OBSCURE, a specification language for abstract data types

by

Thomas Lehmann and Jacques Loeckx

A 19/90

Saarbrücken, November 1990

submitted for publication

Thomas Lehmann and Jacques Loeckx
Fachbereich 14 - Informatik
Im Stadtwald
Universität des Saarlandes
D6600 Saarbrücken

Electronic Mail: lehmann@cs.uni-sb.de, loeckx@cs.uni-sb.de

Contents

1 Introduction

The specification of abstract data types has been studied extensively during the last decade (see e.g. [6]). As a result several specification methods have been proposed. A specification drawn up according to an *algebraic* or *axiomatic specification method* essentially consists of a set of formulas in a given logic. This logic is, for instance, equational logic, conditional equational logic or first-order predicate logic. The semantics of a specification is then defined as a class of models satisfying the set of formulas. This class is, for instance, the class of all models or the class of the "initial" models. In the latter case the models of the class are isomorphic to each other and one speaks of initial semantics. One speaks of loose semantics when the class contains models that are not isomorphic to each other. On the other hand a specification drawn up according to a *constructive specification method* essentially consists of a set of "programs" [4, 11, 15]. Such a specification defines a single model or, equivalently, a class of isomorphic models.

To be feasible the design of large specifications must be modularized. This may be achieved by the use of a specification language. Essentially, a specification language builds specifications out of smaller ones. Recently, several specification languages have been proposed, for instance *Clear* [3, 19], *OBJ3* [10], *ACT-ONE* [6], *ACT-TWO* [8], *PLUSS* [2], *ASL* [24], *ASF* [1], *Extended ML* [21].

The goal of the present paper is to present one more specification language. It is called *OBSCURE*[1] and differs from the other specification languages in

[1]The name is intended to avoid any confusion with the specification language *Clear*.

at least one of the following three aspects.

First, instead of being a language for specifications *OBSCURE* is a language for module specifications (in the sense of [7]) or, equivalently, for parameterized specifications. In other words, a specification in *OBSCURE* is not interpreted as an algebra but as a mapping from algebras into algebras. In this respect *OBSCURE* bears similarities with a specification language such as *ACT-TWO*.

The second feature of *OBSCURE* is more fundamental: *OBSCURE* is "model-oriented" rather than "theory-oriented" in the following sense. Being developed for algebraic specification methods classical specification languages handle with theories (viz. the theories defined by the specifications) rather than with algebras (viz. the models of the specifications). Instead, *OBSCURE* handles with models, viz. with functors mapping algebras into algebras. Establishing a theory for such a model is a problem of logic and is, strictly speaking, outside the realm of the specification language *OBSCURE*. By this feature *OBSCURE* bears similarities with the specification language *Extended ML*. In fact, using the terminology of [20], *OBSCURE* and *Extended ML* both constitute specification languages for parameterized algebras. An advantage of model-oriented specification languages is that they avoid persistency problems. In fact, putting two models together is similar to putting two pieces of program together: the pieces of program do not harm each other. On the other hand, model-oriented specifications may be less abstract than theory-oriented ones. A more precise statement of this point may be found in Section 7.

As a third feature *OBSCURE* does not depend on the specification method used. It is even nearly institution independent. By this feature *OBSCURE* is similar to *Clear*, *ASL* and *Extended ML*.

Section 2 presents an informal overview of the specification language *OBSCURE* and contains examples illustrating the different constructs. Section 3 introduces the terminology used. Section 4 describes a subset of *OBSCURE* called the kernel language. The full language is described in Section 5. Examples illustrating the expressive power of *OBSCURE* are in Section 6. Finally Section 7 discusses some extensions to *OBSCURE* such as the use of loose specifications. Moreover, it shortly presents a specification environment that was recently completed.

2 An informal overview

The goal of this section is to provide an overview of the specification language *OBSCURE*. To this end the main constructs of the language are illustrated on an example. The treatment is very informal but the reader knowing other specification languages may get a flavor of what *OBSCURE* is like. The context-free grammar of the Appendix may also be helpful.

A specification language builds specifications out of others. In order to be able to start such a construction the language has to provide means to draw up specifications "from scratch". In *OBSCURE* such specifications are called *atomic*. They may be drawn up according to any specification method. They may even be pieces of code such as *Ada* packages. In the examples of this section we systematically use the algebraic specification method based on (conditional) equational logic with initial semantics and we assume that the reader is acquainted with this classical method. The use of a constructive specification method, viz. the algorithmic specification method, will be illustrated in Section 6. The following atomic specification specifies lists of natural numbers:

atomspec
import sorts *nat*, *bool*
  opns $\_ =_{nat} \_ : nat \times nat \to bool$
    $if\_then\_else\_fi : bool \times bool \times bool \to bool$
    $true : \to bool$
    $false : \to bool$
create sorts *list*
  opns $\epsilon : \to list$
    $\_ \circ \_ : list \times nat \to list$
    $\_ \in \_ : nat \times list \to bool$
semantics
  vardec $n, m : nat, l : list$
  eqns $(n \in \epsilon) = false$
    $(n \in (l \circ m)) = if\ (n =_{nat} m)\ then\ true\ else\ n \in l\ fi$
endatom

An atomic specification is interpreted as a functor mapping algebras of the imported signature into algebras of the exported signature. The imported signature consists of the sorts and operations following the keyword **import**; the exported signature contains in addition the sorts and operations following the keyword **create**. Hence in the case of the example the imported signature consists of the sorts *nat* and *bool* and of the operations $=_{nat}$, *if_then_else_fi*, *true*, *false*; the exported signature moreover contains the sort *list* and the operations ϵ, $\circ$, $\in$. This functor is strongly persistent (in the sense of [6]) and

hence maps any algebra (of the imported signature) into its free extension. Note that the functor is applicable to *each* algebra of the imported signature and not only to the (intended) algebra consisting of natural numbers and booleans. Hence, in *OBSCURE* the sorts and operations of the imported signature of a specification behave like formal parameters. For this reason a specification in *OBSCURE* bears similarities with a parameterized specification in the sense of [6]. Note also that the example is a "didactical" one: in a practical version of *OBSCURE* — such as the one used in the specification environment of Section 7 — the sorts *bool* and *nat* are "standard" and their operations are imported implicitly.

In order to be able to refer to a specification it is necessary to give it a name. The object thus obtained is called a *module*. An example of such a module is:

module NATLIST **is**

...

endmodule

where ... stands for the text of the above specification.

An *instantiation* (or: *call*) of a module is obtained by writing

NATLIST

This instantiation constitutes a shorthand denotation for the specification contained by the module.

Modules may be parameterized. An example is the following module specifying the parameterized data type "list of elements":

module LIST (**sorts** *el*, **opns** $Eq : el \times el \rightarrow bool$) **is**
atomspec
  import sorts *el*, *bool*
    opns $Eq : el \times el \rightarrow bool$
      $if\_then\_else\_fi : bool \times bool \times bool \rightarrow bool$
      $true : \rightarrow bool$
      $false : \rightarrow bool$
  create sorts *list*
    opns $\epsilon : \rightarrow list$
      $\_ \circ \_ : list \times el \rightarrow list$
      $\_ \in \_ : el \times list \rightarrow bool$
  semantics
    vardec $e, e' : el, l : list$
    eqns $(e \in \epsilon) = false$
      $(e \in (l \circ e')) = if\ Eq(e,e')\ then\ true\ else\ e \in l\ fi$

endatom
endmodule

Note that the specification contained by this module is identical with the one contained by the module NATLIST except for the substitution of the names *nat* and $=_{nat}$ by *el* and *Eq* respectively. This substitution is not essential and was performed for didactical reasons only. A possible instantiation of this module is

$$\text{LIST } (\textbf{sorts } nat, \textbf{ opns } \_ =_{nat} \_ : nat \times nat \rightarrow bool)$$

This instantiation stands for the (text of the) specification contained by the module

$$\text{LIST } (\textbf{sorts } el, \textbf{ opns } Eq : el \times el \rightarrow bool)$$

in which each occurrence of *el* and *Eq* is replaced by *nat* and $=_{nat}$ respectively. Hence, in our example the instantiations NATLIST and

$$\text{LIST } (\textbf{sorts } nat, \textbf{ opns } \_ =_{nat} \_ : nat \times nat \rightarrow bool)$$

happen to denote the same specification. Note again, that a practical version of *OBSCURE* allows abbreviations such as LIST (**sorts** *nat*, **opns** $=_{nat}$).

In general a module may have several parameters each of which may be a sort or an operation. These sorts and operations must be among the imported sorts and operations. The difference between the imported sorts and operations that are parameters and those that are not is merely pragmatic: at each instantiation only the sorts and operations that are parameters are automatically renamed. Note that the parameter passing mechanism of *OBSCURE* is merely based on "renaming": essentially, the formal parameters are substituted by the actual ones in the text of the specification. Note also that in *OBSCURE* the parameters are sorts and/or operations while in most specification languages parameters are modules; in other words, in *OBSCURE* the interface between modules is realized by sorts and operations rather than by modules.

We now want to examine the different constructs of the specification language *OBSCURE*. The constructs performing union, composition, renaming and forgetting are classical and are provided by nearly any specification language. The constructs **e-axioms** and **i-axioms** allow to express semantical

constraints. The constructs **subset** and **quotient** yield subalgebras and quotient algebras respectively. These different constructs are now very informally discussed. For more precision the reader is referred to Section 4.

Consider again the module NATLIST specifying lists of natural numbers. Let similarly NATSTRING be a module specifying strings of natural numbers. The *union* of their specifications is denoted

NATLIST + NATSTRING

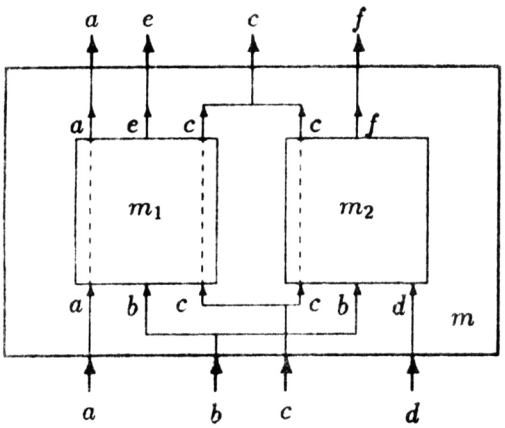
and specifies lists of natural numbers and strings of natural numbers. More precisely, the imported signature of the specification NATLIST + NATSTRING is the union of the imported signatures of NATLIST and NATSTRING — and similarly for the exported signature. The union of two specifications is consistent only under certain conditions. For instance, if a name (of a sort or an operation) occurs in both the exported signature of NATLIST and the exported signature of NATSTRING it must have the same meaning in both specifications in order to avoid an ambiguity ("name clash") in the exported signature of the union. It will be shown that these different consistency conditions are satisfied under certain simple syntactical conditions called *context conditions*. A graphical representation of the union of two specifications may be found in Figure 1(a). Again, it is possible to turn the specification resulting from the union into a module, e.g.

module NAT-LIST-AND-STRING **is**
    NATLIST + NATSTRING
endmodule

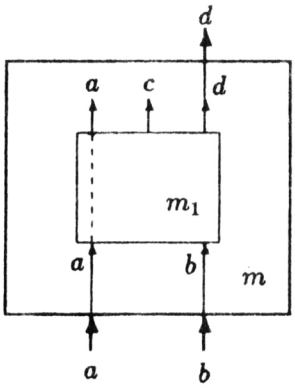
Consider again the module NATLIST. Remember that the imported signature consists of the sort *nat*, the operation $=_{nat}$ and the sort *bool* together with some boolean operations. Let now NAT be a module specifying the natural numbers such that its exported signature coincides with the imported signature of NATLIST; its imported signature contains the sort *bool* together with some boolean operations. The *composition*

NATLIST ∘ NAT

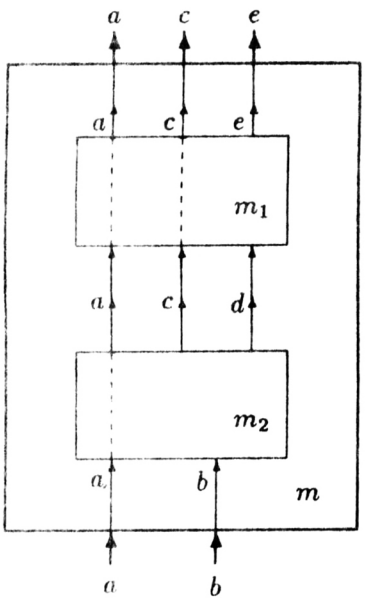
is a specification, the exported signature of which is that of NATLIST and the imported signature of which is that of NAT. Hence composition corresponds to what is sometimes called a refinement step: the top-down design of a specification consists in successively designing modules such as NATLIST and NAT and in composing them — until the imported signature of the resulting specification is empty or, at least, contains only "known" sorts and operations. As for the union — as well as for the constructs discussed below — composition



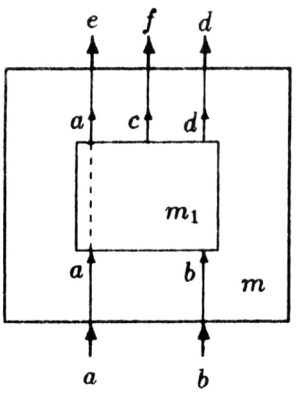
(a) $m = (m_1 + m_2)$



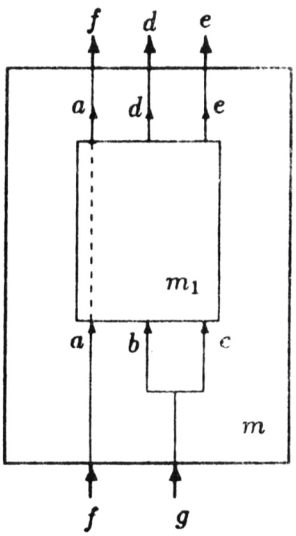
(c) $m = m_1$ **forget** a, c



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



(d) $m = m_1$ **e-rename** a, c **as** e, f



(e) $m = m_1$ **i-rename** a, b, c **as** f, g, g

Figure 1: Graphical illustration of a few constructs of the kernel language of *OBSCURE*. 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.

may only be applied if certain context conditions are satisfied. A graphical representation of the construct may be found in Figure 1(b).

Forgetting (or: *hiding*) allows to get rid of some exported sorts and operations. For instance the specification

NATLIST
forget opns _ ∘ _ : *list* × *nat* → *list*

is identical with NATLIST except for the fact that its exported signature does not contain the operation ∘ (see Figure 1(c)). Forgetting a sort implies forgetting additionally all operations in which this sort occurs. Of course, it is possible to forget sorts or operations from the exported signature only.

It is possible to *rename* sorts and operations from the imported or the exported signature using constructs characterized by the keywords **i-rename** and **e-rename** as illustrated in Figure 1(e) and 1(d) respectively. The latter case is illustrated by the specification:

NATLIST
e-rename sorts *list* **opns** ϵ : → *list*
  as sorts *natlist* **opns** *Enatl* : → *natlist*

The exported signature of this specification is the same as that of NATLIST but with the sort *natlist* (instead of *list*) and the operations

Enatl : → *natlist*
_ ∘ _ : *natlist* × *nat* → *natlist*
_ ∈ _ : *nat* × *natlist* → *bool*

A more elaborate example making use of the parameterized module LIST (and of an abbreviated notation) is the following module. It illustrates how to avoid name clashes during union of both instantiations of the module LIST:

module LISTS-OF-NAT-AND-STRING **is**
  (LIST (**sorts** *nat*, **opns** $=_{nat}$)
   e-rename sorts *list* **opns** ϵ : → *list*
     as sorts *natlist* **opns** *Enatl*)
  +
  (LIST (**sorts** *string*, **opns** $=_{string}$)
   e-rename sorts *list* **opns** ϵ : → *list*
     as sorts *strlist* **opns** *Estrl*)
endmodule

The next construct is the construct of *imported axioms*. It is characterized by the keyword **i-axioms** and allows to express semantical constraints on the imported algebras. The following specification illustrates its use.

module AXLIST (**sorts** *el*, **opns** *Eq* : *el* × *el* → *bool*) **is**

LIST
i-axioms
vardec $e,e',e'' : el$
$Eq(e,e) = true$
$Eq(e,e') = true \supset Eq(e',e) = true$
$Eq(e,e') = true \wedge Eq(e',e'') = true \supset Eq(e,e'') = true$
endmodule

The reader will have understood that

LIST

stands for the module instantiation

LIST (**sorts** *el*, **opns** $Eq : el \times el \rightarrow bool$)

In fact, by notational convention the actual parameters of an instantiation may be omitted if they happen to be identical with the formal ones. Note that the axioms are expressed in first-order predicate logic with equality. Hence $=$, $\supset$ and $\wedge$ are not operations of sort *bool* but are the connectives of predicate logic. Semantically, this module is identical with the module LIST except for the following: the functor representing the meaning of the specification is undefined when applied to an (imported) algebra that fails to satisfy the axioms, i.e. in which Eq is not (interpreted as) an equivalence relation. From a practical point of view the construct of imported axioms obliges the user of the module AXLIST to prove that the axioms hold in the algebra that is effectively imported. In the course of the design of a (complete) specification the user will reach a point where he will know sufficient properties of this algebra to be able to prove (or disprove) this property. He may, for instance, be able to perform the required proof in the "frame" of the specification
AXLIST (**sorts** *nat*, **opns** $=_{nat}$)
$\circ$ NAT

The proof then consists in showing that the operation $=_{nat}$ specified by the module NAT constitutes an equivalence relation. If the proof succeeds, the specification AXLIST(...) $\circ$ NAT is correct in the sense that the axioms are "always" satisfied or, in other words, that the functor representing the meaning of AXLIST(...) $\circ$ NAT is defined whenever the functor defining the meaning of NAT is. Clearly the construct of imported axioms bears strong similarities with a precondition of an imperative program.

The next construct is the construct of *exported axioms* and is characterized by the keyword **e-axioms**. While imported axioms are required to be formulas

of the imported signature, exported axioms must be formulas of the exported signature. The construct may be used to certify that the data type specified satisfies a given property. It consequently bears similarities with a postcondition of an imperative program. A trivial specification illustrating the use of the construct is

LIST
e-axioms
  vardec e, e' : *el*, l : *list*
  $((e \in l) = true) \supset ((e \in l \circ e') = true)$

From a theoretical point of view the construct again restricts the domain of the functor representing the meaning of LIST. From a practical point of view, the user has to prove that the axioms hold in the exported algebra. It is interesting to note that the construct of exported axioms allows to compare different specifications in the following sense. Suppose that an atomic specification is drawn up twice: once with the algebraic specification method and once with the algorithmic one [15]. The equations of the algebraic specification are now added as exported axioms to the algorithmic specification. Loosely speaking one may say that the proof imposed by the construct of exported axioms consists in showing that the algebra defined by the algorithmic specification is a model of the algebraic specification.

The constructs *subset* and *quotient* build subalgebras and quotient algebras respectively. In order to illustrate these constructs we first enrich the module LIST with five operations performing conditional insertion, deleting an element from a list or checking whether a list is a sublist of another list, whether it contains duplicates or whether it contains the same elements as another list:

module RICHLIST (**sorts** *el*, **opns** Eq : $el \times el \rightarrow bool$) **is**
atomspec
import sorts *list*, *el*, *bool*
        opns ...
create opns $Insert$: $list \times el \rightarrow list$
        $Delete$: $list \times el \rightarrow list$
        $\_ \subseteq \_$: $list \times list \rightarrow bool$
        $Nodup$: $list \rightarrow bool$
        Ev : $list \times list \rightarrow bool$
semantics
  vardec e,e' : *el*, l,l' : *list*
  eqns $(e \in l) = true \supset Insert(l,e) = l$
        $(e \in l) = false \supset Insert(l,e) = l \circ e$
        $Delete(\varepsilon,e) = \varepsilon$
        $Eq(e,e') = true \supset Delete(l \circ e',e) = l$
        $Eq(e,e') = false \supset Delete(l \circ e',e) = Delete(l,e) \circ e'$

$(\varepsilon \subseteq l) = true$
$(e \in l') = true \supset (l \circ e \subseteq l') = (l \subseteq Delete(l',e))$
$(e \in l') = false \supset (l \circ e \subseteq l') = false$
$Nodup(\varepsilon) = true$
$(e \in l) = true \supset Nodup(l \circ e) = false$
$(e \in l) = false \supset Nodup(l \circ e) = Nodup(l)$
$(l \subseteq l') = true \supset Ev(l,l') = (l' \subseteq l)$
$(l \subseteq l') = false \supset Ev(l,l') = false$

endatom
∘ LIST
endmodule

We now obtain a specification of multisets by a quotient construct. For didactical reasons we add an export renaming:

module MULTISETS (**sorts** *el*, **opns** *Eq* : $el \times el \rightarrow bool$) **is**
RICHLIST
quotient of sorts *list* **by**
vardec l,l' : *list*. $Ev(l,l') = true$
e-rename sorts *list* **as sorts** *multiset*
endmodule

Informally, the quotient construct identifies lists that differ only by the order of their elements. Clearly, this construction is semantically sound only if the operation *Ev* constitutes a congruence relation. In [15] it is shown how a formula expressing this congruence property may be effectively constructed. Again, it is up to the user to prove the validity of this formula.

By a subset construct we may derive from the specification of multisets a specification of sets. For semantical reasons that will be explained below, we have first to remove the operation ∘:

module SETS (**sorts** *el*, **opns** *Eq* : $el \times el \rightarrow bool$) **is**
MULTISETS
forget opns _ ∘ _ : $multiset \times el \rightarrow multiset$
subset of sorts *multiset* **by**
vardec *m* : *multiset*. $Nodup(m) = true$
e-rename sorts *multiset* **as sorts** *set*
endmodule

Informally, the subset construct removes from the carrier set all multisets with duplicates. Clearly, this construction is semantically sound only if all operations respect the property *Nodup*. *Insert*, for instance, has to yield a multiset without duplicates when applied to a multiset without duplicates. For this reason it was necessary to remove the operation ∘. Again, a formula expressing the consistency of the construct may be effectively constructed [15]

and its validity has to be proved by the user.

Note that the subset and quotient constructs can not be dispensed with when the atomic specifications are drawn up according to the algorithmic specification method. In the case of algebraic specifications the effect of these constructs may be simulated by additional equations such as

$$Insert(Insert(l, e), e') = Insert(Insert(l, e'), e)$$

We nevertheless feel that even for these specifications the use of the subset and quotient constructs may be helpful, because they build subalgebras and quotient algebras in a more explicit and — in some sense — more constructive way.

3 Basic notions

Most of the notions and notation introduced in the section are classical (cf. [6, 5]). As a difference an operation is defined to be an operation name *together with* an arity. This allows to better bridge the gap between the algebraic notion and its syntax in a specification language.

3.1 Algebras

3.1.1 Syntax

We start from two not further specified notions: the notion of a *sort* and the notion of an *operation name*.

An *operation* is a $(k+2)$-tuple

$$n : s_1 \times \ldots \times s_k \rightarrow s$$

where n is an operation name and $s_1, \ldots, s_k, s$ are sorts ($k \geq 0$). The $(k+1)$-tuple $s_1 \times \ldots \times s_k \rightarrow s$ is called the *arity* of the operation. Note that the equality of two operations implies the equality of their names *and* the equality of their arities.

A *signature* is a pair $\Sigma = (S, O)$ where S is a set sorts and O a set of operations such that for each operation

$$n : s_1 \times \ldots \times s_k \rightarrow s$$

of O one has $s_1, \ldots, s_k, s \in S$.

If Σ and Σ' are signatures, expressions such as $\Sigma \cup \Sigma'$ or $\Sigma \subseteq \Sigma'$ are meant componentwise.

3.1.2 Semantics

Let $\Sigma = (S, O)$ be a signature. A (Σ-)*algebra* is given by

(i) a set $A(s)$ for each $s \in S$, called the *carrier set* of sort s;

(ii) a (possibly partial) function

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

for each operation $o = (n : s_1 \times \ldots \times s_k \rightarrow s_{k+1})$ of O, $(k \geq 0)$.

The class of all Σ-algebras is denoted Alg_Σ.

Let $\Sigma \subseteq \Sigma'$ be signatures and let A be a Σ'-algebra. The Σ-algebra obtained by *restriction of A to the signature* Σ is denoted $A \mid \Sigma$.

3.2 Module functors

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

A *module functor* for the module signature (Σ_i, Σ_e) is a (possibly partial) function

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

satisfying the following *persistency condition*:

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

Informally, the persistency condition expresses that the meaning of any inherited sort or operation c has the same meaning in the imported algebra and the exported algebra.

Module functors are used to model the semantics of non-loose *OBSCURE* specifications. A generalization for loose specifications will be briefly discussed in Section 7.

Category theorists should note that these functors have no morphism part (cf. [22]).

3.3 Signature morphisms

3.3.1 Definition

Let $\Sigma = (S, O)$ and $\Sigma' = (S', O')$ be signatures. A *signature morphism* (*on* Σ), say $\sigma : \Sigma \rightarrow \Sigma'$, is a pair $\sigma = (\sigma_S, \sigma_O)$ where

- $\sigma_S : S \rightarrow S'$ is a (total) function;
- $\sigma_O : O \rightarrow O'$ is a (total) function such that for each $o = (n : s_1 \times \ldots \times s_k \rightarrow s)$ from O $(k \geq 0)$, one has

$$\sigma_O(o) = (n' : \sigma_S(s_1) \times \ldots \times \sigma_S(s_k) \rightarrow \sigma_S(s))$$

  for some operation name n'.

Informally, a signature morphism "renames" the signature Σ while respecting the arities of the operations.

If $\sigma : \Sigma \rightarrow \Sigma'$ is a signature morphism and A a Σ'-algebra, the *σ-reduct of A* is the Σ-algebra $A \mid \sigma$ defined by

$$(A \mid \sigma)(c) = A(\sigma(c))$$

for each sort and operation $c \in \Sigma$.

3.3.2 Renaming pairs

In most specification languages signature morphisms are described by so-called renaming pairs indicating the correspondence between the old and the new names. Let Σ be a signature. A *renaming pair* (*on the signature* Σ) is of the form

$$((\textbf{sorts } s_1, \ldots, s_k, \textbf{ opns } o_1, \ldots, o_l), (\textbf{sorts } s'_1, \ldots, s'_k, \textbf{ opns } o'_1, \ldots, o'_l))$$

with $s_1, \ldots s_k, s'_1, \ldots, s'_k$ sorts, $o_1, \ldots, o_l, o'_1, \ldots, o'_l$ operations $(k \geq 0, l \geq 0)$, such that:

(i) $s_1, \ldots, s_k, o_1, \ldots, o_l$ are from Σ;

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

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

(iv) for each i, $1 \leq i \leq k$, the following holds:
if the arity of o_i is $t_1 \times \ldots \times t_p \rightarrow t_{p+1}$ $(p \geq 0)$, then the arity of o'_i is $t'_1 \times \ldots \times t'_p \rightarrow t'_{p+1}$ with

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

for each j $(1 \leq j \leq p+1)$.

Informally, the condition (iv) expresses that the sorts of the new operations are already correctly renamed.

It is understood that the keyword **sorts** may be omitted if $k = 0$. A similar remark holds for **opns**.

We now associate signature morphisms to renaming pairs. Let π be a renaming pair

$$((\textbf{sorts } s_1, \ldots, s_k, \ \textbf{opns } o_1, \ldots, o_l).(\textbf{sorts } s'_1, \ldots, s'_k, \ \textbf{opns } o'_1, \ldots, o'_l))$$

on a signature $\Sigma = (S, O)$. Let $\Sigma' = (S', O')$ be an arbitrary signature with $\Sigma' \supseteq \Sigma$. Put $S'' = S' \cup \{s'_1, \ldots, s'_k\}$ and let $\sigma_S : S' \rightarrow S''$ be given by

$$\sigma_S(s) = \begin{cases} s'_p & \text{if } s = s_p \text{ for some } p, \ 1 \leq p \leq k \\ s & \text{otherwise} \end{cases}$$

for each $s \in S'$. Finally, let $\Sigma'' = (S'', O'')$ be the signature defined by

$$O'' = \{(n : \sigma_S(t_1) \times \ldots \times \sigma_S(t_p) \rightarrow \sigma_S(t)) \mid (n : t_1 \times \ldots \times t_p \rightarrow t) \in O'\} \cup \{o'_1, \ldots, o'_l\}.$$

The *signature morphism on* Σ' $\sigma : \Sigma' \rightarrow \Sigma''$ *induced by the renaming pair* π is the pair $\sigma = (\sigma_S, \sigma_O)$ where $\sigma_O : O' \rightarrow O''$ and

$$\sigma_O(o) = \begin{cases} o'_q & \text{if } o = o_q \text{ for some } q \ (1 \leq q \leq l) \\ n : \sigma_S(t_1) \times \ldots \times \sigma_S(t_p) \rightarrow \sigma_S(t) & \text{otherwise} \end{cases}$$

for each $o = (n : t_1 \times \ldots \times t_p \rightarrow t) \in O'$ $(p \geq 0)$.

3.4 Subalgebras, quotient algebras

We now introduce two constructions yielding subalgebras and quotient algebras respectively. They constitute special cases of the classical ones (see e.g. [6]) in that they "act" on a single sort.

Let Σ be a signature and A a Σ–algebra. Let further s be a sort from Σ and P a subset of the carrier set $A(s)$. The *subalgebra generated by the algebra A and the set P* is the Σ–algebra B defined by:

(i) for each sort t from Σ:

$$B(t) = \begin{cases} P & \text{if } t = s \\ A(t) & \text{otherwise} \end{cases}$$

(ii) for each operation $o = (n : t_1 \times \cdots \times t_k \to t_{k+1})$ from Σ:

$$B(o) = A(o) \mid (B(t_1) \times \ldots \times B(t_k))$$

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

> for each operation $o = (n : t_1 \times \ldots \times t_k \to t_{k+1})$ from Σ $(k \geq 0)$:
>
> $$A(o)(B(t_1) \times \ldots \times B(t_k)) \subseteq B(t_{k+1})$$

Informally, the closure condition expresses that elements from the subset are mapped into elements of the subset.

Let Σ, A and s be as above. Let Q be an equivalence relation in the carrier set $A(s)$. In order to simplify the wording of the definitions it is useful to provide the other carrier sets with an equivalence relation as well, viz. with the identity relation:
for all sorts t from Σ:

$$Q_t = \begin{cases} Q & \text{if } t = s \\ \{(a,a) \mid a \in A(t)\} & \text{else.} \end{cases}$$

The *quotient algebra generated by the algebra A and the equivalence relation Q* is the Σ–algebra B defined by:

(i) for each sort t from Σ:

$$B(t) = \{[a] \mid a \in A(t)\}$$

(where $[a]$ denotes the equivalence class of a generated by Q_t)

(ii) for each operation $o = (n : t_1 \times \ldots \times t_k \rightarrow t_{k+1})$ from Σ $(k \geq 0)$:

$$B(o)([a_1], \ldots, [a_k]) = \begin{cases} [A(o)(a_1, \ldots, a_k)] & \text{if } A(o)(a_1, \ldots, a_k) \text{ is defined} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

for each $a_i \in A(t_i)$ $(1 \leq i \leq k)$.

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

> for each operation $o : t_1 \times \ldots \times t_k \rightarrow t_{k+1}$ $(k \geq 0)$ and for all $a_i, b_i \in A(t_i)$ with $(a_i, b_i) \in Q_{t_i}$ $(1 \leq i \leq k)$:
> *either* $A(o)(a_1, \ldots, a_k)$ and $A(o)(b_1, \ldots, b_k)$ are both defined and $(A(o)(a_1, \ldots, a_k), A(o)(b_1, \ldots, b_k)) \in Q_{t_{k+1}}$
> *or* $A(o)(a_1, \ldots, a_k)$ and $A(o)(b_1, \ldots, b_k)$ are both undefined.

Informally, the condition expresses that equivalent arguments lead to equivalent values.

3.5 Logic

As we want *OBSCURE* to be compatible with different logics, we do not want to fix a particular logic. Instead we merely assume that the logic satisfies the following properties:

(i) for each signature Σ the logic provides a set $\mathbf{WFF}(\Sigma)$ of syntactical objects called *formulas*;

(ii) for each signature Σ the logic provides a relation $\models$, called *satisfaction relation*, with

$$\models \subseteq Alg_\Sigma \times \mathbf{WFF}(\Sigma)$$

(iii) for each signature Σ and each sort s

- there exist formulas, called *properties* (*of sort* s), that define a subset of the carrier set $A(s)$ for each algebra $A \in Alg_\Sigma$;
- there exist formulas, called *relations* (*of sort* s), that define a relation in the carrier set $A(s)$ for each algebra $A \in Alg_\Sigma$.

We now shortly comment on this definition.

In the case we restrict ourselves to algebras with total operations a possible logic is first-order predicate logic. Let us consider the exported signature of the module RICHLIST of Section 2 and let l, l' be variables of sort *list*. A property is, for instance, the formula

$$Nodup(l) = true$$

and a relation

$$Ev(l, l') = true.$$

In the case partial operations are allowed possible logics are, for instance, *LCF* [17] or the logic presented in [15].

4 The kernel language

The kernel language constitutes a subset of *OBSCURE*. It contains no (parameterized) modules and only the "essential" constructs. The syntax is very elementary but sufficient to allow a precise and formal description of the semantics.

The kernel language is a language of specifications with non-loose semantics. A generalization for specifications with loose semantics is treated in Section 7.

The description of the kernel language is parameterized for the specification method and the logic. More precisely, the description is based on a not further defined set **At** of atomic specifications and on a not further defined logic. It is merely assumed that the specification method according to which the atomic specifications have been drawn up associates with each specification of **At** a module signature and a module functor.

4.1 Syntax

The kernel language is defined as a set of specifications. Its syntax is defined in a classical way by a context-free grammar and by context conditions defining a subset of the language defined by this grammar.

4.1.1 The context-free grammar

In the now following context-free grammar the nonterminals $<spec>$, $<atomspec>$, $<lso>$ and $<formula>$ stand for "specification", "atomic specification", "list of sorts and operations" and "formula" respectively.

$$
\begin{array}{lll}
<spec> & ::= & <atomspec> \mid \qquad \text{(R1)} \\
 & & (<spec> + <spec>) \mid \qquad \text{(R2)} \\
 & & (<spec> \circ <spec>) \mid \qquad \text{(R3)} \\
 & & (<spec> \textbf{ forget } <lso>) \mid \qquad \text{(R4)} \\
 & & (<spec> \textbf{ e-rename } <lso> \textbf{ as } <lso>) \mid \qquad \text{(R5)} \\
 & & (<spec> \textbf{ i-rename } <lso> \textbf{ as } <lso>) \mid \qquad \text{(R6)} \\
 & & (<spec> \textbf{ e-axioms } <formula>) \mid \qquad \text{(R7)} \\
 & & (<spec> \textbf{ i-axioms } <formula>) \mid \qquad \text{(R8)} \\
 & & (<spec> \textbf{ subset of } <sort> \textbf{ by } <formula>) \mid \qquad \text{(R9)} \\
 & & (<spec> \textbf{ quotient of } <sort> \textbf{ by } <formula>) \mid \qquad \text{(R10)}
\end{array}
$$

$$
\begin{array}{lll}
<lso> & ::= & \textbf{sorts } <ls>, \textbf{ opns } <lo> \mid \textbf{sorts } <ls> \mid \textbf{opns } <lo> \\
<ls> & ::= & <sort> \mid <ls>, <sort> \\
<lo> & ::= & <operation> \mid <lo>, <operation> \\
<operation> & ::= & <operation\ name> : \rightarrow <sort> \mid \\
 & & <operation\ name> : <ls> \rightarrow <sort>
\end{array}
$$

The non-terminals $<atomspec>$ and $<formula>$ are not further defined in order to account for the independence from the specification method and the logic. The non-terminals $<sort>$ and $<operation\ name>$ are not further defined in order to avoid fixing notational details that are irrelevant at this stage of the description.

The brackets of the rules (R2) to (R10) may be dropped whenever no ambiguity arises and in the case of left associativity.

4.1.2 The context conditions

The context conditions define a subset of the context-free language defined above. This language is called the *kernel language* of *OBSCURE*; its sentences are called *specifications*.

As the context conditions make use of the module signature, it is appropriate to define simultaneously a function, say $\mathcal{S}$, mapping the specifications from the kernel language into their module signatures. The function $\mathcal{S}$ is called the *signature function*. We define the notations $\mathcal{S}_i$ and $\mathcal{S}_e$ by

$$\mathcal{S}(m) = (\mathcal{S}_i(m), \mathcal{S}_e(m))$$

for any specification m from the kernel language.

The now following inductive definition of the set of specifications constituting the kernel language and of the signature function $\mathcal{S}$ follows the pattern of the context-free rules (R1) to (R10). The context conditions are marked with (i), (ii), ... Their main role is to make sure that the values of the function $\mathcal{S}$ are module signatures. The intuitive meaning of the different context conditions is commented after Definition 1. The definition of the signature function is illustrated on Figure 1. By the way, Definition 1 defines the kernel language completely in that it makes the context-free rules (R1) to (R10) superfluous.

Definition 1 The set of all specifications constituting the kernel language and the signature function $\mathcal{S}$ are defined inductively as follows:

(1) Each atomic specification at from **At** is a specification (of the kernel language); the module signature $\mathcal{S}(at)$ is fixed by the specification method according to which at is drawn up.

(2) (*Union*) if m_1, m_2 are specifications and if

(i) $\mathcal{S}_e(m_1) \cap \mathcal{S}_e(m_2) \subseteq \mathcal{S}_i(m_1) \cap \mathcal{S}_i(m_2)$

(ii) $\mathcal{S}_e(m_1) \cap \mathcal{S}_i(m_2) \subseteq \mathcal{S}_i(m_1)$

(iii) $\mathcal{S}_e(m_2) \cap \mathcal{S}_i(m_1) \subseteq \mathcal{S}_i(m_2)$

then

- $(m_1 + m_2)$ is a specification
- $\mathcal{S}(m_1 + m_2) = \mathcal{S}(m_1) \cup \mathcal{S}(m_2)$;

(3) (*Composition*) if m_1, m_2 and if

(i) $\mathcal{S}_e(m_2) = \mathcal{S}_i(m_1)$

(ii) $\mathcal{S}_i(m_2) \cap \mathcal{S}_e(m_1) \subseteq \mathcal{S}_i(m_1)$

then

- $(m_1 \circ m_2)$ is a specification
- $\mathcal{S}(m_1 \circ m_2) = (\mathcal{S}_i(m_2), \mathcal{S}_e(m_1))$;

(4) (*Forgetting*) if m is a specification, if *lso* is a list of sorts and operations and if

(i) $\mathcal{S}_e(m) \setminus lso$ is a signature[2]

then

- (m **forget** *lso*) is a specification
- $\mathcal{S}(m$ **forget** $lso) = (\mathcal{S}_i(m), \mathcal{S}_e(m) \setminus lso)$;

[2] In this notation the list *lso* of sorts and operations is identified with the (pair consisting of the) corresponding sets.

(5) (*Export renaming*) if m is a specification, if *lso*1, *lso*2 are lists of sorts and operations and if

(i) $(lso1, lso2)$ is a renaming pair on the signature $\mathcal{S}_e(m)$; call ρ the signature morphism on $\mathcal{S}_e(m)$ induced by $(lso1, lso2)$

(ii) the signature morphism ρ is injective on $\mathcal{S}_e(m)$

(iii) none of the sorts and operations of *lso*2 are from $\mathcal{S}_i(m)$

then

- (m **e-rename** *lso*1 **as** *lso*2) is a specification
- $\mathcal{S}(m$ **e-rename** *lso*1 **as** *lso*2$) = (\mathcal{S}_i(m), \rho(\mathcal{S}_e(m)))$;

(6) (*Import renaming*) if m is a specification and *lso*1, *lso*2 are lists of sorts and operations, and if

(i) $(lso1, lso2)$ is a renaming pair on the signature $\mathcal{S}_i(m)$; call ρ the signature morphism on $\mathcal{S}_i(m) \cup \mathcal{S}_e(m)$ induced by $(lso1, lso2)$

(ii) the signature morphism ρ is injective on the operations of $\mathcal{S}_e(m) \setminus \mathcal{S}_i(m)$

(iii) the sorts and operations of *lso*1 are all from $\mathcal{S}_i(m)$

(iv) $\rho(so) \notin \rho(\mathcal{S}_e(m) \setminus \mathcal{S}_i(m))$ for each sort or operation so of $\mathcal{S}_i(m)$

then

- (m **i-rename** *lso*1 **as** *lso*2) is a specification
- $\mathcal{S}(m$ **i-rename** *lso*1 **as** *lso*2$) = \rho(\mathcal{S}(m))$;

(7) (*Export axioms*) if m is a specification and if

(i) $w \in \mathrm{WFF}(\mathcal{S}_e(m))$

then

- (m **e-axioms** w) is a specification
- $\mathcal{S}(m$ **e-axioms** $w) = \mathcal{S}(m)$;

(8) (*Import axioms*) if m is a specification and if

(i) $w \in \mathrm{WFF}(\mathcal{S}_i(m))$

then

- (m **i-axioms** w) is a specification
- $\mathcal{S}(m$ **i-axioms** $w) = \mathcal{S}(m)$;

(9) (*Subalgebra*) if m is a specification and if

(i) $w \in \mathrm{WFF}(\mathcal{S}_e(m))$

(ii) w is a property of sort s

(iii) s is a sort from $\mathcal{S}_e(m) \setminus \mathcal{S}_i(m)$

then

- (m **subset of** s **by** w) is a specification
- $\mathcal{S}(m$ **subset of** s **by** $w) = \mathcal{S}(m)$;

(10) (*Quotient algebra*) if m is a specification and if

(i) $w \in \mathrm{WFF}(\mathcal{S}_e(m))$

(ii) w is a relation of sort s

(iii) s is a sort from $\mathcal{S}_e(m) \setminus \mathcal{S}_i(m)$

then

- (m **quotient of** s **by** w) is a specification
- $\mathcal{S}(m$ **quotient of** s **by** $w) = \mathcal{S}(m)$. □

The significance of these context conditions becomes clear in the proofs of Theorem 1 and 2. We now shortly comment on the most "difficult" ones. The context condition (2)(i) expresses that a sort or operation exported by both m_1 and m_2 is inherited by m_1 and m_2. The condition (2)(ii) expresses that a sort or operation exported by m_1 and imported by m_2 is inherited by m_1. The condition (2)(iii) is similar. The condition (3)(ii) expresses that a sort or operation exported by m_1 and imported by m_2 is inherited by m_1 and m_2. The condition (5)(ii) avoids name clashes within the exported signature. Similarly, (5)(iii) avoids clashes between the new exported names and the imported ones. The condition (6)(iii) allows to rename only imported sorts and operations. Note that contrasting with the preceding construct the renaming has not to be injective on $\mathcal{S}_i(m)$, i.e. different names may be given the same new name; the utility of this possibility will become clear in the discussion of the parameter passing mechanism in Section 5.2: it must be possible that different formal parameters get the same actual value. The condition (6)(iv) avoids clashes between the new imported names and the (new) non-inherited exported ones. Note that according to condition (6)(i) the renaming pair is on the signature $\mathcal{S}_i(m)$ but the signature morphism on the signature $\mathcal{S}_i(m) \cup \mathcal{S}_e(m)$. This accounts for the fact that the renaming of a sort from $\mathcal{S}_i(m)$ may modify the arity of an operation from $\mathcal{S}_e(m) \setminus \mathcal{S}_i(m)$. Finally, the condition (6)(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 (9)(ii) and (10)(ii) refer to the construction of subalgebras and quotient algebras in Section 3.4 and Section 3.5. The following theorem states that the context conditions guarantee that the range of the signature function $\mathcal{S}$ consists of module signatures.

Theorem 1 *$\mathcal{S}(m)$ is a module signature for each specification m of the kernel language.* □

The proof of the theorem may be found in [14].

4.2 Semantics

The semantics of the kernel language is defined denotationally. More precisely, a function $\mathcal{M}$ called *meaning function* is introduced that maps each specification of the kernel language into a module functor. Theorem 2 states that the values of the meaning function $\mathcal{M}$ resulting from the now following Definition 2 are effectively module functors satisfying the persistency condition. The definition of $\mathcal{M}$ is along the same pattern as the definition of the signature function $\mathcal{S}$ in Definition 1: The cases (1) to (10) and the notation correspond to those of Definition 1. In order to abbreviate the definition we use the following shorthand notation for the cases (2) to (10): instead of

$$\text{"for all algebras } A \in Alg_{\mathcal{S}_i(n)} : \quad \mathcal{M}(n)(A) = \begin{cases} E & \text{if } C \text{ holds} \\ \textit{undefined} & \text{otherwise"} \end{cases}$$

we write "$\mathcal{M}(n)(A) = E$ iff C".

Definition 2 The meaning function $\mathcal{M}$ is defined inductively as follows:

(1) for each atomic specification at, $\mathcal{M}(at)$ is the module functor associated with this atomic specification;

(2) (*Union*)

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

iff $\mathcal{M}(m_1)(A \mid \mathcal{S}_i(m_1))$ and $\mathcal{M}(m_2)(A \mid \mathcal{S}_i(m_2))$ are both defined;

(3) (*Composition*)

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

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

(4) (*Forgetting*)

$$\mathcal{M}(m \textbf{ forget } lso)(A) = \mathcal{M}(m)(A) \mid (\mathcal{S}_e(m) \setminus lso)$$

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

(5) (*Export renaming*)

$$\mathcal{M}(m \textbf{ e-rename } lso1 \textbf{ as } lso2)(A) = \mathcal{M}(m)(A) \mid \rho^{-1}$$

iff $\mathcal{M}(m)(A)$ is defined where ρ is the signature morphism on $\mathcal{S}_e(m)$ induced by the renaming pair $(lso1, lso2)$;

(6) (*Import renaming*)
$\mathcal{M}(m$ **i-rename** $lso1$ **as** $lso2)(A)$ is defined iff $\mathcal{M}(m)(A \mid (\rho \mid S_i(m)))$ is defined where ρ is the signature morphism on $S_i(m) \cup S_e(m)$ induced by the renaming pair $(lso1, lso2)$;
in this case $\mathcal{M}(m$ **i-rename** $lso1$ **as** $lso2)(A)$ is the $\rho(S_e(m))$-**algebra** given by

$$\mathcal{M}(m \textbf{ i-rename } lso1 \textbf{ as } lso2)(A)(\rho(so)) = \mathcal{M}(m)(A \mid (\rho \mid S_i(m)))(so)$$

where so is a sort (or operation) from $S_e(m)$;

(7) (*Export axioms*)

$$\mathcal{M}(m \textbf{ e-axioms } w)(A) = \mathcal{M}(m)(A)$$

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

(8) (*Import axioms*)

$$\mathcal{M}(m \textbf{ i-axioms } w)(A) = \mathcal{M}(m)(A)$$

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

(9) (*Subalgebra*)
$\mathcal{M}(m$ **subset of** s **by** $w)(A) =$
the subalgebra generated by the algebra $\mathcal{M}(m)(A)$ and the subset of the carrier set $A(s)$ defined by the property w
iff $\mathcal{M}(m)(A)$ is defined and the algebra $\mathcal{M}(m)(A)$ satisfies the closure condition of Section 3.4;

(10) (*Quotient algebra*)
$\mathcal{M}(m$ **quotient of** s **by** $w)(A) =$
the quotient algebra generated by the algebra $\mathcal{M}(m)(A)$ and the relation Q in the carrier set $A(s)$ defined by the formula w
iff $\mathcal{M}(m)(A)$ is defined, Q is an equivalence relation and $\mathcal{M}(m)(A)$ satisfies the congruence condition of Section 3.4. □

We now shortly comment on this definition.

As for case (1) it should be remembered that we restrict ourselves to specification methods with non-loose semantics. In the particular case of algebraic specifications with initial semantics the module functor $\mathcal{M}(at)$ is essentially the functor, say $\mathcal{I}(at)$, mapping each algebra of the imported signature into its free extension. The main difference between $\mathcal{M}(at)$ and $\mathcal{I}(at)$ is that $\mathcal{M}(at)(A)$ is undefined whenever $\mathcal{I}(at)(A)$ is not a strongly persistent extension of the (imported) algebra A. In the particular case of algorithmic specifications the module functor $\mathcal{M}(at)$ is always total (see [15]).

The cases (2) and (3) are illustrated by Figure 1(a) and 1(b). The cases (4), (5), (6) may seem complicated but correspond exactly to what is intuitively intended by forgetting and renaming (see Figure 1(c), 1(d), 1(e)). By context conditions (5)(i) and (ii), ρ in (5) is an injective signature morphism $\mathcal{S}_e(m) \rightarrow \Sigma'$, where Σ' is constructed as described in Section 3.3 and therefore the signature morphism $\rho^{-1} : \rho(\mathcal{S}_e(m)) \rightarrow \mathcal{S}_e(m)$ may be formed. In (6), the signature morphism ρ is on $\mathcal{S}_e(m) \cup \mathcal{S}_i(m)$ by the context condition (6)(i); hence it may therefore be restricted to $\mathcal{S}_i(m)$. In the cases (7) and (8) the domain of the module functor is restricted to those (imported) algebras that lead the formula w to be satisfied. The cases (9) and (10) are best illustrated by the modules SETS and MULTISETS of Section 2.

The following theorem states that the definition of the meaning function is consistent and, in particular, that its range consists of module functors satisfying the persistency condition of Section 3.2.

Theorem 2 *For each specification m of the kernel language $\mathcal{M}(m)$ is a module functor for the module signature $\mathcal{S}(m)$.* □

The proof of the theorem may be found in [14]. It heavily relies on the context conditions of Definition 1.

5 The full language

The kernel language described above is too elementary for practical use. In particular, it has no module concept, does not allow parameterization, possesses constructs — such as the construct of composition — that are too primitive, leads to a clumsy notation and does not allow a strong typing of terms. The full language of *OBSCURE* is intended to avoid these deficiencies. It is defined as the extension of the kernel language.

A context free grammar of *OBSCURE* that does not take into account the syntactical sugar of Section 5.4 may be found in the Appendix.

5.1 Modules

The kernel language is first extended by a concept for non-parameterized modules. The case of parameterized modules is delayed until Section 5.2.

Sections 5.1.1 and 5.1.2 introduce the notion of a module and generalize the

notion of a specification. Section 5.1.3 shortly discusses the notion of a modularized specification.

The description of a module concept requires the introduction of an environment. Informally, an environment is a table listing the modules already introduced. Formally, an *environment* is a partial function mapping module names into specifications. In Sections 5.1.1 and 5.1.2 we assume that the specifications into which an environment maps module names are specifications of the kernel language. The general case is shortly discussed in Section 5.1.3.

5.1.1 Syntax

The context-free grammar of the kernel language is augmented by two rules. They define *module declarations* and *module instantiations* respectively:

< *module* >	:: =	**module** < *module name* > **is**	
		< *spec* > **endmodule**	(M1)
< *spec* >	:: =	< *module name* >	(R11)

The definition of the non-terminal < *module name* > is left pending. Note that we are now interested in two notions, viz. modules and specifications.

The context conditions now moreover depend on the environment in which the syntactical unit occurs. In other words, the set of all (syntactically correct) modules and the set of all (syntactically correct) specifications are defined with respect to an environment. In order to cope with this new situation, the signature function $\mathcal{S}$ of Section 4.1.2 is generalized in a classical way: instead of mapping specifications into module signatures the function $\mathcal{S}$ maps the specifications into functions which in their turn map environments into module signatures. We are now able to express the context conditions for the rules (M1) and (R11) and to update those for the rules (R1) to (R10). More precisely, Definition 3 constitutes a generalization of Definition 1 (Section 4.1.2) and inductively defines the set of all modules and the set of all specifications for a given environment.

Definition 3 Let *env* be an arbitrary environment mapping module names into specifications of the kernel language. The set of all modules for *env*, the set of all specifications for *env* and the signature function $\mathcal{S}$ are inductively defined as follows:

(M1) (*Module declaration*) if n is a module name and m a specification for the environment *env*, and if

(i) $env(n)$ is undefined

then

- **module** n **is** m **endmodule** is a module for the environment *env*

(R1) to (R10) As (1) to (10) of **Definition** 1 but with each **expression** of the form $\mathcal{S}(m)$ replaced by $\mathcal{S}(m)(env)$. Moreover, instead of

"... **specification(s)** ..."

one should read

"... specification(s) for the environment *env*".

(R11) (*Module instantiation*) if n **is a module name and** if

(i) $env(n)$ is defined

then

- n is a specification for the environment *env*
- $\mathcal{S}(n)(env) = \mathcal{S}(env(n))(env)$ □

Informally, context condition (M1)(i) expresses that the name n is "new". The context condition (R11)(i) requires that the module name already "exists" i.e. has been "already" declared. Note that together these context conditions exclude "recursive instantiations".

5.1.2 Semantics

The meaning function $\mathcal{M}$ is generalized in the same way as the signature function $\mathcal{S}$ above. We adopt the shorthand notation of Definition 2 (Section 4.2) as well as the notation of the context conditions above.

Definition 4 The meaning function $\mathcal{M}$ is defined inductively:

(R1) to (R10) As for (1) to (10) of Definition 2 but with all expressions of the form $\mathcal{M}(m)(A)$ replaced by $\mathcal{M}(m)(env)(A)$.

(R11) (*Module instantiation*)

$$\mathcal{M}(n)(env)(A) = \mathcal{M}(env(n))(env)(A)$$

iff $\mathcal{M}(env(n))(env)(A)$ is defined. □

5.1.3 Modularized specifications

The modularized design of a specification essentially consists in drawing up an environment by successively designing its modules. Alternatively, a *modularized specification* may be considered as a sequence of modules; starting from an empty or a "standard" environment this sequence builds up the desired environment.

We dispense with a formal definition of these notions. Actually, such a definition is classical and puts no problems. The main difference with the definitions above is that (the semantics of) a module declaration has now a side-effect in that it adds the specification to the environment. Moreover, an environment now maps module names into specifications that may contain module instantiations. By the way, the context conditions make sure that an environment built up by a modularized specification is hierarchical in the following way: consider the graph by making an arrow lead from a module name m to a module name n if and only if the specification of m contains an instantiation of n. This graph contains no cycles.

5.2 Parameterized modules

Informally, a module declaration and a module instantiation now both contain a list of sorts and operations. These lists may be viewed as the formal and actual parameter lists respectively. Together they constitute a renaming pair that may be used in an import renaming construct. This import renaming constitutes the parameter passing mechanism.

Formally, the context free grammar is extended by two rules:

$<$*module*$>$	::=	**module** $<$*module name*$>$($<$*lso*$>$) **is**	
		$<$*spec*$>$ **endmodule**	(M2)
$<$*spec*$>$	::=	$<$*module name*$>$($<$*lso*$>$)	(R12)

An example of a module declaration is (cf. Section 2):

module LIST (**sorts** *el*, **opns** $Eq : el \times el \rightarrow bool$) **is** ... **endmodule**

A corresponding instantiation is:

LIST (**sorts** *nat*, **opns** $\_ =_{nat} \_ : nat \times nat \rightarrow bool$)

An environment now maps each module name of its domain into a pair con-

sisting of a specification and a list of sorts and operations. This list represents the formal parameter list of the specification.

Definition 3 remains valid but is extended by the two following cases (M2) and (R12):

(M2) (*Module declaration with parameters*)
if n is a module name, *lso* a list of sorts and operations and m a specification for the environment *env* and if

(i) $env(n)$ is undefined

(ii) the sorts of *lso* are pairwise different

(iii) the operations of *lso* are pairwise different

(iv) the sorts and operations of *lso* are all from $\mathcal{S}_i(m)(env)$

then

- **module** $n(lso)$ **is** m **endmodule** is a module for the environment *env*

Informally, the conditions (ii) and (iii) entitle *lso* to be the first element of a renaming pair. The condition (iv) allows this renaming pair to be used in an import renaming construct.

(R12) (*Module instantiation with parameters*)
if n is a module name, *lso* a list of sorts and operations, and if

(i) $env(n)$ is defined; put $env(n) = (m, lso')$

(ii) (lso', lso) constitutes a renaming pair on the signature $\mathcal{S}_i(m)(env)$; call ρ the signature morphism on the signature $\mathcal{S}_i(m)(env) \cup \mathcal{S}_e(m)(env)$ induced by (lso', lso)

(iii) the signature morphism ρ is injective on the operations of $\mathcal{S}_e(m)(env) \setminus \mathcal{S}_i(m)(env)$

(iv) the sorts and operations of lso' are all from $\mathcal{S}_i(m)(env)$

(v) $\rho(so) \notin \rho(\mathcal{S}_e(m)(env) \setminus \mathcal{S}_i(m)(env))$ for each sort or operation so of $\mathcal{S}_i(m)(env)$

then

- $n(lso)$ is a specification for the environment *env*

- $\mathcal{S}(n(lso))(env) = \mathcal{S}(m$ **i-rename** lso' **as** $lso)(env)$ where m and lso' are defined by $env(n) = (m, lso')$

Informally, the conditions (ii) to (v) allow the use of lso' and lso in an import renaming construct (see Definition 1(6)).

Using the same notational conventions as above we obtain the following additional cases for the definition of the meaning function $\mathcal{M}$:

(R12) (*Module instantiation with parameters*)

$$\mathcal{M}(n(lso))(env)(A) = \mathcal{M}(m \textbf{ i-rename } lso' \textbf{ as } lso)(env)(A)$$

iff $\mathcal{M}(m$ **i-rename** lso' **as** $lso)(env)(A)$ is defined where m and lso' are defined by $env(n) = (m, lso')$.

A generalization of Section 5.1.3 for parameterized modules is straightforward.

It is interesting to note that the use of parameters is essentially an elegant way to express a renaming of imported sorts and operations. Hence the difference between parameters on the one hand and imported sorts and operations that are not parameters on the other hand merely lies in the fact that the former are renamed "automatically".

5.3 Macros

The context conditions of the composition construct are particularly stringent: for

$$(m_1 \circ m_2)$$

to be correct it is required that $\mathcal{S}_e(m_2) = \mathcal{S}_i(m_1)$ (see Definition 1(3)). We therefore introduce a more liberal construct called the construct of *liberal composition* and denoted as

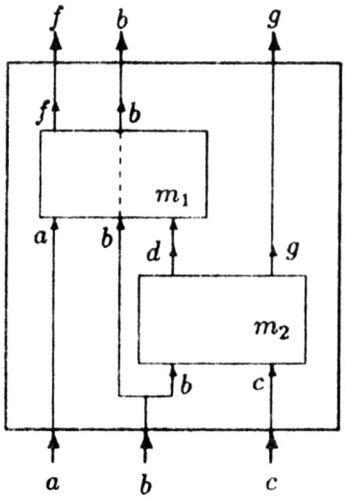
$$(m_1 \odot m_2)$$

Formally this new construct is defined as a macro, i.e. as a shorthand notation for a sequence of constructs of the kernel language. Let for any signature Σ

$$1_\Sigma$$

denote an atomic specification defining the identity module functor for the module signature (Σ, Σ). (It is assumed that the specification method used allows to write such "empty" specifications.) By definition

$$(m_1 \odot m_2)$$



$m = (m_1 \odot m_2)$

Figure 2: Graphical illustration of the construct of liberal composition. The conventions are those of Figure 1.

stands for the specification

$$((m_1 + 1_{\Sigma_1}) \circ (m_2 + 1_{\Sigma_2}))$$

with Σ_1, Σ_2 defined by

$$\begin{aligned}
\Sigma_1 &= (\mathcal{S}_e(m_2) \setminus \mathcal{S}_i(m_1)) \cup \{s \mid s \text{ is a sort of the arity of an operation of } \mathcal{S}_e(m_2) \setminus \mathcal{S}_i(m_1)\} \\
\Sigma_2 &= (\mathcal{S}_i(m_1) \setminus \mathcal{S}_e(m_2)) \cup \{s \mid s \text{ is a sort of the arity of an operation of } \mathcal{S}_i(m_1) \setminus \mathcal{S}_e(m_2)\},
\end{aligned}$$

i.e. Σ_1 and Σ_2 are the smallest signatures containing $\mathcal{S}_e(m_2) \setminus \mathcal{S}_i(m_1)$ and $\mathcal{S}_i(m_1) \setminus \mathcal{S}_e(m_2)$ respectively.

The context conditions and the semantics of $(m_1 \odot m_2)$ are of course those of the specification it stands for.

Two further constructs — denoted $\circ_R$ and $\odot_R$ respectively — are also defined as macros:

$$m_1 \circ_R m_2 \text{ stands for } m_2 \circ m_1$$
$$m_1 \odot_R m_2 \text{ stands for } m_2 \odot m_1$$

Informally, the constructs $\circ$ and $\odot$ correspond to a top-down design while $\circ_R$ and $\odot_R$ correspond to a bottom-up design.

5.4 Syntactical sugar

Among the possible abbreviations let us mention the two following ones to be used in Section 6:

- the arity of an operation may be omitted whenever this does not lead to an ambiguity;
- the instantiation

  $$n(lso)$$

  of a parameterized module (see rule (R12)) may be written

  $$n$$

  whenever $env(n) = (m, lso)$; in other words, the actual parameters may be omitted if they coincide with the formal ones (cf. module AXLIST of Section 2).

A further useful notational simplification is described in the now following Section 5.5.

5.5 Export parameters

Consider again the module declaration of LIST in Section 2:

module LIST (**sorts** *el*, **opns** ...) **is** ... **endmodule**

Its exported signature contains, among others, the sort *list*. Consider now the instantiations

LIST (**sorts** *nat*, **opns** ...)

and

LIST (**sorts** *string*, **opns** ...)

Both have the sort *list* in their exported signatures. Hence, before it is possible to "unite" these instantiations by the construct + it is necessary to rename at least one of these sorts *list*. This requires an export renaming — as illustrated in the module LISTS-OF-NAT-AND-STRING of Section 2. The idea of export parameters is to provide a means for performing this export renaming automatically at each instantiation. Informally, export parameters behave like (normal) parameters but induce an export renaming rather than an import renaming.

Formally, we add two context-free rules:

< *module* >	::=	**module** < *module name* >(< *lso* >)(< *lso* >) **is** < *spec* > **endmodule**	(M3)
< *spec* >	::=	< *module name* >(< *lso* >)(< *lso* >)	(R13)

An environment now maps module names into triples consisting of a specification and two lists of sorts and operations viz. the formal (normal) parameters and the formal export parameters. The definition of the context conditions, the signature function $\mathcal{S}$ and the meaning function $\mathcal{M}$ is left to the reader. Let us merely mention that the instantiation

$$n(lso1)(lso2)$$

may be viewed as a shorthand notation for the specification

m
i-rename $lso1'$ **as** $lso1$
e-rename $lso2'$ **as** $lso2$

where $m, lso1'$ and $lso2'$ are defined by $env(n) = (m, lso1', lso2')$.

5.6 Strong typing

Most logics introduce a notion of terms and associate with each term one or more sorts. For instance, if the signature contains an operation $n : \rightarrow s$ then n may be defined to be a term of sort s. A term language (over a signature)

is called *strongly typed* if each term has at most one sort. Now, according to Section 3.1.1 an operation is characterized by its name *and* its arity. Hence, a signature may contain operations such as $n : \rightarrow s$ and $n : \rightarrow s'$ with $s \neq s'$. In that case the term n has two types and the term language over such a signature is not strongly typed.

Some specification languages such as *OBJ3* and *PLUSS* have made the deliberate choice to do without strong typing. On the other hand strong typing makes the language more robust. For this reason classical imperative languages such as *Pascal* use strong typing.

The specification language *OBSCURE* described above is not strongly typed. We now shortly indicate how a mechanism for strong typing may be included into the language. To this end we assume that it is possible to guarantee strong typing by imposing a condition on the signature. Of course, the existence and the nature of such a condition — called *typing condition* — depends on the (term language of the) logic. A simple but stringent typing condition valid for "classical" logics is that all operations of the signature have different operation names. A more liberal typing condition — called *overloading condition* — is the following: for any two operations

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

$$n : t_1 \times \ldots \times t_k \rightarrow t_{k+1}$$

with the same operation name n and the same number k of arguments there exists i $(1 \leq i \leq k)$, such that $s_i \neq t_i$. A typing condition for a term language allowing prefix, infix and mixfix notation may be found in [9].

When used with a logic equipped with a typing condition *OBSCURE* may be provided with strong typing by adding for each construct (at most) two context conditions. These context conditions express that the pair of signatures constituting the value of the signature $\mathcal{S}$ satisfy the typing condition. For instance, in the case of the union construct (see Definition 1(2)) these context conditions are:

(iv) $\mathcal{S}_i(m_1) \cup \mathcal{S}_i(m_2)$ satisfies the typing condition

(v) $\mathcal{S}_e(m_1) \cup \mathcal{S}_e(m_2)$ satisfies the typing condition

In the case of the composition construct (see Definition 1(3)) there are no extra context conditions.

6 Some more examples

While the examples of Section 2 essentially illustrate the meaning of the different constructs of the kernel language we now want to illustrate the power of the module concept. Contrasting with Section 2 we will use the algorithmic specification method. The pertaining logic is essentially first-order predicate logic and is described in [15]. The typing condition is type inferencing (cf. Section 5.6) extended for infix and mixfix notation [9].

A context free grammar for atomic specifications drawn up according to the algorithmic specification method may be found in the Appendix. Actually, instead of defining here the complete syntax and the semantics of these specifications we prefer to illustrate them on an example. To this end we consider again the specification of lists of elements already treated in the module LIST of Section 2 . This time we make use of the notational conventions of Section 5 and, in particular, of export parameters. As a further facility we, in particular, omit writing down those imported sorts and operations that may be univocally deduced from the rest of the specification. Finally, the sort *bool* and the pertaining operations will be omitted.

module LIST (**sorts** *el*, **opns** *Eq*) (**sorts** *list*, **opns** ϵ) **is**
atomspec
  import sorts *el*
    opns $Eq : el \times el \rightarrow bool$
  create sorts *list*
    opns constr $\epsilon : \rightarrow list$
      constr $\_ \circ \_ : list \times el \rightarrow list$
      $\_ \in \_ : el \times list \rightarrow bool$
  semantics
    vardec $e, e' : el, l, l' : list$
    programs
    $(e \in l) \Leftarrow$ **case** l **of**
      ϵ : *false*
      $l' \circ e'$: **if** $Eq(e,e')$ **then** *true* **else** $e \in l'$ **fi**
      esac
endatom
endmodule

Informally, the carrier set of sort *list* is the term language generated by the constructors i.e. by the operations labeled **constr**. The semantics of the constructors is the Herbrand interpretation; the semantics of the other operations (in this case $\in$) is defined by the corresponding recursive program. A precise definition of the algorithmic specification method may be found in [15]. Clearly, the method has similarities with the corresponding constructs

of the programming languages *Miranda* [23] or *Standard ML* [18].

The data type "pair of elements" may be specified by:
module PAIR (**sorts** *el*1, *el*2) (**sorts** *pair*) **is**
atomspec
  import sorts *el*1, *el*2
  create sorts *pair*
    opns constr [_,_] : *el*1 × *el*2 → *pair*
      First : *pair* → *el*1
      Second : *pair* → *el*2
  semantics
    vardec p : *pair*, e_1 : *el*1, e_2 : *el*2
    programs
    First(p) ⇐ **case** p **of** $[e_1, e_2]$: e_1 **esac**
    Second(p) ⇐ **case** p **of** $[e_1, e_2]$: e_2 **esac**
endatom
endmodule

Pairs of lists of the same element sort, say *el*, are then:
module PAIR-OF-LISTS (**sorts** *el*) (**sorts** *listpair*, *list*, **opns** ϵ) **is**
    PAIR (**sorts** *list*, *list*)(**sorts** *listpair*)
    ⊙ LIST
endmodule

Pairs of lists of natural numbers are then:
module PAIR-OF-LISTS-OF-NAT **is**
  PAIR-OF-LISTS (**sorts** *nat*)(**sorts** *natlistpair*, *natlist*, **opns** *Emptynatlist*)
endmodule

The imported signature of this module contains the sort *nat*. The exported signature contains, in particular, the sorts *nat*, *natlist* and *natlistpair* and the operations

Emptynatlist : → *natlist*
_ ∘ _ : *natlist* × *nat* → *natlist*
_ ∈ _ : *nat* × *natlist* → *bool*
[_,_] : *natlist* × *natlist* → *natlistpair*
First : *natlistpair* → *natlist*
Second : *natlistpair* → *natlist*.

Along the same lines one may write a module defining, for instance, pairs of pairs of lists of natural numbers, or a module defining pairs of lists of natural numbers together with pairs of lists of strings. Note that in the latter case the required renamings are "automatical" — provided the actual export

parameters have been chosen to be different.

It is not difficult to derive from the module PAIR a module defining finite maps. To this end it is sufficient to make sure that for any two different carriers (e_1, e_2) and (e_1', e_2') of sort *pair* it is the case that $e_1 \neq e_1'$. This may be obtained by the subset construct — as explained in Section 2. Again, it is necessary to first add the desired operations and to forget the operation $[\_,\_]$ in order to satisfy the closure condition.

Semantical constraints may also be written in the form of a module, viz. an identity module with imported axioms:

module EQUIVALENCE (**sorts** *el*, **opns** *Eq*) **is**
atomspec
  import sorts *el*
    opns $Eq : el \times el \rightarrow bool$
endatom
i-axioms
  vardec $e, e', e'' : el$
  $Eq(e,e) = true$
  $Eq(e,e') = true \supset Eq(e',e) = true$
  $Eq(e,e') = true \wedge Eq(e',e'') = true \supset Eq(e,e'') = true$
endmodule

Hence

module EQLIST (**sorts** *el*, **opns** *Eq*) (**sorts** *list*, **opns** ε) **is**
  LIST ∘ EQUIVALENCE
endmodule

specifies lists of elements with an equivalence relation (cf. the module AXLIST of Section 2).

We close this list of examples with two remarks. First, the reader may have had difficulties to check the various context conditions or even to keep track of the module signature. In fact, *OBSCURE* is a language for writing specifications with the help of a computer. A corresponding specification environment has been programmed and will be briefly discussed in Section 7. Second, especially atomic specifications contain several keywords that may be abbreviated or even removed without introducing ambiguities. Hence, at the prive of a less explicit notation many of the example specifications given above may be shortened.

7 Concluding remarks

It is easy to generalize *OBSCURE* for loose specifications or, more precisely, to make *OBSCURE* accept atomic specifications with loose semantics. Contrasting with Section 3.2 a module functor is now defined as a (total) function

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

where $\mathcal{P}(Alg_{\Sigma_e})$ denotes the class of all subclasses of Alg_{Σ_e}. The syntax of *OBSCURE* including the context conditions and the definition of the signature function $\mathcal{S}$ remains unchanged. The definition of the new meaning function is obtained by a straightforward generalization of Definition 2 (Section 4.2). The interested reader may find this generalized definition in [14].

A notion of implementation for *OBSCURE* may be found in [12].

As indicated in the introduction *OBSCURE* handles with models rather than with theories. More precisely, starting from the models defined by atomic specifications *OBSCURE* allows to construct new models. The theories of these new models are not explicitly defined by *OBSCURE*. Instead, we have developed a calculus containing an inference rule for each *OBSCURE* construct [13]. The formulas of this calculus link *OBSCURE* specifications to logic in the same way as Hoare formulas link imperative programs to logic.

While being compatible with any specification method *OBSCURE* is particularly well adapted to algorithmic specifications. The main reason is that the module functor of any atomic algorithmic specification is total. Moreover, algorithmic specifications allow to introduce partial operations and lead to a simple and efficient rapid prototyping algorithm. Of course, by their very definition algorithmic specifications are less abstract than algebraic ones in that they constitute "programs" rather than "specifications". Hence the abstract flavor of an *OBSCURE* specification with atomic algorithmic specifications essentially stems from the constructs **forget**, **subset**, **quotient**, **i-axioms** and **e-axioms** that "abstract" from the underlying model. On the other hand, while it is principally true that algebraic specifications are more abstract than algorithmic ones this property appears to be pointless in many "practical" cases: as indicated in [16] the difference between algebraic and algorithmic specifications is often purely notational.

Recently a specification environment for *OBSCURE* has been completed [9]. The specification method implemented is the algorithmic one, the logic is that of [15] and the typing condition is the overloading condition (cf. Section 5.6) extended for infix and mixfix notation. The specification environment performs the syntactical analysis of specifications, tests the various context conditions and computes the module signature. It moreover generates formulas

expressing the closure and congruence conditions of the subset and quotient constructs. It contains a data base of modules modeling the notion of environment introduced in Sections 5.1 and 5.2. Finally, it allows rapid prototyping. The specification environment runs on a *Sun 3/60* computer and is embedded in *Emacs*. More information may be found in the handbook [9]. It is planned to complete the specification environment by a verifier supporting the user in the proofs demanded by the meaning function.

Among the examples of non-trivial *OBSCURE* specifications developed with the help of the specification environment we may cite a specification of the *UNIX* file system and a specification of a test method for sufficient completeness of term rewriting systems.

Acknowledgments. The design of *OBSCURE* was spread over several years. Hence it is impossible to mention all those who helped us to improve the language by their criticism. Instead we wish to thank Jürgen Fuchs, Annette Hoffmann, Liane Meiss, Joachim Philippi, Michael Stolz, Ralf Treinen, Stephan Uhrig and Jörg Zeyer who took an active part in the design of the specification environment of *OBSCURE*.

References

[1] Bergstra, J.A., Heering, J., Klint, P.: *Algebraic Specification.* ACM–Press, 1989

[2] Bidoit, M.: *PLUSS, a language for the development of modular algebraic specifications.* Thèse d'Etat, Univ. Paris–Sud, 1989

[3] Burstall, R.M., Goguen, J.A.: The semantics of CLEAR, a specification language. In: *Proc. 1979 Copenhagen Winter School.* (Lect. Notes Comput. Sci., vol. 86, pp. 292–332) Berlin, Heidelberg, New–York : Springer 1980

[4] Cartwright, R.: A constructive alternative to abstract data type definitions. In: *Proc. 1980 LISP Conf., Stanford Univ.*, 46–55 (1980)

[5] Ehrich, H.D., Gogolla, M., Lipeck, U.: *Algebraische Spezifikation abstrakter Datentypen. Leitfäden und Monographien der Informatik.* Stuttgart : Teubner 1989

[6] Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics.* Volume 6 of *EATCS Monographs on Theor. Comput. Sci.*. Berlin, Heidelberg, New–York : Springer 1985

[7] Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specification 2 — Module Specifications and Constraints.* Volume 21 of *EATCS Monographs on Theor. Comput. Sci.*. Berlin, Heidelberg, New–York : Springer 1990

[8] Fey, W.: *Pragmatics, Concepts, Syntax, Semantics and Correctness notions of ACT TWO: An algebraic module specification and interconnection language.* Forschungsberichte des FB Informatik Nr. 1988/26, TU Berlin, 1988

[9] Fuchs, J., Hoffmann, A., Meiss, L., Philippi, J., Stolz, M., Wolf, M., Zeyer, J.: The OBSCURE Manual. Version 1.0. Univ. Saarbrücken, 1990

[10] Goguen, J.A., Winkler, T.: *Introducing OBJ3.* Int. Rep. SRI–CSL–88–9, Comput. Sci. Laboratory SRI International, 1988

[11] Klaeren, H.A.: A constructive method for abstract algebraic software specification. *Theor. Comput. Sci.* **30**, 139–204 (1984)

[12] Lehmann, T.: *Ein abstrakter Implementierungsbegriff für die Spezifikationssprache OBSCURE.* PhD thesis, Univ. Saarbrücken, 1990

[13] Lehmann, T., Loeckx, J.: A module calculus for the specification language of OBSCURE. in preparation, 1990

[14] Lehmann, T., Loeckx, J.: The specification language of OBSCURE. In: Sannella, D., Tarlecki, A. (eds.), *Proc. of the 5th Workshop on Specification of Abstract Data Types.* Gullane, Scotland 1987. (Lect. Notes Comput. Sci., vol. 332, pp. 131–153) Berlin, Heidelberg, New–York : Springer 1988

[15] Loeckx, J.: Algorithmic specifications: a constructive specification method for abstract data types. *ACM Trans. Prog. Lang. Syst.* **9**, 646–685 (1987)

[16] Loeckx, J.: The specification system OBSCURE. *Bull. EATCS* **40**, 169–171 (1990)

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

[18] Milner, R., Tofte, M., Harper, R.: *The Definition of Standard ML.* Cambridge, Mass.: MIT Press 1990

[19] Sannella, D.: A set-theoretic semantics of CLEAR. *Acta Inf.* **21**, 443–472 (1984)

[20] Sannella, D., Sokolowski, S., Tarlecki, A.: *Toward formal development of programs from algebraic specifications : parameterisation revisited.* Int. Rep. 6/90, Univ. Bremen, 1990

[21] Sannella, D., Tarlecki, A.: Program specification and development in Standard ML. In: *Proc. 12th ACM Symp. on Princ. of Prog. Lang.*, New Orleans, USA, pp. 67–77, 1985

[22] Sannella, D., Tarlecki, A.: Toward formal development of ML programs: foundations and methodology. In: Diaz, J., Orejas, F. (eds.) *TAPSOFT'89.* Proceedings, Barcelona 1989. (Lect. Notes Comput. Sci., vol. 352, pp. 375–389) Berlin, Heidelberg, New–York : Springer 1989

[23] Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. In: Goos, G., Hartmanis, J. (eds.) *Functional Programming and Computer Architecture.* (Lect. Notes Comput. Sci., vol. 201, pp. 1–16) Berlin, Heidelberg, New–York : Springer 1985

[24] Wirsing, M.: Structured algebraic specifications: a kernel language. *Theor. Comput. Sci.* **42**, 124–249 (1986)

APPENDIX

A context-free grammar for *OBSCURE*

The definition of the non-terminals < *sort* >, < *operation name* > and < *module name* > is left pending. The same remark holds for non-terminals such as < *list of equalities* > and < *genuine formula* > that depend on the logic.

< *module* >	:: =	**module** < *module name* > **is**	
		< *spec* > **endmodule** |	(M1)
		module < *module name* >(< *lso* >)	
		is < *spec* > **endmodule** |	(M2)
		module < *module name* >(< *lso* >)(< *lso* >)	
		is < *spec* > **endmodule**	(M3)
< *spec* >	::=	< **atomspec** > |	(R1)
		(< *spec* > + < *spec* >) |	(R2)
		(< *spec* > ∘ < *spec* >) |	(R3)
		(< *spec* > **forget** < *lso* >) |	(R4)
		(< *spec* > **e-rename** < *lso* > **as** < *lso* >) |	(R5)
		(< *spec* > **i-rename** < *lso* > **as** < *lso* >) |	(R6)
		(< *spec* > **e-axioms** < *formula* >) |	(R7)
		(< *spec* > **i-axioms** < *formula* >) |	(R8)
		(< *spec* > **subset of** < *sort* > **by** < *formula* >) |	(R9)
		(< *spec* > **quotient of** < *sort* > **by** < *formula* >) |	(R10)
		< *module name* > |	(R11)
		< *module name* >(< *lso* >) |	(R12)
		< *module name* >(< *lso* >)(< *lso* >) |	(R13)
		(< *spec* > ⊙ < *spec* >) |	(R14)
		(< *spec* > $\circ_R$ < *spec* >) |	(R15)
		(< *spec* > $\odot_R$ < *spec* >) |	(R16)
< *lso* >	::=	**sorts** < *ls* >, **opns** < *lo* > | **sorts** < *ls* > | **opns** < *lo* >	
< *ls* >	::=	< *sort* > | < *ls* >,< *sort* >	
< *lo* >	::=	< *operation* > | < *lo* >,< *operation* >	
< *operation* >	::=	< *operation name* > : → < *sort* > |	
		< *operation name* > : < *ls* > → < *sort* >	

The following rules depend on the specification method:

< *atomspec* >	::=	**atomspec import** < *lso* > **create** < *lso* >
		semantics < *algebraic spec* > **endatom** |
		atomspec import < *lso* > **create** < *lsoc* >
		semantics < *algorithmic spec* > **endatom** |
		...
< *algebraic spec* >	::=	**vardec** < *list of variables* >
		eqns < *list of equalities* >
< *algorithmic spec* >	::=	**vardec** < *list of variables* >
		programs < *list of programs* >
< *lsoc* >	::=	**sorts** < *ls* >, **opns** < *loc* > | **sorts** < *ls* > |
		opns < *loc* >
< *loc* >	::=	< *operation* > | **constr** < *operation* > |
		< *loc* >,< *operation* >

The following rules depend on the logic:

< *formula* >	::=	**vardec** < *list of variables* >.< *list of gen form* >
< *list of gen form* >	::=	< *genuine formula* > |
		< *list of gen form* >.< *genuine formula* >