THE SOL OBJECT-ORIENTED DATABASE LANGUAGE

R. Zicari, F. Cacace, C. Capelli, A. Galipo', A. Pirovano, A. Romboli

Politecnico di Milano
Dipartimento di Elettronica
Piazza Leonardo da Vinci, 32
20123 Milano, Italy

G. Lamperti

TxT Ingegneria Informatica
via Socrate, 41
20128 Milano, Italy

- Work supported by EEC under the ESPRIT-II project 2443 "Stretch"-

Abstract. SOL is a language for databases with tuples, sets, lists, object-identity and multiple inheritance. Other features of SOL are: The existence of a generic type which allows the definition of the schema by step-wise refinements, and the use of null values to express incomplete information in objects. A uniform way of coding both methods and programs is provided through an algebra for objects . The algebra is used both for querying and updating a SOL database. SOL has been defined and implemented as part of the Esprit-II project 2443 "*Stretch*".

1 INTRODUCTION

This paper presents an overview of SOL (Stretch Object-Oriented database Language). The main features of SOL are summarized as follows:

• The SOL data model is a typical object-oriented data model, with *inheritance* hierarchies and object sharing. A novel feature of the SOL data model is existence of a *generic* type which allows the definition of a SOL schema by *step-wise refinements.*

• *Null values* are used to express incomplete information for objects.

• An algebra for objects, called *EREMO,* is used both for coding methods and programs. *EREMO* is used both for expressing SOL queries and updates. In this way, in contrast to other object-oriented database languages, for example the one of O2 [Lecluse et al89], and ORION [Kim89], there is no need to distinguish between the language for implementing methods (in most cases an imperative language) and the language for expressing non-procedural queries (in most cases an SQL-like set-oriented language). The SOL approach solves the "impedance mismatch" problem which still exists, despite all, in many of the proposed object-oriented database languages [BCD89], [CDLR89], [Kim89], [OOP88]. The *EREMO* algebra respects the encapsulation principle and takes advantage of *inheritance* hierarchies and object identifiers.

SOL has been implemented on top of the ALGRES advanced nested-relational system. ALGRES is a powerful rapid prototyping platform which offers an extended nested relational data-model and a language for data definition and manipulation based on an extended algebra for nested relations [Ceri et al.88], [CCLLZ90].

This paper presents the main features of SOL: The data model in Section 2, and the algebra for objects in Section 3. Each of the two sections also includes a comparison with recent similar proposals. The conclusions are reported in section 4. In Appendix we show a simplified version of an application implemented in SOL in the context of the *STRETCH* project.

2. THE SOL MODEL

The SOL data model is similar to that of IQL [Abiteboul90], [Abiteboul Kanellakis 89], LOGRES [Cacace et al 90], and O_2 [Lecluse et al. 89], but with some differences as described in subsections 2.2.4 and 2.5.

2.1 OBJECTS, TYPES, VALUES and CLASSES

SOL entities are **objects** and **values** as in IQL [Abiteboul 90], LOGRES [Cacace et al. 90], and O_2 [Lecluse et al.89]. A SOL entity has a **type**. A type expression is built starting from **elementary types,** and using one of the following **type constructors:** tuple, set, multiset (i.e. a set with duplicates) and list.

Every object is uniquely identified by an object identifier (**oid** in the following). To each oid is associated the value of the object.
A function v maps each oid into a value: $v: O \rightarrow V$
where O is the set of oids and V the set of all values, which will be defined in Section 2.1.2.

The function v defines the SOL instance (Section 2.2.2).

A type in SOL is associated to a *class* . A class defines the structure of a set of objects with the same type. A SOL class declaration contains the class name and its type. These concepts are defined more formally in the rest of this section.

2.1.1 Types

SOL elementary types, denoted as D, are:

$$D = \text{integer} \mid \text{real} \mid \text{string} \mid \text{boolean} \mid \text{text}$$

Let C be the set of class names, L be the set of labels used to name types, O the set of oids.
The type constructors are:

( ): tuple
{ }: set
[ ]: multiset
< >: list

A type expression (or simply **type)** τ is:

$$\tau \rightarrow \emptyset \mid D \mid C \mid (L_1{:}\tau,\ldots,L_k{:}\tau) \mid \{\tau\} \mid [\tau] \mid \langle \tau \rangle$$

where $\emptyset$ denotes the empty type, $D \in D$, $C \in C$ and $L_i \in L$. A **type definition** is defined as L: τ, where $L \in L$.

Each type expression defines a set of values which are compatible with the defined type as follows:

$\text{comp}(\emptyset) = \emptyset$
$\text{comp}(\text{integer}) = \mathbf{I}$
$\text{comp}(\text{real}) = \mathbf{R}$
$\text{comp}(\text{string}) = \mathbf{S}$
$\text{comp}(\text{boolean}) = \{\text{true}, \text{false}\}$
$\text{comp}(\text{text}) = \mathbf{T}$
$\text{comp}(C) = O$
$\text{comp}((L_1{:}\tau,\ldots,L_k{:}\tau)) = \{ (v_1,\ldots,v_k) \mid v_i \in \text{comp}(\tau_i) \}$
$\text{comp}(\{\tau\}) = \{ \{ v_i \} \mid \forall i > 0, v_i \in \text{comp}(\tau) \}$
$\text{comp}([\tau]) = \{ [v_i, n_i] \mid \forall i > 0, n_i > 0, v_i \in \text{comp}(\tau) \}$
$\text{comp}(\langle\tau\rangle) = \{ w \mid w \text{ is a finite sequence of elements } v_i, v_i \text{ comp}(\tau) \}$

where **I** is the set of integers, **R** the set of reals, O the set of oids, etc. Note that the value of a multiset includes the number n_i of occurrences of each element.
A tuple (or set, multiset, list) constructor allows introducing internal labels into the type definitions; for instance if we want the tuple (t_1,t_2) to be labelled N, we write $T = N(t_1,t_2)$. Such a label is not mandatory though. This is explained in the following example.

Example:

The type : (first_name: string,
family_name: string,
age: integer,
date_of_birth: string,
place_of_birth: string)

is equivalent to:

personal_data : (first_name: string,
family_name: string,
age: integer,
date_of_birth: string,
place_of_birth: string)

Example. The following is a class declaration:

```
class PERSON is
        struct (first_name: string,
                family_name: string,
                age: integer,
                date_of_birth: string,
                place_of_birth: string)
end PERSON
```

An object is created with an explicit operator, *new*, which takes a class name as parameter and gives as a result an oid of an object which is included in the class.

Each class has associated the set of oids of the objects of the class. Such a set is called the *class extension* (a more formal definition of a class extension is given in section 2.1.5).

Example We create an object of class PERSON with the following declaration:

#p=new(PERSON)

where #p is a label which contains the oid of the newly created object.

We now define a *subtyping* relationship between two types. *Subtyping* is a feature of the typing discipline [Cardelli 84] [Balsters,Fokkinga 89].

We speak of subtyping when [Balsters Fokkinga89]:

• a partial order exists on types, and from types σ and τ, with $\sigma \leq \tau$ there exists a ("conversion") operation $cv_{\sigma \leq \tau}$ that behaves like a function mapping arguments of type σ into results of type τ .

• an expression e of type σ is allowed to occur at a position where something of type τ is required, provided that $\sigma \leq \tau$ and that the operation $cv_{\sigma \leq \tau}$ is applied (implicitly) to the value of e. We call τ the *supertype* of σ, and σ the *subtype* of τ.

We have extended Cardelli's notion of sybtyping between tuple-types [Cardelli88] to any SOL type, as follows:

We say that τ_1 is a subtype of τ_2, denoted $\tau_1 \leq \tau_2$, if and only if one of the following conditions holds (see also [Lecluse et al. 89]):

1- $\tau_1 \in D \cup C \cup \emptyset$ and $\tau_2 = \tau_1$.
2- $\tau_1, \tau_2 \in C$ and $struct(\tau_1) \leq struct(\tau_2)$.
3- τ_1 is $(L_i: \tau_i)$, $1 \leq i \leq p$, τ_2 is $(L_k: \tau_k)$, $1 \leq k \leq q$, $q \leq p$, $\forall k \exists! i: L_i=L_k, \tau_i \leq \tau_k$.
4- τ_1 is $\{\tau_1'\}$, τ_2 is $\{\tau_2'\}$ and $\tau_1' \leq \tau_2'$.
5- τ_1 is $[\tau_1']$, τ_2 is $[\tau_2']$ and $\tau_1' \leq \tau_2'$.

6- τ_1 is $<\tau_1'>$, τ_2 is $<\tau_2'>$ and $\tau_1' \leq \tau_2'$.

where *struct* is a mapping from *C* to the set of type expressions; *struct* is induced by ν, as explained in the following (Section 2.3.2).

2.1.2 Values

In classical object-oriented languages such as Smalltalk [Goldberg Robson 83], the value encapsulated in an object is always an atom or a tuple of other objects. In object-oriented database systems this value is a tuple or a set of objects. Following the approach of O_2 [Lecluse et al. 89], SOL beside objects provides *values*.

Values are recursively built starting from domains of elementary types using type constructors, as follows:

1- each element of **I**, **R**, **S**, **T**, *O*, {true, false} is a value;
2- ∅ is a value;
3- if $v_1,...,v_k$ are values, $k \geq 0$,
   $(v_1,...,v_k)$, $\{ v_1,...,v_k \}$, $[(v_1,n_1),...,(v_k ,n_k)]$, $< v_1,...,v_k >$ are values;

4- *unk*, *dne*, *open* are values.

The set of all values which can be built in the SOL language is denoted by *V*.

Example: Consider the following two classes:

```
class  PERSON is
struct ( first_name: string,
         family_name: string,
         age: integer,
         date_of_birth: string,
         place_of_birth: string,
         address: ADDRESS )
end PERSON


class ADDRESS is
struct ( city: string,
         street: string,
         number: integer )
```

Suppose we have defined two objects of class PERSON and ADDRESS respectively (object identifiers are written using a #):

#1 : ("John", "Smith", 30, "12-04-60", "London", #2)

#2 : ("Manchester", "Parker", 34)

If one does not want to model ADDRESS as an object, it is possible in SOL to define the class PERSON in a different way, using a so-called *complex attribute*, as follows:

```
class  PERSON is
struct ( first_name: string,
         family_name: string,
         age: integer,
         date_of_birth: string,
         place_of_birth: string,
       address:: (city: string,
                  street: string,
                  number: integer  )
 end PERSON
```

Now we have only one object:

#1 : ("John", "Smith", 30, "12-04-60", "London", ("Manchester", "Parker", 34))

A complex attribute , *address* in the example, has a *structured value*. Structured values can be used every time there is no need to define an independent object.

Among the values allowed for SOL basic attributes, *null* values are permitted. We follow the approach proposed in [Gottlob Zicari88] and define the following types of null values for attributes of basic type: unknown (*unk*), does not exist (*dne*), and *open* . The semantics of such null values is given in [Gottlob Zicari88]. The domain of SOL basic types therefore includes null values (section 2.1.2). Null values are used to express incomplete information for objects [Zicari 90] as the following example shows.

Example: Consider the two class declarations:

```
class LESSON is
struct ( name: string,
         sublessons: < sublesson: LESSON_TREE >,
         lesson_text_list: < page: text >,
        question_list: < question: QUESTION > )
end LESSON_TREE
```

```
Class QUESTION is
struct  < ( question: text ,
            possible_answers: < ( answer: string,  score: integer ) >)  >
end QUESTION
```

and the following objects:

#p1: ("User Interface", { } , <"This is a lesson on the user interface.....">, <#q1,#q2>)

#q1: < ("How do you invoke the user interface?", < ("By clicking the user icon", **unk**), ("Using shut-down", 0) >) , ("How do you return to the main menu?", < ("PF1 key", **unk**) >) >

In the example we have :

- object lesson #p1 does not have sublessons (the corresponding value is the empty set);
- object question #q1 has two unknown scores.

2.1.3 Generic type

SOL allows the definition of a class with a *generic* type [Zicari 90]associated. A *generic* type corresponds to the empty type Ø. The value of an object of type *generic* is*not* defined, and is denoted with ⊥. This corresponds to saying that the value function v is a partial function.A generic type is useful in defining a SOL schema by step-wise refinements, as the following example shows:

Example We create a class DEAN with type generic:

class DEAN is
 struct *generic*
end DEAN

Objects for such class do not have values (we write v (#oid) to denote the value of the object):

#d1 = new(DEAN)
v (#d1) = ⊥

We can refer to a generic class within another class:

```
class UNIVERSITY is
 struct (name: string,
        dean: DEAN )
end UNIVERSITY

#u1 = new(UNIVERSITY)
v (#u1) = ("Politecnico di Milano", #d1)
```

Because the generic type does not have value associated, it respects the inclusion semantics for subtyping. In particular we have $\emptyset \leq \emptyset$, and $\emptyset \leq \tau_i$ for each type in the system. It does *not* hold $\tau_i \leq \emptyset$, if $\tau_i \neq \emptyset$. Therefore, we can have a generic class in an inheritance hierarchy as a subclass of a class with type non generic, but not vice-versa.

Example:

```
class PERSON is
struct ( first_name: string,
        family_name: string,
        age: integer,
        date_of_birth: string,
        place_of_birth: string )
end PERSON

class STUDENT inherits PERSON is
 struct generic
end STUDENT
```

When a generic class is updated to a different type τ then objects of that class get a default value.

2.1.4 Object sharing

Object sharing is used whenever an attribute A in a class C is of type C_1, where C_1 is an element of *C*. The value of the attribute A is the oid of an object of class C_1. An object may be contained into one or more objects, as illustrated by the following example.

Example:

In the following declaration each object of class SYSTEM refers to objects of other classes, namely MATERIAL, CONNECTOR, PROCEDURE and to objects of the same class SYSTEM.

```
class SYSTEM is
struct ( name:string,
        part#: integer,
        serial#: integer,
        date_of_making: string,
        T_min: real,
        T_max:real,
        made_of: { ( material: MATERIAL, quantity: real ) },
        connected_to: { ( system: SYSTEM, connectors: { link: CONNECTOR } ) },
        brand: string,
        model: string,
        subsystems: { system: SYSTEM },
        procedures: { procedure: PROCEDURE } )
end SYSTEM
```

2.1.5 Inheritance

The SOL data model is based on the inheritance relationships among classes.The semantics of inheritance is given using the *subtyping* relationship as follows:

A *class hierarchy* is a triple (C, *struct*, $\angle$), where C is the finite set of class names, *struct* is a mapping from C to types, and $\angle$ is a strict partial ordering on C.

Inheritance (isa relationship) between two classes C_1 and C_2 is expressed in the language by adding the statement C_1 *inherits* C_2. This means that each object of the class C_1 also belongs to the class C_2. C_1 is called a *subclass* of C_2. Conversely C_2 is called a *superclass* of C_1.

An inheritance hierarchy (C, *struct*, $\angle$) is consistent if for any two classes C, C' of C, where C' is a subclass of C, we have *struct* (C') $\leq$ *struct*(C).

For example, a consistent inheritance hierarchy is defined as in the following example.

Example:

```
class PERSON is
struct ( name: string,
         salary : integer,
        friends: {friend: PERSON } )
end PERSON

class MANAGER inherits PERSON is
struct ( name: string,
         salary: integer,
        friends : { friend: STUDENT   } )
end MANAGER

class STUDENT inherits PERSON is
struct ( company: string,
         role: string,
         lessons_attended : < ( lesson: LESSON,  score: integer ) >,
         total_score: integer,
         additional_info: text  )
end STUDENT
```

Note that inherited attributes from a superclass need not to be repeated in the subclass (unless the associated type is different).

So for example, one could re-write class MANAGER in the following equivalent way:

```
class MANAGER inherits PERSON is
 struct ( friends: { friend: STUDENT } )
end MANAGER
```

At the instance level, we model is-a hierarchies by inserting the oid's of sub-classes within the oid's of the superclasses.

The type associated to the class MANAGER is:

```
   τ: ( name: string, salary: integer, friends: {friend:  STUDENT } )
```

The type associated to the class STUDENT is:

```
    τ': ( name:string,
          salary: integer,
          friends:{ friend:PERSON },
          company: string,
          role: string,
          lessons_attended < ( lesson: LESSON, score: integer ) >,
          total_score: integer, additional_info: text )
```

To each class is associated the set of oids of the objects of the class, which is called **class extension.**

A class extension can be defined more precisely as follows[Abiteboul90] :

We define a function π which maps each name in C to a finite set of oid's such that $C \neq C'$ implies $\pi(C) \cap \pi(C') = \emptyset$ (where $C, C' \in C$). We call the set $\pi(C)$ the *local extension* of the class C. If C is an inheritance hierarchy, then we define a *class extension* as the set
$\pi_{in}(C) : \pi_{in}(C) = \cup\{ \pi(C') \mid C' \in C, C' \leq C\}$ (for each C).

Example:

Consider the class PERSON , its subclass STUDENT, and objects:

#p1, #p2, #p3 of class PERSON
#s1, #s2, #s3, #s4 of class STUDENT

we have:

local extension of PERSON ={#p1, #p2, #p3 }
extension of PERSON ={ #p1, #p2, #p3, #s1, #s2, #s3, #s4 }
local extension of STUDENT = { #s1, #s2, #s3, #s4 }
extension of STUDENT = { #s1, #s2, #s3, #s4 }

We assume that STUDENT does not have subclasses.

2.1.6 Multiple Inheritance

In SOL multiple inheritance is allowed, namely, the possibility of declaring a class as a subclass of two or more classes.A special class called OBJECT (see 2.2.3) is always a common ancestor class for each class in the schema. In the language multiple inheritance between a class C_3 (with type τ_3) and two direct superclasses C_1 (with type τ_1) and C_2 (with type τ_2) is expressed as follows: class C_3 *inherits* C_1, C_2
The above is a consistent declaration iff $(\tau_3 \leq \tau_1)$ and $(\tau_3 \leq \tau_2)$.

In the definition of multiple inheritance, name conflicts may occur.For solving name conflicts in multiple inheritance we use the special keyword *from* to rename the label of an attribute.

Example:
class PERSON is
struct (name: string,.....)

class FISH is
struct (name: string,....)

Suppose we define a class MERMAID which inherits from PERSON and FISH. We write:

(i) class MERMAID inherits PERSON, FISH is
struct (name from PERSON.name)

or in alternative, the following are other possible legal definitions :

(ii) class MERMAID inherits PERSON, FISH is
struct (name from FISH.name)

(iii) class MERMAID inherits PERSON, FISH is
struct (p_name from PERSON.name,
f_name from FISH.name)

Note that in declaration (iii) both attributes labelled *name* in classes *Person* and*Fish* are inherited in class *Mermaid* by changing their names.

2.2 SOL DATABASE

A SOL database is composed of a schema and an instance.

2.2.1 SOL Schema

A SOL database is fully described by the ν function, defined in Section 2.1, which associates a value to each oid. Classes are themselves considered as objects, they are defined by the ν function as well, as described in Section 2.2.3.
Given an oid **o**, we will indicate its value as ν(o); if the value is a tuple, we will use a dot notation ν (o).*attr* to denote the value of a particular attribute. Note that each class has its own oid; if **o** is the oid of a class, ν(o) contains (Section 2.2) the class name (denoted ν(o).name), its type (ν(o).struct), its extension (ν(o).ext), and the associated methods (see Section 2.2.3).

A SOL *schema* is a set of classes related to each other by ***inheritance*** relationships and object sharing. In order to describe a correct SOL database, the ν function must satisfy a number of constraints. These constraints can be divided into *schema constraints* and *instance constraints*..

Schema constraints are the following:

1- the ***inheritance*** relationship must be a-cyclic;
2- types associated by ν to each $C \in \mathcal{C}$ must be correct SOL types, according to the definition of Section 2.2.1;
3- class methods must have correct types (see section 2.4);
4- if C_1 *isa* C_2 then it must be $\tau_{C1} \leq \tau_{C2}$;
5- for any pair of oids (o1,o2) corresponding to classes, $\nu(o_1).name \neq \nu(o_2).name$;
6- for any class C different from OBJECT , its type τ_C is not τ_0 (cfr. Section 2.2.3).

2.2.2 SOL instance

A SOL instance defines the objects in the system. Objects belongs to classes. There are some constraints on the SOL instance.
In particular, *instance constraints* for the function ν are the following (we write ν(C) for ν(o), meaning that o is the oid of the class C; ν(C).ext to denote the extension of the class C, ν(C).struct to denote the type of the class C):

1- if C_1 *isa* C_2, then $\nu(C_2).ext \supseteq \nu(C_1).ext$;
2- if $\nu(C_1).ext \cap \nu(C_2).ext. \neq \emptyset$, then (C_1 isa^+ C_2) or (C_2 isa^+ C_1), where isa^+ is the transitive closure of the is-a relationship.
3-**type compatibility**: if $o_1 \in \nu(C_2).ext$, $\nu(o_1)$ must be in comp(τ_3), where τ_3 is a subtype of $\nu(C_2)$.struct;
4- **referential integrity**: if C1 occurs in $\nu(C_2)$.struct, then for any $o' \in \nu(C_2).ext$, $\mathbf{projection}_{C1}$ (ν(o'))$\in$ $\nu(C_1)$.ext ; where **projection** is the usual relational projection operator.

- Condition 1. says that the extension of a subclass is contained in the extension of the superclass;
- Condition 2. says that subclasses of the same class have disjoint extensions in the SOL model, unless they have a common descendant;
- Condition 3. says that the type of an object in a class extension must be subtype of the type of the class;
- Condition 4 defines object sharing.

2.2.3 Metaclasses

SOL is a reflexive language, i.e. each information describing the database (usually called meta-information, or data dictionary) is defined and manipulated within the language. This is obtained with the introduction in the language of a particular type of classes called *meta-classes.*
Meta-classes have been introduced first in object-oriented languages, such as CLOS[Clos87], and in the Smalltalk system [Goldberg Robson 80].

SOL defines eight metaclasses: *OBJECT*, *CLASS*, *CLASS_IN_ISA*, *CLASS_SHARED_BY*, *CONNECTED_CLASS*, *METHOD*, *STRUCTURE*, *SCHEMA*.

The top of every SOL schema is the system class *OBJECT* with associated type τ_0. The class *OBJECT* is the superclass of each class in the schema. By definition we have $\tau_i \le \tau_0$, for each type τ_i defined in the schema. The type τ_0 cannot be used to build user-defined types; by definition we have comp(τ_0)=V, where V is the set of all values which can be built in SOL.

In SOL, a class is considered as an object of the special meta-class **Class**. The **value** of an object of the class *CLASS* is the meta-information corresponding to a class instance , i.e. its name, type, methods (see Section 2.3), the set of classes from which it inherits, its extension (Section 2.3.1).

In particular, the correspondence between the oid of a class and its name is bijective. This is exploited in the EREMO algebra (section 3.) by using class names instead of oids.

2.2.4 Comparison with related approaches

The SOL data model is rather similar to the data models provided by other object-oriented database systems such as IQL [Abiteboul90],O_2 [Lecluse et al.89], Encore [ShawZdonik89] to name a few, but with some differences.

In particular, the data model of IQL allows union and intersection of types while in SOL the equivalent to the union of types is defined only for the top class OBJECT. No intersection of types is provided. IQL and LOGRES both define associations beside classes. SOL does not provide associations. In IQL multiple inheritance is not provided. SOL and LOGRES provide multiple inheritance. However, LOGRES does not have an OBJECT class and therefore constraints multiple inheritance on the existence of a common ancestor class. In SOL no constraints on multiple inheritance are given. Essential features of the SOL data model are the generic type and the possibility of expressing null values, both features are missing in IQL, LOGRES , O_2 and Encore.

Another distinct feature of SOL is the possibility to express the data dictionary in the model through meta-classes. IQL, LOGRES, O_2 and Encore do not support meta-classes.

2.3 METHODS

In SOL object values are manipulated only by methods. A method is just a function which has some typing constraints. A method has a *signature* which defines the type of its input parameter and the type of the output parameter (if any). In SOL, methods are attached to classes and therefore are part of the schema. The definition of a method is done in two steps: first the method signature is given, then its body. The name of the class to which the method is associated can be omitted from the method signature.In such a case, it is implicitly considered when the signature of a method is analyzed.

Example:

```
class PERSON is
  struct (....)
has
  method Get_name () → string is
end PERSON
```

This is equivalent to the following signature:

Get_name (p:PERSON) → string is

Methods are coded using the EREMO algebra (see section 3.). The body of the method is delimited by *begin end*:

```
method body Get_name () → string is
  begin name end
```

Method can be associated to *generic* classes as well.

Example:

```
class MONUMENT is
 struct generic
has
 method Number_of_visitors() → integer
end MONUMENT
method body Number_of_visitors() → integer
 {return a default integer}
```

A class with a generic type may be used in a method signature:

Example: method X (m: MONUMENT)

2.3.1 Method inheritance.

Methods are inherited as well. If a method is associated to a class p, it is inherited by all classes p', such that p' is a (direct or indirect) subclass of p.

2.3.2 Name conflicts

Multiple inheritance may cause name conflicts for methods.We decided to treat method name conflicts in the same way as for name of attributes using a *from* clause (see section 2.2.) as the following example shows:

Example:

Consider the following classes:

Class C is struct (...) has method m ( )

Class C1 is struct (...) has method m( )

Class C3 inherits C, C1

There are three possible legal ways to inherit a method m in C3:

(i) C3 has method m() from C.m ( ) ; (C inherits method m from C).

(ii) C3 has method m() from C1.m ( ) ; (C inherits method m from C1)

(iii) C3 has method m1() from C1.m(), m2() from C.m( ) ;

(C inherits both methods labelled m from C and C1 by re-defining their names).

2.3.3 Method Overloading (rules for consistency)

SOL allows method overloading. It is therefore possible to re-define a method (with same name) in an *is-a* hierarchy. The re-definition of the method must respect a compatibility rule with respect to its signature.

We use the following rule of subtyping among functional types [Cardelli88]:

σ' and σ are types, if $\sigma' \leq \sigma$ and $\tau \leq \tau'$ then $\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'$

Example

The following declarations define a consistent overloading of a method m in an *is-a* hierarchy :

Class C is struct (...)
has method m(c: C)→ (p:C')

Class C' inherits C is struct(...)
has method m(x:C')→(y: C)

SOL methods (and functions see section 2.4) use late binding.

Because methods are associated to classes, we have that a ***class declaration*** also contains for each class C :

- the set of **methods** which can be applied to the objects of the class (this set contains both methods defined locally in C and inherited from the superclasses of C).

2.4 Functions

SOL beside methods allows a set-oriented manipulation of objects by using *functions*. Functions are not attached to classes as methods. They have a signature which defines the domain and codomain of the function; domain and codomain are typed. Objects referred to in a function parameter can only be accessed by using appropriate methods.

Functions are polymorphic with respect to the input parameters; polymorphism is based on the notion of **weak** subtypying as defined in Section 3.2.2.

Functions body are written, as for methods using the EREMO algebra for objects.

Example These are examples of two function declarations:

function **Get_system_component** (Systems: { system: SYSTEM}) → { name: string }

function **Find_material** (materials: {MATERIAL}, m_name: string, m_code: string) → boolean

2.5 Comparison with related approaches

IQL and its extension do not have covariance for method overloading and do not attach methods to classes. The latter is equivalent to SOL functions. O_2 attaches methods to classes. It also imposes a covariance covariance on method overloading and specifies only one method when a name conflict in multiple inheritance occurs. LOGRES does not have methods. A distinct feature of SOL is the existence both of methods and functions. Functions allow the manipulation of class extensions, thus allowing a set-oriented manipulation for objects within the language (see section 3).

3. SOL DATA MANIPULATION LANGUAGE

3.1 EREMO: An algebra for objects

In this section we informally introduce the algebra for objects EREMO. EREMO (Extending Relational Environment for Manipulating Objects) allows the manipulation of objects using a *set-oriented* algebraic approach.

The EREMO algebra respects encapsulation. We use EREMO to write SOL methods, functions and programs. It is however conceptually possible to use a non-encapsulated version of EREMO to write some special type of applications. In the non-encapsulated version of EREMO, attributes of a class structure are seen as particular methods which give the value of the corresponding element of the structure.

In this paper we only consider the algebra which respects the encapsulation.

In particular, EREMO can be used to:

- write method bodies;
- write function bodies;
- write SOL programs;
- perform object and schema updates [Zicari 91];
- write queries .

3.2 EREMO operators

EREMO consists of a complete set of algebraic operators to handle complex values, grouped as follows:

- comparison and membership operators
- set operators
- projection
- selection
- join
- re-structuring operators
- aggregate operators
- conditional operator
- fixpoint operator
- assignment

Algebraic operators consider oids as a particular type of elementary *values* for which special operators are defined .

Operators can be combined to form an algebraic expression, with the usual meaning.

The use of EREMO algebraic operators makes possible to manipulate set of objects at the time, thus allowing a more declarative style of programming than in most conventional object-oriented database systems [Lecluse et al89].

3.2.1 Comparison and membership operators

Comparison operators are the following: $=, >, \geq, <, \leq, \neq$, for basic and structured values and **in** for structured values.There is an overloading in the definition of these operators.

Object identity is obtained as equality of oids. The **in** operator tests the membership of a value to a collection of values of the corresponding type. $>,\geq,<,\leq$, are used in the case of collection to express set inclusion, with the appropriate semantics.

3.2.2 Set operators

Set operators are the following: UNION, DIFFERENCE, INTERSECTION. Their semantics is different in case of sets ,multisets, lists. Set operators exploit a type of polymorphism based on the so called *weak subtyping* defined as follows:
We say that $\tau 1$ is a *weak subtype* of $\tau 2$ (written $\tau 1 << \tau 2$) if one of the following conditions holds:

1- $\tau_1 \in D \cup C \cup \emptyset$ and $\tau_2 = \tau_1$.
2- $\tau_1, \tau_2 \in C$, there exists a τ_3 such that: τ_1 *is-a* τ_3 , and τ_2 *is-a* τ_3 .
3- τ_1 is $(L_i: \tau_i)$, $1 \leq i \leq p$, τ_2 is $(L_k: \tau_k)$, $1 \leq k \leq q$, $q \leq p$, $\forall k\ \exists!\ i$: $L_i=L_k, \tau_i << \tau_k$.
4- τ_1 is $\{\tau_1'\}$, τ_2 is $\{\tau_2'\}$ and $\tau_1' << \tau_2'$.
5- τ_1 is $[\tau_1']$, τ_2 is $[\tau_2']$ and $\tau_1' << \tau_2'$.
6- τ_1 is $<\tau_1'>$, τ_2 is $<\tau_2'>$ and $\tau_1' << \tau_2'$.

Note that the definition of weak subtyping is similar to that of subtyping except for condition (2). Condition (2) says that in case of a set , the operation can be performed between two classes whose type is compatible with that of a common superclass. The result of such operation is a class with the type of the common superclass.

3.2.3 Projection

Projection is the usual operator. The result of a projection is in general a collection having as attributes the specified ones. Projection can be done for attributes which are locally defined in a class, and not indirectly for inherited attributes (if any).

3.2.4 Selection

Selection has the usual meaning.The structure of the result is identical to the structure of the operand.The predicate of a selection may contain an EXIST operator which returns a true value iff there exists at least one element of the collection which satisfies the predicate.
It may also contains an ALL operator which returns true iff for each element the predicate is verified. The quantification level of the select predicate can be nested in case the operand (a collection) contains complex elements.

Example. "Select those students having L5 as next lesson":

SELECT [Next_lesson() = L5] STUDENT.ext()

3.2.5 Join operator

Join is a binary operation defined in the usual way on two collections of the same category :set, multiset, list.

3.2.6 Re-structuring operators

They inlude the usual NEST, UNNEST operators of the nested relations model.

3.2.7 Aggregate operators

Aggregate operators are applied to collections and return a value corresponding to the specified operation.They are:

min, max, average, count.

The general form of an aggregate operator is:

operation [expr] V

where *operation* belongs to one of the above lists, expr is an expression of type compatible with *operation* and V is the operand value (a collection).

Example. "Find the average age of a set of persons":

average [oid.age()] PERSON.ext()

3.2.8 Conditional operator

The conditional operator returns a value depending on the predicates evaluation inside its specification part. The general form is:

COND [**if** p_1 **then** $expr_1$
elsif p_2 **then** $expr_2$
...
elsif p_{n-1} **then** $expr_{n-1}$
otherwise $expr_n$]

where p_i and $expr_i$, i = 1..n, are respectively a predicate and an expression. The list of predicates is evaluated and if a predicate is true, then the corresponding expression is returned as computed value else if none of the predicates is verified, the last expression (corresponding to **otherwise**) is the result. The otherwise branch can be omitted. In this case, the result is the *unknown* values when all predicates are false.

3.2.9 Fixpoint operator

The unary fixpoint operator allows the definition of recursive algebraic expressions.
We show the use of this operator to compute the classical "bill_of_material" problem referred to our ITS example.

Example. "Find all components of a set of systems":

```
FIXPOINT [  subcomponents ←[UNION [ system.subsystems() ],
                                   JOIN [ system = oid ]
                                     systems SYSTEM.ext(),
              subcomponents <= systems,
             UNION systems subcomponents  ]  systems
```

3.2.10 Assignment operator

The assignment operator associates the result of an expression to a value in the following form:

V ← expr

where V is the name of a value while *expr* is a generic algebraic expression. If the value has been declared with a structure definition, the type of *expr* must be compatible with the one of V, otherwise the structure of V is automatically inferred by the one of *expr*. As a particular case, if *expr* returns an object of type class C then V contains the oid of the object. A value V can be assigned many times. This implies that, in case of oids, the association between V and the oid can change.

3.3 SOL Programs

A SOL program consists of three separate units:

- A schema unit:
- An implementation unit
- A query/update unit

The *schema unit* contains the declarations of the schema, i.e. the definitions of the structure of the classes, and the signature of the methods associated to classes and of functions.

The unit is composed of two subsections: one for classes and one for functions.

```
Schema <Schema_name> is
 Class section:
   <class definitions>

 Function section:
   <function definition>

<class definition> : = <class structure>, { <method signature>}

<function definition> : = <function signature>
```

The *implementation unit* contains the body (implementation) of all methods and functions in the schema. It is composed of two sections:A class section, which indicates for each class its associated methods, and a function section. The signature of both methods and functions is also repeated here together with their implementation. The body of both methods and functions is written using EREMO algebraic expressions.

```
Schema <Schema_name> body is
Class section:
  <methods body>
Function section:
  <functions body>

<method body> := <method signature>, <method code>

<function body> := <function signature> , <function code>
```

The *quey/update unit* corresponds to a set of SOL statements. A SOL statement is an invocation of a method or is an EREMO algebraic expression. A method call may contain as a parameter another method call. In SOL there is no distinction between the language for the implementation of methods (functions) and the language for querying and updating of the database . The unifying language is provided by the EREMO algebra.

The following is an example of a query unit .

Example:

Query unit is:

```
def value SeniorStudent is
    { (student:STUDENT,
        name: string,
        age: integer      ) }

SeniorStudent <-  PROJECT [name( ), age( )] self
                  IN (Student.older(21)) and (Student.given_exams("Software Engineering")

DISPLAY [ ] SeniorStudent
```

The result of an EREMO expression can be associated to an identifier (section 3.3.10). This creates a *temporary value*. (SeniorStudent in the example). Differently from classes, values are not encapsulated, they are used for storing results of SOL computation and relationships between objects.

Examples of SOL programs are described in [Zic91b].

4. CONCLUSIONS

We have presented the SOL object-oriented database programming language. The various features of SOL have been described by examples. The SOL language has been implemented on top of the Algres system. The implementation of SOL on Algres meant as a rapid prototype gave us useful insight into the SOL features and provided an important validation to some of the language design decisions. SOL now constitutes one of two languages which compose the multi-paradigm language interface [Zicari Ceri Tanca 91] implemented on top of ALGRES, being the other one a rule-based database programming language called LOGRES [Cacace et al 90].

Acknowledgments

Rolf De By, Herman Balsters, Stefano Ceri, Stefano Crespi-Reghizzi, and Letizia Tanca provided useful comments on earlier drafts of this paper.

REFERENCES

[Abiteboul90] Abiteboul S., "Towards a Deductive Object-Oriented Database Language", Journal of Data and Knowldege Engineering, to appear.

[Abiteboul Kanellakis 89] Abiteboul S., Kanellakis P.C., "Object Identity as a Query Language Primitive", Proc. ACM-SIGMOD, Portland, Oregon, June 1989.

[Balsters Fokkinga 89] Balsters H., Fokkinga M., "Subtyping can have a Simple Semantics", Theoretical Computer Science, to appear.

[Bertino et al. 89] Bertino E., Negri M., Pelagatti G., Sbattella L., "Object-Oriented Query Languages" The Notion and the Issues", Politecnico di Milano, Report no. 89.054, October 1989.

[BCD89] Bancilhon F., Cluet S., Delobel C., "A Query Language for an Object-Oriented Database System", proc. Second Workshop on Database Programming Languages, Salishan, Oregon, June 1989, Morgan Kaufman.

[Cacace 90] Cacace F., "Implementing an Object-Oriented Data Model in Extended Relational Alagebra: Choices and Complexity", Politecnico di Milano, Report no. 90.009, 1990

[Cacace et al 90] Cacace F., S. Ceri, S. Crespi-Reghizzi, L. Tanca, R. Zicari, "Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm", Proc. ACM-SIGMOD, Atlantic City, May 1990.

[Cardelli88] Cardelli L., "A Semantics of Multiple Inheritance", Information and Computation 76, Academic Press, 138-164, 1988.

[Ceri et al 88] Ceri S. et. al, "The ALGRES Project", proc. EDBT 88, Venice, 1988. Springer Verlag, Lecture Notes in Computer Science, no., 1988.

[Zicari Ceri Tanca 91] Zicari R. , Ceri S., Tanca L., "Interoperability between a Rule-based Language and an Object-Oriencnted Database Language", First Int. Workshop on Interoperability in Multidatabse Systems, Kyoto, April, 1991.

[CCLLZ90] Ceri, S., S. Crespi-Reghizzi, G. Lamperti, L.Lavazza, R. Zicari, "ALGRES: A System for the Specification and Prototyping of Complex Applications", IEEE SOFTWARE, July 1990.

[CDLR89] Cluet S., Delobel C., Lecluse C., Richard P., "Reloop, an Algebra Based Query Language for an Object-Oriented Database System", proc. First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, December 1989.

[Clos87] Bobrow et al., "Common Lisp Object System Specifications", X3 87-002, February 1987.

[Goldberg Robson 80] Goldberg A., D. Robson, "Smalltalk80: The Language and its Implementation", Addison Wesley, 1983.

[Gottlob Zicari 88] Gottlob,G., Zicari R., "Closed World Databases Opened through Null Values", proc. VLDB, August 1988, Los Angeles.

[ITS ISIDE] "The User Interface of the ITS ", ISIDE 1133, doc-as-esp-swd-008, May 1989.

[Lecluse et al. 89] Lecluse C., Richard P., Velez F., "O_2, an Object-Oriented Data Model", Proc. ACM-SIGMOD, Chicago,June 1988.

[Kim89] Kim W., "A Model of Queries for Object-Oriented Databases", proc. 15th VLDB, Amsterdam, August, 1989.

[OOP88] Conferencce on Object-Oriented Programming Systems, Languages and Applications", SIGMOD RECORD, (J.Joseph, C. Thomposon, D. Wells eds.), vol.18, no.3, September 1989.

[Shaw Zdonik89]Shaw G.M., Zdonik S.B., "An Object-Oriented Query Algebra", IEEE Data Engineering, September 1989, vol.12, no. 3.

[Zicari 90] Zicari R., "Incomplete Information in Object-Oriented Databases", proc. IFIP Working Conference on Database Semantics(DS-4), July 1990, Windermere.

[Zicari 91] Zicari R., "A Framework for Schema Updates in an Object-oriented Database System", in Proc. IEEE Data Engineering Conf., Japan, 1991.

[Zic91b] Zicari. R, Cacace F., C. Capelli, A. Galipo`, A. Pirovano, A. Romboli, G. Lamperti, The SOL Object-Oriented Database Language, Politecniuco di Milano, Report 91.053, November 1991.

APPENDIX SOL ITS Application

We describe a simplified version of an application developed and implemented in SOL within the ESPRIT-II project *STRETCH*.

A.1 Intelligent Training System : description

We consider a subset of the *STRETCH* Intelligent Training System (ITS) application which involves managing data-components of the hydraulic system of an helicopter. The goal of the ITS application was to define an "intelligent" system which helps a student in learning the various maintenance procedures for specific helicopter components. We use here a considerable smaller subset of such an application.

The schema of the simplified ITS application consists of ten classes, as reported in figure 1. The SOL schema can be logically partitioned in two parts: the first one which stores the description of the various components of the hydraulic sub-system of the helicopter (called *technical database* in the rest) and the second one which stores the structure of the lessons and the student personal data (called *didactic database* in the rest). The technical database is composed of six classes:SYSTEM, MATERIAL, CONNECTOR, PROCEDURE, TROUBLE-SHOOTING, MAINTENANCE. The didactic database is composed of four classes: LESSON_TREE, PERSON, STUDENT, and QUESTION. (We indicate class names with capital letters).

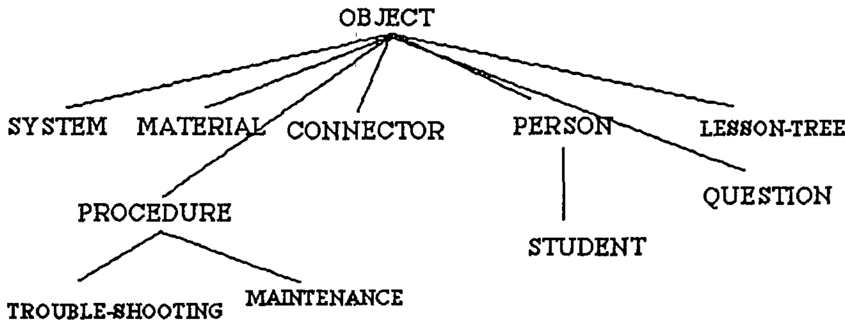

Fig. 1 ITS Schema

A.2 ITS SOL DEFINITION

A SOL program is composed of three units: A schema unit, an Implementation unit, and a Query/Update unit [SOL90]. We present the SOL schema definition for the ITS.

SCHEMA UNIT

We first define the technical database schema.

A class SYSTEM describes the structure of a generic "component" of the hydraulic sub-system. Its associated type is a tuple with twelve attributes. Each attribute is either single-valued or multi-valued. (We recall that types in SOL are either basic types, such as integer, real, or complex types built with the type constructors tuple, set, multiset, and list denoted ( ), { }, [ ], < > respectively, and class names). The attributes of the SYSTEM class are: the system name(*name*), a system part number (*part#*), a serial number (*serial#*), the date in which the component has been produced (*date_of_making*), min. and max. running temperatures of the system (*t_min, t_max*), the materials which constitute the system together with their quantity (*made_of*), the set of other subsystems (*subsystems*) which are connected to the one described through some type of connectors (*connected_to*), the set of procedures which are associated with the described system, the brand (*brand*) and model of the system (*model*). The attribute *made_of* is a set of tuple of two attributes: material of class MATERIAL and quantity. The attribute *connected_to* is a set of tuples of two attributes: *system* of class SYSTEM, and *connectors* which is a set of *link*, where *link* is of class CONNECTOR. The attribute *subsystem* is a set of tuples of one attribute: *system* of class SYSTEM (this is a recursive definition) , and the attribute *procedures* which is a set of tuple of *procedure* of class PROCEDURE. Classes MATERIAL, CONNECTOR and PROCEDURE are in *part_of* relationships with the class SYSTEM.

The corresponding SOL declaration is as follows:

```
Class SYSTEM is
struct ( name: string,
        part#: integer,
        serial#: integer,
         date_of_making: string,
        T_min: real,
        T_max: real,
        made_of: { (material: MATERIAL, quantity: real) },
        connected_to: { ( system:SYSTEM, connectors: { link: CONNECTOR } ) },
        brand: string,
        model: string,
        subsystems: {system: SYSTEM},
        procedures: {procedure: PROCEDURE} )

end SYSTEM
```

Two functions **Get_system_components** and **Select_by_material** are defined:

function **Get_system_components** (systems: { system: SYSTEM }) → { name: string }

The function, given a set of systems, computes all its subsystem components .

function **Select_by_material** (systems: { system: SYSTEM }, name_of_material: string) →{name: string}

The function, given the set of systems and the name of a specific material, finds the name of the systems which are made of the indicated material .

A class MATERIAL contains all the different materials which constitute the hydraulic system. It has a tuple type with three attributes:

```
Class MATERIAL is
struct ( name: string,
        code: string,
        manufacter: string)

end MATERIAL
```

Another function is defined:

function **Find_material** (materials: {MATERIAL}, name: string, code: string) → boolean

This function, given a set of materials and the name and code of a specific material , verifies if the given material is listed. It returns a boolean.

A class CONNECTOR contains all different types of connectors used to link together the subsystems composing the hydraulic system. It has a tuple type with three attributes. The type of one of the attributes is text, as it is used to store text.
The various part of the system are associated to procedures for their ordinary maintenance or when a fault is detected. The class PROCEDURE factors out the common characteristics of a procedure. Classes TROUBLE_SHOOTING and MAINTENANCE both describes special procedures, one invoked when a fault is detected and the other one used for normal maintenance. TROUBLE_SHOOTING and MAINTENANCE are subclasses of the superclass PROCEDURE and inherit the attributes and methods of class PROCEDURE.

```
class CONNECTOR is
struct ( connector_code: string,
        brand : string,
        properties: text )


Class PROCEDURE is
struct ( name: string,
         tools: {tool: string},
        time_required: real,
        ref_manual: text )
 has
```

```
  method Procedure_time() → real  (This method returns the time required to perform a procedure) .

end PROCEDURE

Class TROUBLE_SHOOTING inherits PROCEDURE is
struct ( cause: text,
         description: text ,
        remedy: text )

Class MAINTENANCE inherits PROCEDURE is
struct (case_of_application: text,
        description: text )
```

The didactic database schema is composed of the following classes: Class PERSON factors out the common characteristics of a person: it has a tuple type with five attributes, all of which are simple. Class STUDENT is a subclass of PERSON and describes the information associated to an ITS student. It has a tuple type of five attributes. In particular the attribute "lessons_attended" is a list of tuples, where each tuple contains two attributes: lesson of type class LESSON_TREE and score obtained in that lesson. The *part_of* relationships here is cyclic. Class LESSON_TREE defines the set of lessons for the application with a tree-structure. Class LESSON_TREE has four attributes, the name of the lesson, the set of sublessons composing the lesson, the list of pages containing the text of the lesson, and a question_list. Each element of the question_list is of class QUESTION. Class QUESTION is a list of tuples, each of which has two attributes: a question formulated to the student, and a list of possible answers each one with an associated score. The SOL declarations are as follows:

```
Class  PERSON is
struct ( first_name: string,
          family_name: string,
         age: integer,
          date_of_birth: string,
          place_of_birth: string )
end PERSON


Class STUDENT inherits PERSON is
struct ( company: string,
         role: string,
        course: LESSON_TREE,
         lessons_attended : < (lesson: LESSON_TREE, score: integer) >,
         total_score: integer,
         additional_info: text )
  has

    method Assign_total_score  (score: integer)   (This method assigns a  given score to the
                                                           attribute total_score )


     method Last_lesson () --> lesson_name: string  (This method returns the name of the last
                                                            lesson attended by the student )

     method  Next_lesson () --> string  (This is a method which, given a student and the name
                                                     of its last lesson attended, computes the next lesson
                                                     the student should attend.)
end STUDENT


Class LESSON_TREE is
struct ( name: string;
         sublessons: < sublesson: LESSON_TREE >,
          lesson_text_list: < page: text >,
        question_list: < question: QUESTION >  )

  has

Method Lesson_name () --> string  (This method returns the name of the lesson ).

end LESSON_TREE
```

```
Class QUESTION is
struct  < (question: text,
          possible_answers: < (answer: string, score: integer) >)  >

 has

Method  Average_score ( )
```

(This is an aggregate method which computes the average of the answers given by the student for each question, and stores the result in the attribute *total score* of the object of class STUDENT) .

```
end QUESTION
```

We now define the body of methods and functions defined in the schema. *EREMO* is used to code both bodies of methods and of functions.The same algebra is also used to code SOL programs, as described later.

A.2.2 IMPLEMENTATION UNIT

```
function body Get_system_components (systems: {system: SYSTEM})→{name: string}
begin
  PROJECT [ system.name() ]
  FIXPOINT [  subsys := AGGREGATE [ UNION / system.subsystems() ]
                        JOIN [ system= oid ]
                         systems SYSTEM.ext(),
              subsys <= systems,
              UNION systems subsys ] systems
end
```

This function computes the classical bill-of-material problem. It uses a fixpoint operator.

```
function body
Select_by_material (systems: {system: SYSTEM}, name_of_material: string )
 → {name: string}
  begin
  PROJECT [  system.name()  ]
  SELECT [ EXIST [ material.name() = name_of_material ] made_of ] systems
 end

function body
Find_material (materials: {MATERIAL} , m_name: string, m_code: string )
→ boolean
 begin   EXIST [ m_name = name AND mcode = code ] materials   end


method body Procedure_time  () → real
  begin time_required  end


method body Assign_total_score  (score: integer)
  begin   total_score <- score  end


method body Last_lesson () → string
  begin    LAST (lessons_attended).lesson.Lesson_name()  end


method body Next_lesson () → string
  begin
  COND [ if total_score < DISCRIMINATOR then
            COND [ if Last_lesson().Child_lesson() = dne  then
                      COND [ if Last_lesson().Father_lesson().Left_brother() = dne  then
                               "failure"
                             otherwise
                                  Last_lesson().Father_lesson().Left_brother()
```

```
                otherwise
                    Last_lesson().Child_lesson() ]
        otherwise
            COND [ if Last_lesson().Right_brother() = dne then
                    COND [ if Father_lesson().Right_brother() = dne then
                            "end of lesson tree"
                        otherwise
                            Father_lesson().Right_brother() ]
                otherwise
                    Last_lesson().Right_brother() ] ]
 end
```

```
method body Lesson_name () → string
 begin  name end
```

```
method body Average_score ( )
  begin  Assign_total_score( AGGREGATE [ average / score ] lessons_attended ) end
```

SOL PROGRAMS

We report now two examples of SOL programs working on the ITS schema.

Program 1

The following program "check if a given material (name and code) is used in the helicopter. It then find all systems which use such a material and display them". The program does the following:

- Declare a tuple value corresponding to name and code of a material
- Read the material instance
- Control if material is included into the extension of class MATERIAL and assign the boolean result to the variable *is_material* (whose structure is defined implicitly by the right part of the assignment) .
- The result of the COND factor depends on the specified predicate. If true (not a material) then the result is the empty set, else it is the returned value of the Select_by_material method.
- Declare a string value corresponding to the name of a system
- Read the instance of the value
- Select the systems corresponding to the system_name
- Display all the system subcomponents for the found systems

SOL Program 1

```
value material is ( name: string, code: string )

READ [ material ]

is_material <- Find_material (MATERIAL.ext(), material.name, material.code)

user_systems <- COND [ if not is_material then {}
                      otherwise
                         Select_by_material (SYSTEM.ext(), comp.name, comp.code) ]

DISPLAY user_systems

value system_name is string

READ [ system_name ]

systems <- Find_systems_by_name(SYSTEM.ext(), system_name)

DISPLAY Get_system_components ( systems )

END SOL Program 1
```

SOL PROGRAM 2

The following program reads the name of a student, then selects all students with the given name. For each of such students, it computes the next lesson the student must attend and display it.

```
** Declare a string value corresponding to a student name **

value student_name is string

** Read the instance of the value **

READ [ student_name ]

** Select all students with the specified name and assign the result to a new value **

students <- Find_students_by_name(STUDENT.ext(), student_name)

** Display the last and next lesson for each found student **

DISPLAY PROJECT [ Last_lesson(), Next_Lesson() ] student
```