GSBL: An Algebraic Specification Language Based on Inheritance

S. Clerici & F. Orejas
Facultat d'Informàtica, Universitat Politècnica de Catalunya
Pau Gargallo 5, (08028) Barcelona, SPAIN

Abstract

At the specification level, inheritance can be defined as subtyping by means of order sorted specifications [GM85]. Subtyping is, obviously, a very important notion, allowing not only to work with a non rigid type structure, but also providing an adequate basis for error handling in algebraic specifications. However, in our opinion, subtyping and order sorted specifications do not play the same rôle as inheritance in program design. In this paper, we will present a hierarchical organization for specifications, based on a different concept of inheritance which, we think, corresponds, methodologically, to the usual inheritance relation defined at the programming level. This new relation allows to work with *incomplete* specifications with several levels of detail and, as a side-effect, it may play the rôle of genericity. The use of this notion of inheritance is shown by means of the GSBL specification language built around this new concept, whose use and formal semantics are sketched.

Key words and phrases: Algebraic specification, inheritance, genericity, specification languages.

1. Introduction

Inheritance in object oriented programming languages can be defined as a "specialization" relation among classes. This specialization relation may take different forms. Conceptually, inheritance may be seen as subtyping. For instance the class of natural numbers may be considered to be a subclass of the integers. Indeed, the formal semantics of inheritance has been given in terms of subtyping [CAR84].

However, from a methodological standpoint, inheritance in program design is more than subtyping. On one hand, implementation (in the sense given in the abstract data type literature [EKMP82]) relations are usually considered as inheritance. For instance, in [MEY87] binary search trees are considered a subclass of tables. On the other hand, inheritance plays a rôle very similar to genericity in languages like Ada. The definition of a class of *lists* having as a client a class of *values* may be seen as equivalent [MEY86] to the definition of a generic package of lists in Ada. The instantiation of generic parameters would cause the same effect as defining lists over any subclass of values.

These two aspects make of inheritance a major design concept, since it provides the basis for both top-down, by deferring the description of some aspects of a superclass that will later be refined in a subclass, and bottom-up object-oriented design, by facilitating the reuse of objects through the hierarchical organization provided by the inheritance and client relations.

At the specification level, inheritance can be defined as subtyping by means of order sorted specifications

[GM85]. Subtyping is a very important notion, allowing not only to work with a non rigid type structure, but also providing an adequate basis for error handling in algebraic specifications. However, in our opinion, subtyping and order sorted specifications do not play the same rôle as inheritance in program design. In this paper, we present a hierarchical organization for specifications, based on a different concept of inheritance which, methodologically, corresponds better to the usual inheritance relation defined at the programming level.

Specification design is a process consisting in building a formal and complete description of a problem (and, possibly, of its intended solution) from an informal, often incomplete and sometimes contradictory set of requirements. Therefore, during this process, the specifier, by interacting with the customer, would have to detect the possible contradictions and to add the necessary detail to make the final specification complete.

As a consequence, specification design cannot be seen as a "linear" process as it is, simplistically, often seen, i.e. a process in which, at every time we have a full description of part of the problem that we, continuously, enlarge until the full problem is specified. Rather, when designing a specification, at any moment we have to deal with specifications describing only partially some aspects of the problem. Also, in subsequent steps we may refine these specifications by *completing* them, or we may have to backtrack and redo part of the work.

Therefore, a language aimed at giving support to the specification design process, especially for building large specifications, should offer the possibility of dealing with incomplete specifications, describing partial aspects of a problem, and of refining these specifications by completing them. Also, it would have to facilitate the design of modular specifications and the organization of its components in a hierarchy reflecting the design process, in order to enhance comprehensibility, to simplify the modification of some of its components, to facilitate its reuse and to assure the correctness of the whole specification if every component is correct.

Algebraic specification languages cover, in some sense, most of these aspects. Modularity is inherent to such languages, built over the abstract data type concept. Several forms of hierarchical organization, based on a notion of refinement, have been proposed. Broy and Wirsing [WPPDB83] based hierarchical specifications over the concept of enrichment or extension. Burstall and Goguen [GB80] proposed a hierarchical organization based on a two-dimensional structure: horizontal refinements correspond, also, to enrichments, while vertical refinements correspond to implementations.

The possibility of dealing with "incomplete" specifications, although not explicitly stated, is also present in most specification languages. However, this is not quite true for the hierarchical organization associated to the refinement relation defined by adding detail, i.e. completing a specification. In our opinion, according to the design process we foresee, the kind of hierarchical organization needed for dealing with large specifications is precisely this one, together with the usual one based on the extension concept, that may be seen as playing the same rôle as the *client* relation in object oriented programming languages [MEY87].

Incomplete specifications may be seen, within the algebraic approach, as specifications with loose semantics, i.e. the class of models defined by the given specification is not an isomorphy class. In this sense, "complete" specifications would have an isomorphic semantics (for instance, initial) and completing (refining) a specification would mean restricting its associated class of models.

Within an incomplete specification some parts may be considered to be completely defined (for instance, the booleans may be considered fully defined within a larger and incomplete specification). The semantics of specifications in which some parts are completely defined may be stated in terms of *constraints* [REI80, BG80, SAN81, EWT82].

As it happens at the programming level, genericity is embedded within our inheritance notion. Parameterized specifications may be seen as incomplete specifications with a completely defined part. For instance, *Lists_of(values)* can be seen as a specification in which the parameter, *values*, is only sketched,

while *lists* (once a given set of values is given) is completely defined. In this sense, parameter passing may be seen as a refinement of the parameterized specification, since the result is "more complete".

However, the converse is not true. The use of parameterized specifications for dealing with incomplete specifications is limited. Specifically, we could have problems if we would want to write a specification of *Lists_of(values)* with an operation *choose*, that we have not decided yet which element out of a list must select: since this operation needs to be considered incompletely defined, it would have to be in the parameter specification. However, this would be technically impossible, since the resulting signature of the parameter specification would be syntactically incorrect.

In the rest of the paper, we will introduce a specification language built around this notion of inheritance, and show some small examples of its use. Also, all this concepts will be made more precise by sketching the formal semantics of the language.

2. Overview of GSBL

GSBL is an algebraic specification language based on the ideas expressed above. That is, in GSBL specifications may be seen as having some sorts and operations not completely defined. From an object-oriented point of view, these sorts and operations correspond to the generic attributes of the object, and the sorts and operations completely defined to the fixed ones.

Then new objects may be defined from old ones in two ways: by extending previously defined specifications, then we say that the new specification is defined **over** the old ones, and by consistently redefining some "incomplete" part of an old specification, then we say that the new specification is a **subclass** of the old one, since we may consider the former a specialization of the latter. We consider, then, the objects of GSBL hierarchically organized by this two relations. Moreover, if a class A is defined over a class B and C is a subclass of B, then a new class D, obtained by substituting, in A, B by C, is considered implicitly defined. D is considered to be over C and subclass of A.

Being more specific, the language mechanism for creating a new object is the **class definition** that has the following scheme (all the clauses are optional).

```
CLASS Class name

   OVER  < overlist >                              --over clause

   SUBCLASS-OF  < subclasslist >                   --subclass clause

   WITH     SORTS  < sortlist >                      --with clause
            OPS  < opslist >
            EQS  < varlist > < equationlist >

   DEFINE   SORTS  < sortlist >                      --define clause
            OPS  < opslist >
            EQS  < varlist > < equationlist >
END-CLASS
```

For instance, the following specification describes any ordered set of values.

```
CLASS With-order
OVER Boolean
WITH  SORTS With-order
      OPS
          _ = _ , _ ≤ _ : With-order  ×  With-order  →  Boolean
```

```
    EQS   { a, b, c: With-order }
          a  == a = true
          a  == b ∨ ¬( a  == b ) = true
          a  == b  ⇒ b  == a  = true
          (a ≤  b )  ∨  ( b ≤  a ) = true
          ( a ≤ b  ∧  b ≤ c )  ⇒ a ≤ c  = true
          (a ≤  b )  ∧ (b ≤  a ) =  (a  == b)

DEFINE  OPS   _≥_ , _<_ , _ > _   : With-order  ×  With-order  →  Boolean
        EQS   { a, b :  With-order }
              a  ≥  b = b  ≤   a
              a  >  b = ¬( a  ≤  b )
              a  <  b  =  b  >  a
END-CLASS
```

The *over* clause **imports** the Boolean specification and establishes an **over** relation between the two specifications. The *with* clause declares the new operations and sorts, introduced by the specification, which are not completely defined. Finally, the *define* clause presents the completely defined sorts and operations.

Hence, the specification for the class *With-order* (in addition to the Boolean subspecification) will have, then, a new sort, with its very name, that is considered to be not completely defined. Also, the specification introduces two new infix operations $==$ *and* $\leq$ with equations that express several properties such us symmetry, transitivity, reflexivity, but, again, without a complete definition. On the contrary the new operations $\geq$, $<$ *and* $>$ are considered to be fully defined by the equations. Of course, this definition may be seen as depending from the precise definition of $==$ *and* $\leq$.

Intuitively, we may consider the signature Σ of the specification SP for a class A as being the union of two pairs $W = \langle S_w, Op_w \rangle$ and $D = \langle S_d, Op_d \rangle$ corresponding to the **non completely defined** and **completely defined** sorts and operations of A, respectively. In the example, we have that *Boolean* is a basic class, with all its sorts and operations completely defined. Then, the whole resulting signature of the *With-Order* specification is $\Sigma_{With\text{-}order} = W \cup D$ where

$$W = \langle \{With\text{-}order\}, \{==, \leq\} \rangle \quad \text{and} \quad D = \Sigma_{Boolean} \cup \langle \emptyset, \{\geq, <, >\} \rangle$$

Note that neither W nor D need to be signatures, because the arity of their operations may involve sorts belonging only to the other part.

Formally, the completely defined parts are **constraints** [REI80, BG80, SAN81, EWT82]. A constraint, as we will see in the following section, may be seen as the complete definition of some sorts and operations in terms of others. In the example, the *With-Order* specification defines two constraints: the first one is the definition of booleans and the second one is the definition of $\geq$, $<$ *and* $>$. That is, every *define* clause establishes a constraint within the specification defining the sorts and operations, declared in the clause, in terms of the rest of the specification.

For example, *Natural* can be defined as an instance of *With-order* using the *subclass* clause as follows.

```
CLASS Natural
  SUBCLASS-OF With-order
  DEFINE  SORTS Natural
          OPS   ==, ≤
                0 : →Natural
                succ : Natural →Natural
          EQS   { a, b : Natural }
                0  == succ(a) = false
                succ(a)  == succ(b)  =  a  ==b
                0 ≤ succ(a) = true
                succ(a) ≤ succ(b)  =  a ≤ b
END-CLASS
```

Natural has been constructed as subclass of *With-order*, the first one **inherites** the whole specification of the second one, with the **implicit renaming** of *With-order* by *Natural*. Hence, note that, to write the new specification, it has only been necessary to declare which sorts and operations, from the W part of the superclass, are now completely defined (namely *Natural*, $=$ and $\leq$), and to add the necessary equations and additional operations (*0* and *succ*). That is, *Natural* has refined the specification *With-order* by completing it. Let us observe that, in the resulting specification for *Natural*, the W part will be empty, since all sorts and operations are now completely defined.

Within a specification, it is reasonable to think that the completely defined parts should be "protected" with respect to the rest of the specification, i.e. a specification is considered correct if the whole specification is *consistent* with respect to the constraints and if every constraint is *sufficiently complete* with respect to the other constraints. For instance, we should not allow an equation in the previous specification stating *true* = *false*, nor a completely defined term generating a *junk* value (not equivalent to *true* or *false*) of sort bool. Otherwise, probably, the set of models of the specification would be empty.

As a consequence, we ask all equations introduced by a new specification, either in the *with* or *define* clauses to be consistent with previous constraints, either coming from an *over* or a *subclass* clause. Also the operations introduced in the *define* clause should not introduce junk on any previous constraint. Finally, for methodological reasons, we also ask for consistency of new equations with respect to any subspecification declared in the *over* clause, being completely defined or not, since, otherwise, the **over** relation would not be a true extension relation. This means that our specifications extend the specifications declared in the *over* clause with a mixture of the *protecting* and *extending* enrichments of OBJ2.

In the previous examples, the *over* and the *subclass* clause have been used as a shorthand for writing specifications. However, as it has been already mentioned, this is not their only rôle. In fact, we can declare an already existing specification as subclass of another one just to establish this connection between them. Then, in this case, subclass declaration is similar to a **view** declaration in OBJ2. Indeed, this clauses are also the declarations that serve to define the **over** and **subclass** relations which are the basis for the hierarchical organization of specifications advocated in this paper. For instance, in the example, *Natural* is **subclass-related** to *With-order* and **over-related** to *Boolean.*

A class may be **subclass-related** to several other classes, i.e. **subclass** is a multiple inheritance relation. Furthermore, a class may be **subclass-related** to another one in more than one way. For example, we may want to define two different ways for ordering sequences of naturals, say comparing only the first elements, or perhaps comparing the length. We may construct this new class in the following way:

CLASS Natseq-with-order
SUBCLASS-OF Natsequence,
With-order [equfirst: $=$; l.e.first: $\leq$; l.first: $<$; g.e.first: $\geq$; g.first: $>$] ,
With-order [equleng: $=$; l.e.leng: $\leq$; l.leng: $<$; g.e.leng: $\geq$; g.leng: $>$]
.....

The rest of the specification would include the equations (and, possibly, the new operations) needed to complete the definition of the inherited and renamed (by means of the renaming indicated inside the square brackets) operations.

On the other hand, two objects may be **over** and **subclass-related** at a time. For example, we may want to specify how to induce an order relation on a set by means of a function from this set to an already ordered one:

```
CLASS Orderable
  OVER Inductor : With-order
  SUBCLASS-OF With-order
  WITH OPS
       f : Orderable → Inductor
  DEFINE OPS   =, ≤
  EQS  { a, b : Orderable }
       a  = b = f( a ) = f( b )
       a ≤ b = f( a ) ≤ f( b )
END-CLASS
```

It may be noted that in the resulting specification some operations are overloaded, namely $=, \leq, \geq, <$ *and* $>$ for *Inductor* and for *Orderable*, but, because of their arity, no confusion arises. If it does, a renaming for the *Orderable* operations would have to be made. In fact, *Orderable* has been declared over *Inductor:With-order* and not just over *With-order* to avoid name confusion.

The two relations are transitive, therefore if we would have had *Orderable* already defined, (and *Natstequence* had the operation *length*) we may have had constructed *Natseq-with-order* as follows

```
CLASS Natseq-with-order
 SUBCLASS-OF Natsequence,
   Orderable  [ Natural : Inductor; first: f; equfirst: = ; l.e.first: ≤ ; l.first: < ; g.e.first: ≥ ; g.first: > ] ,
   Orderable  [ Natural : Inductor; length: f; equleng: =; l.e.leng: ≤ ; l.leng: < ; g.e.leng: ≥ ; g.leng: > ]
END-CLASS
```

Then, because of the transitivity of the subclass relation, *Natseq-with-order* will become a subclass of *With-order*, and thus it will be possible to define a new specification in which *Natseq-with-order* plays the rôle of *Inductor*, and so on.

In GSBL genericity is a consequence of the effects of the combination of the over and the subclass relations. As it was already said, if a class A is defined over a class B and C is a subclass of B, then a new class D, obtained by substituting, in A, B by C, is considered implicitly defined. Then, D is considered to be over C and subclass of A. This causes the same effect as parameter passing in usual algebraic specification languages. For instance, if a class *Ordered_sequences* is defined over *With-order*, this would, implicitly, cause the definition of a class of ordered sequences of naturals over *Naturals*.

In what follows, and to end this section, we will see a small example of generic programming [GOG84] using GSBL. Let us first consider a very frequent problem in programming, finding the maximum of a structure of elements having an order relation defined over them. Independently of the kind of structure, we only need a way for traversing it and examining its elements. Then, we first provide a specification of *Traversable* structures.

```
CLASS Traversable
  OVER   Boolean
         Element : ANY
  WITH   SORTS  Traversable
         OPS    first :  Traversable  →  Element
                rest :   Traversable  →  Traversable
                end? :   Traversable  →  Boolean
END-CLASS
```

where *ANY* is a predefined class whose specification is

```
CLASS ANY
  WITH SORTS ANY
END-CLASS
```

that may be seen as a superclass of any specification. For example the usual *stack of elements*, may be viewed as an instance of Traversable as follows:

Stack SUBCLASS-OF Traversable [top: first ; pop: rest ; emptystack? : end?]

Also, an example of trees as traversable structures may be seen in the appendix. Now, if we want to obtain the maximum of a traversable structure we may construct the following class:

```
CLASS   Maximizable
 OVER  Element : With-order
 SUBCLASS-OF Traversable
DEFINE   OPS
            maxim : Maximizable    →  Element
            update: Maximizable  ×  Element  →  Element
          EQS   { m: Element;   x: Maximizable }
            maxim( x ) = update( rest( x ), first( x ) )
            update( x, m ) = IF end?( x )
                        THEN    m
                        ELSE  IF m < first( x )
                          THEN update( rest( x ), first( x ) )
                          ELSE update( rest( x ),  m )
END-CLASS
```

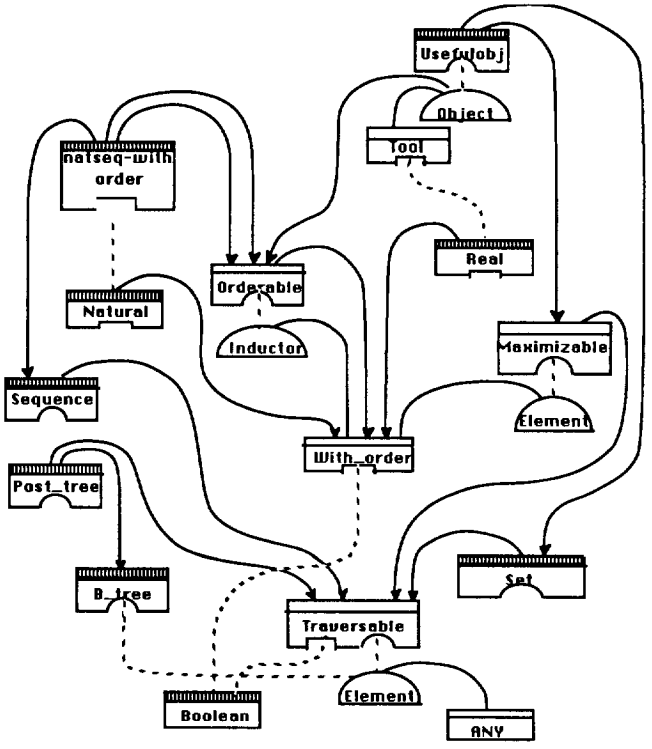
Let us now suppose that the problem is to find the most useful object belonging to a set of tools, where tool is a generic type whose elements have a profit and a weight (see the full specification in the Appendix), assuming that the utility of such objects is defined by the quotient profit/weight. Then the new specification is:

```
CLASS  Useful-objects
 OVER  Object:Tool
 SUBCLASS-OF    Set [ Object :  Element ] ,
                Maximizable[   Object: Element;  the-most-useful:  maxim
                                any : first;   anyless: rest;  Ø ?: end?    ]
  WHERE     Object SUBCLASS-OF Orderable[  Real: Inductor  ;  utility: f
                                eq-useful: equiv ;    less-useful: <   ]
   DEFINE  OPS  utility
            EQS     { a : Object  }
            utility(a ) = profit(a) /  weight(a)
  END-WHERE
END-CLASS
```

where the *WHERE* clause is a local declaration inside the *SUBCLASS* clause. It is used to induce an order relation over objects, so that *Object* is a subclass of *With-order*, and the declaration of *Useful-objects* as subclass of *Maximizable* is correct. A similar effect would have been achieved declaring, in a separate class definition, *Object* as subclass of *Orderable* and, then, *Useful-objects* over *Object*.

We could continue this example until arriving, for instance, to the specification of a solution of the knapsack problem by means of a greedy approach. The rest of the specification can be found in the Appendix.

To end, the following picture shows the connections among the specifications that we have presented up to now. The dashed lines denote the **over** relation, the arrows the **subclass-of** relation, and the full lines without head arrows specifications equivalent up to renaming. The different shapes of the boxes, used to represent classes, correspond to a graphic notation that is of no interest now.



3. Semantics of GSBL

In this section we will sketch the ideas over which the formal semantics of GSBL is defined. First, we will introduce some basic definitions and results concerning algebraic specifications with constraints. Then, we will overview the formal semantics of the main constructions of the language. The reader is assumed to have some background on algebraic specification (for instance, [EM85]).

Definition 1

A **presentation** P is a tuple $< \Sigma, E >$, where Σ is a signature i.e. a pair $< S, Op >$ where S is a set of sorts and Op a set of operations, and E is a set of Σ-equations.

A **constraint** C on P is a pair of presentations (P_1, P'_1) such that $P_1 \subseteq P'_1$ and $P'_1 \subseteq P$. A constraint C is **persistent** iff for any P'_1-algebra A, we have $(A|_{P_1})|^{P'_1} \equiv A$

Notation : $\_|_P : Alg_{P'} \longrightarrow Alg_P$ *and* $\_|^{P'} : Alg_P \longrightarrow Alg_{P'}$ *denote, respectively, the forgetful functor and the free functor associated to the inclusion* $P \subseteq P'$

A constraint, as we have already said, indicates that the P'_1 part of the specification P is completely defined, once the P_1 part is given. This may be made precise with the following definition:

Definition 2

An algebra $A \in Alg_P$ satisfies a constraint (P_1, P'_1) with $P'_1 \subseteq P$ iff $(A|_{P_1})|^{P'_1} \equiv A|_{P'_1}$

This means that a P-algebra satisfies the constraint (P_1,P'_1) if the P'_1 part of the algebra is freely generated from the P_1 part. Technically, this is called an *initial or data constraint* because it "follows" the initial algebra semantics philosophy. In particular, if we want the models of a presentation to be an extension of a basic data type, for instance the booleans, then we would need to define a constraint, in which the first part is the empty specification and the second part is the boolean specification. As a consequence, an algebra would satisfy the constraint if it is an extension of the boolean initial algebra. The kind of specifications we will deal with are, therefore, specifications with constraints:

Definition 3

A **specification** SP is a pair $<P, \zeta>$ where P is a presentation and ζ is a family of constraints on P.

As we have said in the previous section, certain properties guarantying the "protection" of the constraints are needed to assure correctness of the specifications.

Definition 4

A presentation P is **compatible** with a constraint (P_1, P'_1), $P'_1 \subseteq P$ iff

$$\forall t_1, t_2 \in T'_1(X_1)_S, s \in S'_1 - S_1 \quad P \vdash t_1 = t_2 \Rightarrow P'_1 \vdash t_1 = t_2$$

A pair of constraints $C_1 = (P_1,P'_1)$, $C_2 = (P_2,P'_2)$ are **mutually non destructive** if $\forall A \in Alg_{P1+P2}$ and for i = 1,2

$$(A|_{P_i})|^{P'_i} \cong (A|^{P'_1+P'_2})|_{P'_i}$$

A specification $SP = <P,\zeta>$ is **correct** if ζ is a family of persistent and mutually non destructive constraints and P is compatible with ζ.

Compatibility of constraints can be characterized proof-theoretically by means of consistency and sufficient completeness properties:

Theorem 5

Two constraints (P_1,P'_1) and (P_2,P'_2) are mutually non destructive iff for i=1,2 it holds:

a. **(consistency)** $\forall s \in S'_i - S_i$ and $t_1, t_2 \in T_{\Sigma'_i}(X_i)_S$,
$P'_1 + P'_2 \vdash t_1 = t_2 \Rightarrow P'_i \vdash t_1 = t_2$

b. **(sufficient completeness)** $\forall s \in S'_i - S_i$ and $t \in T_{\Sigma'_1+\Sigma'_2}(X_1+X_2)_S$
$\exists\, t' \in T_{\Sigma'_i+\Sigma_1+\Sigma_2}(X_1+X_2)_S$ such that $P'_1 + P'_2 \vdash t = t'$

Now, we may define the semantics of a specification:

Definition 6

The **semantics** of a specification SP is defined by the following class of models

$$Mod(SP) = \{ A \in Alg_P / P \text{ satisfies } \zeta \}$$

From now on, we will assume the correctness of all specifications we are dealing with, since that assures the existence of models. Moreover, it also assures the compatibility, between the syntactic and the semantic level, of certain constructions used in the definition of algebraic specification languages. We do not include these results since we feel that they are slightly out of the scope of this paper.

The over relation of our language is based on the following notion of extension:

Definition 7

A specification SP_1 is an **extension** of another specification SP_2 if

a. $SP_2 \subseteq SP_1$

b. $\forall\ t_1, t_2 \quad T_{\Sigma 2}(X_2)$, $SP_1 \vdash t_1 = t_2 \Rightarrow SP_2 \vdash t_1 = t_2$

We do not ask for sufficient completeness properties since they are implicit in the correctness of SP_1 and SP2. To define the subclass relation we will need a notion of specification morphism:

Definition 8

A **specification morphism** $f: <R_1, \zeta_1> \rightarrow <R_2, \zeta_2>$ is a presentation mophism $f: R_1 \rightarrow R_2$, such that $\forall\ A \in Alg_{P'2}$ if A satisfies (P_2, P'_2) then U_f ($A|_{f(P'_1)}$) satisfies (P_1, P'_1), where U_f is the forgetful functor associated to f.

The idea of this definition is that we ask SP_2 to contain "stronger" constraints than SP_1. Now, to end with this part, the following theorem is the basis for defining the result of the parameter passing-like mechanism of GSBL (combining the subclass and the over relation) and assuring its correctness.

Theorem 9

Let SP_2 be an extension of SP_1, let $f: SP_1 \rightarrow SP'_1$ be a specification morphism (wolog, see the following note, we assume that $(SP_2 - SP_1) \cap SP'_1 = \emptyset$). The result of substituting SP1 by SP1' in SP2 is the specification $SP'_2 = <P'_2, \zeta'_2>$ where $P'_2 = P'_1 + f'(P_2)$, $\zeta'_2 = \zeta'_1 + f(\zeta_2)$ and f' is the presentation morphism defined as extension of f in the usual pushout construction. Then we have:

1. SP'_2 is a correct specification, i.e. ζ'_2 is a family of persistent and mutually non destructive constraints and P'_2 is compatible with respect to ζ'_2.
2. f' is a specification morphism.

Note. If $(SP_2 - SP_1) \cap SP'_1 \neq \emptyset$ we consider that f ' renames the common sorts and operations names.

Now, we may provide the semantics for GSBL. The basic notion for its definition is the concept of environment. An environment is a set of specifications related by extensions and morphisms defining the **over** and the **subclass-of** relations, respectively. Every language construction modify the environment adding new specifications and relationships among them. In particular, the effect of a correct *class definition* on the environment is obtained by the composition of the effects of all its constituent clauses.

A *class definition* denotes a specification, whose presentation is obtained by putting together the presentations of the subspecification inherited by the *over* and *subclass* clauses and, then, including the sorts and operations declared in the *with* and *define* clauses. The constraints of this specification are, also, all the inherited constraints together with a new constraint (P_1, P_2), where P_2 is the whole presentation associated to the class definition and P_1 is P_2 minus the sorts, operations and equations of the *define* clause, in case there is one. Then, the effect of a class definition on the environment consists on adding to it the specification associated to the class, together with all the new relations declared in the *over* and *subclass* clauses. In addition, these clauses may contain some local declarations that also add to the environment new specifications and relations.

Before defining more precisely the effect of all the clauses of a class declaration, let us first define the

effect of a *class instantiation*, i.e. the substitution, as defined in GSBL, within a class C_1, of a class C_2 by a class C_3, when C_1 is **over** C_2 and C_3 is a subclass of C_2 via a specification morphism *f*. Substitutions of this kind may occur in a subclass declaration like:

Object SUBCLASS-OF Orderable [Real : Inductor; utility: f; eq-useful: =; less-useful: <]

or within local declarations of the form:

Input: Traversable[Data:Element; select: first]

in the *over* and the *subclass* clauses. This definition, as it was already said, is based on theorem 9, but it is slightly more complicated. The causes for this additional complication are the following:

- The definition of the subclass C_3 may be implicitly given in the instantiation declaration. For instance, in the previous declaration Traversable is defined over *Element*, and, implicitly, a new class called *Data*, that maybe was not in the environment, is now being implicitly defined. In the example the new class *Data* would just be a renaming of the class *Element*.

- Also, the specification morphism establishing the subclass relationship between C_2 and C_3 may be implicitly given, partially or totally. In GSBL two elements (sorts, operations or classes) with the same name (and the same arity, in the case of operations, or the same components, in the case of classes) are identified. Therefore, there is no need to establish an explicit binding between two such elements. This has the advantage of an economy of writing. However, the price to be paid is a possible loss of correctness, since inconsistencies may be provoked. Later on, we shall again discuss this issue.

Now, the semantics of a class instantiation is the following. Given a class C **over** $C_1, \ldots, C_n$, and a binding *b*, defined by the list $[e_1: e'_1, \ldots, e_m: e'_m]$, where the e_i are either sorts, operations or names of classes, then if *b* is a correct binding for C in the current environment (i.e. if the resulting specification is correct), C [*b*] denotes the class obtained as follows:

- First the C_{over} subspecification is obtained from the disjoint union of $C_1, \ldots, C_n$ (i.e. if $C_1, \ldots, C_n$ contain elements with the same name they will not be yet identified). For instance, in the definition of *Input*, C_{over} would be the union of *Boolean* and *Element* (a renaming of *ANY*), since traversable is defined over these specifications.

- Next, the specification C'_{over} is defined as a subclass of C_{over} by renaming some parts of it consistently with the binding *b*. In the definition of *Input*, C'_{over} would be obtained renaming in C_{over} the sort *Element* by *Data* and the operation *first* by *select*.

- Then, applying the construction of theorem 9, we obtain the specification C'':

$$\begin{array}{ccc} C_{over} & \hookrightarrow & C \\ h \downarrow & & \downarrow \\ C'_{over} & \hookrightarrow & C'' \end{array}$$

that is C'' would contain all the elements of C'_{over} plus the new sorts, operations, equations and constraints introduced by C.

- Finally, C[*b*] is obtained from C'' identifying all the elements with the same name. As it was said before, this identification can cause an inconsistency in the resulting specification. Then, one could think, a priori, that the correctness of the whole specification would have to be tested, with a consequent loss of modularity. But, in fact, if only subspecifications are identified the subclass relation guarantees correctness. And when isolated

sorts and operations are also separately identified, it is possible to find conditions in which some kind of local consistency, between the affected subspecifications, is sufficient to ensure that the resulting class C [*b*] is a correct specification.

Now, we may define the semantics of all the clauses of a class definition. The semantics of an *Over* clause of the form *OVER* C_1 ,... , C_n in the *class definition* for A, consists in the following modifications of the environment:

1. Constructing the A_{over} subspecification of A. Also, at this moment A will denote A_{over} in the environment

2. Adding to the environment the relationships A **over** C_i, and

3. If a class variable definition B: C[*b*] is inside the *over* clause, then it causes the creation of a local class B instantiating C[*b*]. It also has as effect the following local relationships betwen B and the classes related to C. For each class D such that C **over** D, B **over** *h* (D) being *h* the morphism defined by *b*, if C **subclass-of** D via *h'* then B **subclass-of** D via $h \circ h'$

The semantics of a *subclass* clause of the form *SUBCLASS-OF* C_1 ,... , C_n in the *class definition* for A consists in the following modifications of the environment:

1. Updating the specification associated to A in the current environment, adding all the inherited elements

2. Adding to the environment the relationships A **subclass-of** C_i, and A **over** C' for each C' such that C_i **over** C'

A *where* clause inside the *subclass* clause, causes the same effect as adding to the environment the result of the local declarations inside the *where*, and, then, evaluating the rest of the *subclass* clause.

The semantics of the *with* and *define* clauses of the form *WITH w.sortlist w.oplist w.equlist , DEFINE sortlist oplist equlist* in the *class definition* for A consist in modifying the environment, by adding to the presentation associated to A, the sorts in *w.sortlist,* operations in *w.oplist,* and equations in *w.equlist* . They also add to the ζ constraints family associated to A a new constraint (A', A''), where A'' is the new presentation associated to A, and A' is A'' minus the sorts, operations and equations of the *define* clause, in case there is one.

For instance, the declaration of the class *Knapsack* (see the appendix) would cause the definition in the current environment of a new specification, obtained by adding the operations and equations from the *WITH* and the *DEFINE* clauses to the union of the specifications *Real, Tool, Pair* (included in the *OVER* clause) and *Sequence* (from the *SUBCLASS* clause), transformed according to the declared substitutions. Also, a new constraint will be added establishing the definition of *remaining-cap, performance* and *put-object* in terms of the rest of the presentation. Additionally, the new **over** and **subclass-of** relations introduced by the class *KNAPSACK* would be included in the environment.

4. Conclusion

In this paper, a new approach for specification design has been introduced. The main idea consists in dealing with incomplete specifications by means of *constraints* and in introducing a new subclass relation, with the meaning of "is more detailed than", allowing for true inheritance in specification design. In particular, this new

notion of inheritance embeds the concept of genericity as it is usually understood in algebraic specification.

Based in this subclass relation, the specification language GSBL has been defined, showing its use in specification design by sketching an example of parameterized programming [Gog84]. Moreover the idias used for defining its formal semantics have been introduced characterizing the correctness criteria that specifications must fulfil.

ACKNOWLEDGEMENTS

This work has been partially supported by Comisión Asesora de Investigación (ref. 2704-83).

4. References

[BG77] Burstall, R.M.; Goguen, J.A. "Putting theories together to make specifications", Proc. V IJCAI, Cambridge Mass., 1977, pp. 1045-1058.

[BG80] Burstall, R.M.; Goguen, J.A. "The semantics of Clear, a specification language", Proc. Winter School on Abstract Software Specification, Springer LNCS 86, pp. 292-332, 1980.

[CAR84] Cardelli, L. "The semantics of multiple inheritance", Proc. Colloquium on the Semantics of Data Types, Sophia-Antipolis, 1984.

[CW85] Cardelli, L.; Wegner, P. "On understanding types, data abstraction and polymorphism", *Computer Surveys* 17, 4 (Dec. 1985), pp. 471-522.

[EKMP82] Ehrig, H., Kreowski, H.-J., Mahr, B., Padawitz, P. "Algebraic implementation of abstract data types", *Theoret. Comp. Sc.* 20 (1982), pp. 209-263.

[EWT82] Ehrig, H., Wagner, E.G. Thatcher, J.W. "Algebraic constraints for specifications and canonical form results", Institut für Software und Theoretische Informatik, T.U. Berlin Bericht Nr. 82-09, 1982.

[EM85] Ehrig, H., Mahr, B. *"Fundamentals of algebraic specification 1"*, EATCS Monographs on Theor. Comp. Sc., Springer-Verlag, 1985.

[FGJM85] Futatsugi, K, Goguen, J.A., Jouannaud, J.-P., Meseguer, J., "Principles of OBJ2", *Proc. 12th POPL*, Austin 1985.

[GOG84] Goguen, J.A. "Parameterized Programming", *IEEE Trans. on Soft. Eng.* SE10, 5 (Sept. 1984), pp. 528 - 543.

[GB80] Goguen, J.A., Burstall, R.M. "CAT, a system for the structured elaboration of correct programs from structured specifications", Report CSL-118, Comp. Sc. Lab., SRI Int., 1980.

[GM85] Goguen, J.A.; Meseguer, J. "Order-sorted algebra I: partial and overloaded operators, errors and inheritance", SRI Int., Comp. Sc. Lab. Rep., 1985.

[MEY86] Meyer B. " Genericity versus Inheritance ", *Proc. ACM conf. Object-Oriented Programming Syst, Languages, and Applications*, ACM, New York, 1986, pp. 391-405

[MEY87] Meyer B. "Reusability: The Case for Object-Oriented Design", *IEEE Trans. Software Eng.* March 1987, pp. 50-65.

[REI80] Reichel, H. "Initially restricting algebraic theories", Proc. MFCS 80, Springer LNCS 88 (1980), pp. 504-514.

[SAN81] Sannella, D. "A new semantics for Clear", Report CSR - 79 - 8, Univ. of Edinburgh, 1981.

[WPPDB83] Wirsing, M., Pepper, P., Partsch, H., Dosch, W., Broy, M. "On hierarchies of abstract data types", *Acta Informatica* 20 (1983), pp. 1-33.

APPENDIX

```
CLASS Postree
SUBCLASS-OF B-tree , Traversable [ empty? : end? ]
DEFINE
 OPS   first, rest
 EQS   { t1 , t2 : Postree    ; x : Element }

   first( maketree( t1 , x, t2 ) ) = IF empty?( t1 ) ∧ empty?( t2 )
                                     THEN x
                                     ELSE IF empty?( t1 )
                                             THEN first( t2 )
                                             ELSE first( t1 )

   rest( maketree( t1 , x, t2 ) ) = IF empty?( t1 ) ∧ empty?( t2 )
                                    THEN t1
                                    ELSE IF empty?( t1 )
                                            THEN maketree( t1 , x, rest( t2 ) )
                                            ELSE maketree( rest( t1 ), x, t2 )
END-CLASS
```

```
CLASS B-tree
 OVER  Boolean
       Element: Any
 DEFINE
 SORTS  B-tree
 OPS   empty : → B-tree
       maketree: B-tree × Element × B-tree → B-tree
       empty ? : B-tree → Boolean
       left , right  : B-tree → B-tree
       root : B-tree → Element
 EQS   { t1 , t2 : B-tree      x : Element }
       left, right , root ( empty ) = ?
       left ( maketree( t1 , x, t2 ) ) = t1
       right ( maketree( t1 , x, t2 ) ) = t2
       root( maketree( t1 , x, t2 ) ) = x
       empty ? ( empty ) = true
       empty ? ( maketree( t1 , x, t2 ) ) = false
END-CLASS
```

```
CLASS Greedy
 OVER  Data , Solution:  ANY
       Input : Traversable [ Data:  Element , select: first ]
WITH
OPS   init: Input  → Solution
      stop-cond.: Input × Solution  → Boolean
      feasible: Solution × Data  →  Boolean
      unite: Solution × Data  → Solution
DEFINE
OPS   solve: Input  → Solution
      apply: Input  × Solution  → Solution
EQS   { c: Input; s: Solution }
   solve( c ) = apply( c, init( c ) )
   apply( c, s ) = IF stop-cond.( c, s ) THEN s
                   ELSE
                   IF feasible( s, select( c ) )
                    THEN apply( rest( c ), unite( s, select( c ) ) )
                    ELSE apply( rest( c ), s )
END-CLASS
```

```
CLASS  Tool
OVER   Real
WITH
  SORT  Tool
  OPS   profit: Tool  →  Real
        weight: Tool  →  Real
END-CLASS
```

```
CLASS  Knapsack
OVER  Real
        Object: Tool
        Obj-quant : Pair [ Object: Elem1; Real: Elem2; * : makepair
                           which: selec1; quant: selec2 ]
SUBCLASS-OF Sequence [ Obj-quant : Element ]
WITH
  OPS capacity:  →  Real
DEFINE
  OPS  remainig-cap, performance:  Knapsack  →  Real
       put-object:  Knapsack  ×  Object  →  Knapsack
  EQS  [ o : Object;  c:  Real;  m:  Knapsack  p: Obj-quant ]
   remaining-cap( empty )= capacity
   remaining-cap( * (m , * ( o, c) ) = remaining-cap( m )- weight (o) . c
   performance ( empty )= 0
   performance( * (m , * ( o, c ) ) = performance( m ) + profit (o) . c
   put-object( m, o) = IF remainig-cap( m ) ≥ weight( o )
                       THEN  put( * ( o, 1 ) ,m)
                       ELSE put( * ( o, remaining-cap(m)/ weight(o) ) , m)
END-CLASS
```

```
CLASS Greedy-knapsack
 OVER  Objects-set: Useful-Objects
       Optim-knap:  Knapsack
SUBCLASS-OF Greedy[ Objects-set: Input;  Optim-knap: Solution;  Object:  Data;
                    empty-knapsack: init;   not-full: feasible;
                    put-object: unite;   fill-knapsack: solve ]
  WHERE
     Objects-set SUBCLASS-OF Traversable[ Object: Element;     the-most-useful: first
                                          remove.optimum : rest ;  Ø  ?: end?     ]
      DEFINE
       OPS   remove-optimum
       EQS   [ x : Object-set  ]
         remove.optimum( x ) = remove( x, the-most-useful( x ) )
   END-WHERE
DEFINE
 OPS   stop-cond;   empty-knapsack;   not-full
 EQS   [  x : Objects-set;  a, b: Object;   m: Optim-knap  ]
   stop-cond( x, m ) =  Ø  ?( x )  ∨( remaining-cap(m)  ≡ 0 )
   empty-knapsack( x ) =  Ø
   not-full( m, a ) =  remaining-cap(m)  >  0
END-CLASS
```