OBSCURE: An interactive specification language for model-oriented specification methods

(Extended abstract)

by

Claus-Werner Lermen and Jacques Loeckx

A 85/12

Saarbrücken, July 1985

Universität des Saarlandes
Fachrichtung 10.2 Informatik
D - 6600 Saarbrücken

to appear

OBSCURE: An interactive specification language for model-oriented specification methods

(Extended abstract)

Claus-Werner Lermen and Jacques Loeckx
Universität des Saarlandes, Saarbrücken, FRG

1. Introduction

Recently several specification languages have proposed in the literature: CLEAR [BG 77, Sa 84], ACT ONE [EM 85], OBJ2 [FGJM 85], ASL [SW 83], Extended ML [ST 85, Sa 85]. The first three of these languages are essentially based on the concept of the initial algebra; ASL and Extended ML use theories, i.e. classes of models. The specification language to be presented here differs from these languages in several respects.

The original version of OBSCURE [Le 85] was developed for the algorithmic specification method [Lo 81, Lo 84]. According to this method an abstract data type is specified by a constructively defined model. The version of OBSCURE to be presented here is more general and is applicable to any specification method based on the use of a single model. Hence it is also applicable to, for instance, the initial algebra specification method, but - at least in its present form - it is not applicable to specification methods based on theories.

By allowing to put algebras or theories together the existing specification languages suggest a bottom-up development of programs. Instead, OBSCURE is designed for top-down development ("development by stepwise refinement"). A specification is therefore defined as a function mapping algebras into algebras; putting specifications together corresponds to the composition of these functions. For instance, the development of an interpreter for a programming language leads to a specification introducing the sort *program* and an operation Interprete mapping programs and input data into output data. The sorts *input-data* and *output-data* as well as the sorts used in the specification - such as, for instance, *statement* or *configuration* - have still to be specified. The specification of *program* is interpreted as a function mapping an algebra containing the sorts *input-data*, *output-data*, etc. into an (extension of this) algebra containing the sort *program*.

As a further characteristic OBSCURE explicitly links program development to program verification. More precisely, an OBSCURE implementation consists of a system with a program development part and a program verification part. (Figure 9 in the Appendix). This allows OBSCURE to dispose for instance of constructs transforming an algebra into a subalgebra or a quotient algebra. The semantic conditions guaranteeing the consistency of these transformations are theorems which have to be proved by the user.

OBSCURE is designed for interactive use. Its language constructs are therefore applied "postfix-like". At each construct the system checks the consistency conditions of syntactic nature. It generates the theorems expressing the consistency conditions of semantic nature and transmits them to the program verification part.

Finally, OBSCURE allows a mild form of polymorphism.

The goal of this paper is to give an overview of the language. A complete formal description together with the proofs of the theorems may be found in [LL 85]. The version of OBSCURE presented here contains no syntactic sugar. This simplifies the description but makes the examples look unappealing. A version with syntactic sugar is in [Le 85].

Section 2 introduces the main syntactic and semantic notions. Section 3 presents OBSCURE without parameterized specifications. Section 4 adds polymorphism. Section 5 introduces parameterized specifications. Section 6 allows the union of identical subspecifications. Section 7 contains conclusions. The Figures are in the Appendix.

2. Syntactic and semantic notions

2.1 Signatures

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

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

where n is an identifier, called *operation name*, and where $s_1, \ldots, s_k$, s are sorts. If S is a set of sorts containing $s_1, \ldots, s_k$, s, the operation is said to be S-*sorted*. A set of operations is S-*sorted*, if its elements are. Note that classically an operation is defined to be an operation name characterizing univocally the sorts $s_1, \ldots, s_n$, s. The present, more general definition eases the introduction of polymorphism in Section 4 but requires the following extra definition: a set of operations is said to be *unambiguous* if any two different operations have different operation names.

A *signature* is a pair $\Sigma = (S,\Omega)$ where S is a set of sorts and Ω a set of operations. It is called an *algebra signature,* if Ω is S-sorted. If Σ, Σ' are signatures, $\Sigma = \Sigma'$, $\Sigma \subseteq \Sigma'$, $\Sigma - \Sigma'$ and "Σ, Σ' are disjoint" are meant componentwise.

2.2 Algebras

Let $\Sigma = (S,\Omega)$ be an algebra signature. A *(Σ-)algebra* is a mapping which associates

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

(ii) with each operation $n : s_1 \times \ldots \times s_k \to s$, $k \geq 0$, a (possibly partial) function

$$A(n : s_1 \times \ldots \times s_k \to s) : A(s_1) \times \ldots \times A(s_k) \to A(s)$$

The set of all Σ-algebras is denoted Alg_Σ.

If Σ, Σ' are algebra signatures with $\Sigma \subseteq \Sigma'$, and if A is a Σ'-algebra, then $A|\Sigma$ is the Σ-algebra defined by

$$(A|\Sigma)(e) = A(e) \qquad \text{for all } e \in S \cup \Omega$$

From now on we will only consider (S,Ω)-algebras A where S contains the special sort *bool*, Ω is unambiguous and contains the special operations <u>true</u> : → *bool* and <u>false</u> : → *bool*, and A maps these special sorts and operations into their usual meaning.

2.3 Algebra extensions

The goal is to introduce functions mapping algebras (containing "global" sorts and operations from a signature Σ_g) into the algebras obtained by adding "new" sorts and operations (from a signature Σ_n). Informally, the "new" sorts and operations are those introduced by a specification; the "global" ones are those which have still to be specified and which therefore occur in the specification as global "variables".

A pair (Σ_g,Σ_n) of signatures is called an *extension signature* if

(i) Σ_g, Σ_n are disjoint;

(ii) Σ_g is an algebra signature;

(iii) $\Sigma_g \cup \Sigma_n$ is an algebra signature.

An *algebra extension* for the extension signature (Σ_g,Σ_n) is a function

$$E : Alg_{\Sigma_g} \to Alg_{\Sigma_g \cup \Sigma_n}$$

such that

$$(E(A) \mid \Sigma_g) = A \qquad \text{for each } A \in Alg_{\Sigma_g} \tag{I}$$

2.4 More on algebras

Two constructions will be recalled which yield an algebra, called subalgebra and quotient algebra respectively. These constructions are well-known from the literature (see e.g. [GM 83, EM 85]).

Let A be a (Σ)-algebra, $\Sigma = (S,\Omega)$, and p a family of (possibly partial predicates

$$p_s : A(s) \rightarrow \{\text{true, false}\} \qquad \text{for each } s \in S.$$

The *subalgebra* generated by A and p is the Σ-algebra B defined by:

$$B(s) = \{c \in A(s) \mid p_s(c) = \text{true}\} \qquad \text{for each } s \in S$$
$$B(n{:}s_1 \times\ldots\times s_k \rightarrow s) = (A(n{:}s_1\times\ldots\times s_k \rightarrow s)) \mid B(s_1)\times\ldots\times B(s_k)$$
$$\text{for each } n : s_1 \times\ldots\times s_k \rightarrow s \in \Omega,\ k \geq 0.$$

Actually, this definition is consistent only if each operation of A satisfies a *closure condition*. Informally, this condition expresses that arguments from the subset lead to a value from the subset.

Let A again be a (Σ)-algebra, $\Sigma = (S,\Omega)$, and q a family of (total!) equivalence relations

$$q_s : A(s) \times A(s) \rightarrow \{\text{true, false}\} \qquad \text{for each } s \in S$$

The *quotient algebra* generated by A and q is the Σ-algebra B defined by:

$$B(s) = \{[c] \mid c \in A(s)\} \qquad \text{for each } s \in S$$
$$B(n : s_1 \times\ldots\times s_k \rightarrow s)([c_1], \ldots , [c_k])$$
$$= [A(n : s_1 \times\ldots\times s_k \rightarrow s)(c_1, \ldots , c_k)]$$
$$\text{for all } c_i \in A(s_i),\ 1 \leq i \leq k$$
$$\text{for all } n : s_1 \times\ldots\times s_k \rightarrow s \in \Omega,\ k \geq 0$$

This definition is consistent only if each operation of A satisfies a *congruence condition*. Informally, the congruence condition expresses that equivalent arguments lead to equivalent values.

3. OBSCURE without parameterized data types

OBSCURE is described by a context-free grammar, by "semantic functions" mapping specifications into algebra extensions and by a list of conditions guaranteeing the consistency of the definition of these semantic functions. The techniques used are inspired from those used for the description of CLEAR in [Sa 84] and ASL in [SW 83].

3.1 Syntax

A context-free syntax of OBSCURE is in Figure 1. A "program" in OBSCURE is a derivation of csp ("composed specification").

The distinction between csp ("composed specification") and sp ("specification") implicitly provides the "operator" compose with a lower priority than the other "operators" such as subset and forget.

The definition of m ("model") depends on the specification method used. In the case of the initial algebra specification method m consists of a list of equalities. In the case of the algorithmic specification method

m ::= constructors lo programs lp

where lp is a list of recursive programs [Lo 84]. The definition of ax ("axiom") depends on the logic used.

3.2 An informal introduction to the semantics

The context-free rule (csp2) of Figure 1 corresponds to a "refinement step" in the top-down development of programs: sp contains a specification of some "global" (i.e. not yet specified) sorts and operations of csp. Hence the composition of csp and sp yields an algebra extension containing less - or, at least, "lower level" - global sorts and operations.

The rule (sp1) specifies "new" sorts and operations by describing a model m with the help of "global" sorts and operations. The rule (sp2) transforms an algebra into a subalgebra. This transformation is a special case of Section 2.4 in that only the carrier set of sort s is restricted to a subset. Similarly (sp3) transforms an algebra into a quotient algebra. By (sp4) it is possible to drop ("hide") sorts and operations. The renaming of sorts and operations according to (sp5) is especially useful for parameterized data types (see Section 5). The axioms introduced by (sp6) have no effect on the algebra extension defined. Instead, they are transmitted to the verification part of the OBSCURE system. With these axioms the user may, for instance, check properties of the data type introduced. Moreover, the axioms may be used to restrict the actual parameters of a parameterized specification (see Section 5).

3.3 Semantics

The semantics are defined with the help of two (families of) semantic functions. The first of these functions, denoted $\mathfrak{S}$, maps a specification into the signature of an algebra extension. The second, denoted $\mathfrak{I}$, maps the specification into the algebra extension itself.

The definition of the semantic function $\mathfrak{S}$ is in Figure 2. The notation $\Sigma[lso1/lso2]$ used in (sp5) denotes the signature obtained from Σ through simultaneous substitution of the sorts and operations of lso1 by those of lso2. It is understood that the substitution of a sort includes the substitution of its occurrences in the operations (for a more precise definition see [LL 85]). Note that "forgetting" and "renaming" applies to "new" sorts and operations only.

The definition of the semantic function $\mathfrak{I}$ is in Figure 3. Most of the equalities define the algebra extension by its value for an arbitrary argument A. The signature of this algebra A is univocally defined by the semantic function $\mathfrak{S}$. The only "difficult" equality in Figure 3 is the one expressing the semantics of compose. Essentially this equality expresses the composition of the functions $\mathfrak{I}_{csp}$ (csp) and $\mathfrak{I}_{sp}$ (sp). The complication stems from the fact that the arguments have first to be restricted to the signature of the function domain and that the parts cut off by this restriction have to be added to the function value. Note that by the union of two algebras we mean the union of their graphs (remember that according to Section 2.2 an algebra is a function). Of course, the union of two algebras yields an algebra only if the resulting graph represents a function and not merely a relation.

3.4 The consistency conditions

The value of the semantic function $\mathfrak{S}$ is defined in Figure 2 as a pair of signatures. Such a pair is an extension signature only if it satisfies the condition (i) to (iii) of Section 2.3. Similarly, the value of the semantic function $\mathfrak{I}$ for a specification sp and an algebra A is defined as a relation. Hence $\mathfrak{I}$(sp) is an algebra extension only if this relation is a function (i.e. $\mathfrak{I}$(sp)(A) is an algebra) and if moreover the condition (I) of Section 2.3 is satisfied.

To guarantee these properties a number of conditions, called *consistency conditions*, are attached to each rule of the context-free grammar of Figure 1. The following remarks may provide a flavor of these conditions. In rule (sp1), $\mathfrak{S}_{lso}$(lso1) and $\mathfrak{S}_{lso}$(lso2) have to satisfy the conditions (i) to (iii) of Section 2.3. In rule (sp2) s has to be a "new" sort, o has to be of the form n : s → *bool* and the closure conditions of Section 2.4 must be satisfied. A similar remark holds for rule (sp3). In (sp4) the sorts and operations of lso have to be "new" ones. A similar remark holds for lso1 in rule (sp5); moreover, lso1 and lso2 must "match". An important condition for (csp2) is that Σ_{n1} and Σ_{n2} have to be disjoint. A complete list of these consistency conditions together with their formal description is in [LL 85].

Let OK (csp) denote that the composed specification csp satisfies the consistency conditions. The following theorems are proved in [LL 85]:

THEOREM 1: If OK (csp) holds for a composed specification csp then $\mathfrak{S}_{csp}$ (csp) is an extension signature.

THEOREM 2: If OK (csp) holds for a composed specification csp then $\mathfrak{I}_{csp}$ (csp) is an algebra extension.

3.5 An example

Figure 4 shows a specification of the data type "set of elements". It contains "superfluous" information which could easily be removed by adding syntactic sugar to the language (cf [Le 85]). The create construct makes us of the algorithmic specification method [Lo 81, Lo 84]. According to this method which, by the way, is very similar to the one proposed in Standard ML [Mi 84], the carrier set of sort *set* is the term language generated by the constructors. Hence, it contains words such as Emptyset and App(Emptyset, O). The interpretation of the operations which are constructors is the Herbrand interpretation. Hence the value of the operation App for the arguments Emptyset and O is the word App(Emptyset, O). The other operations are defined as recursive programs in the sense of [Ma 74, LS 84]; a precise syntax and semantics may be found in [Lo 84]. It is important to note that after execution of the create construct the elements of the carrier set may be accessed through the ("new") operations only. The formulas in the axioms construct have to be formulated in an appropriate logic, for instance the one proposed in [Lo 84]. The forget construct is necessary because the operation App does not satisfy the closure conditions implied by the subsequent subset construct which eliminates the carriers containing duplicates. The quotient construct identifies carriers differing only by the order of occurrence of their elements. Note that it is possible to do without the subset construct by making the quotient construct also identify carriers differing only by duplicates.

An illustration of the use of compose and rename is delayed until Section 5.

4. Introducing a mild version of polymorphism

According to the notion of unambiguity introduced in Section 2.1 an operation name univocally defines its operation. Actually, an operation name never occurs "naked" in a specification. It rather occurs "at least" in a term, viz. within an axiom or a recursive program. This suggests the following, more general definition: a set of operations is *unambiguous*, if for any two different operations of the form

$$n : s_1 \times \ldots \times s_k \rightarrow s$$
$$n : t_1 \times \ldots \times t_k \rightarrow t$$

$k \geq 0$, there exists i, $1 \leq i \leq k$, such that $s_i \neq t_i$. Hence, an unambiguous set of operations may, for instance, contain

Memberof : *intset* × *int* → *bool*
and Memberof : *stringset* × *string* → *bool*
(take i = 1 or i = 2) but not
Emptyset : → *intset*
and Emptyset : → *stringset*

5. Introducing parameterized specifications

The present Section introduces "procedures" with and without parameters. Procedures with parameters constitute parameterized specifications. Parameterless procedures allow to modularize the design: instead of developing a single specification by composing "elementary" ones, each elementary specification is given a name and called when needed.

The use of procedures requires the introduction of an environment which binds "procedure names" to "procedure bodies". Two approaches are possible: in the operational approach names are bound to specifications, i.e. to pieces of text; in the denotational approach names are bound to algebra extensions i.e. to the meaning of the pieces of text. In most specification languages the approach taken is essentially the denotational one (see e.g. [Sa 84, EM 85]). The approach taken here is the operational one. As an advantage it leads to a very simple copy-rule semantics.

Formally, an environment is defined as a function

$$n \rightarrow lso \times csp$$

where the sorts and operations of the list lso constitute the formal parameters and the composed specification csp the procedure body.

5.1 Syntax and Semantics

The syntax is in Figure 5.

The semantic function $\mathfrak{S}$ is in Figure 6. Note that $\mathfrak{S}_{csp}$ and $\mathfrak{S}_{sp}$ now have the environment as an extra argument. The notation ...[lso1/lso] used in (sp8) is that of Section 3.3 applied componentwise.

The semantic function $\mathfrak{I}$ is in Figure 7. The notion csp^{lso2}_{lso1} expresses the copy rule. More precisely, for the constructs of the rules (csp1) to (sp7), the notation expresses the simultaneous substitution of the sorts and operations of lso1 ("formal parameters") by those of lso2 ("actual parameters"). For the rule (sp8) the notation is defined by

$$(\underline{\text{call}}\ \ n(lso))^{lso2}_{lso1} =$$

$$\underline{\text{let}}\ \ e(n) = (lso', csp)\ \underline{\text{in}}$$

$$(csp^{lso}_{lso'})^{lso2}_{lso1}$$

A complete formal definition of the notation may be found in [LL 85].

5.2 The consistency conditions

The consistency conditions are essentially those of Section 3.4 together with conditions for the rules of Figure 5. For instance, the sorts and operations of lso in rule (d1) have to be global sorts and operations of csp. For (sp8) the name n must already have been declared and the actual and formal parameters must match.

Again, it is possible to prove the theorems of Section 3.4. The proof of Theorem 2 requires a lemma which, roughly speaking, expresses that if OK(csp) holds and lso1, lso2 match, then OK(csp^{lso2}_{lso1}) holds. For details and proofs see [LL 85].

Note that, while global sorts and operations are used before being specified (i.e. before being "created"), procedure names may only be used after having been declared. The reason for this restriction lies in the interactive nature of OBSCURE which requires that all syntactic consistency conditions may be checked at each step of the development.

5.3 An example

An examples introducing "pairs of sets of sets of integers" is in Figure 8. Line (1) turns the specification of Figure 4 into a procedure. Line(2) to (5) introduce the sort "set of sets of integers". According to the top-down development principle this sort is introduced by making use of the global sort "set of integers" which is specified in lines (4) to (5). Note that at least one of the renamings of line (3) and (5) is necessary in order to avoid name collisions and ambiguity at the execution of the combine construct. In line(6) the specification is shoved off into the environment as a parameterless procedure. The exact definition of the model in line (7) is dispensed with. The "new" sorts generated by the program of Figure 8 are *pair*, *setofint* and *setofsetofint* (but not *set*). "New" operations are for instance

Insert : *setofsetofint* × *setofint* → *setofsetofint*

and Insert : *setofint* × *int* → *setofint*

The global sorts are *int* and *bool*, the global operations Equal : *int* × *int* → *bool*, true : → *bool* and false :→ *bool*. The reader who has difficulties in keeping track of all these global and new sorts and operations should remember that OBSCURE is a language for interactive use and that the system updates and displays the global and new sorts and operations at each step of the development.

6. Allowing the union of identical subspecifications

One of the consistency conditions of the combine construct requires the sets Σ_{n1} and Σ_{n2} of "new" sorts and operations to be disjoint (see Section 3.4). This condition is unnecessarily stringent as illustrated by the following example. Let $A_{s,t}$ stand for a procedure call with global sorts s and t, B^s_t for a procedure call with "new" sort s and global sort t and C^t for a procedure call with "new" sort t. Then the specification

$$A_{s,t} \ \underline{\text{combine}} \ B^s_t \ \underline{\text{combine}} \ C^t \qquad (1)$$

is ok but the "equivalent" specification

$$A_{s,t} \ \underline{\text{combine}} \ C^t \ \underline{\text{combine}} \ \underline{\text{spec}} \ B^s_t \ \underline{\text{combine}} \ C^t \ \underline{\text{endspec}} \qquad (2)$$

is not, because the arguments of the second combine construct both have t among their "new" sorts. While a perspicuous user is expected to write (1) rather than (2), forbidding (2) appears not reasonable. Hence it is necessary to relax the consistency condition of the combine construct by allowing Σ_{n1} and Σ_{n2} to have common sorts and operations with the same "origin". Note that the same problem occurs in "bottom-up" specification languages such as CLEAR for a similar - but not identical - reason.

To this end a so-called *history function* is introduced. It maps each "new" sort or operation into the name of the procedure - together with the actual parameters - in which it was introduced by a create construct. The function is defined along the same lines as the semantic functions (see [LL 85]).

It is worthwhile to note that the solution proposed changes neither the syntax nor the semantics of OBSCURE. It merely modifies that part of the OBSCURE system which tests the consistency conditions.

7. Conclusions

OBSCURE is a simple, yet powerful specification language differing from other specification languages by its underlying principle of top-down development, by its interactive nature and by its link to verification. Thanks to an operational approach the parameterization concept has a simple semantics. A mild version of polymorphism and the union of common subspecifications are obtained without modifying the syntax and semantics of the language.

A further development of OBSCURE concerns its generalization for classes of models (*loose* specifications) and the inclusion of the concept of implementation (of a data type by another data type).

An implementation of OBSCURE based on [Le 85] is under development (Figure 9). The program development part will be completed before the end of 1985. The verification part is inspired from the AFFIRM-system [Th 79, Mu 80, Lo 80].

We are especially indebted to Rod Burstall and Don Sannella for several helpful discussions.

References

[BG 77] Burstall, R.M., Goguen, J.A., Putting theories together to make specifications, *Proc. 5th Joint Conf. on Art. Int.*, Cambridge, pp. 1045 - 1058 (1977)

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

[FGJM 85] Futatsugi, K., Goguen, J., Jouannaud, J.-P., Meseguer, J., Principles of OBJ2, *Proc. 12th POPL Conf.*, pp. 52 - 66 (1985)

[GM 83] Goguen, J.A., Meseguer, J., *Initiality, Induction and Computability*, SRI-CSL Techn. Rep. 140, Stanford Research Institute, December 1983

[Le 85] Lermen, C.W., The specification language OBSCURE, Int. Rep. A 85/11, Univ. Saarbrucken (1985)

[LL 85] Lermen, C.W., Loeckx, J., OBSCURE: An interactive specification language for model-oriented specification methods, Int. Rep. A 85/12, Univ. Saarbrucken (Sept. 1985)

[Lo 80] Loeckx, J., Proving properties of algorithmic specifications of abstract data types in AFFIRM. AFFIRM-Memo-29-JL, USC-ISI, Marina del Rey, 1980

[Lo 81] Loeckx, J., Algorithmic specifications of abstract data types, Proc. ICALP 81, LNCS 115 (1981), 129 - 147

[Lo 84] Loeckx, J., Algorithmic specifications: A constructive specification method for abstract data types. Int. Rep. A 84/03, Univ. Saarbrucken (1984). To appear in *TOPLAS*

[LS 84] Loeckx, J., Sieber, K., *The Foundations of Program Verification*, J. Wiley/Teubner-Verlag, New York/Stuttgart (1984)

[Ma 74] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill (1974)

[Mi 84] Milner, R., The Standard ML Core Language, Int. Rep. CSR-168-84, Univ. Edinburgh (1984)

[Mu 80] Musser, D.R., Abstract data type specification in the AFFIRM System, *IEEE Trans. on Softw. Eng.* SE-6, pp. 24 - 32 (1980)

[Sa 84] Sannella, D., A set-theoretic semantics for CLEAR, *Acta Inform.* 21, 5, 443 - 472 (1984)

[Sa 85] Sannella, D., The semantics of Extended ML, draft (May 1985)

[ST 85] Sannella, D., Tarlecki, A., Program specification and development in standard ML, *Proc. 12th POPL Conf.*, pp. 67 - 77 (1985)

[SW 83] Sannella, D., Wirsing, M., A kernel language for algebraic specification and implementation, Proc. Int. Coll. FCT, LNCS 158, 413 - 427, 1983

[Th 79] Thompson, D.H. (Ed.), AFFIRM Reference Manual. Internal Report, USC-ISI, Marina del Rey (1979)

Syntactic categories.

csp : composed specification
sp : specification
lso : list of sorts and operations
ls : list of sorts
lo : list of operations
lax : list of axioms

s : sort
o : operation
ax : axiom
m : model
n : name

Syntax.

csp ::= sp | (csp 1)
csp **compose** sp (csp 2)
sp ::= **create** **new** lso **model** m **global** lso | (sp 1)
sp **subset** **of** s **by** o | (sp 2)
sp **quotient** **of** s **by** o | (sp 3)
sp **forget** lso | (sp 4)
sp **rename** lso **as** lso | (sp 5)
sp **axioms** lax | (sp 6)
spec csp **endspec** (sp 7)
lso ::= ε | **sorts** ls | **opns** lo **sorts** ls **opns** lo
ls ::= s | ls s
lo ::= o | lo o
lax ::= ax | lax ax
o ::= n : → s | n : ls → s

FIGURE 1: The syntax of OBSCURE (without parameterized specifications).

$\mathcal{S}_{csp}$: csp → extension signature

$\mathcal{S}_{csp}(sp) = \mathcal{S}_{sp}(sp)$ (csp 1)

$\mathcal{S}_{csp}(csp\ \underline{compose}\ sp) =$ (csp 2)

$\underline{let}\ \mathcal{S}_{csp}(csp) = (\Sigma_{g1}, \Sigma_{n1})\ \underline{in}$

$\underline{let}\ \mathcal{S}_{sp}(sp) = (\Sigma_{g2}, \Sigma_{n2})\ \underline{in}$

$((\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2}\ ,\ \Sigma_{n1} \cup \Sigma_{n2})$

$\mathcal{S}_{sp}$: sp → extension signature

$\mathcal{S}_{sp}(\underline{create}\ \underline{new}\ lso1\ \underline{model}\ m\ \underline{global}\ lso2) = (\mathcal{S}_{lso}(lso2), \mathcal{S}_{lso}(lso1))$ (sp 1)

$\mathcal{S}_{sp}(sp\ \underline{subset}\ \underline{of}\ s\ \underline{by}\ o) = \mathcal{S}_{sp}(sp)$ (sp 2)

$\mathcal{S}_{sp}(sp\ \underline{quotient}\ \underline{of}\ s\ \underline{by}\ o) = \mathcal{S}_{sp}(sp)$ (sp 3)

$\mathcal{S}_{sp}(sp\ \underline{forget}\ lso) = \underline{let}\ \mathcal{S}_{sp}(sp) = (\Sigma_g, \Sigma_n)\ \underline{in}\ (\Sigma_g, \Sigma_n - \mathcal{S}_{lso}(lso))$ (sp 4)

$\mathcal{S}_{sp}(sp\ \underline{rename}\ lso1\ \underline{as}\ lso2) = \underline{let}\ \mathcal{S}_{sp}(sp) = (\Sigma_g, \Sigma_n)\ \underline{in}$ (sp 5)

$(\Sigma_g, \Sigma_n[lso1/lso2])$

$\mathcal{S}_{sp}(sp\ \underline{axioms}\ lax) = \mathcal{S}_{sp}(sp)$ (sp 6)

$\mathcal{S}_{sp}(\underline{spec}\ csp\ \underline{endspec}) = \mathcal{S}_{csp}(csp)$ (sp 7)

$\mathcal{S}_{lso}$: lso → signature

$\mathcal{S}_{lso}(\varepsilon) = (\emptyset, \emptyset)$

$\mathcal{S}_{lso}(\underline{sorts}\ ls) = (\mathcal{S}_{ls}(ls), \emptyset)$

$\mathcal{S}_{lso}(\underline{opns}\ lo) = (\emptyset, \mathcal{S}_{lo}(lo))$

$\mathcal{S}_{lso}(\underline{sorts}\ ls\ \underline{opns}\ lo) = (\mathcal{S}_{ls}(ls), \mathcal{S}_{lo}(lo))$

$\mathcal{S}_{ls}$: ls → set-of-sorts

$\mathcal{S}_{ls}(s) = \{s\}$

$\mathcal{S}_{ls}(ls\ s) = \mathcal{S}_{ls}(ls) \cup \{s\}$

$\mathcal{S}_{lo}$: lo → set-of-operations

$\mathcal{S}_{lo}(o) = \{o\}$

$\mathcal{S}_{lo}(lo\ o) = \mathcal{S}_{lo}(lo) \cup \{o\}$

FIGURE 2: The family of semantic functions $\mathcal{S}$ for OBSCURE without parameterized specifications

$\mathfrak{I}_{csp}$: csp → algebra extension
$\mathfrak{I}_{csp}(sp) = \mathfrak{I}_{sp}(sp)$ (csp 1)
$\mathfrak{I}_{csp}$(csp $\underline{\text{compose}}$ sp) (A) = (csp 2)
$\underline{\text{let}}$ $\mathfrak{S}_{csp}(csp) = (\Sigma_{g1}, \Sigma_{n1})$ $\underline{\text{in}}$
$\underline{\text{let}}$ $\mathfrak{S}_{sp}(sp) = (\Sigma_{g2}, \Sigma_{n2})$ $\underline{\text{in}}$
$\mathfrak{I}_{csp}(csp)((\mathfrak{I}_{sp}(sp)(A \mid \Sigma_{g2}) \cup A) \mid \Sigma_{g1})$
$\cup \ \mathfrak{I}_{sp}(sp)(A \mid \Sigma_{g2})$

$\mathfrak{I}_{sp}$: sp → algebra extension
$\mathfrak{I}_{sp}$($\underline{\text{create}}$ $\underline{\text{new}}$ lso1 $\underline{\text{model}}$ m $\underline{\text{global}}$ lso2) (A) = A ∪ $\mathfrak{I}_{m}(m)(A)$ (sp 1)
$\mathfrak{I}_{sp}$(sp $\underline{\text{subset}}$ $\underline{\text{of}}$ s $\underline{\text{by}}$ o) (A) = (sp 2)
$\underline{\text{let}}$ $\mathfrak{S}_{sp}(sp) = ((S_g, \Omega_g), (S_n, \Omega_n))$ $\underline{\text{in}}$
$\underline{\text{let}}$ p be the family of functions p_t, $t \in S_g \cup S_n$, with:
p_t a function which maps any element from
$\mathfrak{I}_{sp}(sp)(A)(t)$ into true, for all $t \neq s$
$p_s = \mathfrak{I}_{sp}(sp)(A)(o)$ $\underline{\text{in}}$
the subalgebra generated by $\mathfrak{I}_{sp}(sp)(A)$ and p
$\mathfrak{I}_{sp}$(sp $\underline{\text{quotient}}$ $\underline{\text{of}}$ s $\underline{\text{by}}$ o) (A) = (sp 3)
$\underline{\text{let}}$ $\mathfrak{S}_{sp}(sp) = ((S_g, \Omega_g), (S_n, \Omega_n))$ $\underline{\text{in}}$
$\underline{\text{let}}$ q be the family of functions q_t, $t \in S_g \cup S_n$, with:
q_t a function which maps a pair of elements from
$\mathfrak{I}_{sp}(sp)(A)(t)$ into true iff they are equal, for all $t \neq s$
$q_s = \mathfrak{I}_{sp}(sp)(A)(o)$ $\underline{\text{in}}$
the quotient algebra generated by $\mathfrak{I}_{sp}(sp)(A)$ and q
$\mathfrak{I}_{sp}$(sp $\underline{\text{forget}}$ lso) (A) = (sp 4)
$\underline{\text{let}}$ $\mathfrak{S}_{lso}(lso) = (S, \Omega)$ $\underline{\text{in}}$
$\mathfrak{I}_{sp}(sp)(A) - \{ (e, A(e)) \mid e \in S \cup \Omega \}$
$\mathfrak{I}_{sp}$(sp $\underline{\text{rename}}$ lso1 $\underline{\text{as}}$ lso2) (A) = B (sp 5)
where $B(e) = \mathfrak{I}_{sp}(sp)(A)(e)$ if e is a sort or operation not occurring in lso2
and where $B(e) = \mathfrak{I}_{sp}(sp)(A)(e')$ if e is a sort or operation occurring in lso2 and e' the corresponding sort or operation in lso1.
$\mathfrak{I}_{sp}$(sp $\underline{\text{axioms}}$ lax) = $\mathfrak{I}_{sp}(sp)$ (sp 6)
$\mathfrak{I}_{sp}$($\underline{\text{spec}}$ csp $\underline{\text{endspec}}$) = $\mathfrak{I}_{csp}(csp)$ (sp 7)

FIGURE 3: The family of semantic functions $\mathfrak{I}$ for OBSCURE without parameterized specifications

```
create
    new  sorts  set
           opns  Emptyset : → set
                 App : set × el → set
                 Insert : set × el → set
                 Memberof : set × el → bool
                 Subset : set × set → bool
                 Nodup : set → bool
                 Eq : set × set → bool
    model
      constructors  Emptyset : → set
                    App : set × el → set
      programs
        Insert(s,e) ⇐ if Memberof(s,e) then s else App(s,e)
        Memberof(s,e) ⇐ case s of
           Emptyset : s
           App(s',e') : if Equal(e,e') then true else Memberof(s',e)
        Subset(s1,s2) ⇐ case s1 of
           Emptyset : true
           App(s1',e) : if Memberof(s2,e) then Subset(s1',s2) else false
        Nodup(s) ⇐ case s of
           Emptyset : true
           App(s',e):if Memberof(s',e) then false else Nodup(s')
        Eq(s1,s2)⇐ if Subset(s1,s2) then Subset(s2,s1) else false
    global  sorts  el, bool
               opns  Equal : el × el → bool
                     true : → bool,  false : → bool
axioms  expressing that Equal : el × el → bool is an equivalence
    relation
forget  opns  App : set × el → set
subset  of  set  by  Nodup : set → bool
quotient  of  set  by  Eq : set × set → bool
```

FIGURE 4: An example of a specification. The global sorts and operations of the extension signature defined by this specification are those listed above under global, the new sorts and operations are those listed under new except for the operation App : *set* × *el* → *set*

Syntactic categories

pr : program
ld : list of declarations
d : declaration

Syntax

pr ::= ld csp
ld ::= ε | ld d
d ::= csp $\underline{\text{is}}$ $\underline{\text{proc}}$ n(lso) (d 1)
sp ::= ... | (as in Figure 1) (sp 1) to (sp 7)
$\underline{\text{call}}$ n(lso) (sp 8)

FIGURE 5: Figure 1 and Figure 5 together constitute the complete syntax of OBSCURE with parameterized specifications

$\mathcal{S}_{pr}$: pr → extension signature
$\mathcal{S}_{pr}$ (ld csp) = $\mathcal{S}_{csp}$ (csp)($\mathcal{S}_{ld}$(ld))

$\mathcal{S}_{ld}$: ld → environment
$\mathcal{S}_{ld}$ (ε) = $\emptyset$
$\mathcal{S}_{ld}$ (ld d) = $\mathcal{S}_{ld}$ (ld) $\cup$ $\mathcal{S}_{d}$ (d)

$\mathcal{S}_{d}$: d → environment
$\mathcal{S}_{d}$ (csp $\underline{\text{is}}$ $\underline{\text{proc}}$ n(lso)) = {(n,(lso,csp))}

$\mathcal{S}_{csp}$: csp → environment → extension signature
similar to Figure 2

$\mathcal{S}_{sp}$: sp → environment → extension signature
...... (similar to Figure 2) (sp 1) to (sp 7)
$\mathcal{S}_{sp}$ ($\underline{\text{call}}$ n(lso))(e) = (sp 8)
$\underline{\text{let}}$ e(n) = (lso1, csp) $\underline{\text{in}}$
$\underline{\text{let}}$ $\mathcal{S}_{csp}$(csp)(e) = (Σ_g,Σ_n) $\underline{\text{in}}$
(Σ_g,Σ_n)[lso1/lso]

FIGURE 6: The family of semantic functions $\mathcal{S}$ for OBSCURE with parameterized specifications

$\mathfrak{I}_{pr}$: pr → algebra extension

$\mathfrak{I}_{pr}$ (ld csp) = $\mathfrak{I}_{csp}$ (csp) $(\mathfrak{E}_{ld}(ld))$

$\mathfrak{I}_{csp}$: csp → environment → algebra extension

similar to Figure 3

$\mathfrak{I}_{sp}$: sp → environment → algebra extension

.... (similar to Figure 3) (sp 1) to (sp 7)

$\mathfrak{I}_{sp}$ (call n(lso))(e) = (sp 8)

let e(n) = (lso1, csp) in

$\mathfrak{I}_{csp}\ (csp^{lso}_{lso1})(e)$

FIGURE 7: The family of semantic functions $\mathfrak{I}$ for OBSCURE with parameterized specifications

```
create                                                          ⎤
                                                                ⎥
  :                                                             ⎥ Figure 4
                                                                ⎥
quotient of set by Eq : set × set → bool                        ⎦
is proc SET (sorts el opns Equal : el × el → bool)                          (1)

call SET (sorts setofint opns Eq : setofint × setofint → bool)  (2)
rename sorts set opns Emptyset : → set
   as sorts setofsetofint opns Emptysetofsetofint : → setofsetofint (3)
combine
   call SET (sorts int opns Equal : int × int → bool)           (4)
   rename sorts set opns Emptyset : → set
       as sorts setofint opns Emptysetofint : → setofint        (5)
is proc SETOFSETOFINT( )                                        (6)

create .... (model introducing the sort pair and the operations
   Pair : el1 × el2 → pair, First : pair → el1, Second : pair → el2)   (7)
is proc PAIR (sorts el1, el2)

call PAIR (sorts setofsetofint, setofsetofint)
combine
   call SETOFSETOFINT( )
```

FIGURE 8: An OBSCURE program introducing pairs of sets of sets of integers (see Section 5.3 for comments)

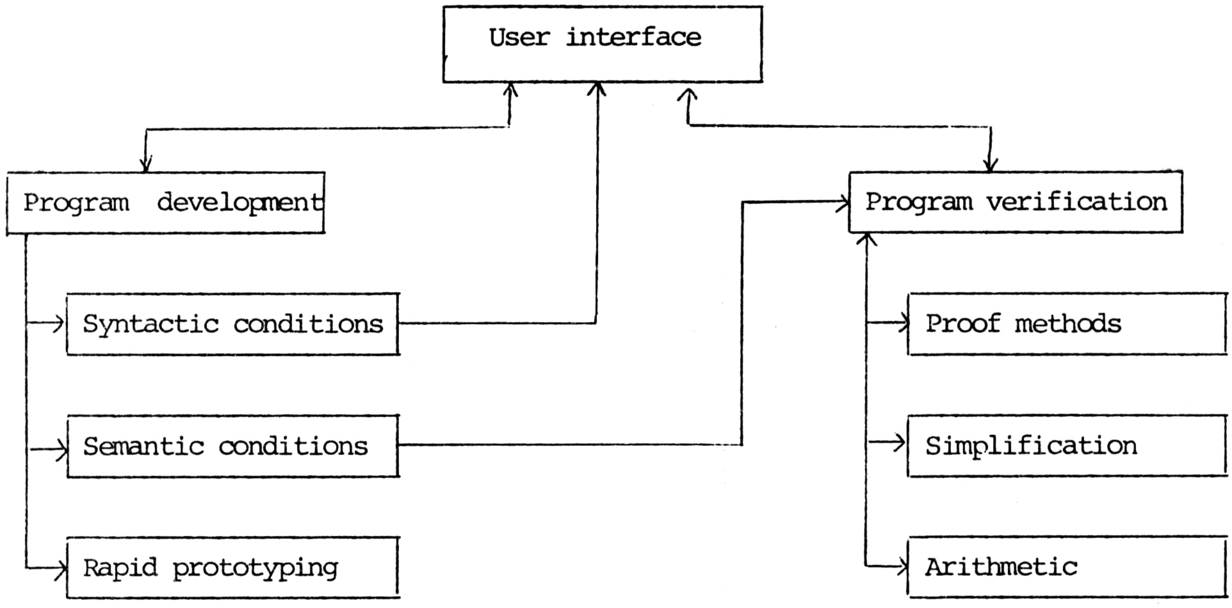

FIGURE 9: A rough scheme of an OBSCURE implementation