The Specification Language of OBSCURE

A 87/07

Thomas Lehmann and
Jacques Loeckx

Fachbereich 10 — Informatik
Universität des Saarlandes
D — 6600 Saarbrücken

to appear

1 Introduction

The idea that abstract types may support the development of correct programs is now well-accepted. Meanwhile several methods have been proposed for the specification of abstract types: operational specifications ([Ho 72], [Sh 81], [Li 81], [NY 83], [LG 86]), algebraic specificatons ([GTW 78], [GHM 78], [TWW 82], [BW 82], [Eh 82], [EM 85]) and constructive specifications ([Ca 80], [Kl 84], [Lo 87], [Bu 87]). While operational specifications are embedded in an imperative language, algebraic specifications are more abstract in that they make use of first-order formulas, usually equalities or Horn-clauses. Constructive specifications offer a similar degree of abstraction but differ by their algorithmic nature.

The design of non-trivial specifications is practicable only if it is performed modularly. To this end specifications are embedded in a specification language. Essentially, such a language allows the construction of specifications out of more elementary ones. In the case of operational specifications the specification language is foreordained to be the embedding imperative language. For algebraic specifications several specification languages have recently been proposed: CLEAR ([Sa 84], [BG 80]), ACT-ONE and ACT-TWO ([EM 85], [Fe 87]), OBJ2 ([FGJM 85]), PLUSS ([Gd 84], [BGM 87]), ASL ([Wi 86]), ASF ([BHK 87]).

Even with the use of a specification language the design of non-trivial specifications with pencil and paper is tedious and error-prone. As a solution the specification language is embedded into an adequate environment. Such an environment supports the interactive design of specifications as well as the (interactive or automatic) verification of their properties. Some more or less elaborate environments have been described or announced in the literature: OBJ2 ([FGJM 85]), an environment for a subset of the specification language PLUSS called ASSPEGIQUE ([BCV 85]), the environment RAP ([Hu 87]), an environment for the specification language ACT-ONE called the ACT-System.

The specification tool discussed in this paper is called *OBSCURE*. It consists of a specification language *together with* an environment for it. The specification language is a simple language similar to Bergstra's term language ([BHK 86]). The environment is a program consisting of a design unit and a verification unit. The design unit allows the interactive design of specifications. More precisely, with the help of a command language the user induces the design unit to stepwise generate syntactically correct specifications. The verification unit allows to prove properties of these specifications and, in particular, to prove their semantical correctness. The main features by which *OBSCURE* differs from the specification languages and environments described in the literature are now briefly discussed.

First, the specification language of *OBSCURE* has been designed as a language to be used in an environment, not as a language to be used with pencil

and paper. As a result the specification language has a very simple syntax and semantics at the expense of more elaborate context conditions. These context conditions put no burden on the user as they are checked automatically and on-line (i.e. at each command) by the design unit. Second, the specification language is independent from the specification method used. It even allows the use of different (algebraic and/or constructive) specification methods within the same specification. This is possible because *OBSCURE* distinguishes between the constructs inherent to the specification method — such as "data constraints" in CLEAR — and those inherent to putting specifications together. Third, *OBSCURE* explicitly distinguishes between the specification language and the command language of the design unit. This is reflected by the fact that procedures (i.e. "parameterized specifications") and user-friendly macros are part of the command language, *not* of the specification language. According to this philosophy parameterized specifications are *not* specifications but rather constitute a tool to construct specifications. Next, apart from the classical operations *OBSCURE* provides means to explicitly construct subalgebras and quotient algebras. Finally, *OBSCURE* directly ties the design of a specification to its verification. In particular, the design unit automatically generates formulas expressing, for instance, certain closure and congruence conditions and transmits them for verification to the verification unit.

The goal of the present paper is to give a formal definition of the specification language of *OBSCURE* . A description of the design unit and its command language may be found in [LL 87b]. An informal introduction to *OBSCURE* and an illustration of its use may be found in [LL 87a].

Section 2 briefly recalls some basic notions. Section 3 contains a description of the specification language; it constitutes the bulk of the paper. The case of loose specifications and the problem of overloading is treated in Section 4. Finally, Section 5 shortly discusses the design unit of *OBSCURE* and its command language.

2 Basic notions

2.1 Algebras

This section recalls a few notions from algebra (cf. [EM 85]).

2.1.1 Syntax

A *sort* is an identifier. An *operation* is a $(k+2)$-tuple, $k \geq 0$,

$$n : s_1 \dots s_k \rightarrow s_{k+1}$$

where n is an identifier, called *operation name*, and $(s_1, \dots, s_{k+1})$ is a $(k+1)$-tuple of sorts, called the *arity*. It is called *S-sorted*, if S is a set of sorts with $s_1, \dots, s_{k+1}$ among its elements. By definition two operations

$$n : s_1 \dots s_k \rightarrow s_{k+1}$$

$$m : t_1 \dots t_l \rightarrow t_{l+1}$$

are equal, if $n = m, k = l$ and $s_i = t_i$ for all i, $1 \leq i \leq k+1$.

A *signature* is a pair (S, O) where S is a set of sorts and O a set of operations. It is called an *algebra signature* if each operation of O is S-sorted.

A *list of sorts and operations* is a $(k+l)$-tuple

$$(s_1, \dots, s_k; o_1, \dots, o_l) \quad (k \geq 0,\ l \geq 0)$$

where $s_1, \dots, s_k$ are sorts and $o_1, \dots, o_l$ operations.

Let $\Sigma = (S, O)$ and $\Sigma' = (S', O')$ be signatures. Expressions such as $\Sigma - \Sigma'$ or $\Sigma \subseteq \Sigma'$ are used to denote $(S - S', O - O')$ or $S \subseteq S'$ and $O \subseteq O'$ respectively. Similarly, if no ambiguity arises one writes Σ instead of $S \cup O$; for instance, $c \in \Sigma$ stands for $c \in S \cup O$. Finally, if $lso = (s_1, \dots s_k; o_1, \dots o_l)$ is a set of sorts and operations, we will occasionally identify lso with the signature

$$(\{s_1, \dots, s_k\}, \{o_1, \dots, o_l\})$$

The following lemma is a straightforward consequence of the definitions:
Lemma 1: If Σ and Σ' are algebra signatures, then so are $\Sigma \cup \Sigma'$ and $\Sigma \cap \Sigma'$.

2.1.2 Semantics

In order to avoid the use of classes and functors we start from a set $\mathcal{U}$ called *universe*.

Let $\Sigma = (S, O)$ be an algebra signature. A (Σ)- *algebra* is a (total) function, say A, which maps

(i) each sort s of S into a set $A(s) \subseteq \mathcal{U}$, called the *carrier set* of sort s;

(ii) each operation $n : s_1 \ldots s_k \rightarrow s_{k+1}$ $(k \geq 0)$ of O into a (possibly partial) function

$$A(n : s_1 \ldots s_k \rightarrow s_{k+1}) : A(s_1) \times \ldots \times A(s_k) \rightsquigarrow A(s_{k+1}).$$

The set of all Σ-algebras is denoted Alg_{Σ}.

Let $\Sigma = (S, O)$ and Σ' be algebra signatures with $\Sigma \subseteq \Sigma'$ and let A be a Σ'-algebra. By definition $A \mid \Sigma$ denotes the Σ-algebra obtained by the restriction of the function A to the domain $S \cup O$.

2.2 Algebra modules

Intuitively, algebra modules are to represent the meaning of specifications of abstract data types (cf. [BHK 86], [EW 85], [EW 86]). Syntactically an algebra module is characterized by an "imported" signature and an "exported" one. In the case of non-loose specifications it is characterized semantically by a function mapping algebras of the imported signature into algebras of the exported one. The basic idea is that an exported algebra is an extension of the imported one. Actually, it is possible to "forget" sorts and operations, i.e. the sorts and operations of an imported algebra are not necessarily all "inherited" by the exported one. Hence, the requirement that the exported algebra is an extension of the imported one is replaced by the requirement that the inherited sorts and operations are "persistent", i.e. that their meaning in the exported algebra is the same as in the imported algebra. To include the case of loose specifications it is sufficient to view an algebra module as a function mapping imported algebras into sets of exported ones. These notions are now made more precise for the case of non-loose specifications. The treatment of loose specifications is delayed until Section 4.2.

A *module signature* is a pair (Σ_i, Σ_e) of algebra signatures. Σ_i is called the *imported* signature, Σ_e the *exported* one. The sorts and operations from $\Sigma_i \cap \Sigma_e$ are called the *inherited* ones.

An *(algebra) module* for the module signature (Σ_i, Σ_e) is a (possibly partial) function

$$M : Alg_{\Sigma_i} \rightsquigarrow Alg_{\Sigma_e}$$

satisfying the following *persistency condition* :

for each algebra $A \in Alg_{\Sigma_i}$ from the domain of M:
for each inherited sort or operation $c \in \Sigma_i \cap \Sigma_e$:
$M(A)(c) = A(c)$

2.3 Logic

The main goal of this Section is to introduce the notion of formulas in the framework of multi-sorted algebras. In order to keep the definition of *OBSCURE* institution-independent the precise syntax and semantics of these formulas is left pending.

Let $\Sigma = (S, O)$ be an algebra signature. A (Σ−) *variable (of sort s)* is a pair

$$(v : s)$$

where v is an identifier and s a sort from S. When s is known one writes v instead of $(v : s)$. It is implicitly assumed that there exists infinitely many variables of each sort. *Formulas* are built up from operations and variables. The precise definition of the set WFF(Σ) of all formulas for the algebra signature Σ is left pending. In the case of predicate logic, for instance, WFF(Σ) consists of the usual (correctly typed) formulas, in the case of equational logic WFF(Σ) contains equalities between terms only.

Let now A be a Σ-algebra. An *assignment* for the algebra A is a function mapping each variable of sort s into an element of the carrier set $A(s)$. The *meaning* of the logic associates with each formula and each assignment a value from $\{true, false\}$. It is usually defined as an extension of the function A, namely

$$A : \mathrm{WFF}(\Sigma) \rightarrow (\mathrm{ASS} \rightarrow \{true, false\})$$

where ASS denotes the set of all assignments for the algebra A and (ASS $\rightarrow$ $\{true, false\}$) the set of all functions on ASS with values in $\{true, false\}$ (cf [LS 87]). Again, the precise definition of this extension is left pending. In the case of first-order predicate logic and total operations the meaning is as usual. In case not all operations are total the meaning has to cope with undefined values. In fact, $A(w)(\sigma)$ evaluates to *true* or *false* for any formula w and assignment σ, even if w contains terms with undefined values. Examples of logics dealing with partial operations are *LCF*([Mi 72]) and the logic described in [Lo 87].

Let Σ be an algebra signature. A formula w from WFF(Σ) is said to be *valid* in a Σ-algebra A if $A(w)(\sigma) = true$ for all assignments σ. One then writes $A \models w$.

These general notions suffice for a definition of *OBSCURE*. Actually, one gets a more explicit definition by assuming the validity of the Coincidence Theorem. To this effect the logic is supposed to provide a — not further specified — notion of a *free occurrence* of a variable in a formula. The Coincidence Theorem states that the value $A(w)(\sigma)$ of a formula w for the assignment σ depends on the value $\sigma(x)$ of only those variables x which occur free in w (see e.g. [En 72], [LS 87]). As a notational abbreviation one writes $A(w)$ instead

of $A(w)(\sigma)$ whenever the formula w is *closed*, i.e. whenever w contains no free occurrences of variables.

2.4 Renamings

A specification language has to allow the renaming of sorts and operations. Such renamings may, for instance, be used to avoid "name clashes". A renaming is an operation on signatures. Essentially, it performs the simultaneous substitution of "old" names by "new" ones. As a technical complication the renaming of a sort also implies the renaming of its occurrences in the (arities of the) operations. In a specification renamings are defined by pairs of lists of sorts and operations. These different notions are now made precise. If he wishes the reader may skip the formal details in a first reading.

2.4.1 Renamings

Let $\Sigma = (S, O)$ be a signature. A *renaming* $(on\Sigma)$ is a pair $\rho = (\rho_S, \rho_O)$ of functions ρ_S on S and ρ_O on O such that for each operation $o = (n : s_1 \ldots s_k \rightarrow s_{k+1})$ from O

$$\rho_O(o) = (n' : s'_1 \ldots s'_k \rightarrow s'_{k+1})$$

with $s'_i = \rho_S(s_i)$ for $1 \leq i \leq k+1$.

The following Lemmata are immediate consequences of the definitions:

Lemma 2: Let ρ be a renaming. If $\Sigma' \subseteq \Sigma$ is an algebra signature, then so is $\rho(\Sigma')$. □

Lemma 3: Let ρ be a renaming on Σ and let $\Sigma' \subseteq \Sigma$ be an algebra signature.

(i) If A is a $\rho(\Sigma')$-algebra, then $A \circ (\rho \mid \Sigma')$ is a Σ'-algebra.

(ii) If ρ is injective on Σ' and if B is a Σ'-algebra, then $B \circ (\rho \mid \Sigma')^{-1}$ is a $\rho(\Sigma')$-algebra. □

2.4.2 Renaming pairs

A *renaming pair (on a signature* Σ) is a pair of lists of sorts and operations from Σ, say

$$((s_1, \ldots, s_k; o_1, \ldots, o_l), (s'_1, \ldots, s'_k; o'_1, \ldots, o'_l)) \qquad (k, l \geq 0)$$

satisfying the following conditions:

a) the sorts $s_1, \ldots, s_k$ are pairwise different;

b) the operations $o_1, \ldots, o_l$ are pairwise different;

c) for each $i, 1 \leq i \leq l$: if $o_i = (n : t_1 \ldots t_m \rightarrow t_{m+1})$ and $o'_i = (n' : t'_1 \ldots t'_{m'} \rightarrow t'_{m'+1})$, $m, m' \geq 0$, then

- $m = m'$;
- for each $1 \leq j \leq m+1$:

$$t'_j = \begin{cases} s'_p & \text{if } t_j = s_p \text{ for some } p,\ 1 \leq p \leq k \\ t_j & \text{otherwise .} \end{cases}$$

Informally, condition c) expresses that in the arities of the new operations the sorts have already been substituted. An example of a renaming pair is

$$((s,\ t,\ u;\ n : stuv \rightarrow w), (w,\ s,\ w;\ m : wswv \rightarrow w))$$

(provided s, t, u are pairwise different). Counterexamples are

$$((s;\ n : u \rightarrow s), (v;\ m : u \rightarrow s))$$

(if $v \neq s$) and

$$((v,\ v;\ n :\rightarrow v), (s, t;\ n :\rightarrow s)).$$

A renaming pair $((s_1, \ldots, s_k; o_1, \ldots, o_l), (s'_1, \ldots, s'_k; o'_1, \ldots, o'_l))$ on a signature Σ induces a renaming ρ on Σ in the following straightforward way:

- for each sort s from Σ:

$$\rho(s) = \begin{cases} s'_j & \text{if } s = s_j \text{ for some } j,\ 1 \leq j \leq k \\ s & \text{otherwise} \end{cases}$$

- for each operation $o = (n : t_1 \ldots t_m \rightarrow t_{m+1})$, $m \geq 0$, from Σ:

$$\rho(o) = \begin{cases} o'_j & \text{if } o = o_j \text{ for some } j, 1 \leq j \leq l \\ (n : \rho(t_1) \ldots \rho(t_m) \rightarrow \rho(t_{m+1})) & \text{otherwise} \end{cases}$$

That ρ is effectively a renaming is a direct consequence from the definitions.

2.5 Subalgebras, quotient algebras

Two constructions are recalled yielding subalgebras and quotient algebras respectively. These constructions are well-known from the literature (see e.g. [EM 85]).

Let A be a Σ-algebra, $\Sigma = (S, O)$, and w a formula from WFF(Σ) containing free occurrences of a single variable, say $(x : s_o)$. This formula defines a

subset, say C, of the carrier set $A(s_o)$ of sort s_o, namely the set of all carriers from $A(s_o)$ that satisfy w. Formally

$$C = \{a \in A(s_o) \mid A(w)(\sigma[x/a]) = \mathit{true} \text{ for all } \sigma \in \text{ASS}\}$$

(where $\sigma[x/a]$ denotes the assignment identical with σ except that its value for the argument x is a. Note, by the way, that the value of $A(w)(\sigma[x/a])$ does not depend on σ by the Coincidence Theorem). This subset in its turn defines a subalgebra of the algebra A, namely the Σ-algebra B defined by

$$B(s) = \begin{cases} A(s) & \text{if } s \in S - \{s_o\} \\ C & \text{if } s = s_o \end{cases}$$

for each $s \in S$;

$$B(o) = A(o) \mid \ (B(s_1) \times \ldots \times B(s_k))$$
$$\text{for each } o = (n : s_1 \ldots s_k \rightarrow s) \in O, \ k \geq 0$$

It is well-known that B is effectively an algebra only if the algebra A satisfies the following *closure condition*:

$$\text{for each } o = (n : s_1 \ldots s_k \rightarrow s) \text{ from } O, k \geq 0 :$$

$$A(o)(B(s_1) \times \ldots \times B(s_k)) \subseteq B(s_{k+1})$$

Informally the condition expresses that elements from the subset are mapped into elements from the subset. Henceforth the algebra B is said to be the ***subalgebra generated by A and w***.

Let A again be a Σ-algebra and w a formula in WFF(Σ) in which exactly two variables of the same sort, say $(x : s_o)$ and $(y : s_o)$, occur free. This formula defines an equivalence relation, say $\sim_{s_o}$, in the carrier set $A(s_o)$, namely the least equivalence relation satisfying

$$\text{for all } a, b \in A(s_o) :$$

$$a \sim_{s_o} b \text{ if } A(w)(\sigma[x/a][y/b]) = \mathit{true} \text{ for all } \sigma \in \text{ASS}$$

In order to simplify the wording of the now following definitions it is useful to provide the other carrier sets with an equivalence relation as well:

$$\text{for all sorts } s \in S - \{s_o\} :$$

$$a \sim_s b \text{ iff } a = b$$

This family of equivalence relations defines a quotient algebra of the algebra A, namely the Σ-algebra B defined by:

$$B(s) = \{[c] \mid c \in A(s)\}^1$$

[1] [c] denotes the equivalence class of c generated by $\sim_s$.

for each $s \in S$;

$$B(o)([c_1],\ldots,[c_k]) = \begin{cases} [A(o)(c_1,\ldots,c_k)] \text{ if } A(o)(c_1,\ldots,c_k) \text{ is defined} \\ \text{undefined otherwise} \end{cases}$$

for each $c_i \in A(s_i), 1 \leq i \leq k$,

for each $o = (n : s_1 \ldots s_k \rightarrow s) \in O, k \geq 0$.

It is well-known that B is effectively an algebra only if the algebra A satisfies the ***congruence condition***:

for each $o = (n : s_1 \ldots s_k \rightarrow s) \in O, \quad k \geq 0$:
  for all $a_i, b_i \in A(s_i)$ with $a_i \sim_{s_i} b_i, \quad 1 \leq i \leq k$:
    either $A(o)(a_1,\ldots,a_k)$ and $A(o)(b_1,\ldots,b_k)$ are both undefined
    or $A(o)(a_1,\ldots,a_k)$ and $A(o)(b_1,\ldots b_k)$ are both defined and, moreover,
      $A(o)(a_1,\ldots,a_k) \sim_s A(o)(b_1,\ldots,b_k)$

Informally, the condition expresses that equivalent arguments lead to equivalent values. Henceforth the algebra B is said to be the ***quotient algebra generated by A and w***.

For a more detailed treatment the reader may consult [EM 85] or [Lo 87].

3 The specification language

The specification language of *OBSCURE* constitutes a mathematical notation for algebra modules. While its structure is very simple, specifications written in it look clumsy. For this reason *OBSCURE* also provides a language which allows to draw up specifications; this language constitutes the command language of the (design unit of the) *OBSCURE* environment. The present Section is devoted to the description of the specification language. The command language will be shortly discussed in Section 5.

Section 3.1 contains an informal introduction to the syntax and semantics motivating the formal definitions in Section 3.2 and 3.3. The treatment of overloading and the case of loose specifications is delayed until Section 4.

3.1 An informal overview of the language

Syntactically the specification language is a formal language, the elements of which are called *specifications*. Each specification has the form of a term. In these terms *atomic specifications* play the role of constants; constructs such as "+", "o" or "□" play the role of operators. More precisely, a specification is either an atomic specification or it has one of the following nine forms:

$$(m_1 + m_2),\ (m_1 \circ m_2),\ (\mathit{lso} \,\square\, m_1),\ ([\mathit{lso}1/\mathit{lso}2]m_1),\ (m_1[\mathit{lso}1/\mathit{lso}2]),$$

$$(\{w\}m_1),\ (m_1\{w\}),\ (w \mid m_1),\ (w \bowtie m_1)$$

where m_1, m_2 are specifications, *lso*, *lso*1, *lso*2 are lists of sorts and operations and w is a formula. Note that the language bears strong similarities with the term language of [BHK 86].

Following a now classical pattern ([EM 85], [Sa 84]) the semantics of the specification language are defined in two steps:

(i) a function S, called *signature function*, maps specifications into module signatures;

(ii) a function $\mathcal{M}$, called *meaning function*, maps specifications into algebra modules; for any specification m the module signature of the algebra module $\mathcal{M}(m)$ is $S(m)$.

The semantics of atomic specifications and of the different constructs of the language are now discussed successively.

An atomic specification is a specification drawn up according to one of the numerous specification methods known from the literature. In the description of the specificaton language the syntax and semantics of these specifications is left pending. It is merely assumed that each atomic specification defines an algebra module. In practice an atomic specification usually consists of

a "heading" and a "specification body". The heading fixes the imported and exported signatures of the module signature and the specification body defines the algebra module. In the case of the initial algebra specification method, for instance, the specification body consists of a set of equalities and the algebra module maps any imported algebra into its free extension. By the way, the predicate "atomic" refers to the semantics, not to the syntax: atomic specifications usually constitute the bulk of (the text of) a specification.

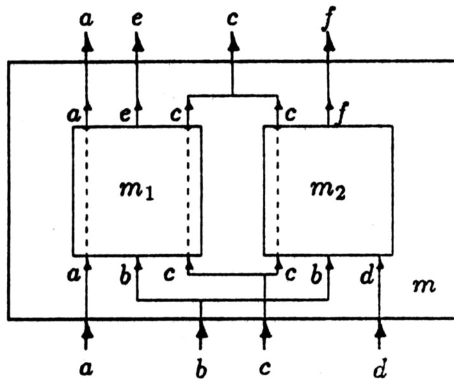
Next, the nine constructs of the specification language are discussed successively. Most of them are illustrated graphically on Figure 1.

The construct "+" puts two specifications together. More precisely, when applied to the specifications m_1 and m_2 the construct yields the specification (m_1+m_2). The module signature $S((m_1+m_2))$ of this specification is defined as the union of the module signatures $S(m_1)$ and $S(m_2)$ (see Figure 1(a)). The module $\mathcal{M}((m_1+m_2))$ is defined similarly: its value is obtained by uniting the values of the modules $\mathcal{M}(m_1)$ and $\mathcal{M}(m_2)$. Hence the construct "+" of *OBSCURE* is similar to the construct "+" of *CLEAR* and "and" of *ACT-ONE*. Actually, a precise definition of the semantics of this construct has to cope with the following technical problem: the algebras accepted as arguments by the modules $\mathcal{M}((m_1+m_2))$, $\mathcal{M}(m_1)$ and $\mathcal{M}(m_2)$ have in general different signatures. Hence the module $\mathcal{M}((m_1+m_2))$ is defined by its value for an arbitrary algebra A of the imported signature of the module signature $S((m_1+m_2))$:

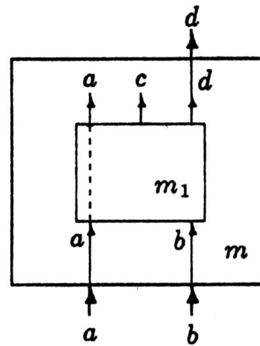
$$\mathcal{M}((m_1+m_2))(A) = \mathcal{M}(m_1)(A \mid S_i(m_1)) \cup \mathcal{M}(m_2)(A \mid S_i(m_2))$$

where $S_i(m_1)$ and $S_i(m_2)$ are the imported signatures of the module signatures $S(m_1)$ and $S(m_2)$ respectively. Note that the restrictions $A \mid S_i(m_1)$ and $A \mid S_i(m_2)$ of the algebra A to the signatures $S_i(m_1)$ and $S_i(m_2)$ yield algebras of the required signatures. The right-hand side of the equality denotes the algebra obtained by the union of the graphs of the algebras $\mathcal{M}(m_1)(A \mid S_i(m_1))$ and $\mathcal{M}(m_2)(A \mid S_i(m_2))$. (Remember that an algebra is a function!). Actually, the union of the graphs of two functions yields (the graph of) a relation which is not necessarily a function. That the right-hand side of the equality nevertheless denotes an algebra follows from "context conditions" on the specifications m_1 and m_2. Essentially, these conditions make sure that there are no "name clashes". A precise formulation of these context conditions and the pertaining proof that the definition of the module $\mathcal{M}((m_1+m_2))$ is consistent is to be found in Sections 3.2 and 3.3.

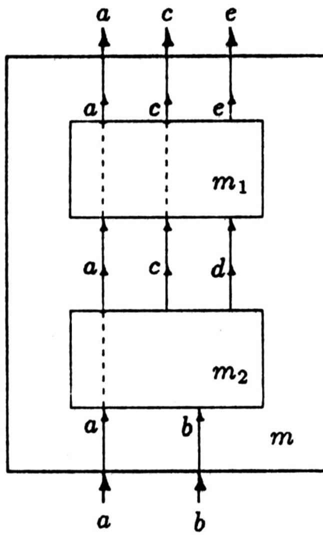
The construct "$\circ$" composes two specifications as illustrated by Figure 1(b). In the terminology of top-down design the specification $(m_1 \circ m_2)$ constitutes a refinement of the specification m_1 by the specification m_2. Hence the construct is similar to the `enrich`-construct of *CLEAR*. Note that according to Figure 1(b) the construct may be applied to specifications m_1 and m_2



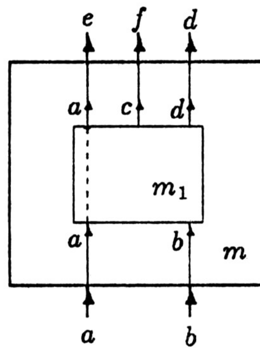
(a) $m = (m_1 + m_2)$



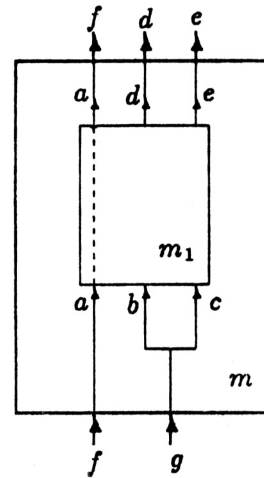
(c) $m = ((a, c) \square m_1)$



(b) $m = (m_1 \circ m_2)$



(d) $m = ([(a, c)/(e, f)]m_1)$



(e) $m = (m_1[(a, b, c)/(f, g, g)])$

FIGURE 1 Graphical illustration of a few constructs of the specification language. In this illustration a specification is represented by a box. The arrows entering a box represent its imported sorts and operations, those leaving a box represent its exported sorts and operations. A dotted line represents an inherited sort or operation. Each of the symbols $a, b, \ldots, f, g$ stands for a sort or an operation.

only if the exported signature of m_2 coincides with the imported signature of m_1.

The next construct allows to forget sorts and operations. More precisly, if m_1 is a specification and lso a list of sorts and operations the specification $(lso \square m_1)$ denotes the module obtained from m_1 by deleting from its exported signature the sorts and operations of lso (see Figure 1(c)). Note that the imported signature remains unchanged. The construct allows, in particular, to get rid of auxiliary (i.e. "hidden") sorts and operations. More importantly, it allows to remove those sorts and operations fail to comply with the closure or congruence conditions of subsequent `subset` or `quotient` constructs.

If m_1 is a specification and $(lso1, lso2)$ a renaming pair then the specification $([lso1/lso2]m_1)$ denotes the module obtained from m_1 by renaming its exported signature according to $(lso1, lso2)$ (see Figure 1(d)). Again, the imported signature remains unchanged. In particular, if an inherited sort or operation is renamed, only its occurrences in the exported signature are modified. The construct may, for instance, be used to avoid name clashes resulting from a subsequent "+"-construct.

Let m_1 and $(lso1, lso2)$ be defined as above. The specification $(m_1[lso1/lso2])$ performs a renaming of the imported signature (see Figure 1(e)). Contrasting with the previous construct the renaming of an inherited sort or operation modifies its occurrences in both the imported and exported signature. The construct allows in particular to simulate the parameter passing mechanism used in the command language: the formal parameters $lso1$ are renamed into actual parameters $lso2$ — as will be explained in Section 5.

Let m be a specification and w a formula. The effect of the specification $(\{w\}m)$ is to make sure that the formula w is valid in the algebra $\mathcal{M}(m)(A)$. The signatures remain unchanged. The construct may be used to express that the data type specified by m satisfies the property denoted by w. Formally, the effect of the construct $(\{w\}m)$ is to restrict the domain of the module $\mathcal{M}(m)$ to those algebras A for which the formula w is valid in the algebra $\mathcal{M}(m)(A)$, i.e. for which

$$\mathcal{M}(m)(A) \models w \tag{1}$$

In practice the user has to prove that (1) holds for all "intended" algebras A. To this end he may make use of the verification unit of the *OBSCURE* environment.

The specification $(m\{w\})$ is similar to the previous one but now the formula w expresses a property of the imported algebra. The construct may be used to explicitly restrict the domain of the module. It will be used in Section 5 to express semantic constraints on the formal parameters of a procedure.

Let m be a specification and w a formula with free occurrences of a single variable. The specification $(w \mid m)$ performs a subalgebra construct. More precisely, $\mathcal{M}((w \mid m))(A)$ is the subalgebra generated by the algebra $\mathcal{M}(m)(A)$ and the formula w (see Section 2.5). The proof that the closure condition is

satisfied is left to the user. Again, he may make use of the verification unit of the *OBSCURE* environment. Note that the module signature remains unchanged but that the meaning of the sort of the variable occurring free in w is "overwritten". The subalgebra construct may, for instance, be used to transform a specification of multisets into a specification of sets by eliminating the multisets containing duplicates. Subalgebra constructs are essential in the algorithmic specification method ([Lo 87]). Algebraic specifications may do without subalgebra constructs but their use may make specifications more transparent and modular.

Finally, the specification $(w \bowtie m)$ performs a quotient algebra construct along the same lines as the subalgebra construct. Again, the user has to prove that the congruence condition of Section 2.5 is satisfied. A quotient algebra may, for instance, be used to transform a specification of lists into a specification of multisets by identifying lists which differ by the order of occurrence of their elements only. The remarks on the necessity of the subalgebra construct carry over.

After this informal overview we now proceed with a formal definition of the syntax and the semantical functions $\mathcal{S}$ and $\mathcal{M}$ of the specification language. The syntax, $\mathcal{S}$ and $\mathcal{M}$ are defined inductively. The syntax and the semantical function $\mathcal{S}$ are defined by simultaneous induction and are treated first.

3.2 The syntax and the semantical function $\mathcal{S}$

The goal of this Section is to define

- a formal language **SPEC** of specifications;
- a function $\mathcal{S}$ mapping any specification of **SPEC** into a module signature.

To this end we start from an algebra signature Σ and a set **AtSPEC**, the elements of which are called *atomic specifications*. It is assumed that a module signature $\mathcal{S}_o(am) \subseteq \Sigma^2$ is associated with each specification am from **AtSPEC**.

The set **SPEC** and the function $\mathcal{S}$ are now defined by simultaneous induction. In this definition $\mathcal{S}$ is defined as a function mapping each specification from **SPEC** into a pair of signatures. That these pairs of signatures are module signatures, i.e. pairs of algebra signatures, is proved in Theorem 4.

Each step in the now following definition is accompanied by conditions denoted $(i), (ii), \ldots$. These conditions are called *context conditions*. The name stems from the fact that these conditions define a subset of the context-free language implicitly introduced by the informal definition of the syntax in Section 3.1. The aim of these context conditions is to guarantee the consistency of the definitions of the semantical functions $\mathcal{S}$ and $\mathcal{M}$, i.e. to make the proofs of Theorem 4 and Theorem 5 feasible.

Most of the formal definitions now following may be clear from the informal description of Section 3.1. Some of the "difficult" context conditions will be shortly discussed after the formal definition. The reader should remember the notations introduced in Section 2.1.1. Moreover, for each specification m $S_i(m)$ and $S_e(m)$ denote the imported and exported signature of the module m. Hence

$$S(m) = (S_i(m), S_e(m)) \quad .$$

Definition (The formal language **SPEC** and the semantical function S)
(Induction basis) If $am \in$ **AtSPEC** then:

- $am \in$ **SPEC**
- $S(am) = S_o(am)$

(Induction step) If $lso, lso1, lso2$ are lists of sorts and operations from Σ, if $w \in \mathrm{WFF}(\Sigma)$ and if $m, m_1, m_2 \in$ **SPEC** then:

(1) if

  (*i*) $S_e(m_1) \cap S_e(m_2) \subseteq S_i(m_1) \cap S_i(m_2)$
  (*ii*) $S_e(m_1) \cap S_i(m_2) \subseteq S_i(m_1)$
  (*iii*) $S_e(m_2) \cap S_i(m_1) \subseteq S_i(m_2)$

  then

  - $(m_1 + m_2) \in$ **SPEC**
  - $S((m_1 + m_2)) = S(m_1) \cup S(m_2)$

(2) if

  (*i*) $S_e(m_2) = S_i(m_1)$
  (*ii*) $S_i(m_2) \cap S_e(m_1) \subseteq S_i(m_1)$

  then

  - $(m_1 \circ m_2) \in$ **SPEC**
  - $S((m_1 \circ m_2)) = (S_i(m_2), S_e(m_1))$

(3) if

  (*i*) $S_e(m) \setminus lso$ is an algebra signature

  then

  - $(lso \square m) \in$ **SPEC**
  - $S((lso \square m)) = (S_i(m), S_e(m) \setminus lso)$

(4) if

(i) $(lso1, lso2)$ is a renaming pair; call ρ the induced renaming

(ii) the renaming ρ is injective on $S_e(m)$

(iii) none of the sorts and operations of $lso2$ are from $S_i(m)$

then

- $([lso1/lso2]m) \in \mathbf{SPEC}$
- $S(([lso1/lso2]m)) = (S_i(m), \rho(S_e(m)))$

(5) if

(i) $(lso1, lso2)$ is a renaming pair; call ρ the induced renaming

(ii) the renaming ρ is injective on the operations of $S_e(m) \setminus S_i(m)$

(iii) the sorts and operations of $lso1$ are all from $S_i(m)$

(iv) $\rho(so) \notin \rho(S_e(m) \setminus S_i(m))$ for each sort or operation so of $S_i(m)$

then

- $(m[lso1/lso2]) \in \mathbf{SPEC}$
- $S((m[lso1/lso2])) = \rho(S(m))$

(6) if

(i) $w \in \mathrm{WFF}(S_e(m))$

then

- $(\{w\}m) \in \mathbf{SPEC}$
- $S((\{w\}m)) = S(m)$

(7) if

(i) $w \in \mathrm{WFF}(S_i(m))$

then

- $(m\{w\}) \in \mathbf{SPEC}$
- $S((m\{w\})) = S(m)$

(8) if

(i) $w \in \mathrm{WFF}(S_e(m))$

(ii) w contains free occurrences of a single variable; call s the sort of this variable

(iii) s is not a sort from $S_i(m)$

then

- $(w \mid m) \in \mathbf{SPEC}$
- $S((w \mid m)) = S(m)$

(9) if

(i) $w \in \mathrm{WFF}(S_e(m))$

(ii) w contains free occurrences of exactly two variables; these variables have the same sort; call s this sort

(iii) s is not a sort from $S_i(m)$

then

- $(w \bowtie m) \in \mathbf{SPEC}$
- $S((w \bowtie m)) = S(m)$ □

Before proceeding we shortly comment on the intuitive meaning of the most "difficult" context conditions. The full significance of these context conditions will become clear in the proofs of Theorem 4 and 5. The context condition (1)(i) expresses that a sort or operation exported by both m_1 and m_2 is an inherited one of m_1 and m_2. The condition (1)(ii) expresses that a sort or operation exported by m_1 and imported by m_2 is inherited by m_1. The condition (1)(iii) is similar. The condition (2)(ii) expresses that a sort or operation exported by m_1 and imported by m_2 is an inherited one. The condition (4)(ii) avoids name clashes within the exported signature. Similarly, (4)(iii) avoids clashes between the new exported names and the imported ones. The condition (5)(iii) allows to rename only imported sorts and operations. Note that contrasting with the preceding construct the renaming has not to be injective on $S_i(m)$, i.e. different names may be given the same new name; the utility of this possibility will become clear during the discussion of the parameter passing mechanism: it must be possible that different formal parameters get the same actual value. The condition (5)(iv) avoids clashes between the new imported names and the (new) non-inherited exported ones. Finally, the condition (5)(ii) expresses that the renaming does not lead to name clashes between the non-inherited exported operations. (Remember that the renaming of an imported sort may modify the arity of a non-inherited exported operation). The conditions (8)(ii) and (9)(ii) refer to the construction of subalgebras and quotient algebras in Section 2.5.

We now prove that S is the desired semantical function, i.e. that the values of S are module signatures.

Theorem 4. $S(m)$ is a module signature for each specification $m \in \mathbf{SPEC}$.

Proof
$S(m)$ has been defined as a pair of signatures. According to the definition of a module signature in Section 2.2 it is sufficient to prove that both signatures are algebra signatures. The proof is by induction on the structure of m and refers to the above Definition.
(Induction basis)
$S(am) = S_o(am)$ is a module signature by assumption.
(Induction step)

(1) $S(m_1) \cup S(m_2) = (S_i(m_1) \cup S_i(m_2), S_e(m_1) \cup S_e(m_2))$ is a module signature by induction hypothesis and Lemma 1.

(2) By induction hypothesis.

(3) By the context condition (3)(i) and the induction hypothesis.

(4) and (5) By induction hypothesis, Lemma 2 and context conditions (4)(i) and (5)(i) respectively.

(6) to (9) By induction hypothesis. □

3.3 The semantical function $\mathcal{M}$

Let the signature Σ, the set of atomic specifications **AtSPEC** and the semantical function S_o on **AtSPEC** be given as above. Let moreover **SPEC** and S be defined as indicated in Section 3.2. The goal of the present Section is to define a function $\mathcal{M}$ mapping any specification of **SPEC**, say m, into an algebra module with module signature $S(m)$. To this end it is assumed that an algebra module $\mathcal{M}_o(am)$ with module signature $S_o(am)$ is associated with each atomic specification am from **AtSPEC**.

The function $\mathcal{M}$ is now defined by its value $\mathcal{M}(m)(A)$ for an arbitrary specification $m \in$ **SPEC** and an arbitrary algebra A of the imported signature $S_i(m)$ of m. That $\mathcal{M}(m)$ is effectively an algebra module will be proved in Theorem 5. The basis of this proof is constituted by the context conditions of the definition of **SPEC**. Hence the comments on these context conditions at the end of Section 3.2 — together with the informal description of the language in Section 3.1 — may help the reader to understand the formal definitions now following.

An algebra module is a partial function. Hence a value $\mathcal{M}(m)(A)$ of $\mathcal{M}(m)$ is not necessarily defined. In fact the last four constructs of the language **SPEC** may introduce partiality, even if the "atomic specifications" $\mathcal{M}_o(am)$ are all total — as will become clear below.

The formal definition of the semantical function $\mathcal{M}$ is by structural induction on its argument m. Hence it closely follows the structure of the inductive definition of **SPEC** in Section 3.1.

Definition (The semantical function $\mathcal{M}$) Writing

$$\text{“}\mathcal{M}(x)(A) = E \text{ iff } C\text{”}$$

as a shorthand for

$$\text{“for all algebras } A \in Alg_{S_i(x)} :$$
$$\mathcal{M}(x)(A) \text{ is defined iff } C \text{ holds; in that case its value is } E\text{”}$$

one defines:

(Induction basis)

$$\mathcal{M}(am)(A) = \mathcal{M}_o(am)(A)$$
$$\text{iff } \mathcal{M}_o(am)(A) \text{ is defined}$$

(Induction step)

(1)

$$\mathcal{M}((m_1 + m_2))(A) = \mathcal{M}(m_1)(A \mid S_i(m_1)) \cup \mathcal{M}(m_2)(A \mid S_i(m_2))$$
$$\text{iff } \mathcal{M}(m_1)(A \mid S_i(m_1)) \text{ and } \mathcal{M}(m_2)(A \mid S_i(m_2)) \text{ are both defined}$$

(2)

$$\mathcal{M}((m_1 \circ m_2))(A) = \mathcal{M}(m_1)(\mathcal{M}(m_2)(A))$$
$$\text{iff } \mathcal{M}(m_2)(A) \text{ and } \mathcal{M}(m_1)(\mathcal{M}(m_2)(A)) \text{ are both defined}$$

(3)

$$\mathcal{M}((lso \,\square\, m))(A) = \mathcal{M}(m)(A) \mid (S_e(m) \setminus lso)$$
$$\text{iff } \mathcal{M}(m)(A) \text{ is defined}$$

(4)

$$\mathcal{M}(([lso1/lso2]m))(A) = (\mathcal{M}(m)(A)) \circ (\rho \mid S_e(m))^{-1}$$
$$\text{iff } \mathcal{M}(m)(A) \text{ is defined}$$

where ρ is the renaming induced by $(lso1,\ lso2)$

(5)

$$\mathcal{M}((m[lso1/lso2]))(A) = \{(so, (\mathcal{M}(m)(A \circ (\rho \mid S_i(m)))(so')) \mid so = \rho(so'), so' \in S_e(m)\}$$
$$\text{iff } \mathcal{M}(m)(A \circ (\rho \mid S_i(m))) \text{ is defined}$$

where ρ is the renaming induced by $(lso1,\ lso2)$

(6)

$$\mathcal{M}((\{w\}m))(A) = \mathcal{M}(m)(A)$$

$$\text{iff } \mathcal{M}(m)(A) \text{ is defined and } \mathcal{M}(m)(A) \models w$$

(7)

$$\mathcal{M}((m\{w\}))(A) = \mathcal{M}(m)(A)$$

$$\text{iff } \mathcal{M}(m)(A) \text{ is defined and } A \models w$$

(8)

$$\mathcal{M}((w \mid m))(A) = \text{ the subalgebra generated by } \mathcal{M}(m)(A) \text{ and } w$$

$$\text{iff } \mathcal{M}(m)(A) \text{ is defined and } \mathcal{M}(m)(A) \text{ satisfies the closure condition}$$

(9)

$$\mathcal{M}((w \bowtie m))(A) = \text{ the quotient algebra generated by } \mathcal{M}(m)(A) \text{ and } w$$

$$\text{iff } \mathcal{M}(m)(A) \text{ is defined}$$

$$\text{and } \mathcal{M}(m)(A) \text{ satisfies the congruence condition}$$

□

Theorem 5. The definition of the semantical function $\mathcal{M}$ is consistent, i.e. for each specification $m \in \mathbf{SPEC}$ it is the case that $\mathcal{M}(m)$ is an algebra module with module signature $S(m)$.

Proof

The proof is by induction on the structure of m. To this end it is sufficient to successively consider the defining equalities of the form

$$\mathcal{M}(x)(A) = E \text{ iff } C$$

and to prove:

(I) whenever C holds the expression E yields an algebra of signature $S_e(x)$ as its value;

(II) $\mathcal{M}(x)$ satisfies the persistency condition, i.e. for each inherited sort or operation so from $S_i(x) \cap S_e(x)$:

$$\mathcal{M}(x)(A)(so) = A(so)$$

The proposition (I) may be replaced by the following three propositions:

(Ia) whenever C holds the value of the expression E is defined;

(Ib) whenever C holds the value of the expression E is a function;

(Ic) this function is an algebra (over $S_e(x)$).

In the now following proof the application of the induction hypothesis is not always explicitly mentioned.

(Induction basis)

The theorem follows from the assumption that $\mathcal{M}_o(am)$ is an algebra module.

(Induction step)

(1) $\mathcal{M}((m_1 + m_2))(A)$

As A is an algebra of signature $S_i(m_1) \cup S_i(m_2)$, $A \mid S_i(m_1)$ is an algebra of signature $S_i(m_1)$. Hence the value $\mathcal{M}(m_1)(A \mid S_i(m_1))$ is well-defined. A similar remark holds for $\mathcal{M}(m_2)(A \mid S_i(m_2))$. This proves (Ia).

To prove (Ib) it is sufficient to prove that any sort or operation $so \in S_e(m_1) \cap S_e(m_2)$ has the same meaning in the algebras $\mathcal{M}(m_1)(A \mid S_i(m_1))$ and $\mathcal{M}(m_2)(A \mid S_i(m_2))$, i.e.

$$\mathcal{M}(m_1)(A \mid S_i(m_1))(so) = \mathcal{M}(m_2)(A \mid S_i(m_2))(so)$$

By context condition (i) one obtains $so \in S_i(m_1) \cap S_i(m_2)$. Hence so is an inherited sort of both m_1 and m_2. By induction hypothesis one obtains

$$\begin{aligned} \mathcal{M}(m_1)(A \mid S_i(m_1))(so) &= (A \mid S_i(m_1))(so) \\ &= A(so) \end{aligned}$$

and a similar equality for m_2. This concludes the proof of (Ib).

(Ic) is a direct consequence of the fact that $\mathcal{M}(m)(A \mid S_i(m_1))$ and $\mathcal{M}(m)(A \mid S_i(m_2))$ are algebras.

To prove (II) let so be an inherited sort or operation of $(m_1 + m_2)$, i.e.

$$so \in (S_i(m_1) \cup S_i(m_2)) \cap (S_e(m_1) \cup S_e(m_2))$$

It has to be proved that so has the same meaning in A and $\mathcal{M}((m_1 + m_2))(A)$. This follows directly from the induction hypothesis using the inclusion

$$(S_i(m_1) \cup S_i(m_2)) \cap (S_e(m_1) \cup S_e(m_2)) \subseteq (S_i(m_1) \cap S_e(m_1)) \cup (S_i(m_2) \cap S_e(m_2))$$

which can be deduced from context conditions (ii) and (iii).

(2) $\mathcal{M}((m_1 \circ m_2))(A)$

(Ia) directly follows from context condition (i).

The proofs of (Ib) and (Ic) are immediate with context condition (i).

Let so be an inherited sort or operation of $(m_1 \circ m_2)$. Hence $so \in S_i(m_2) \cap S_e(m_1)$. By context condition (ii) one obtains $so \in S_i(m_1) \cap S_i(m_2)$. Hence (II) holds.

(3) $\mathcal{M}((lso \,\Box\, m))(A)$

The proofs of (Ia), (Ib) and (Ic) are immediate.

An inherited sort or operation of $(lso \,\Box\, m)$ is an inherited one in m. This proves (II).

(4) $\mathcal{M}(([lso1/lso2]m))(A)$

The renaming ρ is well-defined by context condition (i). The inverse function $(\rho \mid S_e(m))^{-1} : \rho(S_e(m)) \rightarrow S_e(m)$ is well-defined by context condition (ii). Hence (Ia), (Ib) and (Ic) follow from Lemma 3(ii). In particular, the signature of the algebra

$$(\mathcal{M}(m)(A)) \circ (\rho \mid S_e(m))^{-1}$$

is clearly $\rho(S_e(m))$.

In order to prove (II) it suffices to show

$$S_i(m) \cap \rho(S_e(m)) \subseteq S_e(m) \qquad (a)$$

and

$$\rho(so) = so \quad \text{for} \quad so \in S_i(m) \cap \rho(S_e(m)) \qquad (b)$$

In fact we can deduce the validity of (II) from (a) and (b) as follows:

For $so \in S_i(m) \cap \rho(S_e(m))$ we have by (a) :

$$so \in S_i(m) \cap S_e(m)$$

and therefore, with (b) and the induction hypothesis:

$$(\mathcal{M}(m)(A) \circ (\rho \mid S_e(m))^{-1})(so) = (\mathcal{M}(m)(A) \circ (\rho \mid S_e(m))^{-1})(\rho(so))$$

$$= \mathcal{M}(m)(A)(so) = A(so)$$

yielding (II).

It remains to prove (a) and (b).

Let so be a sort or operation from $S_i(m) \cap \rho(S_e(m))$. By context condition (iii) $so \notin lso2$. Let $so' \in S_e(m)$ be the sort for which $\rho(so') = so$.

Then so' cannot be an element of $lso1$. (because $\rho(so') \notin lso2$). If so and so' are sorts, it directly follows

$$so' = \rho(so') = so$$

proving (a) and (b) for sorts.

If so and so' are operations one has

$$\begin{array}{ll} so = & (n : s_1 \ldots s_k \rightarrow s_{k+1}) \\ so' = & (n : s'_1 \ldots s'_k \rightarrow s'_{k+1}) \qquad (k \geq 0) \end{array}$$

with $\rho(s'_i) = s_i$ $(1 \leq i \leq k+1)$ by the very definition of renaming. Since $S_e(m)$ and, by Lemma 1, $S_i(m) \cap \rho(S_e(m))$ are algebra signatures we have

$$\begin{array}{ll} & s'_i \in S_e(m) \\ \text{and} & s_i \in S_i(m) \cap \rho(S_e(m)) \quad (1 \leq i \leq k+1). \end{array}$$

By context condition (iii) $s_i \notin lso2 \cup S_i(m) = \emptyset$, hence

$$s'_i \notin lso1 \qquad (1 \leq i \leq k+1).$$

Hence

$$s'_i = \rho(s'_i) = s_i \qquad (1 \leq i \leq k+1).$$

This yields $so = so'$ proving (a) and (b) for operations.

(5) $\mathcal{M}((m[lso1/lso2])(A)$

Let us first prove (Ia). Clearly, $\rho \mid S_i(m)$ is a function mapping $S_i(m)$ into $\rho(S_i(m))$. By assumption A is an algebra with signature $\rho(S_i(m))$. Hence

$$A \circ (\rho \mid S_i(m))$$

is an algebra with signature $S_i(m)$ (see Lemma 3(i)). As $\mathcal{M}(m)$ is a module with signature $(S_i(m), S_e(m))$, it accepts $A \circ (\rho \mid S_i(m))$ as an argument and yields an algebra $\mathcal{M}(m)(A \circ (\rho \mid S(m)))$ with signature $S_e(m)$. This algebra accepts so' as an argument. Hence the relation denoted by the righthand side of the equality is well-defined.

To prove (Ib) we prove that this relation is a function. In order to shorten the notation we put $B = \mathcal{M}(m)(A \circ (\rho \mid S_i(m)))$. Let $so', so'' \in S_e(m), so' \neq so''$, be sorts or operations such that

$$\rho(so') = \rho(so'') \qquad (c)$$

It is sufficient to prove that

$$B(so') = B(so'')$$

We distinguish three cases:

- so' and so'' are both inherited in m. In that case

$$\begin{aligned} B(so') &= (A \circ (\rho \mid S_i(m)))(so') \quad \text{by persistency} \\ &= A(\rho(so')) \end{aligned}$$

and similarly for so''. Hence the property results from (c).

- so' is inherited in m but so'' is not. Hence $so' \in S_i(m)$ and $so'' \in S_e(m) \setminus S_i(m)$. By context condition (iv), $\rho(so') \notin \rho(S_e(m) \setminus S_i(m))$. Hence $\rho(so') \neq \rho(so'')$ which contradicts (c).

- neither so' nor so'' are inherited in m, i.e. so' and so'' are both from $S_e(m) \setminus S_i(m)$. If so', so'' are sorts, then $\rho(so') = so'$ and $\rho(so'') = so''$ by context condition (iii). Hence $\rho(so') \neq \rho(so'')$ which contradicts (c). If so', so'' are operations then context condition (ii) implies $\rho(so') \neq \rho(so'')$ contradicting (c).

  To prove (Ic) put $C = \mathcal{M}((m[lso1/lso2]))(A)$. We have to show that for any operation $o = (n : s_1 \ldots s_k \rightarrow s_{k+1}) \in \rho(S_e(m))$, $(k \geq 0)$, it is the case that:

  α) The domain of $C(o)$ is contained in $C(s_1) \times \ldots \times C(s_k)$

  β) $C(o)(C(s_1) \times \ldots \times C(s_k)) \subseteq C(s_{k+1})$.

  By definition of C (and B) we have:

$$C(so) = B(so')$$

  for all $so \in \rho(S_e(m))$ and $so' \in S_e(m)$ with $so = \rho(so')$. Let $o' = (n' : s'_1 \ldots s'_k \rightarrow s'_{k+1}) \in S_e(m)$ be such that $\rho(o') = o$. Note that

$$\rho(s'_i) = s_i \quad (1 \leq i \leq k+1) \qquad (d)$$

  The domains of $C(o)$ and $B(o')$ coincide by definition. They are contained in $B(s'_1) \times \ldots \times B(s'_k)$ because B is a $S_e(m)$-algebra. By (d) and the definition of C we obtain

$$C(s_i) = B(s'_i) \qquad (1 \leq i \leq k+1)$$

  proving α). A similar argument shows the validity of β).

Finally, we prove (II). Let

$$so \in \rho(S_i(m)) \cap \rho(S_e(m))$$

be a sort or operation. We have to prove that

$$A(so) = (\mathcal{M}(m)(A \circ (\rho \mid S_i(m))))(so') \qquad (e)$$

for a sort or operation $so' \in S_e(m)$ satisfying $so = \rho(so')$. Now we have the obvious inclusion

$$\rho(S_i(m)) \cap \rho(S_e(m)) \subseteq \rho(S_e(m)) = \rho(S_e(m) \cap S_i(m)) \cup \rho(S_e(m) \setminus S_i(m)).$$

By context condition (iv)

$$\rho(S_i(m)) \cap \rho(S_e(m) \setminus S_i(m)) = \emptyset$$

yielding

$$\rho(S_i(m)) \cap \rho(S_e(m)) \subseteq \rho(S_e(m) \cap S_i(m)).$$

Hence it is possible to choose so' from $S_e(m) \cap S_i(m)$ and

$$\begin{aligned} \mathcal{M}(m)(A \circ (\rho \mid S_i(m)))(so') &= (A \circ (\rho \mid S_i(m)))(so') \\ &= A(\rho(so')) = A(so) \end{aligned}$$

by persistency of $\mathcal{M}(m)$ yielding (e).

(6) $\mathcal{M}((\{w\}m))(A)$

$\mathcal{M}(m)(A) \models w$ is well-defined by context condition (i). This proves (Ia). The proofs of (Ib), (Ic) and (II) are immediate.

(7) $\mathcal{M}((m\{w\}))(A)$

As for (6).

(8) $\mathcal{M}((w \mid m))(A)$

The subalgebra generated by $\mathcal{M}(m)(A)$ and w is well-defined by the context conditions (i) and (ii) (and by the fact that the closure condition is satisfied). This proves (Ia), (Ib) and (Ic).

Let so be an inherited sort or operation of $(w \mid m)$. If it is a sort it cannot be s by context condition (iii). If it is an operation the sort s cannot occur in its arity — again by context condition (iii) and because $S_i(m)$ is an algebra signature. Hence the meaning of so is not modified by the subalgebra construction. This proves (II).

(9) $\mathcal{M}((w \bowtie m))(A)$

As for (8). □

4 Two Generalizations

4.1 Overloading

An operation has been defined as consisting of its name together with its arity. Actually, in terms or formulas it is usual to denote an operation by its name only. This postulates that it is possible to distinguish between operations with the same operation name. This disambiguation is classically performed by type inferencing. To be applicable type inferencing requires that any two operations with the same operation name differ by the number or the sort of their arguments. These notions are now made more precise.

A signature is called *unambiguous* if for any two different operations

$$\begin{array}{ll} n: & s_1 \dots s_k \rightarrow s_{k+1} \\ n: & t_1 \dots t_k \rightarrow t_{k+1} \end{array} ,$$

$k \geq 0$, with the same operation name n and the same number k of arguments there exists i, $1 \leq i \leq k$, such that $s_i \neq t_i$. An unambiguous signature may for instance contain the operations

$$n : s\, t \rightarrow u$$

$$n : s\, t' \rightarrow u$$

(provided $t \neq t'$) but not

$$n : \rightarrow u$$

$$n : \rightarrow t$$

It is easy to adapt the definition of the specification language of Section 3 to algebras with unambiguous signatures. While the definitions of the semantical functions $\mathcal{S}$ and $\mathcal{M}$ remain unchanged the formal language **SPEC** is slightly restricted by additional context conditions. More precisely, each of the constructs

$$(m_1 + m_2)$$

$$(m_1 \circ m_2)$$

$$([lso1/lso2]m)$$

$$(m[lso1/lso2])$$

is provided with additional context conditions expressing that the resulting module signature consists of a pair of unambiguous signatures. For instance. the additional context conditions for the construct $(m_1 + m_2)$ are:

(iv) $\mathcal{S}_i(m_1) \cup \mathcal{S}_i(m_2)$ is unambiguous

(v) $\mathcal{S}_e(m_1) \cup \mathcal{S}_e(m_2)$ is unambiguous

Clearly, these restrictions do not affect the validity of the Theorems 4 and 5.

4.2 Loose specifications

The specification language described in Section 3 is not able to handle loose specifications. In fact, it was assumed that any atomic specification has a unique model (up to isomorphism). To handle loose specifications it is necessary to generalize the notion of an algebra module and to modify the semantics of the specification language accordingly. These generalizations are now shortly described.

A *loose (algebra) module* for the module signature (Σ_i, Σ_e) is a total function

$$M : Alg_{\Sigma_i} \rightarrow \mathcal{P}(Alg_{\Sigma_e})$$

(where $\mathcal{P}(Alg_{\Sigma_e})$ denotes the power set of Alg_{Σ_e}) satisfying the following *persistency condition*:

for each algebra $A \in Alg_{\Sigma_i}$:
for each algebra $B \in M(A)$:
for each inherited sort or operation $c \in \Sigma_i \cap \Sigma_e$:
$B(c) = A(c)$.

Informally, a loose module maps any imported algebra into a set of exported ones. Note that loose modules are total functions while — according to Section 2.2 — non-loose ones are partial. The reason is that in loose modules the undefined value is "simulated" by the empty set.

The generalization of the specification language of Section 3 for loose specifications is straightforward. The definition of the formal language **SPEC** and the semantical function $\mathcal{S}$ remain unchanged. On the other hand $\mathcal{M}_o$ now assigns a loose module $\mathcal{M}_o(am)$ to each atomic specification $am \in$ **AtSPEC**. The definition of the semantical function $\mathcal{M}$ is generalized by componentwise application. More precisely, the definition of Section 3.3 is replaced by:

(Induction basis)

$$\mathcal{M}(am)(A) = \mathcal{M}_o(am)(A)$$

(Induction step)

(1) $\mathcal{M}((m_1 + m_2))(A) =$
$\{B \cup C \mid B \in \mathcal{M}(m_1)(A \mid \mathcal{S}_i(m_1)), C \in \mathcal{M}(m_2)(A \mid \mathcal{S}_i(m_2))\}$

(2) $\mathcal{M}((m_1 \circ m_2))(A) = \bigcup\{\mathcal{M}(m_1)(B) \mid B \in \mathcal{M}(m_2)(A)\}$

(3) $\mathcal{M}((lso \,\Box\, m))(A) = \{B \mid (\mathcal{S}_e(m) \setminus lso) \mid B \in \mathcal{M}(m)(A)\}$

(4) $\mathcal{M}(([lso1/lso2]m))(A) = \{B \circ (\varrho \mid \mathcal{S}_e(m))^{-1} \mid B \in \mathcal{M}(m)(A)\}$
— where ϱ is the renaming induced by $(lso1, lso2)$

(5) $\mathcal{M}((m[lso1/lso2]))(A) =$
$\{\{(so, B(so')) \mid so = \varrho(so'), so' \in \mathcal{S}_e(m)\} \mid B \in \mathcal{M}(m)(A \circ (\varrho \mid \mathcal{S}_i(m)))\}$
— where ϱ is the renaming induced by $(lso1, lso2)$

(6) $\mathcal{M}((\{w\}m))(A) = \{B \in \mathcal{M}(m)(A) \mid B \models w\}$

(7) $\mathcal{M}((m\{w\}))(A) = \begin{cases} \mathcal{M}(m)(A) & \text{if } A \models w \\ \emptyset & \text{otherwise} \end{cases}$

(8)

$$\mathcal{M}((w \mid m))(A) = \\ \{\text{the subalgebra generated by } B \text{ and } w \mid B \in \mathcal{M}(m)(A)\}$$

(9)

$$\mathcal{M}((w \bowtie m))(A) = \\ \{\text{the quotient algebra generated by } B \text{ and } w \mid B \in \mathcal{M}(m)(A)\}.$$

The proof of Theorem 5 carries over without any difficulty (see [Kn 87]).

It is interesting to note that a user of *OBSCURE* has not to explicitly indicate which of the two definitions of $\mathcal{M}$ he uses. If some of his atomic specifications are loose the relevant definition of $\mathcal{M}$ is of course the present one. If none are loose both the present definition and the definition of Section 3.3 apply.

5 The command language

While the specification language described in Section 3 and 4 has a clear mathematical structure it is difficult to write specifications in it using pencil and paper. This difficulty stems from the clumsy notation, the elaborate context conditions and the primitivity of the constructs. To overcome this difficulty *OBSCURE* provides an environment. The design unit of this environment supports the user in the development of specifications written in the specification language. To control this design unit the specifier makes use of a command language. This command language may be viewed as yet another specification language. As such it differs from the specification language described in Section 3 by its syntax, by additional powerful language constructs ("macros") and by the possibility to draw up parameterized specifications. Alternatively, the command language may be viewed as a programming language. A program written in this language generates a specification of the specification language, i.e. yields a specification of **SPEC** as its value. We here adopt the former viewpoint.

The goal of the present Section is to roughly describe the main features by which the command language differs from the specification language. To this end we successively discuss the syntax, the parameterization mechanism and the macros of the command language. A complete and formal description of the command language may be found in [LL 87b]. An illustration of its use in the form of a protocol of a session with the design unit is in [LL 87a].

Syntactically, a specification in the command language consists of a sequence of commands. The language contains, in particular, a command for each construct of the specification language. This linear structure of the command language allows the incremental design of specifications. More precisely, after each command the design unit automatically checks some context conditions and displays the current module signature. In the case of the commands for the constructs of the specification language the context conditions are essentially those of Section 3.2. For a command yielding a subalgebra or a quotient algebra the design unit moreover automatically generates a formula expressing respectively the closure or the congruence condition.

Parameterization is realized with the help of two commands called *procedure declaration* and *procedure call*. A procedure declaration has the form

$$\texttt{is proc}\ n(lso)$$

where n is an (arbitrarily chosen) name and lso a list of sorts and operations from the (current) imported signature. Its effect is to turn the current specification, i.e. the specification drawn up so far, into a procedure. More precisely, n is the name of the procedure, the sorts and operations from lso are its formal parameters, and the current specification constitutes the procedure

body. A procedure call has the form

$$\texttt{call } n(lso')$$

where n is the name of a — previously declared — procedure and lso' a list of sorts and operations constituting the actual parameters. Its effect is to create the specification

$$(m[lso/lso'])$$

where m is the procedure body and lso are the formal parameters of the procedure n; it is understood that $(m[lso/lso'])$ is the construct of Section 3. As a grossly simplified example a specification for the sort *set* (of elements) with the sort *element* among its imported sorts is turned into a procedure with name SET by the command

$$\texttt{is proc } \mathrm{SET}(element)$$

The procedure call

$$\texttt{call } \mathrm{SET}(integer)$$

yields a specification with *integer* instead of *element* among its imported sorts. The carriers of the exported sort *set* are now sets of integers rather than sets of elements.

A macro is essentially a shorthand for a sequence of commands. The role of the macros is to simplify the design of a specification. The command language contains, for instance, a macro performing "generalized composition". Its effect is similar to that of the construct $(m_1 \circ m_2)$ but it circumvents the stringent context condition $(2)(i)$ if Section 3.2.

An implementation of the command language in the form of a design unit should of course offer additional facilities for editing, error correction or rapid prototyping. Details on such an implementation may be found in [FH 87].

References

[BCV 85] Bidoit, M., Choppy, C., Voisin, F., "The ASSPEGIQUE specification environment — Motivations and design", *Int. Rep., Univ. Paris-Sud* (Oct. 1985)

[BG 80] Burstall, R.M., Goguen, J.A., "The semantics of Clear, a specification language", *Proc. 1979 Copenhagen Winter School, LNCS* **86** (1980), pp. 292 - 332

[BGM 87] Bidoit, M., Gaudel, M.C., Mauboussin, A., "How to make algebraic specifications more understandable? — An experiment with the PLUSS specification language", *Int. Rep. 343, Univ. Paris-Sud* (Apr. 1987)

[BHK 86] Bergstra, J.A., Heering, J., Klint, P., "Module Algebra", *Report CS-RXXX, Centre for Math. and Comp. Sc., Amsterdam* (1986), submitted for publication

[BHK 87] Bergstra, J.A., Heering, J., Klint, P., "ASF — An algebraic specification formalism", *Report CS-R 8705, Centre for Math. and Comp. Sc., Amsterdam* (1987)

[Bu 87] Burstall, R.M., "Inductively Defined Functions in Functional Programming Languages", *Int. Rep. ECS-LFCS-87-25, Univ. Edinburgh* (April 1987)

[BW 82] Broy, M., Wirsing, M., "Partial abstract types", *Acta Inform.* **18** (1982), pp. 47 - 64

[Ca 80] Cartwright, R., "A constructive alternative to abstract data type definitions", *Proc. 1980 LISP Conf., Stanford Univ.* (1980), pp. 46 - 55

[Eh 82] Ehrich, H.D., "On the theory of specifications, implementation and parameterization of abstract data types", *Journal ACM* **29** (1982), pp. 206 - 227

[En 72] Enderton, H.B., "A Mathematical Introduction to Logic", *Academic Press* (1972)

[EM 85] Ehrig, H., Mahr, B., "Fundamentals of Algebraic Specification", *Springer-Verlag* (1985)

[EW 85] Ehrig, H., Weber, H., "Algebraic Specification of Modules", in "Formal Models in Programming", *Proc. of the IFIP TC2 Work. Conf. on the Role of Abstr. Models in Inform. Process.* (ed. E.J. Neuhold and G. Chroust), *North-Holland* (1985)

[EW 86] Ehrig, H., Weber, H., "Programming in the Large with Algebraic Module Specifications", *Invited Paper for IFIP-Congress, Dublin* (1986)

[Fe 87] Fey, W., "Concepts, syntax and semantics of ACT-TWO", presented at the *5th Workshop on Specification of Abstract Data Types, Edinburgh* (Sept. 1987)

[FGJM 85] Futatsugi, K., Goguen, J., Jouannaud, J.P., Meseguer, J., "Principles of OBJ2", *Proc. POPL 85* (1985), pp. 52 – 66

[FH 87] Fuchs, J., Hoffmann, A., Loeckx, J., Meiss, L., Philippi, J., Zeyer, J., "Benutzerhandbuch des *OBSCURE*-Systems — Teil 1: Der Editor", *Int. Rep., Univ. Saarbrücken* (1987)

[Gd 84] Gaudel, M.C., "A first introduction to PLUSS", *Int. Rep., Univ. Paris-Sud* (Dec. 1984)

[GHM 78] Guttag, J.V., Horowitz, E., Musser, D.R., "Abstract data types and software validation", *Comm. ACM* **21** (1978), pp. 1048 – 1069

[GTW 78] Goguen, J.A., Thatcher, J.W., Wagner, E.G., "An initial algebra approach to the specification, correctness and implementation of abstract data types", *Current Trends in Programming Methodology IV* (Yeh, R., ed.), *Prentice-Hall* (1978), pp. 80 – 149

[Ho 72] Hoare, C.A.R., "Proof of correctness of data representations", *Acta Inf.* **1**, 4 (1972), pp. 271 – 281

[Hu 87] Hussmann, H., "Rapid Prototyping for Algebraic Specifications — RAP System User's Manual" (Revised edition), *Int. Rep. MIP-8504, Univ. Passau* (1987)

[Kl 84] Klaeren, H.A., "A constructive method for abstract algebraic software specification", *Theor. Comp. Sc.* **30**, 2 (1984), pp. 139 – 204

[Kn 87] Klein, B., "Zwei Erweiterungen von *OBSCURE*", *Diplomarbeit, FB 10, Univ. Saarbrücken* (1987)

[LG 86] Liskov, B., Guttag, J., "Abstraction and specification in program development", *The MIT Electrical Engin. and Comp. Sc. Series, McGraw-Hill* (1986)

[Li 81] Liskov, B., et al, "CLU Reference Manual", *LNCS* **114** (1981)

[LL 87a] Lehmann, T., Loeckx, J., "*OBSCURE*: A specification environment for abstract data types", *Int. Rep. A06/87, Univ. Saarbrücken*, submitted for publication (1987)

[LL 87b] Lehmann, T., Loeckx, J., "The design of specifications in *OBSCURE*", *Int. Rep., Univ. Saarbrücken* (1987)

[Lo 87] Loeckx, J., "Algorithmic Specifications: A Constructive Specification Method for Abstract Data Types", to appear in *TOPLAS* (Oct. 1987)

[LS 87] Loeckx, J., Sieber, K., "The Foundations of Program Verification" (Second edition), *Wiley/Teubner* (1987)

[Mi 72] Milner, R., "Logic for computable functions: description of a machine implementation", *SIGPLAN NOTICES* **7** (1972), pp. 1 – 6

[NY 83] Nakajima, R., Yuasa, T., "The IOTA Programming System", *LNCS* **160** (1983)

[Sa 84] Sannella, D., "A set-theoretic semantics for Clear", *Acta Informatica* **21**, 5 (1984), pp. 443 – 472

[Sh 81] Shaw, M., "ALPHARD, Form and Content", *Springer-Verlag* (1981)

[TWW 82] Thatcher, J.W., Wagner, E.G., Wright, J.B., "Data type specification: Parameterization and the power of specification techniques", *TOPLAS* **4** (1982), pp. 711 – 732

[Wi 86] Wirsing, M., "Structured algebraic specifications: A kernel language", *Theor. Comp. Sc.* **42**, 2 (1986), pp. 124 – 249