[.], .[. : .], $.^{-1}$;
  based on NAT_3 **as** (**nat**, 0, .+1, .−1, .=0);
  include SEQU(**m**);
  functs |.|: **sequ** → **nat**, (*length*)
  .[.]: (**sequ** × **nat**) → **m**, (*indexed access*)
  .[. : .]: (**sequ** × **nat** × **nat**) → **sequ**, ("*slicing*")
  $.^{-1}$: **sequ** → **sequ**; (*reversal*)
  axioms ∀ *x*: **m**; *r, s, t*: **sequ**; *i, k*: **nat** ||
  $|<>| \equiv 0$,
  $|x + s| \equiv |s| + 1$,
  $s[1] \equiv$ **first** s,
  $s[i] \equiv$ (**rest** s)$[i-1]$ **provided** $1 < i$,
  $s[i : k] \equiv$ **if** $i > k$ **then** <> **else** $s[i] + s[i+1 : k]$ **fi**
  provided $1 \leq i, k \leq |s|$,
  $s^{-1} \equiv$ **if** $s =<>$ **then** <> **else** $s[|s|] + (s[1 : |s| - 1])^{-1}$ **fi**
endoftype

Of course, INDSEQU again could be extended by further operations. Also extensions of SEQU by other operations are obvious, as are extensions of other basic data types such as sets, bags, or maps. For examples, cf. [Partsch 90].

If we extended our type formalism to also allow 'higher-order types' [Möller 87], we also could define an extension of SEQU that comprises the (higher-order) operations and predicates from [Bird 87]. Thus, within the algebraic formalism, we could also profit from the well-known advantages of using higher-order operations and predicates to express commonalities and generic aspects in a concise, abstract way. However, such an extension to higher-order types would also require a (slightly) more complicated theory. Therefore, a detailed treatment is not included in this tutorial text.

4.3 Modes

Certain types and type schemes, such as Cartesian product and direct sum, occur so frequently that it is reasonable to introduce particular shorthand notations called **modes** [Bauer et al. 85]. Syntactically, modes are introduced by a **mode declaration.** Their semantics is defined via instantiations of the associated type schemes.

A **product** specifies objects that are composed of a finite number k>0 of other objects, called **components**, together with operations for *construction* and *selection.*

A special case of products are pairs. Arbitrary object kinds **m** and **m'** can be combined into pairs by using the type scheme

type PAIR(**m**, **m′**);
  params **m**, **m′**: **sort**;
  exports **pair**, *mk*, *sel*, *sel′*
  sort **pair**;
  functs *mk*: (**m** × **m′**) → **pair**, (*pairing*)
    sel: **pair** → **m**, (*selection of first component*)
    sel′: **pair** → **m′** (*selection of second component*)
  axioms ∀ *x*: **m**; *x′*: **m′** ||
    $sel(mk(x, x')) \equiv x$,
    $sel'(mk(x, x')) \equiv x'$
endoftype

The type scheme PAIR is monomorphic relative to its parameter sorts. The constructor operation *mp* – like all operations defined by types – is strict in its arguments. As an abbreviation for the above type definition and its use in

include PAIR(**m**, **m′**) **as** (**pair**, *mp*, *s*, *s′*) (4.3.1)

we now introduce the mode declaration

mode **pair** = *mp*(*s*: **m**, *s′*: **m′**). (4.3.2)

The generalization of (4.3.1) and (4.3.2) to an arbitrary product PRODUCT_k is obvious.

The pairs as introduced by (4.3.2) provide, except for construction and selection, no further operations. Frequently, however, at least the *induced equality*, i.e. componentwise equality, is wanted, provided equalities *eq* and *eq′* on the component types are available. In order to define pairs with induced equality, we simply extend the type scheme PAIR by an operation

equ: (**pair** × **pair**) → **bool**

defined by (∀ *x*, *y*: **m**; *x′*, *y′*: **m′**)

$$equ(mp(x, x'), mp(y, y')) \equiv eq(x, y) \wedge eq'(x', y')$$

and use

mode **pair**(*equ*) = *mp*(*s*: **m**(*eq*), *s′*: **m′**(*eq′*))

as a shorthand notation for the instantiation of the extended definition of pairs (cf. [Bauer et al. 87]).

The (direct) **sum** specifies the disjoint union of a finite number k > 0 of carrier sets, which are called **variants** of the sum. In addition to *injection* and (partially defined) *projection* operations, which are analogous to the constructor and selector operations in products, one also needs *discriminating predicates.*

The sum of 2 carriers is introduced by

mode **m** = *v*(*p*: **m**) | *v′*(*p′*: **m′**) (4.3.3)

which is defined to be an abbreviation for

include SUM_2(**m**, **m**′) **as** (**sum**, v, **.is** v, p, v', **.is** v', p') (4.3.4)

where SUM_2(**m**, **m**′) is defined as follows:

type SUM_2(**m**, **m**′)
params **m**, **m**′: **sort**;
exports **s**, $mk, ismk, pr, mk', ismk', pr'$;
sorts **s**;
functs mk: **m** → **s**, (*injection*)
mk': **m**′ → **s**,
$ismk, ismk'$: **s** → **bool**, (*discrimination*)
pr: **s** → **m**, (*projection*)
pr': **s** → **m**′
axioms ∀ x : **m**; x': **m**′ ||
defined($pr(x)$) ⇒ $ismk(x)$ ≡ **true**,
$ismk(mk(x))$ ≡ **true**,
$ismk'(mk(x))$ ≡ **false**,
$pr(mk(x))$ ≡ x,
defined($pr'(x')$) ⇒ $ismk'(x')$ ≡ **true**,
$ismk'(mk'(x'))$ ≡ **true**,
$ismk(mk'(x'))$ ≡ **false**,
$pr'(mk'(x'))$ ≡ x'
endoftype

Again, the generalization of (4.3.3) and (4.3.4) to an arbitrary number of variants is obvious. As $\mathrm{PRODUCT}_k$, the scheme SUM_k is monomorphic relative to the parameter sorts.

Example
A typical example for a sum mode declaration is

mode result = *error*(*message*: **string**) | *correct*(*res*: **int**). ■

In connection with the sum mode it is also allowed to have nullary variants (without projections), which are variants which define new constant symbols.

Example
A typical example of this kind is

mode color = *red* | *blue* | *green*

that introduces a new object kind **color** consisting of the constants *red*, *blue*, and *green*, resp. ■

Furthermore, sums may also be quasi-ordered such that all elements of one variant are preceded in the quasi-ordering by all elements of another variant. For details see [Bauer et al. 85].

Similar to products, sums can be formally extended by an equality in a straightforward way. Since, however, this equality simply coincides with the equality on the variants of the sum, we do not introduce a particular notation.

The combined use of product and sum also gives meaning to **recursive mode declarations.**

Example
By the above definitions,

mode nat = 0 | *succ(pred:* **nat**)

is equivalent to the definition of NAT_3 as given in section 3.5 (with **.is** 0 for .=0).■

As a further notational device in connection with modes, **submodes** may be used as a convenient shorthand notation for expressing restrictions on objects. The meaning of submode declarations such as

mode month = (x: **nat** || $1 \leq x \leq 12$)

is intuitively clear. For a formal definition see again [Bauer et al. 85].

In contrast to [Bauer et al. 85], we will also use modes for abbreviating type instantiations. Thus,

mode m = T(**n**) (4.3.5)

is defined to be an abbreviation for

include T(**n**) **as** (**m**, ...). (4.3.6)

Example
Using (4.3.5) and (4.3.6), sequences of natural numbers thus could have been defined simply by

mode nat = NAT_4;
mode natsequ = SEQU(**nat**, =). ■

4.4 Formulation of Concepts by Algebraic Types and Modes

In the following we are going to exemplify how the basic types and their extensions may be used for the representation of (mathematical) concepts (cf. section 2.1). We will confine ourselves to the concepts "finite directed graph" and "cube". From these examples the representation of other concepts as (systems of) algebraic types and/or modes should be straightforward.

Finite Directed Graphs. Assuming a type NODE with sort **node** and equality test $=$, a basic constructive definition for *finite directed graphs* can be given as follows (cf. [Bauer et al. 89]):

type DGRAPH
  exports **dgraph**, *eg*, *inc*, *isarc*;
  based on NODE, BOOL;
  sort **dgraph**;
  functs *eg*: $\rightarrow$ **dgraph**, (*empty graph*)
    inc: (**dgraph** $\times$ **node** $\times$ **node**) $\rightarrow$ **dgraph**, (*addition of connected nodes*)
    isarc: (**dgraph** $\times$ **node** $\times$ **node**) $\rightarrow$ **bool**; (*test on edges*)
  axioms $\forall$ *g*: **dgraph**; *x*, *y*, *u*, *v*: **node** ||
    $isarc(eg, x, y) \equiv$ **false**,
    $isarc(inc(g, x, y), u, v) \equiv (((x = u) \wedge (y = v)) \vee isarc(g, u, v))$
endoftype

This specification of DGRAPH is based on a fixed set of nodes (defined by NODE). A specification of finite directed graphs for different kinds of nodes as well as various extensions can be found in [Partsch 90].

Based on DGRAPH, it is possible to give definitions of the representations of graphs as exemplified in section 2.1. Thus, e.g., a representation of finite directed graphs according to (2.1.2) is as follows:

type DGRAPH′
  exports DGRAPH, **nodeset**, *in*, *out*;
  include DGRAPH,
    SET(**node**, $=$) **as** (**nodeset**, ...);
  functs *in*, *out*: (**node** $\times$ **dgraph**) $\rightarrow$ **nodeset**; (*predecessors, successors*)
  axioms $\forall$ *g*: **dgraph**; *x*, *y*, *z*: **node** ||
    $in(x, eg) \equiv \emptyset$,
    $in(x, inc(g, y, z)) \equiv$ **if** $x = z$ **then** $in(x, g) \cup \{y\}$ **else** $in(x, g)$ **fi**,
    $out(x, eg) \equiv \emptyset$,
    $out(x, inc(g, y, z)) \equiv$ **if** $x = y$ **then** $out(x, g) \cup \{z\}$ **else** $out(x, g)$ **fi**
endoftype

Similarly, definitions for the other two representations could be given. Thus, e.g., a definition for the representation (2.1.3) might use an appropriate instantiation of the type MATRIX:

type DGRAPH″
  exports **dgraph**, *eg*, *inc*, *isarc*;
  based on NODE, BOOL;
  include MATRIX(**node**, **bool**, =) **as** (**dgraph**, ...);
  functs *eg*: → **dgraph**, (*empty graph*)
  inc: (**dgraph** × **node** × **node**) → **dgraph**, (*addition of connected nodes*)
  isarc: (**dgraph** × **node** × **node**) → **bool**; (*test on edges*)
  axioms ∀ *g*: **dgraph**; *x*, *y*: **node** ||
  eg ≡ *init*(**false**),
  inc(*g*, *x*, *y*) ≡ *g*[*x*, *y*]←**true**,
  isarc(*g*, *x*, *y*) ≡ *g*[*x*, *y*]
endoftype

A definition for the representation (2.1.1) can be given simply using mode declarations:

mode nodeset = SET(**node**, =);
mode edge(=) = *me*(*in*: **node**(=), *out*: **node**(=));
mode edgeset = SET(**edge**, =),
mode graph = *mg*(*nodes*: **nodeset**, *edges*: **edgeset** ||
  ∀ *e*: **edge** || *e* ∈ *edges* ⇒ *in*(*e*) ∈ *nodes* ∧ *out*(*e*) ∈ *nodes*)

In all these definitions, only few operations (on graphs) are defined. Of course, it is possible to extend these definitions by arbitrary operations. Examples can again be found in [Partsch 90].

Cubes. As another example for illustrating the formalization of concepts as algebraic types, we consider (the combinatoric properties of) the notion "cube".

Intuitively, with respect to the notion of a cube, three different kinds of entities are involved: faces, edges, and vertices. Accordingly, depending on whether we consider faces, edges, or vertices as primitive entities, different representations of the concept cube will emerge from relating the respective primitives. The obvious possibilities are

- 6-tuples of related faces
- 12-tuples of related edges
- 8-tuples of related vertices.

In the sequel we will concentrate on the first possibility. Specifications based on the other ones follow similar patterns.

Assuming the availability of a type FACE which defines a sort **f** (for "face") and an equality = on objects of sort **f**, the combinatoric properties of cubes can be axiomatically defined by 6-tuples of related faces as follows:

type F-CUBE
  exports FACE, *iscube*;
  based on BOOL;
  include FACE;
  functs $.\|.: (\mathbf{f} \times \mathbf{f}) \to \mathbf{bool}$, (*parallel*)
  $.\perp.: (\mathbf{f} \times \mathbf{f}) \to \mathbf{bool}$, (*perpendicular to*)
  iscube: $(\mathbf{f} \times \mathbf{f} \times \mathbf{f} \times \mathbf{f} \times \mathbf{f} \times \mathbf{f}) \to \mathbf{bool}$;
  axioms $\forall\ a, b, c, d, e, f$: **f** ||
  include EQUIV(**f**, .||.), (*1*)
  $a \perp b \equiv \neg(a \parallel b)$, (*2*)
  $(a \perp b \wedge a \perp c \wedge b \perp c \equiv \mathbf{true}) \Rightarrow$
  $(d \parallel a \vee d \parallel b \vee d \parallel c \equiv \mathbf{true})$, (*3*)
  $(a = b \equiv \mathbf{true}) \Rightarrow (a \parallel b \equiv \mathbf{true})$, (*4*)
  $iscube(a, b, c, d, e, f) \equiv$ (*5*)
  $\neg(a = b) \wedge \neg(c = d) \wedge \neg(e = f) \wedge$
  $a \parallel b \wedge c \parallel d \wedge e \parallel f \wedge a \perp c \wedge a \perp e \wedge c \perp e$
endoftype

The sufficient completness of this definition is obvious: By the axioms of EQUIV, any subterm of the form a || b can be reduced to either **true** or **false**. Since, moreover, any subterm of the form a ⊥ b can be reduced to a term over (a || b), the axioms of BOOL guarantee that arbitrary terms (over .||. and . ⊥ .) are reducible to either **true** or **false**.

Whether this formalization really captures the intuitive notion of (the combinatoric properties of) a cube is a different question which cannot be answered formally, but only made plausible. To this end, from this definition additional properties can be derived which profitably can be used either for checking the adequacy of the formal specification or in program development. Examples are:

Lemma 1
For faces *a, b, c, d, e, f* we have:
a. $(a \parallel b \wedge a \perp c \equiv \mathbf{true}) \Rightarrow (b \perp c \equiv \mathbf{true})$;
b. $iscube(a, b, c, d, e, f) \Rightarrow |\{a, b, c, d, e, f\}| = 6$;
c. $iscube(a, b, c, d, e, f) \equiv iscube(c, d, a, b, e, f)$;
d. $iscube(a, b, c, d, e, f) \equiv iscube(b, a, c, d, e, f)$;
e. $iscube(a, b, c, d, e, f) \equiv iscube(c, d, e, f, a, b)$.

Proof: Straightforward from the axioms and basic rules of logic. ■

Lemma 2
For elements *a, b, c, d, e, f* of type **f** there are exactly 15 equivalence classes with respect to *iscube*.

Proof:
First we show that all possible tuples of arguments for *iscube* can be represented by at most 15 tuples formed out of *a, b, c, d, e, f*:
From lemma 1 it follows that for all tuples starting with an element different from *a* there is an equivalent one (with respect to *iscube*) starting with *a*. Thus,

it is sufficient to consider tuples starting with *ab, ac, ad, ae, af.* Likewise it follows that for all tuples starting with *abd, abe, abf* there is an equivalent one starting with *abc.* By analogous reasoning we conclude that for all tuples there are equivalent ones starting with *abc, acb, adb, aeb, afb.* Once more using the same reasoning we finally find out that the tuples

a, b, c, d, e, f	*a, c, b, d, e, f*	*a, d, b, c, e, f*	*a, e, b, c, d, f*	*a, f, b, c, d, e*
a, b, c, e, d, f	*a, c, b, e, d, f*	*a, d, b, e, c, f*	*a, e, b, d, c, f*	*a, f, b, d, c, e*
a, b, c, f, d, e	*a, c, b, f, d, e*	*a, d, b, f, c, e*	*a, e, b, f, c, d*	*a, f, b, e, c, d*

are sufficient to represent all possible argument tuples to *iscube.* Finally, it is easy to see that none of these tuples is equivalent to another (with respect to *iscube*), which concludes the proof. ■

Lemma 3
For faces *a, b, c, d, e, f* which are mutually different w.r.t. =, we have:

$$iscube(a, b, c, d, e, f) \Leftrightarrow$$
$$a \perp c \wedge a \perp e \wedge c \perp e \wedge b \perp c \wedge b \perp e \wedge d \perp e \wedge$$
$$a \perp d \wedge a \perp f \wedge c \perp f \wedge b \perp d \wedge b \perp f \wedge d \perp f$$

Proof: Straightforward from the definition of *iscube* and lemma 1. ■

5 Examples

In this section we would like to illustrate how to use the specification formalism introduced in the previous sections by means of somewhat more complex examples.

5.1 The Bounded Buffer

As part of a simple system of communicating agents, we consider the problem of specifying the behaviour of a "bounded buffer". The problem is as follows:

> *There is a buffer of restricted length. Information can be sent to the buffer for storing. If the buffer is not full, the information is stored and an OK-message is given. Information can also be retrieved from the buffer according to priorities tagged to the information. Storing (retrieval) results in issuing an error message, if the buffer is full (empty).*

In order to formally specify the problem, we assume to have available a type INFO which defines an object kind **info** (of information) and an operation *pty*: **info** → **nat** which assigns a priority to an information. Disregarding the problem of boundedness, the intended buffer obviously behaves like a priority queue (of objects of type **info**) which may be specified by

type PQUEUE
exports **pqueue**, *put*, *max*, *rest*, *init*, |.|;
based on INFO, NAT_4 **as** (**nat**, 0, .+1, .−1,.,.,.);
sorts **pqueue**,
functs *init*: → **pqueue**,
put: (**pqueue** × **info**) → **pqueue**,
max: **pqueue** → **info**,
rest: **pqueue** → **pqueue**,
|.|: **pqueue** → **nat**;
axioms ∀ *q*: **pqueue**; *i*: **info** ||
$|init| \equiv 0$,
$|put(q,i)| \equiv |q| + 1$,
defined$(max(q)) \Rightarrow (0 < |q|) \equiv$ **true**,
$max(put(init,i)) \equiv i$,
$max(put(q,i)) \equiv$ **if** $pty(max(q)) < pty(i)$ **then** i **else** $max(q)$ **fi**
provided $0 < |q|$,
defined$(rest(q)) \Rightarrow (0 < |q|) \equiv$ **true**,
$rest(put(init,i)) \equiv init$,
$rest(put(q,i)) \equiv$ **if** $pty(max(q)) < pty(i)$ **then** q **else** $put(rest(q),i)$ **fi**
provided $0 < |q|$
endoftype

Taking boundedness into account, the intended buffer can be specified in terms of priority queues as follows:

mode buffer = (q: **pqueue** || $|q| < N \vee |q| = N$);

According to the informal description, input to the buffer is either information to be stored or an attempt to retrieve stored information. Thus, a single piece of input to the buffer can be specified by

mode inelem = *sto*(*inf*: **info**) | *retr*.

Likewise, a single piece of output from the buffer is either an OK-message, an error message or an information. This can be formalized by

mode outelem = *OK* | *s-error* | *r-error* | *mk-res*(*inf*: **info**).

The behaviour of the intended buffer is completely characterized by specifying a function *buffer* which maps an arbitrary sequence of input elements onto a sequence of output elements. Thus, altogether, we have the following specification:

type BUFFER(N)
params N: **nat**;
exports **input, output,** *buffer;*
based on INFO, PQUEUE, SEQU, NAT_4;
mode inelem = *sto*(*inf*: **info**) | *retr*;
mode input = SEQU(**inelem**);
mode outelem = *OK* | *s-error* | *r-error* | *mk-res*(*inf*: **info**);
mode output = SEQU(**outelem**);
mode buffer = (q: **pqueue** || $|q| < N \vee |q| = N$);
functs *buffer*: **input** → **output**,
buff: (**input** × **buffer**) → **output**;
axioms ∀ *in*: **input**; b: **buffer**; i: **info** ||
buffer(*in*) ≡ *buff*(*in*, *init*),
buff(<>, b) ≡ <>,
buff(*sto*(i) + *in*, b) ≡ **if** $|b| < N$ **then** *OK* + *buff*(*in*, *put*(b, i))
else *s-error* + *buff*(*in*, b) **fi**,
buff(*retr* + *in*, b) ≡ **if** $|b| > 0$ **then** *mk-res*(*max*(b))
+*buff*(*in*, *rest*(b))
else *r-error* + *buff*(*in*, b) **fi**
endoftype

5.2 The "Cube Problem"

An informal statement of the problem was given in section 1. Following the methodological guidelines from section 2, formalizing the problem requires formalization of the problem domain, i.e., input, output, and constituents of the problem, as well as formalizing the problem proper as an expression in terms of the formalization of the problem domain. Each of these aspects will be separately looked at in turn, assuming the availability of basic concepts such as bags, indexed sequences or tuples, for which an intuitively obvious notation will be used as long as we are reasoning on the conceptual level. In our subsequent treatment of the problem, in particular, the influence of various design decisions during formalization will be commented on.

Input. In our attempt to formalize the input to the problem, we follow a top-down approach. We start from a first rough approximation of the conceptual data structures involved and then gradually refine them until we reach a sufficiently detailed and adequate description.

Input
In a most general view, according to the informal description given in section 1, the "input" to the problem consists of a bag of 6 pieces, i.e.

pbag $=_{\text{def}}$ (B: **bag of piece** || $|B| = 6$),

assuming the availability of a primitive type **piece** that formally specifies the individual pieces of the puzzle. Additionally, we know that the surface area covered by all pieces

when arranged in a cube has to be $6 \times 5^2 = 150$:

$$\Sigma_{p \in B}\ \textit{area-covered-by}(p) = 150.$$

Thus, together, as a first approximation, we specify the input to the problem by

$$\textbf{pbag} =_{\text{def}} (B\text{: } \textbf{bag of piece} \parallel |B| = 6 \wedge \Sigma_{p \in B}\ \textit{area-covered-by}(p) = 150).$$

For a further, more adequate refinement, a detailed specification of **piece** is needed.

Individual pieces

In order to formally describe the pieces, there are obviously different possibilities. Irrespective of these, however, all pieces have common characteristics which need not be specified explicitly and may be considered invariants. Each piece has:

- 4 sides, perpendicular to each other;
- a fixed "kernel" of size $3^2 = 9$ square units;
- a thickness of 1 unit; and
- upto 5 "teeth" per side, each of size 1 square unit.

According to these common characteristics, the area covered by a single piece p is composed of the size of the kernel and the areas covered by the teeth. With respect to the latter, we further have to distinguish between "middle teeth" (which contribute to the total area of the cube in two dimensions) and "corner teeth" (contributing to three dimensions). Assuming the availability of appropriate auxiliary operations, the area covered by a single piece thus can be defined by

$$\textit{area-covered-by}(p) =_{\text{def}} 9 + 2 \times \textit{middle-teeth}(p) + 3 \times \textit{corner-teeth}(p).$$

The individual "geometry" of each piece is given by the presence, resp. absence, of "teeth" at its sides. Thus, a straightforward formalization of pieces might be given by

$$\begin{aligned} &\textbf{side} =_{\text{def}} (s\text{: } \textbf{indexedsequ of bool} \parallel |s| = 5), \\ &\textbf{piece} =_{\text{def}}\ \textit{mp}(s_1\text{: } \textbf{side}, s_2\text{: } \textbf{side}, s_3\text{: } \textbf{side}, s_4\text{: } \textbf{side}) \end{aligned} \tag{5.2.1}$$

(or, equivalently: **piece** = (p: **indexedsequ of side** $\parallel |p| = 4$), implicitly assuming a fixed orientation of the sides, e.g. clockwise).

However, this straightforward formalization does not take into account that two consecutive sides share a common corner, i.e., for each side s_i of a piece p

$$s_i[5] = s_{(i \textbf{ mod } 4)+1}[1] \tag{5.2.2}$$

has to be required. Furthermore, in order to be mechanically stable, there must not be "isolated teeth" at the corners, i.e., for each side s_i of a piece p

$$s_i[5] \Rightarrow s_i[4] \vee s_{(i \textbf{ mod } 4)+1}[2] \tag{5.2.3}$$

has to be required, too.

Thus, rather than using pieces as defined in (5.2.1), we need

$$\textbf{rpiece} =_{\text{def}} (p\text{: } \textbf{piece} \parallel \mathit{isagp}(p)) \text{ where} \qquad (5.2.4)$$
$$\mathit{isagp}(p) =_{\text{def}}$$
$$\forall_{1 \le i \le 4}\ (p.s_i[5] = p.s_{(i \bmod 4)+1}[1]) \wedge (p.s_i[5] \Rightarrow p.s_i[4] \vee p.s_{(i \bmod 4)+1}[2]).$$

Here, the well-known dot notation is used to denote the selection of the components of a tuple. If selection is used in conjunction with indexing or slicing, it is assumed that selection has higher priority.

The simple closed forms of the properties (5.2.2) and (5.2.3) are obviously a consequence of our decision to assume a fixed orientation of the sides of a piece. Otherwise, explicit conditions on all pairs of adjacent sides would have been needed. On the basis of this definition we are now also in a position to formalize the auxiliary functions *middle-teeth and corner-teeth*, e.g. by

$$\mathit{middle\text{-}teeth}(p) =_{\text{def}} \Sigma_{i=1..4}\ (\mathit{btn}(p.s_i[2]) + \mathit{btn}(p.s_i[3]) + \mathit{btn}(p.s_i[4])),$$
$$\mathit{corner\text{-}teeth}(p) =_{\text{def}} \Sigma_{i=1..4}\ \mathit{btn}(p.s_i[1]),$$
$$\mathit{btn}(b) =_{\text{def}} \textbf{if } b \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi}.$$

Furthermore, definitions of functions to access side i ($1 \le i \le 4$) of a piece or its left corner, its middle part, and its right corner, which will be needed to formalize that two sides fit together, are straightforward:

$$\mathit{full\text{-}side}(i, p) =_{\text{def}} p.s_i,$$
$$\mathit{left\text{-}corner}(i, p) =_{\text{def}} p.s_i[1],$$
$$\mathit{middle\text{-}part}(i, p) =_{\text{def}} p.s_i[2 : 4],$$
$$\mathit{right\text{-}corner}(i, p) =_{\text{def}} p.s_i[5].$$

Summing up, in order to formally describe the input to our problem, we may use

$$\textbf{pbag} =_{\text{def}} (B\text{: } \textbf{bag of rpiece} \parallel\!|B| = 6 \wedge \Sigma_{p \in B}\ \mathit{area\text{-}covered\text{-}by}(p) = 150) \qquad (5.2.5)$$

with definitions of **rpiece** and the auxiliary functions as given above.

Other definitions for individual pieces
Of course, there are alternative representations of pieces. Examples are:

- representation by corners (of unit size) and (the middle parts of) sides (of size 3). Here, due to considering corners as individual entities, an equivalent of requirement (5.2.2) is not explicitly needed. However, access to a side or its parts has to be changed.
- representation by sides (of size 4), assuming that each corner (of unit size) is uniquely attached to one of its adjacent sides. Here too, an explicit equivalent of requirement (5.2.2) is not needed, but again access to a side and its constituents has to be changed.
- representation by (the very middle tooth of) sides (of unit size) and corners (of size 3). Again, an explicit equivalent of requirement (5.2.2) is not needed. Moreover, requirement (5.2.3) is already given in the definition of corner. However, accessing a side or its parts becomes somewhat more difficult.

"Views"

So far we did not pay attention to the fact that the description of a piece depends on the way we look at it. Obviously, by rotating a piece (clockwise or anti-clockwise) by 90 degrees it remains the same, although its description changes – irrespective of which formalization is used. Similarly, flipping a piece (vertically or horizontally) changes its representation, but, of course, not the piece itself.

As a straightforward approach to formally describe all these various RviewsS of a piece, we may use generating functions *id* ("identity"), *r* ("rotate"), *f* ("flip"), each of functionality

rpiece → rpiece,

which are described by the following properties (with ∘ denoting function composition and assuming that no other equalities hold than those explicitly stated):

$$\begin{aligned} &r \circ r \circ r \circ r = id, \\ &f \circ f = id, \\ &r \circ id = id \circ r = r, \\ &f \circ id = id \circ f = f, \\ &r \circ f = f \circ r \circ r \circ r. \end{aligned} \tag{5.2.6}$$

Obviously, $\{id, r, f\}$, together with the axioms (5.2.6) generates a group with the elements

$$\{id, r, r^2, r^3, f, fr, fr^2, fr^3\}. \tag{5.2.7}$$

The different views, generated by $\{id, r, r^2, r^3, f, fr, fr^2, fr^3\}$, of piece 1 from the example given in the beginning may be visualized as follows:

Constructive definitions of the generating functions of the various "views" of a piece are also obvious. Assuming, e.g., definition (5.2.4) for pieces, we have

$$\begin{aligned} &id(p) \mapsto p, \\ &r(p) \mapsto mp(p.s_2, p.s_3, p.s_4, p.s_1), \\ &f(p) \mapsto mp([p.s_3]^{-1}, [p.s_2]^{-1}, [p.s_1]^{-1}, [p.s_4]^{-1}) \end{aligned}$$

where verifying that these definitions preserve the essential properties (5.2.2) and (5.2.3) is straightforward.

From these definitions, deriving explicit definitions for the remaining elements of the group (5.2.7) is straightforward, e.g.

$$r^2(p) = r(r(p)) = r(mp(p.s_2,\ p.s_3, p.s_4,\ p.s_1)) = mp(p.s_3,\ p.s_4,\ p.s_1,\ p.s_2).$$

In order to formally define the aspect of "views", we may use

$$\textbf{viewid} =_{\text{def}} \{id,\ r,\ r^2,\ r^3,\ f,\ fr,\ fr^2,\ fr^3\}$$

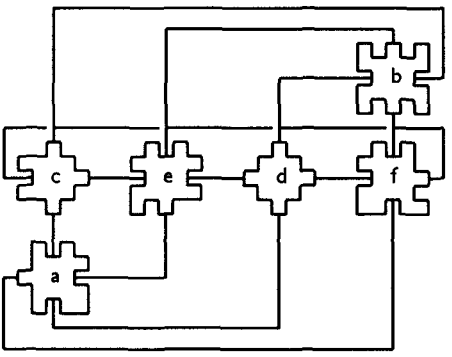
and a function *view* of functionality (**viewid** × **rpiece**) → **rpiece**, that "applies" the "view function" denoted by an object of type **viewid** to a piece.

Since *view* is a finite mapping, there are, of course, obvious alternative representations by appropriate data structures, such as maps, arrays, 8-tuples, etc.

Output. For an adequate formalization of the output of our initially given problem, F-CUBE (cf. section 4.4.2) seems to be a reasonable candidate, since the elements of type **f** obviously can be interpreted by (views of) the pieces of the puzzle and the relation $\perp$ by "has a common side with". Thus, as a first approximation for formally specifying the output of our problem, we choose a tuple (a, b, c, d, e, f) of (views of) pieces which forms a cube, i.e., for which $iscube(a, b, c, d, e, f)$ holds where *iscube* is defined as in F-CUBE or as in lemma 3 (cf. section 4.4.2).

This first approximation, however, is not yet satisfactory. Since the puzzle ultimately aims at building a cube (provided it exists), an adequate formalization of the output should also give sufficient information on how to actually construct the cube from the pieces.

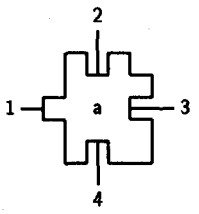
One possibility for giving this information is by arranging the pieces in a two-dimensional way such that common sides of pieces immediately can be recognized. Thus, e.g., for a tuple (a, b, c, d, e, f), which is known to form a cube, the following representation could be used (where lines indicate common sides):



Finally, for a suitable definition of *iscube*, we also have to take into account that two pieces are only allowed to form one common edge of the intended cube, if the teeth of adjacent sides fit together. This has two implications with respect to the formalization of output:

- rather than just interpreting elements of type **f** by (views of) pieces, we have to use a stronger relation (instead of $\perp$) within the definition of *iscube* which reflects the idea of "fitting together"; and
- instead of just requiring that two pieces have a common side, we also have to be more precise on which side of the one piece has to fit which side of the other one.

One possibility for dealing with the latter problem is by assuming a fixed representation for each view of a piece, e.g.,



In order to cope with the former problem, we use an appropriately defined predicate *fits* which assumes such a fixed representation of the pieces. With these modifications, output to our original problem now can be formally specified even in linear form, viz. by a sequence $<a, b, c, d, e, f>$ of views of the original pieces which satisfies the predicate *iscube* defined (following lemma 3) by

$$\begin{aligned} &\mathit{iscube}(a,b,c,d,e,f) =_{\mathbf{def}} \\ &\quad \mathit{fits}(a,2,c,4) \wedge \mathit{fits}(a,3,e,4) \wedge \mathit{fits}(c,3,e,1) \wedge \mathit{fits}(b,3,c,2) \wedge \mathit{fits}(b,2,e,2) \wedge \\ &\quad \mathit{fits}(d,1,e,3) \wedge \mathit{fits}(a,4,d,4) \wedge \mathit{fits}(a,1,f,4) \wedge \mathit{fits}(c,1,f,3) \wedge \mathit{fits}(b,1,d,2) \wedge \\ &\quad \mathit{fits}(b,4,f,2) \wedge \mathit{fits}(d,3,f,1). \end{aligned}$$

In order to give a formal definition of *fits*, we recall the properties of the pieces of the puzzle: Two sides of two pieces fit together, if their respective teeth are complementary with the exception that facing corner teeth of both pieces may not be present. Thus, an obvious specification for *fits* is the following one (assuming a fixed orientation of the sides of a piece):

$$\begin{aligned} &\mathit{fits}(p,i,q,j) =_{\mathbf{def}} \\ &\quad \forall_{1 \le k \le 3}\, \neg(\mathit{middle\text{-}part}(i,p)[k] \Leftrightarrow \mathit{middle\text{-}part}(j,q)[3-k+1]) \wedge \\ &\quad \neg(\mathit{left\text{-}corner}(i,p) \wedge \mathit{right\text{-}corner}(j,q)) \wedge \neg(\mathit{right\text{-}corner}(i,p) \wedge \mathit{left\text{-}corner}(j,q)). \end{aligned}$$

Formulation of the Problem Proper. Having now available formal specifications of input and output to the problem, it remains to formally specify the problem proper. This, however, is straightforward from the informal description: Given a bag b of 6 pieces, we want to know if a cube can be built from these pieces, and if so, how. Formally, this is specified by

if *formscube*(b) **then** *somecube*(b) **else false fi where**
psequ $=_{\text{def}}$ **sequ of rpiece**;
formscube(b) $=_{\text{def}}$ $\exists$ s: **psequ** $\|$ *isperm*(b, s) $\triangle$ *iscube*(s);
somecube(b) $=_{\text{def}}$ **some** s: **psequ** $\|$ *isperm*(b, s) $\triangle$ *iscube*(s);

Here we have used an existential quantifier and a "comprehensive choice" (denoted by **some**) to formulate specification expressions. Intuitively, the meaning of these operators is obvious, for a formal definition we refer the reader to [Bauer et al. 85]. The symbol "$\triangle$" is used to denote sequential conjunction.

It remains to give a formal specification of *isperm*, i.e., to formally specify what it means that a sequence s of the above kind is an arrangement of a given bag b of pieces. But this is obvious: both have to consist of the same pieces, however, maybe in a different view. Thus, the formal relationship between the input bag and the output sequence may be described by

$$\begin{aligned} &\mathit{isperm}(b, s) =_{\text{def}} \\ &\quad |b| = |s| \triangle \exists\, p\text{: } \mathbf{rpiece},\ v\text{: } \mathbf{viewid} \parallel \\ &\qquad\qquad p \in b \wedge \mathit{view}(v, p) = \mathbf{first}\ s \wedge \mathit{isperm}(b - p, \mathbf{rest}\ s). \end{aligned}$$

A Complete Formal Specification. A complete formal specification of our initial problem now is immediately obtained by collecting the various parts developed so far and translating them into our specification formalism. However, we have to take some design decisions, e.g. with respect to the representation of the pieces of the puzzle, which will also have consequences for some of the auxiliary functions. Furthermore, we will still use certain obvious abbreviations such as universal quantification over a restricted domain or iterated sums.

For the formalization of pieces we use

type PIECE;
  exports side, s, rpiece;
  based on BOOL, NAT, INDSEQU;
  mode s = INDSEQU(**bool**);
  mode side = (s: **s** $\|$ $|s| = 5$);
  mode piece = INDSEQU(**side**);
  mode rpiece = (p: **piece** $\|$ $|p| = 4 \triangle$ *isagp*(p));
  functs *isagp*: **piece** $\rightarrow$ **bool**;
  axioms $\forall$ p: **piece** $\|$
    $\mathit{isagp}(p) \equiv \forall_{1 \le i \le 4}\ (p[\mathrm{i}][5] = p[(\mathrm{i}\ \mathbf{mod}\ 4){+}1][1]) \wedge$
      $(p[\mathrm{i}][5] \Rightarrow p[\mathrm{i}][4] \vee p[(\mathrm{i}\ \mathbf{mod}\ 4){+}1][2])$
endoftype

Based on the definition of pieces, the input to the problem can be formalized by

type PBAG
  exports **pbag**;
  based on PIECE, BOOL, NAT, BAG;
  mode rpbag = BAG(**rpiece**);
  mode pbag = (B: **rpbag** || $|B| = 6 \wedge \Sigma_{p \in B}$ *area-covered-by*(p) = 150);
  functs *area-covered-by*: **rpiece** → **nat**,
    middle-teeth: **rpiece** → **nat**,
    corner-teeth: **rpiece** → **nat**,
    btn: **bool** → **nat**;
  axioms ∀ p: **rpiece**; b: **bool** ||
    area-covered-by(p) ≡ 9 + 2 × *middle-teeth*(p) + 3 × *corner-teeth*(p),
    middle-teeth(p) ≡ $\Sigma_{i=1..4}$ (*btn*($p[i][2]$) + *btn*($p[i][3]$) + *btn*($p[i][4]$)),
    corner-teeth(p) ≡ $\Sigma_{i=1..4}$ *btn*($p[i][1]$),
    btn(b) ≡ **if** b **then** 1 **else** 0 **fi**
endoftype

For the formalization of views we use the idea of "view identifiers" as introduced above, as well as generating functions *rot* (for "rotate") and *flip*. Thus we obtain:

type VIEW
  exports **viewid**, *view*;
  based on PIECE;
  mode viewid = $id \mid r \mid r^2 \mid r^3 \mid f \mid fr \mid fr^2 \mid fr^3$;
  functs *view*: (**viewid** × **rpiece**) → **rpiece**,
    rot: **rpiece** → **rpiece**,
    flip: **rpiece** → **rpiece**;
  axioms ∀ i: **viewid**; p: **rpiece**; s: **side** ||
    view(i, p) ≡
      if i **is** id **then** p
      elsf i **is** r **then** *rot*(p)
      elsf i **is** r^2 **then** *rot*(*rot*(p))
      elsf i **is** r^3 **then** *rot*(*rot*(*rot*(p)))
      elsf i **is** f **then** *flip*(p)
      elsf i **is** fr **then** *flip*(*rot*(p))
      elsf i **is** fr^2 **then** *flip*(*rot*(*rot*(p)))
      elsf i **is** fr^3 **then** *flip*(*rot*(*rot*(*rot*(p)))) **fi**,
  rot(p) ≡ $p[4] + p[1:3]$,
  flip(p) ≡ $[p[3]]^{-1} + ([p[2]]^{-1} + ([p[1]]^{-1} + ([p[4]]^{-1} + <>)))$
endoftype

Based on the definition of pieces, the output of the problem can be formalized by

type PSEQU
exports **psequ**, *iscube*;
based on RPIECE, BOOL, NAT, INDSEQU;
mode psequ = INDSEQU(**rpiece**);
mode mside = (*m*: **s** || |*m*| = 3);
functs *iscube*: **psequ** → **bool**,
fits: (**rpiece** × **nat** × **rpiece** × **nat**) → **bool**,
left-corner: (**nat** × **rpiece**) → **bool**,
middle-part: (**nat** × **rpiece**) → **mside**,
right-corner: (**nat** × **rpiece**) → **bool**;
axioms ∀ *s*: **psequ**; *p*, *q*: **rpiece**; *i*, *j*: **nat** ||
defined(*iscube*(*s*)) ⇔ |*s*| = 6,
iscube(*s*) ≡
$fits(s[1],2,s[3],4) \wedge fits(s[1],3,s[5],4) \wedge fits(s[3],3,s[5],1) \wedge$
$fits(s[2],3,s[3],2) \wedge fits(s[2],2,s[5],2) \wedge fits(s[4],1,s[5],3) \wedge$
$fits(s[1],4,s[4],4) \wedge fits(s[1],1,s[6],4) \wedge fits(s[3],1,s[6],3) \wedge$
$fits(s[2],1,s[4],2) \wedge fits(s[2],4,s[6],2) \wedge fits(s[4],3,s[6],1)$;
fits(*p*, *i*, *q*, *j*) ≡
$\forall_{1 \le k \le 3} \neg(middle\text{-}part(i,p)[k] \Leftrightarrow middle\text{-}part(j,q)[3-k+1]) \wedge$
$\neg(left\text{-}corner(i,p) \wedge right\text{-}corner(j,q)) \wedge$
$\neg(right\text{-}corner(i,p) \wedge left\text{-}corner(j,q))$,
left-corner(*i*, *p*) ≡ *p*[*i*][1],
middle-part(*i*, *p*) ≡ *p*[*i*][2 : 4],
right-corner(*i*, *p*) ≡ *p*[*i*][5]
endoftype

Based on these definitions, the problem proper can be formalized as given above.

This specification has been successfully used as a starting point for a (rather straightforward) transformational development which ended in a (fairly efficient) backtrack program to solve the "cube problem" (as was requested in section 1). In this program (the transformed versions of) the definitions of *isagp* and **pbag** are profitably used to check the validity of the input (represented by boolean arrays). Efficiency of the resulting program as compared with a naive backtrack program is obtained by exploiting lemma 3 from section 4.4 during the development.

6 Concluding Remarks

In this paper we have introduced algebraic specifications as a means for fomally specifying problems – or at least as a solid basis on which a comprehensive specification formalism can be built. Concepts, theoretical background, and abstract syntax have been borrowed from the language CIP-L (cf. [Bauer et al. 85]) on purpose. Of course, there are many other algebraically based specification languages, some of which are mentioned in section 1. Although these languages may differ quite substantially with respect to notation or the particular theoretical basis, most of them basically comprise more or less the same concepts which justifies our inital claim to view CIP-L as a typical representative for an algebraic specification language. A more detailed synoptical treatment of several algebraic

specification languages can be found in [Wirsing 89]. There, in particular, also lots of interesting apsects with respect to theory are surveyed and discussed.

Within our paper the advantage of formal specifications over informal ones was taken for granted. In fact, although formal specification can be argued to be an additional cost factor for software development, it is rather to be seen as an investment that essentially pays afterwards during implementation and "maintenance". Detailed experiences in this respect are reported in [Möller, Partsch 86].

The advantages of algebraic specification over conventional (semi-formal) approaches to requirements engineering are also obvious. Algebraic specifications have a precise (formal) semantics, and thus provide the mandatory prerequisite for reasoning about completeness and consistency. They fit nicely into the paradigm of "transformational programming" (cf. [Partsch 90]) and thus make a substantial part of a conceptually integrated methodology for software development. And last, but not least, algebraic specifications also contribute to solving the important problem of adequacy of a formal specification (cf. section 2.3). In this respect not only the expressive power with respect to formulating axioms has to be mentioned, but, above all, the various possibilities of "checking" adequacy, i.e. checking how far a formal problem description coincides with the original problem. It is obvious that adequacy cannot be proved: the best to be achieved is making it plausible. In this respect, algebraic specifications provide several possibilities all of which are based on redundancy:

- It is in principle possible to give different, independent formalizations of the same problem, and then to (formally) prove their equivalence;
- It is possible to derive additional (redundant) properties ("theorems") from the axiomatization, and to examine these for adequacy;
- Certain restricted algebraic specifications are "executable" (cf. section 3.5) and provide a means for "rapid prototyping".

In order to illustrate the use of algebraic specifications, we had to restrict ourselves to relatively small examples for obvious reasons. More comprehensive examples can be found elsewhere, among them a line-oriented text editor [Partsch 90], the kernel of a transformation system [Bauer et al. 87], a screen-oriented editor [Feijs 90], part of the Macintosh Toolbox Event Manager [Burton et al. 89], or (substantial parts of) Microsoft Word [Wijshoff 92]. In addition, several companies have started the experimental use of algebraic specification languages in their daily work.

Algebraic specifications are an appropriate approach to problem specification and a reasonable basis for a suitable formalism for requirements engineering, because they meet nearly all properties desired of such a formalism (cf. [Partsch 91]). Nevertheless, for practical work, extensions of the pure formalism will be needed to enhance expressiveness, such as higher-order functions (cf. [Möller 87]), specification-building operations (like those in ASL [Wirsing 83]), and relations to formulate indeterminism and certain non-functional requirements. Furthermore, extensions by modal and temporal logic (for real-time and other behavioural aspects) or traces (for parallel and distributed systems) can be thought of. Experiments with these and similar kinds of extensions are on the way.

Acknowledgement

Constructive criticism and valuable remarks by J. Boyle, M. Geerling, C. Morgan, and D. Tuijnman on earlier versions of this paper are herewith gratefully acknowledged.

References

Agresti, W.M. (ed.): New paradigms for software development. Washington, D.C.: IEEE Computer Society Press 1986

Backhouse, R.C.: Program construction and verification. London: Prentice-Hall 1986

Balzer, R.: Final report on GIST. USC/ISI, Marina del Rey, Technical Report 1981

Balzer, R., Cheatham, T.E.Jr., Green, C.: Software technology in the 1990's: using a new paradigm. IEEE Computer **16**:11, 39-45 (1983)

Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H.: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**, Berlin: Springer 1985

Bauer, F.L., Ehler, H., Horsch, A., Möller, B., Partsch, H., Paukner, O., Pepper, P.: The Munich project CIP. Volume II: The transformation system CIP-S. Lecture Notes in Computer Science **292**, Berlin: Springer 1987

Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: Programming by formal reasoning - computer-aided intuition-guided programming. IEEE Transactions on Software Engineering, **15**:2, 165-180 (1989)

Bergstra, J.A., Heering, J., Klint, P. (eds.): Algebraic specification. ACM Press Frontier Series. New York: Addison-Wesley 1989

Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.): Logic of programming and calculi of discrete design. NATO ASI Series, Series F: Computer and System Sciences, vol. 36. Berlin: Springer 1987, pp. 5-42

Bird, R.S., Wadler, P.L.: Introduction to functional programming. Hemel Hempstead: Prentice Hall International 1988

Bjørner, D., Jones, C.B.: Formal specification and software development. Englewood Cliffs, N.J.: Prentice-Hall 1982

Broy, M.: Predicative specifications for functional programs describing communicating networks. Information Processing Letters **25**, 93-101 (1987)

Burstall, R.M., Goguen, J.A.: Semantics of CLEAR, a specification language. In: Bjørner, D. (ed.): Abstract software specifications. Lecture Notes in Computer Science **86**, Berlin: Springer 1980, pp. 292-332

Burton, C.T., Cook, S.J., Gikas, S., Rowson, J.R., Sommerville, S.T.: Specifying the Apple MacintoshTM Toolbox Event Manager. Formal Aspects of Computing **1**, 147-171 (1989)

Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, N.J.: Prentice-Hall 1976

Dubois, E., Hagelstein, J., Rifaut, A.: Formal requirements engineering with EREA. Philips Journal of Research **43**:3/4, 393-414 (1988)

Feather, M.S.: A survey and classification of some program transformation approaches and techniques. In: Meertens, L.G.L.T. (ed.): Proc. IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, Germany, 1986. Amsterdam: North-Holland 1987, pp. 165-196

Feijs, L.M.G.: A formalization of design methods. A λ- calculus approach to system design with an application to text editing. Technical University of Eindhoven, Ph. D. thesis, 1990

Feijs, L.M.G., Jonkers, H.B.M., Obbink, J.H., Koymans, C.P.J., Renardel de Lavalette, G.R., Rodenburg, P.H.: A survey of the design language COLD. ESPRIT '86: Results and achievements. Amsterdam: North-Holland 1987, 631-644

Fey, W.: Introduction to algebraic specification in ACT TWO. TU Berlin, Fachbereich 20, Technical Report 86-13, 1986

Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. Proc. 12th Ann. ACM Symp. on Principles of Programming Languages, New Orleans, Miss., 1985, pp. 52-66

Gaudel, M.C.: Toward structured algebraic specification. ESPRIT '85: Status report of continuing work. Part I. Amsterdam: North-Holland 1986, pp. 493- 510

Goldberg, A.T.: Knowledge-based programming: a survey of program design and construction techniques. IEEE Transactions on Software Engineering SE- **12**:7, 752-768 (1986)

Geser, A., Hußmann, H.: Experiences with the RAP-system – a specification interpreter combining term rewriting and resolution. In: Robinet, B., Wilhelm, R. (eds.): ESOP 86. Lecture Notes in Computer Science **213**, Berlin: Springer 1986, pp. 339-350.

Gries, D.: The science of programming. Berlin: Springer 1981

Guttag, J.V., Horning, J.J.: Preliminary Report on the LARCH shared language. Xerox Research, Palo Alto, Technical Report CSL 83-6, 1983

Hehner, E.C.R., Gupta, L.E., Malton, A.J.: Predicative Methodology. Acta Informatica **23**, 487-505 (1986)

Henderson, P.: Functional programming: application and implementation. Englewood Cliffs, N.J.: Prentice-Hall 1980

Special Collection on Requirement Analysis. IEEE Transactions on Software Engineering SE-**3**:1, 2-84 (1977)

IEEE Computer, 18:4 (1985)

Jones, C.B.: Systematic software development using VDM. Englewood Cliffs, N.J.: Prentice-Hall 1986

Möller, B.: Higher-order algebraic specifications. Technische Universität München, Fakultät für Mathematik und Informatik, Habilitation thesis, 1987

Möller, B., Partsch, H.: Formal specification of large-scale software – objectives, design decisions and experiences in a concrete software project. In: Meertens, L.G.L.T. (ed.): Proc. IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, Germany, 1986. Amsterdam: North-Holland 1987, pp. 491-515

Partsch, H.: Specification and transformation of programs. Berlin: Springer 1990

Partsch, H.: Requirements Engineering. München: Oldenbourg 1991

Rzepka, W., Ohno, Y.: Requirements Engineering environments: Software tools for modelling user needs. IEEE Computer, **18**:4, 9-12 (1985)

Spivey, J.M.: Understanding Z. A specification language and its formal semantics. Cambridge, U.K.: Cambridge University Press 1988

Webster's New World Dictionary. Second College Edition. Cleveland: William Collings + World Publishing 1974

Wijshoff, F.: Formal specification of existing software — A case study: MsWord for the MacIntosh. University of Nijmegen, Diploma thesis, 1992

Wirsing, M.: A Specification Language. Technische Universität München, Fachbereich Mathematik und Informatik, Habilitation thesis, 1983

Wirsing, M.: Algebraic specification. Universität Passau, Fakultät für Mathematik und Informatik, Technical Report MIP-8914, 1989. Also in: Van Leeuven, J. (ed.): Handbook for theoretical Computer Science. Amsterdam: North-Holland 1990

Wirsing, M., Pepper, P., Partsch, H., Dosch, W., Broy, M.: On hierarchies of abstract data types. Technische Universität München, Institut für Informatik, Technical Report TUM-I8007. Also: Acta Informatica **20**, 1-33 (1983)