A formal description of the specification language OBSCURE

by

Jacques Loeckx

A 85/15

Saarbrücken, November 1985

ABSTRACT

The present report gives a complete formal definition of the specification language OBSCURE. An informal introduction to this language may be found in [LL 85].

1.	Introduction	1
2.	Syntactic and semantic notions	3
2.1	Signatures	3
2.2	Algebras	4
2.3	Algebra extensions	5
2.4	Subalgebras and quotient algebras	6
3.	A formal description of OBSCURE	7
3.1	The syntax	8
3.2	An informal introduction to the semantics	8
3.3	Some more definitions and notations	11
3.3.1	Matching pairs	11
3.3.2	The notation [lso/lso']	12
3.3.3	Environments	14
3.4	The semantic functions	15
3.5	The context conditions	21
4.	The consistency proof	21
5.	Uniting identical subspecifications	26
5.1	Introduction	26
5.2	The provenance function	27
5.3	The context conditions	28
5.4	The consistency proof	30
5.5	Discussion	31
6.	Two simple examples	32
6.1	The data type "sets of elements"	32
6.2	The data type "pairs of sets of sets of integers"	34
7.	Conclusions	34
	References	37
	Appendix	A1 - A14

1. Introduction

Recently several specification languages based on the use of abstract data types have been 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 specification language OBSCURE differs from these languages in at least four respects.

In the existing languages a specification is classically interpreted as an algebra or a theory. The specification of elaborate abstract data types is then performed "bottom-up" by putting algebras or theories together. Instead, a specification in OBSCURE introduces "new" types on the basis of not yet specified - henceforth called "global" - ones. More formally, an OBSCURE specification is interpreted as a function mapping an algebra (containing the global types) into an extension of this algebra (containing in addition the new types). The specification of elaborate abstract data types may therefore be performed "top-down" by "stepwise refinement": for each global type introduced a specification is constructed which introduces this type on the basis of other, more "elementary" (global) types. The final specification is obtained by "composing" these different specifications. Take, as an example, the development of an interpreter for a programming language. First, the data type *program* is specified on the basis of data types such as *input-data* and *output-data*; the type *program* contains, in particular, the operation *Interprete* mapping programs and input data into output data. Next, the data type *input-data* is specified, etc. The development is completed when the remaining global types are "known" ones, for example data types provided by a usual programming language.

Next, OBSCURE explicitly links program development to program verification. This is reflected by the fact that an OBSCURE implementation consists of a developement module and a verification module (see Figure 9 in Section 7). This feature allows OBSCURE to contain, for instance, language constructs for transforming an algebra into a subalgebra or a quotient algebra.

It is well-known that such transformations make sense only if certain (semantic) constraints are satisfied, viz. a closure condition in the case of a subalgebra and a congruence condition in the case of a quotient algebra (see e.g. [EM 85, MG 84]). The verification module may then be used to prove that these constraints are satisfied.

OBSCURE is designed for interactive use. A specification in OBSCURE essentially consists of a sequence of elementary constructs. At each construct the development module of the OBSCURE implementation automatically checks the syntactic constraints. Moreover, it generates the theorems expressing the semantic constraints - such as those implied by the construct yielding a subalgebra or quotient algebra - and transmits them to the verification module. The proof of these theorems by the verification module is likewise performed interactively.

Finally, by performing type inferencing OBSCURE allows a mild form of polymorphism.

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 present version of OBSCURE is more general in that it is applicable to any specification method using a single model. It is, for instance, also applicable to the initial algebra specification method.

The goal of the present paper is to give a complete formal description of the language. An informal introduction may be found in [LL 85]. Note that the version of OBSCURE presented here contains no syntactic sugar. This simplifies the description but makes the examples look unappealing. Ideas for syntactic sugar may be found in [Le 85].

Section 2 introduces the main syntactic and semantic notions. Section 3 presents a formal description of OBSCURE. Section 4 proves that the definitions of Section 3 are consistent. Sec-

tion 5 allows OBSCURE to unite identical subspecifications. Section 6 and 7 contain examples and conclusions. The proofs of the Lemmas and Theorems are in the Appendix.

2. Syntactic and semantic notions

2.1 Signatures

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

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

where n is an identifier, called *operation name*, and where $s_1, \ldots, s_k$, s are sorts. The (k + 1)-tuple

$$s_1 \; \ldots \; s_k \rightarrow s$$

is called the *type* of the operation. If S is a set of sorts containing $s_1, \ldots, s_k$, s, then the operation is said to be S-*sorted*. A set of operations is S-*sorted*, if its elements are. Note that an operation is classically defined to be an operation name characterizing univocally its operation type. The present, more refined definition allows polymorphism.

A set of operations is called *unambiguous*, if for any two operations of the form

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

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

Memberof : *intset int* → *bool*

and Memberof : *stringset string* → *bool*

(take i = 1 or i = 2) but not

Emptyset : → *intset*

and Emptyset : → *stringset*

Informally, if a set of operations is unambiguous, type inferencing allows to determine the operation for each occurrence of an operation name in a term or a formula.

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

(i) Ω is S-sorted;

(ii) S contains the special sort *bool*;

(iii) Ω contains the special operations true : $\rightarrow$ *bool* and false : $\rightarrow$ *bool*;

(iv) Ω is unambiguous.

Informally, the conditions (ii) and (iii) will allow the construction of subalgebras and quotient algebras; the condition (iv) allows type inferencing.

2.2 Algebras

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

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

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

The set of all Σ-algebras is denoted Alg_Σ.

If $\Sigma = (S,\Omega)$ and Σ' are algebra signatures with $\Sigma \subseteq \Sigma'$, and if A is a Σ'-algebra, then $A|\Sigma$ is an abbreviated notation for the Σ-algebra $A \mid (S \cup \Omega)$ (where "$\mid$" denotes the restriction of a function).

An algebra is *standard* if it maps *bool*, true : $\rightarrow$ *bool* and false : $\rightarrow$ *bool* into their usual meaning. Henceforth only standard algebras are considered.

2.3 Algebra extensions

As indicated in Section 1 a specification is going to be interpreted as a function mapping an algebra (containing "global" sorts and operations from a signature Σ_g) into an algebra obtained by adding "new" sorts and operations (from a signature Σ_n). The "new" sorts and operations are those introduced by the specification; the "global" ones are those which have still to be specified (in the course of the top-down development) and which therefore are similar to global variables in an Algol-like program. These functions are called "algebra extensions" and are characterized syntactically by a pair of signatures, called "extension signatures".

Formally, 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.

Σ_g is called the *global*, Σ_n the *new* signature.

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

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

such that for each $A \in Alg_{\Sigma_g}$:

either $E(A)$ is undefined

or $A \subseteq E(A)$.

Informally, the condition "$A \subseteq E(A)$" is equivalent with

$$(E(A) \mid \Sigma_g) = A$$

and expresses that the algebra $E(A)$ is an extension of the algebra A. By defining an algebra extension as a partial rather than a total function we will be able to account for semantic constraints imposed by the specifications. In fact, the main effect of such a constraint will be seen to restrict the domain of an extension to the algebras satisfying the constraint (see Section 3.3).

2.4 Subalgebras and quotient algebras

Two constructions are recalled which yield an algebra, called subalgebra and quotient algebra respectively. These constructions which were already mentioned in Section 1, are well-known from the literature (see e.g. [MG 84, EM 85]).

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

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

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

$$B(s) = \{c \in A(s) \mid p_s(c) = true\} \qquad \text{for each } s \in S$$

$$B(n : s_1 \ldots s_k \rightarrow s) = (A(n : s_1 \ldots s_k \rightarrow s)) \mid B(s_1)\times\ldots\times B(s_k)$$

$$\text{for each } n : s_1 \ldots s_k \rightarrow s \in \Omega,\ k \geq 0.$$

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

for each operation $n : s_1 \ldots s_k \rightarrow s \in \Omega$, $k \geq 0$:

for all $(a_1,\ldots,a_k) \in A(s_1) \times\ldots\times A(s_k)$:

if $p_{s_i}(a_i) = true$ for all i, $1 \leq i \leq k$,

then $p_s(A(n : s_1 \ldots s_k \rightarrow s)(a_1, \ldots, a_k)) = true$

Henceforth we will only consider special families p, viz. families such that for all sorts s the function

$$p_s : A(s) \rightarrow \{true, false\}$$

yields the value "true" for any argument, except for one distinguished sort, say t. We then speak of the *subalgebra generated by* A *and* p_t and mean the subalgebra generated by A and p.

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

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

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

$$B(s) = \{[c] \mid c \in A(s)\} \qquad \text{for each } s \in S$$

$$B(n : s_1 \ldots s_k \rightarrow s)([c_1], \ldots, [c_k])$$
$$= [A(n : s_1 \ldots s_k \rightarrow s)(c_1, \ldots, c_k)]$$
$$\text{for all } c_i \in A(s_i),\ 1 \le i \le k$$
$$\text{for all } n : s_1 \ldots s_k \rightarrow s \in \Omega,\ k \ge 0$$

where [...] denotes an equivalence class. This definition is consistent only if each operation of the algebra A satisfies a *congruence condition*. Informally, a congruence condition expresses that equivalent arguments lead to equivalent values. Formally:

for each operation $n : s_1 \ldots s_k \rightarrow s \in \Omega,\ k \ge 0$:
for all $(a_1, \ldots, a_k), (a'_1, \ldots, a'_k) \in A(s_1) \times \ldots \times A(s_k)$:
if $q_{s_i}(a_i, a'_i) = \text{true}$ for all i, $1 \le i \le k$,
then $q_s(A(n : s_1 \ldots s_k \rightarrow s)(a_1, \ldots, a_k),$
$A(n : s_1 \ldots s_k \rightarrow s)(a'_1, \ldots, a'_k)) = \text{true}$.

Henceforth we will only consider special families q, viz. families such that for all sorts s the function

$$q_s : A(s) \times A(s) \rightarrow \{\text{true}, \text{false}\}$$

yields the value "true" for any argument, except for one distinguished sort, say t. We then speak of the *quotient algebra generated by* A *and* q_t and mean the quotient algebra generated by A and q.

3. A formal description of OBSCURE

OBSCURE is described by a context-free grammar (Figure 1), by a "semantic function" $\mathcal{S}$ mapping specifications into extension signatures (Figure 2), by a "semantic function" $\mathcal{J}$ mapping specifications into algebra extensions (Figure 3) and by context conditions (Figure 4) guaranteeing the consistency of the definitions of the semantic functions. The techniques used are inspired from the description of CLEAR in [Sa 84] and of ASL in [SW 83].

The description of OBSCURE leaves open which specification method and logic are used.

3.1 The syntax

The context-free syntax for OBSCURE is in Figure 1.

The distinction between csp ("composed specification") and sp ("specification") provides the "operator" compose with a lower priority than the other "operators". This allows to save bracket pairs and to write, for instance

sp1 compose sp2 forget lso

instead of

sp1 compose (sp2 forget lso)

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]. Similarly, the definition of f ("formula") depends on the logic used. In the case of the algorithmic specification method this is the logic of strict algebras [Lo 84].

3.2 An informal introduction to the semantics

The context-free rule (sp1) of Figure 1 introduces the "new" sorts and operations listed under new lso by describing a model m. The description of this model makes use of the sorts and operations listed under global lso. Hence the rule (sp1) "creates" an algebra extension, the global and new signature of which are fixed by global lso new lso respectively. The rule (sp2) transforms an algebra into a subalgebra, s being the distinguished sort. 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

Syntactic categories

pr : program	lf : list of formulas
csp : composed specification	d : declaration
sp : specification	s : sort
ld : list of declarations	o : operation
lso : list of sorts and operations	f : formula
ls : list of sorts	m : model
lo : list of operations	n : name

Syntax

```
pr ::= ld csp                                             (pr1)
ld ::= ε | ld d                                   (ld1), (ld2)
d  ::= csp is proc n(lso)                                 (d1)
csp ::= sp |                                              (csp1)
        csp compose sp                                    (csp2)
sp ::= create new lso model m global lso |                (sp1)
       sp subset of s by o |                              (sp2)
       sp quotient of s by o |                            (sp3)
       sp forget lso |                                    (sp4)
       sp rename lso as lso |                             (sp5)
       sp axioms lf  |                                    (sp6)
       (csp) |                                            (sp7)
       call n(lso)                                        (sp8)
lso ::= ε | sorts ls | opns lo | sorts ls opns lo (lso1)bis(lso4)
ls ::= s | ls s                                   (ls1), (ls2)
lo ::= o | lo o                                   (lo1), (lo2)
lf ::= f | lf f                                   (lf1), (lf2)
o  ::= n : → s | n : ls → s                       (o1),  (of2)
```

FIGURE 1: The context-free syntax of OBSCURE

operations according to (sp5) is especially useful for parameterized data types. The formulas introduced by (sp6) are transmitted for proof to the verification module of the OBSCURE system. With these formulas the user may, for instance, check properties of the data type introduced. Moreover, the formulas may be used to constrain the actual parameters of a parameterized specification.

The rule (sp8) performs a procedure call: n is the name of the procedure, the sorts and operations of lso are the actual parameters.

The context-free rule (csp2) "composes" the specifications csp and sp. If some of the global sorts and operations of csp are introduced by sp (as new sorts and operations), the application of the rule constitutes a "refinement step" in the top-down development: in the resulting specification these common sorts and operations are no longer global. For example, let s, t, u, v, w each denote a sort or operation. If C_t^s denotes a (composed) specification with s "new" and t global, and D_u^t a specification with t "new" and u global, then

$$C_t^s \ \underline{\text{compose}} \ D_u^t$$

is a (composed) specification with s and t "new" and u "global". More generally, if $C_{t,v}^s$ is a specification with s "new" and t, v global, and if $D_{v,u}^{t,w}$ is defined similarly, then

$$C_{t,v}^s \ \underline{\text{compose}} \ D_{v,u}^{t,w}$$

is a specification with s, t, w "new" and v, u global.

Rule (d1) corresponds to a procedure declaration: csp is the procedure body, n is the procedure name and the sorts and operations of lso are the formal parameters. If the procedure name n is declared by

csp <u>is</u> <u>proc</u> n(lso)

the meaning of the procedure call

<u>call</u> n(lso1)

is that of the procedure body csp with the sorts and operations of lso (i.e. the formal parameters) replaced by those of lso1 (i.e. the actual parameters). The context conditions which will be discussed in Section 3.5 require that a procedure body only contains calls to procedures already declared (in the list of declarations). This excludes, in particular, recursive procedure calls.

While procedures with and without parameters have essentially the same syntax and semantics, their use results from different concerns. Procedures with parameters constitute parameterized specifications and avoid respecifying similar data types. Parameterless procedures allow to modularize the design: instead of developing a big single specification by successively composing "elementary" ones, each of these elementary specifications may be given a name and called at "composition time".

By the way, parameterized specifications suggest the notion of a library and, hence, bottom-up development. Actually, for practical reasons it is not possible to do without parameterized specifications, even in a specification language based on top-down development.

3.3 Some more definitions and notations

Before the semantics of OBSCURE can be described it is necessary to introduce the following definitions.

3.3.1 Matching pairs

Using the syntactic categories and notations of Figure 1 let lso be a list of sorts and operations of the form

$$\underline{\text{sorts}}\ s_1 \ldots s_k \quad \underline{\text{opns}}\ o_1 \ldots o_l$$

with $k, l \geq 0$. (It is understood that the word <u>sorts</u> is missing when $k = 0$, and similarly for $l = 0$). The list of sorts and operations lso is said *to have no duplicates* if:

s_i, s_j are pairwise disjoint for all i, j with $1 < i \leq j \leq k$

and

o_i, o_j are pairwise disjoint for all i, j with $1 < i \leq j \leq l$.

Let

$$\text{lso} = \underline{\text{sorts}}\ s_1 \ \dots\ s_k\ \underline{\text{opns}}\ o_1\ \dots\ o_l$$
$$\text{lso'} = \underline{\text{sorts}}\ s'_1 \ \dots\ s'_{k'}, \underline{\text{opns}}\ o'_1\ \dots\ o'_{l'},$$

be two lists of sorts and operations, k, l, k', l' ≥ 0. Assume that lso has no duplicates. Then the pair lso, lso' is said to *match* if

(i) k = k' and l = l'

(ii) for all i, 1 ≤ i ≤ l:

$\underline{\text{let}}\ o_i = n_i : \text{optype}_i\ \underline{\text{in}}$

$\underline{\text{let}}\ o'_i = n'_i : \text{optype}'_i\ \underline{\text{in}}$

$\text{optype}'_i = (\text{optype}_i)^{s'_1, \dots, s'_k}_{s_1, \dots, s_k}$

In this definition optype stands for the syntactical category defined by

optype ::= → s | ls → s

and $(\dots)^{s'_1, \dots, s'_k}_{s_1, \dots, s_k}$ denotes the simultaneous substitution of the identifiers $s_1, \dots, s_k$ by $s'_1, \dots, s'_k$. Note that this simultaneous substitution is defined because lso has no duplicates.

An example of a matching pair is

$$\left.\begin{array}{l} \underline{\text{sorts}}\ \ s\ \ t\ \ \underline{\text{opns}}\ \ n : s\ u \rightarrow t \\ \underline{\text{sorts}}\ \ v\ \ v\ \ \underline{\text{opns}}\ \ m : v\ u \rightarrow v \end{array}\right\} (1)$$

(provided s ≠ t). Counterexamples are

$\underline{\text{sorts}}$ s $\underline{\text{opns}}$ n : u → s

$\underline{\text{sorts}}$ v $\underline{\text{opns}}$ m : u → s

(if v ≠ s) and

$\underline{\text{sorts}}$ v v

$\underline{\text{sorts}}$ s t

3.3.2 The notation [lso/lso']

Let lso, lso' be two lists of sorts and operations. Assume that lso has no duplicates and that lso, lso' match. Let

$$lso = \underline{sorts}\ s_1 \ldots s_k\ \underline{opns}\ o_1 \ldots o_l$$

$$lso' = \underline{sorts}\ s'_1 \ldots s'_k\ \underline{opns}\ o'_1 \ldots o'_l\ ,$$

$k, l \geq 0$.

Let s be a sort. We define

$$s[lso/lso'] = \begin{cases} s'_i\ , \text{ if } s = s_i \text{ for some } i,\ 1 \leq i \leq k \\ s \text{ else.} \end{cases}$$

Let

$$o = n : optype$$

be an operation. We define

$$o[lso/lso'] = \begin{cases} o'_i, \text{ if } o = o_i \text{ for some } i,\ 1 \leq i \leq l \\ n : (optype)^{s'_1,\ldots,s'_k}_{s_1,\ldots,s_k} \text{ else.} \end{cases}$$

For example, with lso, lso' as in the example (1) of Section 3.2.1:

$$(n : s\ u \rightarrow t)\ [lso/lso'] = m : v\ u \rightarrow v$$

$$(n' : s\ t \rightarrow t)\ [lso/lso'] = n' : v\ v \rightarrow v$$

Let be

S a set of sorts

Ω a set of operations

(S,Ω) a signature

(Σ_g,Σ_n) an extension signature.

We define

$$S[lso/lso'] = \{s[lso/lso'] \mid s \in S\}$$

$$\Omega[lso/lso'] = \{o[lso/lso'] \mid o \in \Omega\}$$

$$(S,\Omega)[lso/lso'] = (S[lso/lso'],\ \Omega[lso/lso'])$$

$$(\Sigma_g,\Sigma_n)\ [lso/lso'] = (\Sigma_g[lso/lso']\ ,\ \Sigma_n[lso/lso'])$$

Finally, let

A be an algebra with signature (S,Ω)

E be an algebra extension with extension signature (Σ_g,Σ_n)

We define

$$A[lso/lso'] = \{(c[lso/lso'],\ A(c)) \mid c \in S \cup \Omega\}$$

$$E[lso/lso'] = \{\ (\ A[lso/lso'],\ (E(A))[lso/lso']\ ) \mid A[lso/lso'] \text{ is an algebra } \}$$

Note that the notation [lso/lso'] does, for instance, not respect the property of unambiguity of a set of operations. For instance, if

$$\Omega = \{n : s \to t,\ n : u \to v\}\ ,$$

$s \neq u$, $u \neq v$, then

$$\Omega[\underline{\text{sorts}}\ \ s\ /\ \underline{\text{sorts}}\ \ u]$$

is ambiguous. Hence, the notation [lso/lso'] applied to an extension signature does not necessarily yield an extension signature. A similar remark holds for algebras and algebra extensions. Note in particular that a necessary condition for

$$A[\underline{\text{sorts}}\ \ s\ t\ /\ \underline{\text{sorts}}\ \ u\ u]$$

to be an algebra is $A(s) = A(t)$. On the other hand, by its very definition the domain of E[lso/lso'] consists only of algebras.

3.3.3 Environments

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 these pieces of text. While in most specification languages the approach taken is essentially the denotational one (e.g. [Sa 84, EM 85]), the approach taken here is the operational one. As an advantage it leads to a very simple semantics ("copy rule semantics").

Formally, let n, lso, csp be the syntactic categories of Figure 1. An *environment* is a function

$$e : n \to lso \times csp$$

Informally, an environment e maps the procedure name n into the formal parameters lso and the procedure body csp. Note that these parameters consist of sorts and operations.

3.4 The semantic functions

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

The definition of the semantic function $\mathfrak{S}$ is in Figure 2. The line (sp1) should be evident from the comments of Section 3.2. The lines (sp2) and (sp3) may be understood by noting that the constructions yielding a subalgebra or a quotient algebra leave the algebra signature unchanged. Line (sp4) implies that only new sorts and operations may be "forgotten". A similar remark holds for line (sp5). Line (sp8) should be clear from the comments of Section 3.2. Line (csp2) performs the union of the global and new signatures of csp and sp but removes the sorts and operations common to Σ_{g1} and Σ_{n2} from the global signature - as explained in Section 3.2. The union in line (ld2) applies to the graphs of the functions $\mathfrak{S}_{ld}$(ld) and $\mathfrak{S}_{d}$(d). Note that the value of $\mathfrak{S}_{pr}$, $\mathfrak{S}_{csp}$ and $\mathfrak{S}_{sp}$ is defined as a pair of signatures. The role of the context conditions which are to be given in Section 3.5 is to make sure that these pairs of signatures are extension signatures. A similar remark holds for the value of $\mathfrak{S}_{ld}$.

The semantic function $\mathfrak{I}$ is defined in Figure 3. Most of the equalities define an algebra extension by its value for an arbitrary argument A. We now shortly discuss the values and the domains of these algebra extensions.

In line (sp1) the value is the union of (the graphs of) the functions A and $\mathfrak{I}_m$(m)(A), the latter function being the algebra defined by the model m. Clearly, a precise definition of $\mathfrak{I}_m$(m) depends on the specification method used. In the case of the algorithmic specification method for instance, $\mathfrak{I}_m$(m)(A) associates with each sort from lso1 a (constructively defined) formal language and with each operation from lso1 a (constructively defined) function (see [Lo 84]). The algebra

extension defined in line (sp1) is total in the sense that the algebra A is an arbitrary $\mathfrak{S}_{lso}$(lso2)-algebra.

Line (sp2) of Figure 3 expresses the construction of a subalgebra - according to the comments in Section 3.2. This construction restricts the domain of the extension to the algebras satisfying the closure condition. The precise definition of $\mathfrak{I}_{lf}$(lf) depends on the logic used. Similar remarks hold for line (sp3).

The value of line (sp4) reflects the fact that the sorts and operations from lso are "new" ones - as will be required by the context conditions. The domain of the algebra extension remains unchanged. Similar remarks hold for line (sp5).

The effect of line (sp6) is to restrict the domain of the algebra extensions to those algebras for which the formulas from lf hold.

Line (sp8) "interpretes" the procedure body csp and then replaces the formal parameters lso1 by the actual parameters lso. The domain of the algebra extension E remains unchanged - up to renaming.

The line (csp2) is best explained by Figure 4(b). Very informally speaking, line (csp2) expresses the composition of the algebra extensions $\mathfrak{I}_{csp}$(csp) and $\mathfrak{I}_{sp}$(sp) for those sorts and operations which are common to Σ_{n2} and Σ_{g1}; for the other sorts and operations it expresses their "union".

Again, the value of the semantic function $\mathfrak{I}$ as defined in Figure 3 is not necessarily an algebra extension. Once more it is the role of the context conditions to put it sure.

At this point it is interesting to note that the development of specifications has to do with the values of the algebra extensions and their verification with the domains of these algebra extensions. In fact, each theorem to be proved during verification expresses that a given algebra - or a given class of algebras - belongs to the domain of a given algebra extension.

$\mathcal{S}pr$: pr → extension signature

$\mathcal{S}_{pr}$ (ld csp) = $\mathcal{S}_{csp}$ (csp)($\mathcal{S}_{ld}$ (ld)) (pr1)

$\mathcal{S}_{ld}$: ld → environment

$\mathcal{S}_{ld}(\varepsilon) = \emptyset$ (ld1)

$\mathcal{S}_{ld}$ (ld d) = $\mathcal{S}_{ld}$ (ld) $\cup$ $\mathcal{S}_{d}$ (d) (ld2)

$\mathcal{S}_{d}$: d → environment

$\mathcal{S}_{d}$ (csp <u>is</u> <u>proc</u> n(lso)) = {(n,(lso,csp))} (d1)

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

$\mathcal{S}_{csp}$ (sp) = $\mathcal{S}_{sp}$ (sp) (csp1)

$\mathcal{S}_{csp}$ (csp <u>compose</u> sp)(e) = (csp2)

<u>let</u> $(\Sigma_{g1},\Sigma_{n1})$ = $\mathcal{S}_{csp}$ (csp)(e) <u>in</u>

<u>let</u> $(\Sigma_{g2},\Sigma_{n2})$ = $\mathcal{S}_{sp}$(sp)(e) <u>in</u>

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

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

$\mathcal{S}_{sp}$(<u>create</u> <u>new</u> lso1 <u>model</u> m <u>global</u> lso2)(e) (sp1)

= ($\mathcal{S}_{lso}$ (lso2), $\mathcal{S}_{lso}$ (lso1))

$\mathcal{S}_{sp}$ (sp <u>subset</u> <u>of</u> s <u>by</u> o) = $\mathcal{S}_{sp}$ (sp) (sp2)

$\mathcal{S}_{sp}$ (sp <u>quotient</u> <u>of</u> s <u>by</u> o) = $\mathcal{S}_{sp}$ (sp) (sp3)

$\mathcal{S}_{sp}$ (sp <u>forget</u> lso)(e) = (sp4)

<u>let</u> (Σ_g,Σ_n) = $\mathcal{S}_{sp}$ (sp)(e) <u>in</u>

$(\Sigma_g, \Sigma_n - \mathcal{S}_{lso}$(lso))

$\mathcal{S}_{sp}$ (sp <u>rename</u> lso1 <u>as</u> lso2)(e) = (sp5)

<u>let</u> (Σ_g, Σ_n) = $\mathcal{S}_{sp}$ (sp)(e) <u>in</u>

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

$\mathcal{S}_{sp}$ (sp <u>axioms</u> lf) = $\mathcal{S}_{sp}$ (sp) (sp6)

$\mathcal{S}_{sp}$ ((csp)) = $\mathcal{S}_{csp}$ (csp) (sp7)

$\mathcal{S}_{sp}$ (<u>call</u> n(lso))(e) = (sp8)

<u>let</u> (lso1, csp) = e(n) <u>in</u>

<u>let</u> (Σ_g, Σ_n) = $\mathcal{S}_{csp}$ (csp)(e) <u>in</u>

(Σ_g, Σ_n)[lso1/lso]

$\mathfrak{S}_{lso}$: lso → signature

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

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

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

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

$\mathfrak{S}_{ls}$: ls → set-of-sorts

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

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

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

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

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

FIGURE 2: The family of semantic functions $\mathfrak{S}$

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

$\mathfrak{I}_{pr}(ld\ csp) = \mathfrak{I}_{csp}(csp)(\mathfrak{S}_{ld}(ld))$ (pr1)

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

$\mathfrak{I}_{csp}(sp) = \mathfrak{I}_{sp}(sp)$ (csp1)

$\mathfrak{I}_{csp}(csp\ \underline{\text{compose}}\ sp)(e)(A) =$ (csp2)

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

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

$\underline{\text{let}}\ B = \mathfrak{I}_{sp}(sp)(e)(A|\Sigma_{g2})\ \underline{\text{in}}$

- $\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1}) \cup B$
  if both $B = \mathfrak{I}_{sp}(sp)(e)(A \mid \Sigma_{g2})$ and
  $\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})$ are
  defined;
- undefined else.

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

$\mathfrak{I}_{sp}(\underline{\text{create}}\ \underline{\text{new}}\ lso1\ \underline{\text{model}}\ m\ \underline{\text{global}}\ lso2)(e)(A) =$ (sp1)

$A \cup \mathfrak{I}_{m}(m)(A)$

$\mathfrak{I}_{sp}(sp\ \underline{\text{subset}}\ \underline{\text{of}}\ s\ \underline{\text{by}}\ o)(e)(A) =$ (sp2)

$\underline{\text{let}}\ ((S_g, \Omega_g), (S_n, \Omega_n)) = \mathfrak{S}_{sp}(sp)(e)\ \underline{\text{in}}$

$\underline{\text{let}}\ p_s = \mathfrak{I}_{sp}(sp)(e)(A)(o)\ \underline{\text{in}}$

- the subalgebra generated by $\mathfrak{I}_{sp}(sp)(e)(A)$ and p_s,
  if $\mathfrak{I}_{sp}(sp)(E)(A)$ is defined
  and $\mathfrak{I}_{sp}(sp)(e)(A)$ satisfies the closure
  conditions;
- undefined else.

$\mathfrak{I}_{sp}$ (sp quotient of s by o)(e)(A) = (sp3)
  let $((S_g,\Omega_g), (S_n,\Omega_n)) = \mathfrak{S}_{sp}$ (sp)(e) in
  let $q_s = \mathfrak{I}_{sp}$ (sp)(e)(A)(o) in
  - the quotient algebra generated by A and q_s, if $\mathfrak{I}_{sp}$ (sp)(e)(A) is defined and $\mathfrak{I}_{sp}$ (sp)(e)(A) satisfies the congruence conditions;
  - undefined else.

$\mathfrak{I}_{sp}$ (sp forget lso)(e)(A) = (sp4)
  let $(\Sigma_g,\Sigma_n) = \mathfrak{S}_{sp}$ (sp)(e) in
  let B = $\mathfrak{I}_{sp}$ (sp)(e)(A) in
  - B | $(\Sigma_g \cup (\Sigma_n - \mathfrak{S}_{lso}$ (lso))), if $\mathfrak{I}_{sp}$ (sp)(e)(A) is defined;
  - undefined else.

$\mathfrak{I}_{sp}$ (sp rename lso1 as lso2)(e)(A) = (sp5)
  let B = $\mathfrak{I}_{sp}$ (sp)(e)(A) in
  - B[lso1/lso2], if $\mathfrak{I}_{sp}$ (sp)(e)(A) is defined;
  - undefined else.

$\mathfrak{I}_{sp}$ (sp axioms lf)(e) (A) = (sp6)
  - $\mathfrak{I}_{sp}$ (sp)(e)(A), if $\mathfrak{I}_{sp}$ (sp)(e)(A) is defined and $\mathfrak{I}_{lf}$ (lf)($\mathfrak{I}_{sp}$(sp)(e)(A)) = true ;
  - undefined else.

$\mathfrak{I}_{sp}$((csp)) = $\mathfrak{I}_{csp}$ (csp) (sp7)

$\mathfrak{I}_{sp}$ (call n(lso))(e) = (sp8)
  let (lso1, csp) = e(n) in
  let E = $\mathfrak{I}_{csp}$(csp)(e) in
  E[lso1/lso]

FIGURE 3: The family of semantic functions $\mathfrak{I}$

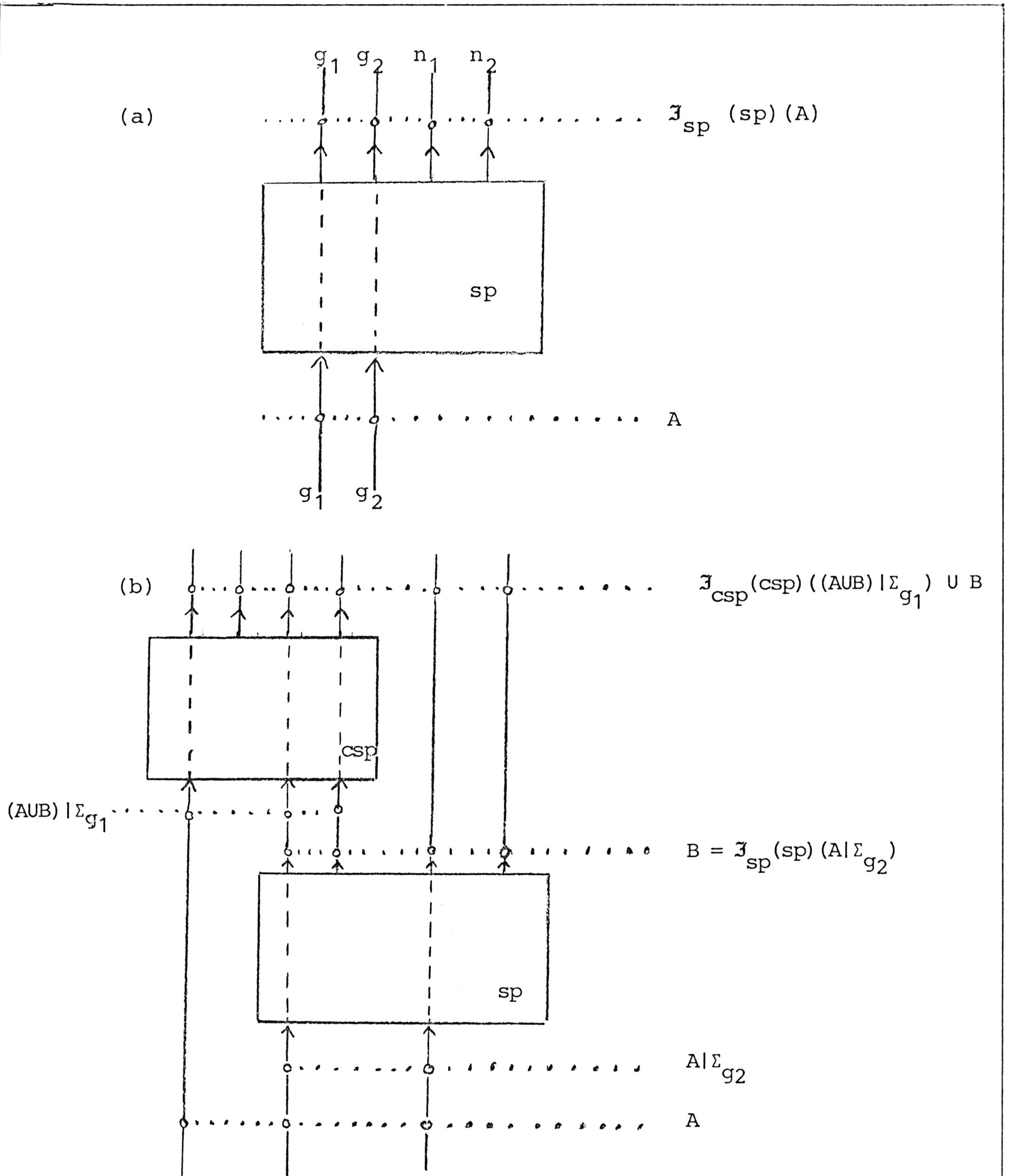

FIGURE 4:

(a) A schematical representation of the semantics of a specification sp. In this Figure g_1, g_2 are sorts and operations from the global signature, n_1, n_2 from the new one.

(b) Using the schematical representation of (a) this Figure illustrates the semantics of the compose construct (line (csp2) of Figure 3)

3.5 The context conditions

The context conditions are defined as a family OK of predicates. This family is defined along the same lines as the semantic functions. A complete description is in Figure 5.

It is important to note that all context conditions are purely syntactic. Hence they may be checked automatically by the development module of an OBSCURE implementation (Figure 9). Moreover, being attached by their very definition to the context-free rules of the language, they may be checked "on-line" during the development of a specification, thus emphasizing the interactive nature of OBSCURE.

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. This restriction stems from the requirement that all context conditions may be checked "on-line".

By the way, a procedure call can not be simulated by "composing" and "renaming" - with global sorts and operations instead of formal parameters and with formal parameters renamed into actual ones.The main reason is that formal parameters are global sorts and operations while renaming only applies to "new" ones.

4. The consistency proof

The goal of this section is to prove that the context conditions of Section 3.5 suffice to guarantee the consistency of the definitions of the semantic functions $\mathfrak{S}$ and $\mathfrak{I}$. The proofs of the Lemmas and Theorems may be found in the Appendix.

OK_{pr} : pr → {true, false}

OK_{pr} (ld csp) = OK_{ld} (ld) ∧ OK_{csp} (csp)($\mathcal{S}_{ld}$(ld)) (pr1)

OK_{ld} : ld → {true, false}

$OK_{ld}(\varepsilon)$ = true (ld1)

OK_{ld} (ld d) = OK_{ld} (ld) ∧ OK_d (d)($\mathcal{S}_{ld}$ (ld)) (ld2)

OK_d : d → environment → {true, false}

OK_d (csp <u>is</u> <u>proc</u> n(lso))(e) = OK_{csp} (csp)(e) ∧ (d1)

(i) lso has no duplicates;

(ii) <u>let</u> $(\Sigma_g,\Sigma_n) = \mathcal{S}_{csp}$ (csp) <u>in</u>
$\mathcal{S}_{lso}$ (lso) $\subseteq \Sigma_g$;

(iii) <u>let</u> $(S,\Omega) = \mathcal{S}_{lso}$ (lso) <u>in</u>
bool ∉ S, <u>true</u> : → *bool* ∉ Ω, <u>false</u> : → *bool* ∉ Ω ;

(iv) e(n) is undefined.

OK_{csp} : csp → environment → {true, false}

OK_{csp} (sp) = OK_{sp} (sp) (csp1)

OK_{csp}(csp <u>compose</u> sp)(e) = OK_{csp}(csp)(e) ∧ OK_{sp}(sp)(e) ∧ (csp2)

<u>let</u> $(\Sigma_{g1},\Sigma_{n1}) = \mathcal{S}_{csp}$ (csp) <u>in</u>

<u>let</u> $(\Sigma_{g2},\Sigma_{n2}) = \mathcal{S}_{sp}$ (sp) <u>in</u>

(i) Σ_{g2}, Σ_{n1} are disjoint ;

(ii) <u>let</u> $(S',\Omega') = \Sigma_{g1} \setminus \Sigma_{n2}$ <u>in</u>
Ω' is S'-sorted;

(iii) Σ_{n1}, Σ_{n2} are disjoint ;

(iv) <u>let</u> $(S,\Omega) = \Sigma_{g1} \cup \Sigma_{g2} \cup \Sigma_{n1} \cup \Sigma_{n2}$ <u>in</u>
Ω is unambiguous.

OK_{sp} : sp → environment → {true, false}

OK_{sp} (<u>create</u> <u>new</u> lso1 <u>model</u> m <u>global</u> lso2)(e) = (sp1)

<u>let</u> $(S_g,\Omega_g) = \mathcal{S}_{lso}$ (lso2) <u>in</u>

<u>let</u> $(S_n,\Omega_n) = \mathcal{S}_{lso}$ (lso1) <u>in</u>

(i) *bool* ∈ S_g , <u>true</u> : → *bool* ∈ Ω_g, <u>false</u> : → *bool* ∈ Ω_g ;

(ii) Ω_g is S_g-sorted ;

(iii) $\mathcal{S}_{lso}$ (lso2), $\mathcal{S}_{lso}$ (lso1) are disjoint ;

(iv) Ω_n is $(S_g \cup S_n)$-sorted ;

(v) $\Omega_g \cup \Omega_n$ is unambiguous ;

(vi) m associates to each sort from S_n a carrier set and to each operation from Ω_n a function (of the corresponding arity).

OK_{sp} (sp subset of s by o)(e) = OK_{sp} (sp)(e) $\wedge$ (sp2)
  let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{sp}$ (sp)(e) in
  let $(S_n, \Omega_n) = \Sigma_n$ in
  (i) $s \in S_n$ and $o \in \Omega_n$;
  (ii) there is a name n such that o = n : s → *bool* ;

OK_{sp} (sp quotient of s by o)(e) = OK_{sp} (sp)(e) $\wedge$ (sp3)
  let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{sp}$ (sp)(e) in
  let $(S_n, \Omega_n) = \Sigma_n$ in
  (i) $s \in S_n$ and $o \in \Omega_n$;
  (ii) there is a name n such that o = n : s × s → *bool* ;

OK_{sp} (sp forget lso)(e) = OK_{sp} (sp)(e) $\wedge$ (sp4)
  let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{sp}$ (sp)(e) in
  (i) let $(S_g, \Omega_g) = \Sigma_g$ in
    let $(S'_n, \Omega'_n) = \Sigma_n - \mathfrak{S}_{lso}$ (lso) in
    Ω'_n is $(S_g \cup S'_n)$-sorted ;
  (ii) $\mathfrak{S}_{lso}(lso) \subseteq \Sigma_n$

OK_{sp} (sp rename lso1 as lso2)(e) = OK_{sp} (sp)(e) $\wedge$ (sp5)
  let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{sp}$ (sp)(e) in
  (i) lso1 has no duplicates ;
  (ii) $\mathfrak{S}_{lso}(lso1) \subseteq \Sigma_n$;
  (iii) lso1, lso2 match ;
  (iv) lso2 has no duplicates ;
  (v) Σ_g, $\mathfrak{S}_{lso}$ (lso2) are disjoint ;
  (vi) $\Sigma_n - \mathfrak{S}_{lso}$ (lso1), $\mathfrak{S}_{lso}$ (lso2) are disjoint ;
  (vii) let $(S_g, \Omega_g) = \Sigma_g$ in
    let $(S'_n, \Omega'_n) = \Sigma_n[lso1/lso2]$ in
    $\Omega_g \cup \Omega'_n$ is unambiguous.

OK_{sp} (sp axioms lf)(e) = OK_{sp} (sp)(e) (sp6)

OK_{sp} ((csp)) = OK_{csp} (csp) (sp7)

OK_{sp} (call n (lso))(e) = (sp8)

(i) e(n) is defined ;

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

let $(\Sigma_g, \Sigma_n) = \mathcal{S}_{csp}$ (csp)(e) in

(ii) lso1, lso match ;

(iii) Σ_g[lso1/lso], Σ_n[lso1/lso] are disjoint ;

(iv) let $(S_n, \Omega_n) = \Sigma_n$ in

for any two operations o, o' $\in \Omega_n$,

o $\neq$ o' implies o[lso/lso'] $\neq$ o'[lso/lso']

(v) let $(S_g, \Omega_g) = \Sigma_g$ in

let $(S_n, \Omega_n) = \Sigma_n$ in

Ω_g[lso1/lso] $\cup$ Ω_n[lso1/lso] is unambiguous.

FIGURE 5: The family of the functions OK expressing the context conditions.

Informally speaking, the first Lemma shows that substitution respects the property of being "sorted".

LEMMA 1: Let lso, lso' be lists of sorts and operations such that lso has no duplicates and lso, lso' match. Let moreover S be a set of sorts and Ω a set of operations. If Ω is S-sorted, then Ω[lso/lso'] is S[lso/lso']-sorted.

The next two Lemmas indicate conditions under which substitution respects algebras and algebra extensions.

LEMMA 2: Let lso, lso' be as in Lemma 1. Let A be a Σ-algebra with $\Sigma = (S,\Omega)$. If

(i) Σ[lso/lso'] is an algebra signature,

(ii) lso' has no duplicates,

(iii) $\Sigma - \mathcal{S}_{lso}$ (lso), $\mathcal{S}_{lso}$ (lso') are disjoint,

then A[lso/lso'] is an algebra.

LEMMA 3: Let E be an algebra extension with extension signature (Σ_g, Σ_n). Let lso, lso' be lists of sorts and operations satisfying the following conditions:

(i) lso has no duplicates ;

(ii) lso, lso' match ;

(iii) (Σ_g, Σ_n)[lso/lso'] is an extension signature ;

(iv) $\mathfrak{S}_{lso}$ (lso) $\subseteq \Sigma_g$;

(v) let $(S_n, \Omega_n) = \Sigma_n$ in
for any two operations o, o' $\in \Omega_n$:
o $\neq$ o' implies o[lso/lso'] $\neq$ o'[lso/lso']

Then E[lso/lso'] is an algebra extension.

Note that in Lemma 2 the condition (i) implies, in particular, that lso does not contain *bool*. Note also that in Lemma 3 lso' may have duplicates.

The next theorem proves that the definitions of the semantic functions $\mathfrak{S}_{sp}$ and $\mathfrak{S}_{csp}$ are consistent. First it is necessary to introduce one more definition.

An environment e is called *correct*, if for each name n, for which the value e(n) is defined, one has:

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

(i) lso has no duplicates ;

(ii) let $(S, \Omega) = \mathfrak{S}_{lso}$ (lso) in
bool $\notin$ S, true : $\to$ *bool* $\notin \Omega$, false : $\to$ *bool* $\notin \Omega$

(iii) $\mathfrak{S}_{csp}$ (csp)(e) is an extension signature

(iv) let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{csp}$ (csp)(e) in
$\mathfrak{S}_{lso}$ (lso) $\subseteq \Sigma_g$

(v) $\mathfrak{I}_{csp}$ (csp)(e) is an algebra extension with extension signature $\mathfrak{S}_{csp}$ (csp)(e) ⊠

THEOREM 1: Let sp, csp be a specification and a composed specification respectively. Let moreover e be a correct environment.

(i) If OK_{sp} (sp)(e) = true, then $\mathfrak{S}_{sp}$ (sp)(e) is an extension signature.

(ii) If OK_{csp} (csp)(e) = true, then $\mathfrak{S}_{csp}$ (csp)(e) is an extension signature.

The following theorem proves that the definition of the semantic functions $\mathfrak{I}_{sp}$ and $\mathfrak{I}_{csp}$ is consistent.

THEOREM 2: Let sp, csp be a specification and a composed specification. Let moreover e be a correct environment.

(i) If OK_{sp} (sp)(e) = true, then $\mathfrak{I}_{sp}$ (sp)(e) is an algebra extension with extension signature $\mathfrak{S}_{sp}$ (sp)(e).

(ii) If OK_{csp} (csp)(e) = true, then $\mathfrak{I}_{csp}$ (csp)(e) is an algebra extension with extension signature $\mathfrak{S}_{csp}$ (csp)(e).

It is now proved that declarations lead to correct environments.

THEOREM 3. Let ld be a list of declarations. If OK_{ld} (ld) = true, then $\mathfrak{S}_{ld}$ (ld) is a correct environment.

The consistency proof of the definitions of the semantic functions $\mathfrak{S}$ and $\mathfrak{I}$ now culminates in the following Theorem.

THEOREM 4. Let pr be a program. If OK_{pr} (pr) = true, then:

(i) $\mathfrak{S}_{pr}$ (pr) is an extension signature;

(ii) $\mathfrak{I}_{pr}$ (pr) is an algebra extension with extension signature $\mathfrak{S}_{pr}$ (pr).

5. Uniting identical subspecifications

5.1 Introduction

The context condition (iii) of (csp2) requires that the new signatures Σ_{n_1} and Σ_{n_2} of the operands of a compose construct be disjoint (see Figure 5). This condition is a stringent one in that it disallows the union of identical data types - as illustrated by the following example. Let s, t, u each denote a sort or operation. Let $A_{s,t}$ denote a procedure call with s and t "new", B_s^t a procedure call with s "new" and t global, and C^t a procedure call with t "new". Then the specification

$A_{s,t}$ <u>compose</u> B_t^s <u>compose</u> C^t (1)

satisfies the context condition mentioned above but the "equivalent" specification

$A_{s,t}$ <u>compose</u> C^t <u>compose</u> (B_t^s <u>compose</u> C^t) (2)

does not. Now, while one may argue that top-down development leads to (1) rather than (2), it seems not reasonable to forbid (2). Hence the context condition mentioned above should be relaxed to allow the union of identical sorts and operations. Note that the same problem occurs for similar reasons in specification languages designed for bottom-up development (see e.g. [Sa 84]).

The idea of the solution to be presented in the next Sections is to attach a "provenance label" to each sort and operation introduced by a procedure call. As sorts and operations introduced by identical procedure calls are themselves identical (syntactically <u>and</u> semantically), such a label should contain the procedure name and the actual parameters. The context condition (iii) of (csp2) may then be relaxed: the new signatures Σ_{n1} and Σ_{n2} may have sorts and operations in common, provided each such sort or operation has the same provenance label. In the formal treatment of the next Sections a family $\mathfrak{P}$ of functions called "provenance function" will attach the labels to the sorts and operations. In an OBSCURE implementation these labels may be stored in a table and automatically updated during the development of the specification.

5.2 The provenance function

The family $\mathfrak{P}$ of provenance functions is defined in Figure 6 along the same lines as the semantic functions. These functions are essentially partial: their value is defined for at most those "new" sorts and operations which result from a procedure call. Informally, line (sp8) of Figure 6 issues the provenance labels. These labels are then

sent through the different constructs. In line (sp4) some labels are removed according to the forget construct. Line (sp5) expresses that renaming also removes labels. Line (csp2) keeps only those labels for which $\Sigma'_g \subseteq \Sigma_g$ and thus implies a genuine weakening of the principle stated in Section 5.1. This weakening which is essential in the consistency proof (Section 5.4) will be discussed in Section 5.5.

Note that the provenance function is defined in Figure 6 as a relation. It will be shown in Section 5.3 that this relation is a function.

5.3 The context conditions

Let OK' be the function obtained by replacing in the definition of OK in Figure 5 the context condition (iii) of (csp2) by:

(iii') <u>let</u> $(S,\Omega) = \Sigma_{n_1} \cap \Sigma_{n_2}$ <u>in</u>
for each $c \in S \cup \Omega$:
there exists p such that
$(c, p) \in \mathfrak{P}_{csp}(csp)(e) \cap \mathfrak{P}_{sp}(sp)(e)$

Informally, this condition expresses that each sort or operations c occurring in both Σ_{n_1} and Σ_{n_2} has the same provenance label p. Note that this condition does <u>not</u> assume that $\mathfrak{P}$ is a function. Instead, this property is proved in the following Lemma.

LEMMA 4: Let sp, csp, e be a specification, a composed specification and a correct environment respectively.
(i) If $OK'_{sp}(sp)(e)$ = true, then $\mathfrak{P}_{sp}(sp)(e)$ is a function.
(ii) If $OK'_{csp}(csp)(e)$ = true, then $\mathfrak{P}_{csp}(csp)(e)$ is a function.

By Lemma 4 it is possible to replace the condition (iii') by:

$\mathfrak{P}_{csp}$: csp → e → so → n × lso

$\mathfrak{P}_{csp}$ (sp) = $\mathfrak{P}_{sp}$ (sp) (csp1)

$\mathfrak{P}_{csp}$ (csp <u>compose</u> sp)(e) = (csp2)

<u>let</u> (Σ_g, Σ_n) = $\mathfrak{S}_{csp}$ (csp <u>compose</u> sp)(e) <u>in</u>

$\{(c,(n,lso)) \in \mathfrak{P}_{csp}(csp)(e) \cup \mathfrak{P}_{sp}(sp)(e) \mid \Sigma'_g \subseteq \Sigma_g$ where $(\Sigma'_g, \Sigma'_n) = \mathfrak{S}_{sp}($<u>call</u> n(lso))(e)}

$\mathfrak{P}_{sp}$: sp → e → so → n × lso

$\mathfrak{P}_{sp}$ (<u>create</u> <u>new</u> lso1 <u>model</u> m <u>global</u> lso2)(e) = ∅ (sp1)

$\mathfrak{P}_{sp}$ (sp <u>subset</u> <u>of</u> s <u>by</u> o)(e)(c) = (sp2)

$\mathfrak{P}_{sp}$ (sp)(e)(c) , if c is a sort different from s, or c is an operation in which s does not occur;
undefined else.

$\mathfrak{P}_{sp}$ (sp <u>quotient</u> <u>of</u> s <u>by</u> o)(e)(c) = (sp3)

$\mathfrak{P}_{sp}$ (sp)(e)(c) , if c is a sort different from s, or c is an operation in which s does not occur;
undefined else.

$\mathfrak{P}_{sp}$ (sp <u>forget</u> lso)(e) = (sp4)

<u>let</u> (Σ_g, Σ_n) = $\mathfrak{S}_{sp}$ (sp)(e) <u>in</u>

<u>let</u> $(S'_n, \Omega'_n) = \Sigma_n - \mathfrak{S}_{lso}$ (lso) <u>in</u>

$\mathfrak{P}_{sp}$ (sp)(e) | $(S'_n \cup \Omega'_n)$

$\mathfrak{P}_{sp}$ (sp <u>rename</u> lso1 <u>as</u> lso2)(e) = (sp5)

<u>let</u> (Σ_g, Σ_n) = $\mathfrak{S}_{sp}$ (sp)(e) <u>in</u>

<u>let</u> $(S'_n, \Omega'_n) = \Sigma_n - \mathfrak{S}_{lso}$ (lso1) <u>in</u>

$\mathfrak{P}_{sp}$ (sp)(e) | $(S'_n \cup \Omega'_n)$

$\mathfrak{P}_{sp}$ (sp <u>axioms</u> lf) = $\mathfrak{P}_{sp}$ (sp) (sp6)

$\mathfrak{P}_{sp}$ ((csp)) = $\mathfrak{P}_{csp}$ (csp) (sp7)

$\mathfrak{P}_{sp}$ (<u>call</u> n(lso))(e) = (sp8)

<u>let</u> (Σ'_g, Σ'_n) = $\mathfrak{S}_{sp}$ (<u>call</u> n(lso))(e) <u>in</u>

<u>let</u> $(S'_n, \Omega'_n) = \Sigma'_n$ <u>in</u>

$\{(c, (n,lso)) \mid c \in S'_n \cup \Omega'_n\}$

<u>FIGURE 6:</u> The family $\mathfrak{P}$ of provenance functions. The syntactic category so is defined by

so ::= s | o ;

the other syntactic categories are those of Figure 1

(iii") $\underline{\text{let}}$ $(S,\Omega) = \Sigma_{n_1} \cap \Sigma_{n_2}$ $\underline{\text{in}}$
for each $c \in S \cup \Omega$:
$\mathfrak{P}_{csp}$ (csp)(e)(c) is defined, and
$\mathfrak{P}_{sp}$ (sp)(e)(c) is defined, and
$\mathfrak{P}_{csp}$ (csp)(e)(c) = $\mathfrak{P}_{sp}$ (sp)(e)(c)

Let OK" be the function obtained by replacing in the definition of OK in Figure 5 the condition (iii) of (csp2) by (iii"). The goal is now to prove Theorem 4 with OK replaced by OK".

5.4 The consistency proof

As indicated in the proof of Theorem 4, Theorem 1 holds with OK replaced by OK" because its proof does not make use of the context condition (iii) of (csp2). The proof of Theorem 2 with OK replaced by OK" requires two properties of the provenance function. They both base on the restriction implied by line (csp2) of Figure 6. The first property is syntactic and is expressed by the following Lemma.

LEMMA 5. Let sp, csp and e be a specification, a composed specification and a correct environment respectively.
(i) Let $(\Sigma_g,\Sigma_n) = \mathfrak{S}_{csp}$ (csp)(e). For each
(c , (n, lso)) $\in \mathfrak{P}_{csp}$ (csp)(e)
one has $\Sigma_g' \subseteq \Sigma_g$ where $(\Sigma_g',\Sigma_n') = \mathfrak{S}_{sp}$ ($\underline{\text{call}}$ n (lso))(e)
(ii) Similar to (i) with sp instead of csp.

The second property essentially expresses that the meaning of the "new" sorts and operations resulting from a procedure call does not change. This property is proved together with a revised version of Theorem 2.

LEMMA 6. Let sp, csp, e be as in Lemma 5.
(i) If OK''_{csp} (csp)(e) = true, then
(1°) for each algebra A such that $\mathfrak{I}_{csp}$ (csp)(e)(A) is defined and for each sort or operation c, name n and list of sorts and operations lso such that
$\mathfrak{P}_{csp}$ (csp)(e)(c) = (n,lso)

it is the case that

$\mathcal{I}_{csp}$ (csp) (e) (A) (c) = $\mathcal{I}_{sp}$(call n(lso)) (e) (A|Σ'_g) (c)

where (Σ'_g, Σ'_n) = $\mathcal{S}_{sp}$ (call n(lso)) (e)

(2°) $\mathcal{I}_{csp}$ (csp) (e) is an algebra extension with extension signature $\mathcal{S}_{csp}$ (csp) (e).

(ii) Similar to (i) with sp instead of csp.

Finally we are able to state the version of Theorem 4 with OK" instead of OK.

THEOREM 5. Let pr be a program. If OK''_{pr} (pr) = true, then:

(i) $\mathcal{S}_{pr}$ (pr) is an extension signature ;

(ii) $\mathcal{I}_{pr}$ (pr) is an algebra extension with extension signature $\mathcal{S}_{pr}$ (pr).

5.5 Discussion

As indicated above the line (csp2) of Figure 6 departs from the principle stated in Section 5.1 in that it keeps only certain labels, namely those for which the global sorts and operations of the procedure call are already global in the specification containing this call. Informally, this restriction is necessary because otherwise these sorts and operations could be dropped or renamed, and subsequently be reintroduced (as "new" sorts and operations) with different semantics. Removing this restriction is in principle possible at the price of a more complex provenance function able to keep track of these droppings and renamings.

It is interesting to note that - contrasting with, for instance, CLEAR [Sa 84] - the solution for uniting identical data types proposed here keeps the syntax and semantics of the specification language unchanged. Instead, it merely relaxes one of the context conditions.

By the way, the reason for introducing the provenance function was to relax one of the context conditions. A similar function

able to keep track of renamings could provide OBSCURE with an interesting aliasing feature - and thus enrich the specification language. It might be interesting to investigate this possibility.

6. Two simple examples

6.1 The data type "sets of elements"

Figure 7 shows a specification of the data type "sets of elements". It contains redundant information which could easily be removed by adding syntactic sugar to the language (cf [Le 85]). The model of the create construct is defined with 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 operations which are declarated to be constructors. Hence, it contains words such as Emptyset and App(Emptyset, 0). The interpretation of the operations which are constructors is the Herbrand interpretation. Hence the value of the operation App for the arguments Emptyset and 0 is the word App(Emptyset, 0). 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 in the logic of strict algebras proposed in [Lo 84]. The forget construct is necessary because the operation App does not satisfy the closure condition implied by the subsequent subset construct. This latter construct 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.

```
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 formulas 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 7: An example of a specification introducing the data type "sets of elements". The global sorts and operations of the extension signature defined by this specification are those listed above under <u>global</u>, the new sorts and operations are those listed under <u>new</u>, except for the operation App : *set el* → *set* (See Section 6.1 for comments).

6.2 The data type "pairs of sets of sets of integers"

Figure 8 introduces the data type "pairs of sets of sets of integers". Line (1) turns the specification of Figure 7 into a procedure. Line (2) to (5) introduce the sort *setofsetofint* and the pertaining operations. According to the top-down development principle this sort is introduced by making use of the global sort *setofint* and the pertaining operations; these are 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 compose construct. In line (6) the specification is sent into the environment as a parameterless procedure. The exact definition of the model in line (7) is dispensed with. The "new" sorts introduced by the program of Figure 9 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 an OBSCURE implementation automatically updates and displays the current global and new sorts and operations.

7. Conclusions

While providing essentially the same expressive power as specification languages such as CLEAR or ACT ONE, the specification language OBSCURE appears to have a very simple formal description. In particular, the use of operational rather than denotational semantics provides a transparent parameterization concept. The union of

```
create                                              ⎤
                                                    ⎥  Figure 7
  :                                                 ⎥
                                                    ⎥
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 setofin → bool) (2)
rename  sorts  set  opns  Emptyset : → set
    as  sorts  setofsetofint  opns  Emptysetofsetofint : → setofsetofint   (3)
compose
    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)
compose
    call  SETOFSETOFINT( )
```

FIGURE 8: An OBSCURE program introducing the data type "pairs of sets of sets of integers" (see Section 6.2 for comments)

identical data types was realized without modifying the language.

OBSCURE differs from the existing specification languages by several features which we believe to be of practical value. First, by interpreting a specification as an algebra extension rather than as an algebra or a theory, the specification language OBSCURE suggests top-down rather than bottom-up development. Second, OBSCURE is designed for interactive, on-line development of specifications. Next, in its design as well as in its implementation OBSCURE is concerned with both the development and the verification of specifications. Finally, OBSCURE allows a mild form of polymorphism on the basis of type inferencing.

Two future developments of OBSCURE are a generalization of the language for loose specifications and the inclusion of a construct expressing implementation (of a data type by another one).

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

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

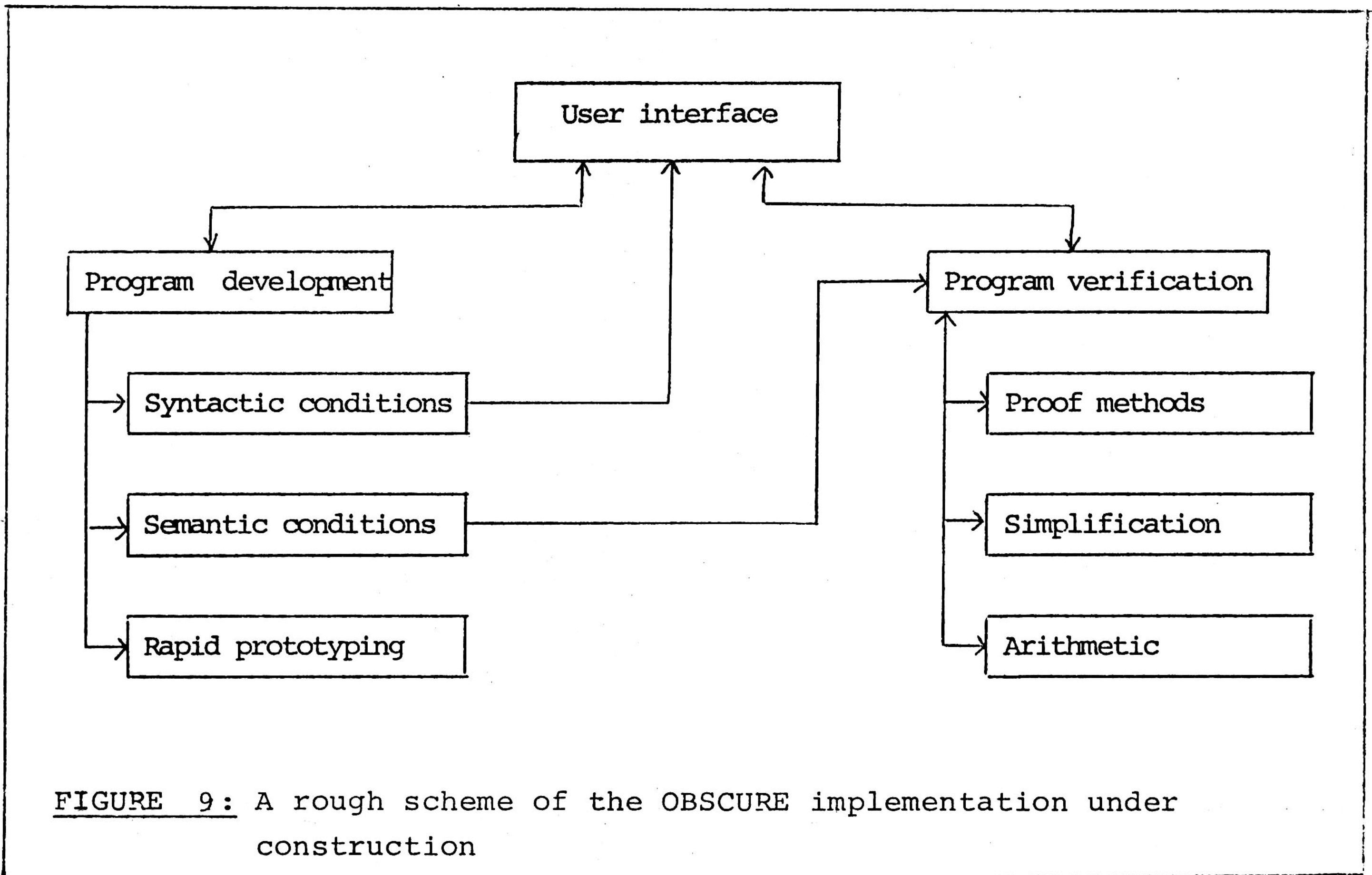

FIGURE 9: A rough scheme of the OBSCURE implementation under construction

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)

[Le 85] Lermen, C.W., The specification language OBSCURE, Int. Rep. A85/11, Univ. Saarbrücken (1985)

[LL 85] Lermen, C.W., Loeckx, J., OBSCURE: A language for interactive top-down specification, Int. Rep. A85/16, Univ. Saarbrücken (November 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. A84/03, Univ. Saarbrücken (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)

[MG 84] Meseguer, J., Goguen, J.A., *Initiality, Induction and Computability* in Nivat, M., Reynolds, J. (Eds.), *Algebraic Methods in Semantics*, Cambridge Univ. Press, 1984

[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 - 172 (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)

APPENDIX

LEMMA 1: Let lso, lso' be lists of sorts and operations such that lso has no duplicates and lso, lso' match. Let moreover S be a set of sorts and Ω a set of operations. If Ω is S-sorted, then Ω[lso/lso'] is S[lso/lso']-sorted.

Proof

Let $\Sigma = (S,\Omega)$.

By definition the elements of Ω[lso/lso'] are of the form o[lso/lso'], $o \in \Omega$.

Consider now an arbitrary element o[lso/lso'] of Ω[lso/lso']. By definition

$$o[lso/lso'] = \begin{cases} o'_i \quad \text{if} \quad o = o_i \quad \text{for some } o_i \text{ of lso} & (1) \\ (o)^{s'_1,\ldots,s'_k}_{s_1,\ldots,s_k} \quad \text{else} & (2) \end{cases}$$

(using the notation of Section 3.3)

In case (2) the lemma directly follows from the definition of S[lso/lso'] and the fact that Ω is S-sorted. In case (1) the lemma follows moreover from the condition (ii) in the definition of matching. ∎

LEMMA 2: Let lso, lso' be as in Lemma 1. Let A be a Σ-algebra with $\Sigma = (S,\Omega)$. If

(i) Σ[lso/lso'] is an algebra signature,

(ii) lso' has no duplicates,

(iii) $\Sigma - \mathfrak{S}_{lso}$ (lso), $\mathfrak{S}_{lso}$ (lso') are disjoint,

then A[lso/lso'] is an algebra.

Proof

It is sufficient to prove that, for any $c, d \in S \cup \Omega$,

c[lso/lso'] = d[lso/lso'] implies c = d

This directly follows from (ii) and (iii). ∎

LEMMA 3: Let E be an algebra extension with extension signature (Σ_g, Σ_n). Let lso, lso' be lists of sorts and operations satisfying the following conditions:

(i) lso has no duplicates ;

(ii) lso, lso' match ;

(iii) (Σ_g, Σ_n)[lso/lso'] is an extension signature ;

(iv) $\mathfrak{S}_{lso}$ (lso) $\subseteq \Sigma_g$;

(v) <u>let</u> $(S_n, \Omega_n) = \Sigma_n$ <u>in</u>
for any two operations o, o' $\in \Omega_n$:
o $\neq$ o' implies o[lso/lso'] $\neq$ o'[lso/lso']

Then E[lso/lso'] is an algebra extension.

Proof

By definition
E[lso/lso'] = {(A[lso/lso'], B[lso/lso']) |
(A,B) $\in$ E, A[lso/lso'] is a Σ_g[lso/lso']-algebra}

It must be proved that:

(a) E[lso/lso'] is a function (rather than merely a relation);

(b) a value of E[lso/lso'] is an extension of its argument;

(c) the values of E[lso/lso'] are algebras .

Ad (a):

Let A, A' $\in Alg_{\Sigma_g}$ be such that A[lso/lso'] = A'[lso/lso']. It must be proved that

(E(A))[lso/lso'] = (E(A'))[lso/lso']

Of course, it is sufficient to prove that A = A'.

Let c be an arbitrary sort or operation from Σ_g. It is sufficient to prove that A(c) = A'(c). By definition of the notation A[lso/lso']

(c[lso/lso'] , A(c)) $\in$ A[lso/lso']

(c[lso/lso'] , A'(c)) $\in$ A'[lso/lso']

As A[lso/lso'] = A'[lso/lso'] , A(c) = A'(c).

Ad (b):

We now prove that

$$A[lso/lso'] \subseteq (E[lso/lso'])(A[lso/lso'])$$

for any $A \in Alg_{\Sigma_g}$.

As E is an extension,

$$A \subseteq E(A)$$

Hence, by definition of the notation [lso/lso'] applied to algebras:

$$A[lso/lso'] \subseteq (E(A))[lso/lso']$$

or, by definition of the notation E[lso/lso']:

$$A[lso/lso'] \subseteq (E[lso/lso'])(A[lso/lso']).$$

Ad (c):

By (b) it is sufficient to prove that

$$(E[lso/lso'])(A[lso/lso'])$$

is a function. By definition of the notation [lso/lso'] applied to algebra extensions:

$$(E[lso/lso'])(A[lso/lso']) = (E(A))[lso/lso'] \qquad (1)$$

and by definition of the notation [lso/lso'] applied to algebra extensions the elements of (E(A))[lso/lso'] are of the form

$$(c[lso/lso'], E(A)(c)) \qquad (2)$$

Hence, let $c,d \in \Sigma_g \cup \Sigma_n$. It is sufficient to show that $c[lso/lso'] = d[lso/lso']$ implies $E(A)(c) = E(A)(d)$. We consider three cases:

(1°) $c,d \in \Sigma_g$

By (b) we have:

$$(E[lso/lso'])(A[lso/lso'])(c[lso/lso']) = (A[lso/lso'])(c[lso/lso'])$$

hence, by (1):

$$((E(A))[lso/lso'])(c[lso/lso']) = (A[lso/lso'])(c[lso/lso'])$$

hence, by (2):

$$E(A)(c) = (A[lso/lso'])(c[lso/lso'])$$

Similarly,

$$E(A)(d) = (A[lso/lso'])(d[lso/lso'])$$

Hence $E(A)(c) = E(A)(d)$. Note that we implicitly used the property that A[lso/lso'] is an algebra.

(2°) $c \in \Sigma_g$, $d \in \Sigma_n$.

In that case c[lso/lso'] = d[lso/lso'] is impossible by the condition (iii) of the Lemma.

(3°) $c, d \in \Sigma_n$.

By the conditions (iv) and (v), c[lso/lso'] = d[lso/lso'] implies c = d. ∎

THEOREM 1: Let sp, csp be a specification and a composed specification respectively. Let moreover e be a correct environment.

(i) If OK_{sp} (sp)(e) = true, then $\mathfrak{S}_{sp}$ (sp)(e) is an extension signature.

(ii) If OK_{csp} (csp)(e) = true, then $\mathfrak{S}_{csp}$ (csp)(e) is an extension signature.

Proof

The values assigned to $\mathfrak{S}_{sp}$ (sp)(e) and $\mathfrak{S}_{csp}$ (csp)(e) by Figure 2 are pairs of signatures. It is necessary to show that the notations used to represent these pairs of signatures (such as ...[lso/lso']) make sense and that these pairs of signatures are extension signatures. Remember that by its definition in Section 2.3 an extension signature is a pair of signatures

$$((S_g, \Omega_g), (S_n, \Omega_n))$$

satisfying the following five properties:

(E1) $\mathit{bool} \in S_g$, <u>true</u> $: \to \mathit{bool} \in \Omega_g$, <u>false</u> $: \to \mathit{bool} \in \Omega_g$;

(E2) Ω_g is S_g-sorted;

(E3) (S_g, Ω_g), (S_n, Ω_n) are disjoint;

(E4) Ω_n is $(S_g \cup S_n)$-sorted;

(E5) $\Omega_g \cup \Omega_n$ is unambiguous.

The parts (i) and (ii) of the Theorem are proved by simultaneous structural induction. The notations used refer to those of Figures 2 and 3. The context conditions referred to are those of Figure 5.

(csp1) The theorem follows from the induction hypothesis. Note in this respect that the induction is on the structure induced by the context-free grammar, <u>not</u> on the length of the string (cf. [LS 84], p. 16, Exercise 1.2-5).

(csp2) Condition (E1) is satisfied because Σ_{g2} satisfies (E1) by induction hypothesis. (E2) follows from the context condition (ii) (and from the induction hypothesis). (E3) follows from the context condition (i) (and, again, from the induction hypothesis). (E4) results from the induction hypothesis. Finally, (E5) results from the context condition (iv).

(sp1) Conditions (E1) to (E5) directly result from the context conditions (i) to (v) respectively.

(sp2), (sp3), (sp6), (sp7) hold by induction hypothesis.

(sp4) The notation $\Sigma_n - \mathcal{S}_{lso}(lso)$ makes sense by the context condition (ii). Conditions (E1) to (E3) and (E5) follow from the induction hypothesis. Condition (E4) follows from the context condition (i).

(sp5) The notation $\Sigma_n[lso1/lso2]$ makes sense because lso1 has no duplicates (see context condition (i)) and lso1, lso2 match (see context condition (iii)). Conditions (E1) and (E2) hold by induction hypothesis, Condition (E3) follows from the context condition (v) (and from the induction hypothesis). As $\Sigma_g = \Sigma_g[lso1/lso2]$ by the context condition (ii) and by induction hypothesis, condition (E4) follows from Lemma 1. Condition (E5) results from the context condition (vii).

(sp8) First note that the value e(n) is defined thanks to the context condition (i). The notation (Σ_g, Σ_n)[lso1/lso] makes sense because lso1 has no duplicates (as e is correct) and lso1, lso match (by the context condition (ii)). Condition (E1) follows from the fact that e is correct: (Σ_g, Σ_n) is an extension signature and *bool*, <u>true</u> : → *bool*, <u>false</u> : → *bool* are not in lso1. Condition (E2) and (E4) follow from Lemma 1. Condition (E3) and (E5) directly follow from the context conditions (iii) and (v). ◻

THEOREM 2: Let sp, csp be a specification and a composed specification. Let moreover e be a correct environment.

(i) If OK_{sp} (sp)(e) = true, then $\mathfrak{I}_{sp}$ (sp)(e) is an algebra extension with extension signature $\mathfrak{S}_{sp}$ (sp)(e).

(ii) If OK_{csp} (csp)(e) = true, then $\mathfrak{I}_{csp}$ (csp)(e) is an algebra extension with extension signature $\mathfrak{S}_{csp}$ (csp)(e).

Proof

As in Theorem 1 it will be necessary to show that the notations used in Figure 3 make sense and that the values assigned to $\mathfrak{I}_{sp}$ (sp)(e) and $\mathfrak{I}_{csp}$ (csp)(e) are algebra extensions. In the case in which these values are defined by E(A), one has to check the following two properties:

(A1) E(A) is a function;

(A2) E(A) $\subseteq$ A

(cf. the definition of an algebra extension in Section 2.3).

The proof follows the same pattern as that of Theorem 1.

(csp1) The proof follows from the induction hypothesis.

(csp2) First, it is necessary to show that the algebras occurring in the expression of $\mathfrak{I}_{csp}$ have the right signature The signature of A is $(\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2}$ and, hence, the notation A | Σ_{g2} makes sense. The signature of the argument of $\mathfrak{I}_{sp}$ (sp)(e) is Σ_{g2} and that of its value is $\Sigma_{g2} \cup \Sigma_{n2}$.

Hence the signature of

$$A \cup \mathfrak{I}_{sp}\ (sp)(e)(A \mid \Sigma_{g2}) = A \cup B$$

is

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

As this signature contains Σ_{g1} the notation

$$(A \cup B) \mid \Sigma_{g1}$$

makes sense. Note that the signature of the value of $\mathfrak{I}_{csp}$ (csp <u>compose</u> sp)(e)(A) is

$$\Sigma_{g1} \cup \Sigma_{g2} \cup \Sigma_{n1} \cup \Sigma_{n2}$$

- as required by Figure 2.

In order to prove Condition (A1) we first prove that

$$A \cup B$$

is a function. By induction hypothesis $B = \mathfrak{I}_{sp}(sp)(e)(A|\Sigma_{g2})$ is. Hence it is sufficient to prove that for each sort or operation c from

$$(\Sigma_{g2} \cup \Sigma_{n2}) \cap ((\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2}) \tag{1}$$

one has

$$B(c) = A(c) \tag{2}$$

As Σ_{g2} and Σ_{n2} are disjoint, (1) becomes

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

i.e. Σ_{g2}

As by induction hypothesis Condition (A2) holds for the left member of (2), (2) becomes

$$A(c) = A(c)$$

By induction hypothesis

$$\mathfrak{I}_{csp}\ (csp)(e)((A \cup B) \mid \Sigma_{g1})$$

and

$$B = \mathfrak{I}_{sp}\ (sp)(e)(A \mid \Sigma_{g2})$$

are functions. Hence, in order to prove Condition (A1) it is sufficient to prove that the union of these two functions is a function, i.e. for each sort or operation c from

$$(\Sigma_{g1} \cup \Sigma_{n1}) \cap (\Sigma_{g2} \cup \Sigma_{n2}) \quad (3)$$

one has

$$\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})(c) = B(c) \quad (4)$$

Because of the context conditions (i) and (iii) one may replace (3) by

$$\Sigma_{g1} \cap (\Sigma_{g2} \cup \Sigma_{n2}) \quad (5)$$

Consider first the case in which c is from $\Sigma_{g1} \cap \Sigma_{g2}$. As by induction hypothesis Condition (A2) holds for each member of (4), (4) becomes

$$(A \cup B)(c) = A(c)$$

i.e. $A(c) = A(c)$

Consider now the case in which c is from $\Sigma_{g1} \cap \Sigma_{n2}$. Again, (4) becomes

$$(A \cup B)(c) = B(c)$$

i.e. $B(c) = B(c)$

Finally, it is necessary to prove Condition (A2). Remember that the signature of A is $(\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2}$. Each sort or operation from $\Sigma_{g1} \setminus \Sigma_{n2}$ belongs to $(A \cup B) \mid \Sigma_{g1}$ and, by induction hypothesis, to

$$\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})$$

Each sort or operation from Σ_{g2} belongs to $A \mid \Sigma_{g2}$ and, by induction hypothesis, to

$$B = \mathfrak{I}_{sp}(sp)(e)(A \mid \Sigma_{g2})$$

(sp1) Condition (A1) results from the context condition (vi). Condition (A2) directly results from the fact that the expression of $\mathfrak{I}_{sp}$ is $A \cup \ldots$.

(sp2) The consistency of the definition follows from the context condition (ii) and from the validity of the closure conditions.Condition (A1) follows from the very definition of a subalgebra. Condition (A2) follows from the context

condition (i) and from the definition of the functions p_t, $t \neq s$.

(sp3) As for (sp2)

(sp4) As $\mathcal{S}_{sp}$ (sp forget lso)(e) = $(\Sigma_g, \Sigma_n - \mathcal{S}_{lso}(lso))$ it is the case that $\Sigma_g \cup (\Sigma_n - \mathcal{S}_{lso}(lso))$ is an algebra signature. As, moreover, the signature of $\mathcal{I}_{sp}$ (sp forget lso)(e)(A) is $\Sigma_g \cup \Sigma_n$ the notation

$$\mathcal{I}_{sp}(sp)(e)(A) \mid (\Sigma_g \cup (\Sigma_n - \mathcal{S}_{lso}(lso)))$$

makes sense. Condition (A1) is verified trivially. Condition (A2) results from the induction hypothesis and the fact that the signature of A is Σ_g.

(sp5) Condition (A1) follows from Lemma 2. That this lemma is applicable results from the following arguments: lso1 has no duplicates and lso1, lso2 match by the context condition (iii); the condition (i) of Lemma 2 results from Theorem 1 (and from the context condition (ii) implying that

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

the condition (ii) of Lemma 2 results from the context condition (iv); the condition (iii) of Lemma 2 results from the context conditions (v) and (vi). Condition (A2) results from the induction hypothesis and from the context condition (v) implying

$$(B[lso/lso']) \mid \Sigma_g = B \mid \Sigma_g$$

(sp6) and (sp7) result from the induction hypothesis.

(sp8) By the context condition (i), e(n) is defined. As e is correct, $\mathcal{I}_{csp}$ (csp)(e) is an algebra extension. The property now follows from Lemma 3. The applicability of this Lemma results from the following arguments. The conditions (i) and (iv) of the Lemma follow from the fact that e is correct. The conditions (ii) and (v) of the Lemma follow from the context conditions (ii) and (iv) respectively. The condition (iii) of the Lemma follows from Theorem 1. ⊠

THEOREM 3. Let ld be a list of declarations. If $OK_{ld}(ld)$ = true, then $\mathfrak{S}_{ld}(ld)$ is a correct environment.

Proof

The proof is by induction on ld.

If ld = ε the theorem holds trivially.

Assume the theorem holds for $\mathfrak{S}_{ld}(ld)$. We now prove that it holds for

$$\mathfrak{S}_{ld}(ld\ d) = \mathfrak{S}_{ld}(ld) \cup \mathfrak{S}_{d}(d)$$

Note first that $\mathfrak{S}_{ld}(ld\ d)$ is an environment by the context condition (iv) and by the induction hypothesis. The conditions (i), (ii) and (iv) in the definition of a correct environment (see Section 4) correspond to the context conditions (i), (iii) and (ii) of (d1) respectively. The conditions (iii) and (v) in the definition of a correct environment hold by Theorem 1 and 2; note that the environment e of these Theorems is $\mathfrak{S}_{ld}(ld)$ and is correct by induction hypothesis. ⊠

THEOREM 4. Let pr be a program. If $OK_{pr}(pr)$ = true, then:

(i) $\mathfrak{S}_{pr}(pr)$ is an extension signature ;

(ii) $\mathfrak{I}_{pr}(pr)$ is an algebra extension with extension signature $\mathfrak{S}_{pr}(pr)$.

Proof

Direct from Theorem 1,2 and 3 ⊠

LEMMA 4. Let sp, csp, e be a specification, a composed specification and a correct environment respectively.

(i) If $OK'_{sp}(sp)(e)$ = true, then $\mathfrak{F}_{sp}(sp)(e)$ is a function.

(ii) If $OK'_{csp}(csp)(e)$ = true, then $\mathfrak{F}_{csp}(csp)(e)$ is a function.

Proof

The proof is again by simultaneous structural induction.

First note that Theorem 1 holds with OK replaced by OK'. The reason is that the proof of Theorem 1 does not make use of the context condition (iii) of (csp2). Hence the expressions $\mathfrak{S}_{sp}(\ldots)(e)$ occurring in Figure 6 make sense (and represent extension signatures).

The proof of the Lemma for the cases (sp1) and (sp8) is trivial.

The proof for the cases (csp1), (sp2), (sp3), (sp4), (sp5), (sp6) and (sp7) directly follows from the induction hypothesis.

The proof for the case (csp2) follows from the condition (iii') and from the induction hypothesis. ∎

LEMMA 5. Let sp, csp and e be a specification, a composed specification and a correct environment respectively.

(i) Let $(\Sigma_g, \Sigma_n) = \mathfrak{S}_{csp}(csp)(e)$. For each $(c, (n, lso)) \in \mathfrak{P}_{csp}(csp)(e)$ one has $\Sigma'_g \subseteq \Sigma_g$ where $(\Sigma'_g, \Sigma'_n) = \mathfrak{S}_{sp}(\underline{\text{call}}\ n\ (lso))(e)$

(ii) Similar to (i) with sp instead of csp.

Proof

The proof is by simultaneous induction as usual.

The cases (csp1), (sp2), (sp3), (sp4), (sp5), (sp6), (sp7)

hold by induction hypothesis.

The cases (csp2) and (sp8) hold by construction.

The case (sp1) holds trivially. ∎

LEMMA 6. Let sp, csp, e be as in Lemma 5.

(i) If OK''_{csp}(csp)(e) = true, then

(1°) for each algebra A such that $\mathfrak{I}_{csp}$(csp)(e)(A) is defined and for each sort or operation c, name n and list of sorts and operations lso such that

$\mathfrak{F}_{csp}$(csp)(e)(c) = (n,lso)

it is the case that

$\mathfrak{I}_{csp}$(csp)(e)(A)(c) = $\mathfrak{I}_{sp}$(call n(lso))(e)(A$|\Sigma'_g$)(c)

where (Σ'_g,Σ'_n) = $\mathfrak{S}_{sp}$(call n(lso))(e)

(2°) $\mathfrak{I}_{csp}$(csp)(e) is an algebra extension with extension signature $\mathfrak{S}_{csp}$(csp)(e).

(ii) Similar to (i) with sp instead of csp.

Proof

The proof is by simultaneous induction of the four properties. Note that the proof of (i)(2°) and (ii)(2°) must only be given for the case (csp2): for the other cases the proof of Theorem 2 does not make use of the context condition (iii) of (csp2) and therefore remains valid. Note also that by Lemma 5 the argument A$|\Sigma'_g$ of $\mathfrak{I}_{sp}$(call n(lso))(e) in the right hand member of the equality in (i)(1°) (and in (ii)(2°)) makes sense.

(csp1) The Lemma follows from the induction hypothesis.

(csp2) Let us first prove (ii)(1°). Assume A, c, n, lso such that

- $\mathfrak{I}_{csp}$(csp compose sp)(e)(A) is defined (I)
- $\mathfrak{F}_{csp}$(csp compose sp)(e)(c) = (n, lso) (II)

It must be proved that

$\mathfrak{I}_{csp}$(csp compose sp)(e)(A)(c) = $\mathfrak{I}_{sp}$(call n(lso))(e)(A$|\Sigma'_g$)(c) (III)

where (Σ'_g,Σ'_n) = $\mathfrak{S}_{sp}$(call n(lso))(e)

From (I) one deduces by definition of $\mathfrak{I}_{csp}$ (see Figure 3):

- $\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})$ is defined (Ia)
- $B = \mathfrak{I}_{sp}(sp)(e)(A \mid \Sigma_{g2})$ is defined (Ib)

From (II) one deduces by definition of $\mathfrak{P}_{csp}$ (see Figure 5):

- either $\mathfrak{P}_{csp}(csp)(e)(c) = (n, lso)$ (IIa)
- or $\mathfrak{P}_{sp}(sp)(e)(c) = (n, lso)$ (IIb)
- or both

From (III) one deduces by definition of $\mathfrak{I}_{csp}$ (see Figure 3) that it is sufficient to prove

- either $\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})(c)$
  $= \mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)(A \mid \Sigma'_g)(c)$ (IIIa)
- or $B(c) = \mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)(A \mid \Sigma'_g)(c)$ (IIIb)

Let us distinguish the cases (IIa) and (IIb). Let us first consider the case (IIb). By (Ib) the induction hypothesis is:

$$B(c) = \mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)(A \mid \Sigma_{g2} \mid \Sigma'_g)(c) \qquad (IV)$$

But by Lemma 5 (and the definition of $\mathfrak{S}_{csp}$) :

$$\Sigma'_g \subseteq (\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2} \qquad (V)$$

hence

$$A \mid \Sigma_{g2} \mid \Sigma'_g = A \mid \Sigma'_g$$

and (IV) reduces to (IIIb). Let us now consider the case (IIa). By (Ia) the induction hypothesis is:

$$\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})(c) = \mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)((A \cup B) \mid \Sigma_{g1} \mid \Sigma'_g)(c) \qquad (VI)$$

Again, by (V)

$$(A \cup B) \mid \Sigma_{g1} \mid \Sigma'_g = (A \cup B) \mid \Sigma'_g \qquad (VII)$$

Now, by assumption A is a $((\Sigma_{g1} \setminus \Sigma_{n2}) \cup \Sigma_{g2})$-algebra. By (V) (and by the induction hypothesis that $A \cup B$ is an algebra):

$$(A \cup B) \mid \Sigma'_g = A \mid \Sigma'_g \qquad (VIII)$$

(VI), (VII) and (VIII) reduce to (IIIa).

Let us now prove (ii)(2°). In the proof of Theorem 2 the context condition (iii) of (csp2) was only used to transform

$$(\Sigma_{g1} \cup \Sigma_{n1}) \cap (\Sigma_{g2} \cup \Sigma_{n2}) \qquad (3)$$

into

$$\Sigma_{g1} \cap (\Sigma_{g2} \cup \Sigma_{n2}) \qquad (5)$$

In the present case (3) can merely be transformed into

$$\Sigma_{g1} \cap (\Sigma_{g2} \cup \Sigma_{n2}) \cup \Sigma_{n1} \cap \Sigma_{n2} \qquad (5')$$

Hence the proof of Theorem 2 carries over provided one also considers the extra case

$$c \in \Sigma_{n1} \cap \Sigma_{n2}$$

For this case it must be proved that

$$\mathfrak{I}_{csp}(csp)(e)((A \cup B) \mid \Sigma_{g1})(c) = B(c) \qquad (IX)$$

By the context condition (iii') of (csp2) (see Section 5.3) and by induction hypothesis (and by the implicit assumption in the proof of Theorem 2 that $\mathfrak{I}_{csp}(\ldots)\ldots$ and $B = \mathfrak{I}_{sp}(\ldots)\ldots$ are defined) one may write (IX) as:

$$\mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)((A \cup B) \mid \Sigma_{g1} \mid \Sigma'_{g})(c) = \mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)(A \mid \Sigma_{g2} \mid \Sigma'_{g})(c) \qquad (X)$$

where $(n, lso) = \mathfrak{P}_{csp}(csp)(e)(c) = \mathfrak{P}_{sp}(sp)(e)(c)$ and where Σ'_{g} is defined as usual. By the same arguments as in the proof of (ii)(1°) both members of (X) may be reduced to

$$\mathfrak{I}_{sp}(\underline{call}\ n(lso))(e)(A \mid \Sigma'_{g})(c)$$

This concludes the proof of the case (csp2).

(sp1) The part (ii)(1°) of the Lemma holds trivially.

(sp2) The part (ii)(1°) of the Lemma follows from the induction hypothesis. In fact, the construction of the subalgebra keeps the carrier sets of the sorts different from s unchanged; a similar remark holds for the operations in which s does not occur.

(sp3) As for (sp2). Note that, strictly speaking, a carrier of a sort different from s is transformed into the equivalence class containing this carrier as its single element. It is no great sin to identify such an equivalence class with its element.

(sp4) through (sp7). Use the induction hypothesis.

(sp8) The proof of (ii)(1^{O}) is direct. ◘

THEOREM 5. Let pr be a program. If OK''_{pr}(pr) = true, then:

(i) $\mathfrak{S}_{pr}$(pr) is an extension signature ;

(ii) $\mathfrak{I}_{pr}$(pr) is an algebra extension with extension signature $\mathfrak{S}_{pr}$(pr).

Proof

The theorem directly results from Theorem 4 (which is also valid with OK" instead of OK), Lemma 5 and Lemma 6.

◘