An Algebraic Theory of Class Specification

FRANCESCO PARISI-PRESICCE and ALFONSO PIERANTONIO
Università degli Studi de L'Aquila

The notion of class (or object pattern) as defined in most object-oriented languages is formalized using known techniques from algebraic specifications. Inheritance can be viewed as a relation between classes, which suggests how classes can be arranged in hierarchies. The hierarchies contain two kinds of information: on the one hand, they indicate how programs are structured and how code is shared among classes; on the other hand, they give information about compatible assignment rules, which are based on subtyping. In order to distinguish between code sharing, which is related to implementational aspects, and functional specialization, which is connected to the external behavior of objects, we introduce an algebraic specification-based formalism, by which one can specify the behavior of a class and state when a class inherits another one. It is shown that reusing inheritance can be reduced to specialization inheritance with respect to a virtual class The class model and the two distinct aspects of inheritance allow the definition of *clean* interconnection mechanisms between classes leading to new classes which inherit from old classes their correctness and their semantics.

Categories and Subject Descriptors: D 1 5 [**Programming Techniques**]: Object-Oriented Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.10 [**Software Engineering**]: Design—*methodologies*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3 3 [**Programming Languages**]: Language Constructs and Features–*abstract data types, modules*; packages; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*

General Terms· Design, Languages

Additional Key Words and Phrases: Algebraic specifications, inheritance, interconnection mechanisms, modularity

1. INTRODUCTION

The 1980s have witnesses a fast growth of the object-oriented methodology and related languages, and it is expected they will be prevalent in the next decade. In object-oriented programming languages, programs are often con-

This work has been supported in part by the Italian CNR under the project *Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo,* subproject *Linguaggi di Nuova Concezione,* by the German *DFG*, and by the European Community under the ESPRIT *Basic Research Working Group COMPASS*.
Authors' addresses: F. Parisi-Presicce, Dipartimento di Scienze dell' Informazione, Universitá di Roma La Sapienza, Via Salaria 113, I-00198, Roma, Italy; email: parisi@vxscaq.acquila.infn.it. A. Pierantonio, TFS, Fachbereich Informatik, Technische Unitersität Berlin, D-10587 Berlin, Germany.

sidered as a collection of *objects* and *messages*. An object, also called *instance*, is a single entity integrating data and operations which operate on these data, usually called *methods*. Messages are execution requests of operations. Objects with similar properties and features belong to the same *class*, which is an *object pattern*.

In recent years the ever greater need for quality software has led to a number of techniques to enhance reusability, extensibility, and compatibility of software components. Among them, inheritance characterizes the object-oriented programming languages. This mechanism allows the use of old classes in the definition of new ones. The usual terminology gives the names of *superclasses* for the former and *subclasses* for the latter.

Unfortunately, the explosive spread of key notions such as inheritance, encapsulation, and subtyping was not followed by the development of a uniform terminology [Nelson 1991]. The term *inheritance*, for example, has been used to mean anything from the preservation of given characteristics with the addition of new ones, to the use, without constraints of any kind, of parts of a class, whether intended to be reused or not.

Even apparently similar meanings of a term can turn out to have distinctively different implementations. To understand and compare better the intended meaning of similar terms in various object-oriented languages and to analyze conceptually different uses of the same word, we propose a formalization of the concept of class. This formalization is based on algebraic specifications, intended in the widest possible sense, although presented in their simplest form for clarity of exposition. The choice of the algebraic framework was dictated by the simplicity of its semantics and the modularity of its tools.

The class model is intended to capture both the corresponding features in existing languages and suggestions borrowed from other areas [Ehrig and Mahr 1990].

One of the features included in most languages is *encapsulation*, which provides for a distinction between what the class implements and what the designer of the class allows to be seen (and therefore used) from the outside. This technique allows the designer to change some internal details without affecting the clients as long as the behavior of the class remains compatible. Some of the features of a class visible to the outside may not be strictly related to each object instance of that class: a separate interface (usually included in the interface of the class containing visible data and methods) is useful to set aside the messages to which the instances can react [Snyder 1986], in order to avoid compromising the encapsulation benefits because of inheritance. The danger of accessing inherited variables in a subclass leaves the designer of the superclass unable to rename, reinterpret, or suppress them without rendering the subclass illegal.

The possibility of modeling genericity [Meyer 1986] has suggested another part of the class, namely, a parameter part, intended to model what are called unconstrained and constrained genericity with the possibility, additionally, to specify some of the properties required of the methods in the

parameter. This technique allows us to write generic software components which are modular and reusable.

Finally, inspired by the work on module specifications [Blum et al. 1987], we propose an explicit import interface, which can be viewed as a (highly!) virtual class on which the rest of the class being built is based. Unlike other languages with features such as *use*, the import interface specifies only *what* is needed, but not *which* class is intended to provide the methods and class variables needed, leaving it to the interconnection mechanism to indicate the appropriate match. The presence of the import as well as of the class interface allows the development of a system bottom-up by extending already-built data types successively, top-down by refining and implementing abstract data types successively as well as *middle-out*, where certain class (or instance) interfaces can at first be realized assuming the existence of certain functionalities expressed in the import interface; then the class interface is enriched or extended to obtain the desired methods and the realized import interface using other classes until all import interfaces are eliminated. Among the advantages, there is the possibility of combining one class with several different classes.

The proposed model of the class consists then of five parts: two interfaces, instance interface and class interface, for the two different roles of the class; another one, the import, requiring a producer for that class; a parameter part to model genericity, and an implementation part that includes the other four, in addition to the hidden features of the class. With such a model as basis, we are then able to distinguish between the notion of inheritance based on specialization and the notion of inheritance based on the reuse of code. We show that the two notions are related and that, in fact, the former is sufficient to express the latter.

In this article we propose a formalization which allows us to analyze, criticize, and compare several object-oriented languages with respect to the features modeled by this notion of class and by our definitions of inheritance which can be interpreted under different views. The languages that we have analyzed belong to the dimension of *class centric* according to the classification in Gabriel et al. [1991]. In these languages, the class is the main *factored description* in which the internal structure of the objects and the behavior of methods is specified. Some of the languages which are considered *operation centric*, i.e., which do not support the message-passing metaphor, are also covered. Object systems have been classified also by Wegner [1987]. Languages are defined to be *object based*, *class based*, and *object oriented* depending on the existence of objects, classes, and inheritance. With the proposed model, we are able to describe the last two categories of classes.

The relationship between the type and class notions is not homogeneous: while in languages such as Trellis/Owl [O'Brien et al. 1987; Schaffert et al. 1986] and POOL [America 1987; America and van der Linden 1990], there is not a one-to-one correspondence between types and classes, and different classes may implement the same type; in BETA [Kristensen et al. 1987], a programming language in the tradition of Simula, one is not allowed to consider different classes as being of the same type.

The notion of type is quite general, and one of the main characteristics of a language is the relationship between the notion of type and that of class. In either case, the type concept deals with a specification of behavior. At the moment, in the space of languages we can find systems such as OOZE [Alencar and Goguen 1991] and the OBJ family, which are able to handle behavior in a proper way. One way to consider type classes as separate concepts is to restrict the specification of behavior to signatures, and deal only with signature types. In the POOL programming language, a type specification is a signature with a list of property identifiers. This specification method is very simple but could play an important role in the task of programming, considering a property identifier as an abbreviation for a formal specification.

Another key notion in object-oriented methodology is inheritance, which allows the definition of new classes starting from variables and methods of old classes. But this definition of inheritance is not precise enough, and it has been pointed out that there are different views of such a mechanism [Snyder 1986]. Although both called inheritance, there is no confusion between the idea of code sharing and the notion of functional specialization. While the BETA programming language gives the possibility to extend in a subclass the old methods defined in a superclass in general, in the other languages taken into consideration, inheritance allows the redefinition of an old method, resulting in a specialization process without being able to say anything about correctness. In such a framework, multiple inheritance can be considered only for sharing and combining code; BETA does not allow multiple inheritance, but POOL does, by decoupling inheritance and subtyping.

Other aspects taken into account concern encapsulation, genericity, and strict typing. We agree with Madsen et al. [1990] and Schaffert et al. [1986] that a programming language must offer a proper tradeoff between flexibility and static checking in order to be useful. The object-oriented programming languages we have analyzed are C++, Eiffel, Trellis/Owl, POOL, BETA, Smalltalk, but also characteristics of other languages such as, OOZE and CLOS, will be mentioned.

In the following sections, we give the formal definition of our model of class and introduce the notions of specialization and reusing inheritance. These two relations show how specialization is different from code reusing. Then we show how to simulate the former by the latter via a virtual class. Two new relations between pairs of classes are presented, relations based on the use of the import interface and the parameter part. One relation indicates whether one class produces an instance interface which satisfies the constraints set forth in the parameter part of another class. The effect of substituting the class for the parameter is shown to be a clean (i.e., whose semantic effect can be predicted) way to obtain a new class which inherits by specialization from the old one. The other relation matches the producer of some methods to a potential consumer of those methods by finding a correspondence between the class interface of a class and the import interface of another class. The effect of matching the needed import with the provided class interface is again seen

as a clean way to obtain a new class which inherits by implementation from the producer and by specialization from the consumer.

2. PRELIMINARY NOTATION

In this section, we review briefly some basic notions of algebraic specifications; details can be found in Ehrig and Mahr [1985] and Wirsing [1991]. A *signature* Σ is a pair (S, OP) where S is a set of *sorts* and OP a set of *constants* and *function symbols.* A *pointed signature* is a signature $\Sigma = (S, OP)$ with a distinguished element of the set S of sorts denoted by $pt(\Sigma)$. By a Σ-algebra $A = (S_A, OP_A)$ on a signature $\Sigma = (S, OP)$, we mean a family $S_A = (A_s)_{s \in S}$ of sets and a set $OP_A = (N_A)_{N \in OP}$ of operations. The category of all Σ-algebras is denoted by $Alg(\Sigma)$. If $\Sigma_1 = (S_1, OP_1)$ and $\Sigma_2 = (S_2, OP_2)$ are signatures, a *signature morphism* $h: \Sigma_1 \to \Sigma_2$ is a pair of consistent functions $(h^S: S_1 \to S_2, h^{OP}: OP_1 \to OP_2)$. Every signature morphism $h: \Sigma_1 \to \Sigma_2$ induces a *forgetful functor* V_h: $Alg(\Sigma_2) \to Alg(\Sigma_1)$. A *pointed signature morphism* is a signature morphism $h: \Sigma_1 \to \Sigma_2$ such that $h^S(pt(\Sigma_1)) = pt(\Sigma_2)$. By an *algebraic specification* $SPEC = (\Sigma, E)$ we intend a pair consisting of a signature Σ and a set E of (positive conditional) equations. If $SPEC_1 = (\Sigma_1, E_1)$ and $SPEC_2 = (\Sigma_2, E_2)$ are two algebraic specifications, a *specification morphism* $f: SPEC_1 \to SPEC_2$ is a signature morphism $f: \Sigma_1 \to \Sigma_2$ such that the translation $f^{\#}(E_1)$ of the equations of $SPEC_1$ is contained in E_2. A *pointed algebraic specification* is an algebraic specification with a pointed signature. A *pointed specification morphism* between pointed specifications is a pointed signature morphism which is also a specification morphism. For notational convenience, when $SPEC = (\Sigma, E)$ is a pointed specification the distinguished sort $pt(\Sigma)$ will be also denoted by $pt(SPEC)$. The algebraic specifications and the specification morphisms form the category CATSPEC of algebraic specifications [Ehrig and Mahr 1985]. For readability some notions of category theory are included in the Appendix.

3. THE CLASS MODEL

In this section we propose a class specification in order to model with generality the class notion present in the current object-oriented programming languages. Such a mathematical notion could be fruitfully exploited to understand better the main mechanisms and their interactions.

The importance of inheritance is widely recognized, but it is not the only peculiar feature of the object-oriented paradigm: encapsulation is considered as important [America 1990]. This protection mechanism allows the designer to draw a boundary between the implementation and the outside. The operations which can be invoked over the instances of a class are just those listed in the external interface of the class itself, and any attempt at executing a private operation results in an error. The minimalization of the interdependencies of separately written software components and the reduction of the amount of implementational details are among the major benefits due to this technique. The concept of an abstract data type is strengthened by

the presence of an external interface because of the separation of the functionalities versus the implementation of an abstraction.

Unfortunately, inheritance can reduce the benefits of encapsulation. In fact accessing, in a subclass, inherited variables leaves the designer of the superclass unable to rename, reinterpret, or remove these variables. On the other hand, there are two categories of clients of a class, those who need to manipulate objects (the clients of the instances) and those who want to reuse somehow the class in order to specialize it or just reuse its code (the clients of the class) [Snyder 1986]. Thus, some languages have two external interfaces, one for each kind of clients, and they are usually defined incrementally since the interface for the class clients has a greater view than the instance clients. We will call these interfaces *instance* and *class interface* (see Table I, later).

Usually, inheritance allows a designer to arrange classes in specialization hierarchies through a *top-down* process. Moreover, we can also define a class by reusing the code of another one or instances from other classes. This results in a *bottom-up* process. An interesting case arises if we allow an explicit import interface where we describe some features which are necessary in order to realize the external behavior of the class. This allows the implementation of a class assuming the existence of some functionalities but disregarding which class will provide them. Unfortunately, most programming languages have constructs for defining modules and reuse them only by listing their names. This approach goes in the opposite direction of abstraction since we need to declare explicitly which is the supplier of a particular abstraction.

The class that we have described in the previous section is composed of a certain number of parts: a parameter part, an import interface, an instance, and a class export interface. All these components declare signatures and their properties.

Definition 3.1 (*Class Specification and Semantics*). A class specification C_{spec} consists of five algebraic specifications *PAR* (parameter part), EXP_i (instance interface), EXP_c (class interface), *IMP* (import interface), and *BOD* (implementation part), and five specification morphisms as in the following commutative diagram.

$$\begin{tikzcd} PAR \arrow[r, "e_i"] \arrow[d, "i"'] & EXP_i \arrow[r, "e_c"] & EXP_c \arrow[d, "v"] \\ IMP \arrow[rr, "s"'] & & BOD \end{tikzcd}$$

The specification EXP_i, EXP_c, and *BOD* are pointed specifications, and e_c and v are pointed specification morphisms.

The semantics $SEM(C_{spec})$ of a class specification is the set of all pairs (A_I, A_{E_c}) of algebras, where $A_I = V_s(A_B)$ and $A_{E_c} = V_v(A_B)$ for some $A_B \in Alg(BOD)$. The pointed sort $pt(EXP_i)$ is called class sort.

Interpretation. Each of the five parts consists not only of signatures, but also of equations, which describe some of the properties of the operations. The interfaces EXP_i and EXP_c describe the external access functions and their

behavior: the former describes the messages which can be sent to the objects that are instances of the class, while the latter contains the methods which can be used by other classes. The part of *BOD* not in EXP_c is hidden from other classes. The specification *BOD* describes an implementation of the exported methods using the ones provided by the *IMP* specification. The import specification *IMP* contains information about *what* is needed by *BOD* to implement EXP_c, but not *which* class can provide it: the latter task is provided by the interconnection mechanisms. The specification *PAR* models genericity, unconstrained if the specification consists of sorts only, constrained when the parameter is required to have operations satisfying certain properties.

Remark 3.2. The semantics chosen for a class specification is only one of a number of possibilities, and the theory presented here can be developed by choosing one of the alternatives. One possibility is a functional semantics (as done in Parisi-Presicce and Pierantonio [1991]) where the meaning of a class specification is a (functorial) transformation which takes a model (an algebra in this case) of *IMP* and returns a model (again an algebra) of EXP_c. In this case, what are needed are results such as the Extension Lemma in Ehrig and Mahr [1985] if, for example, the semantics is the composition of the forgetful functor V_v: $Alg(BOD) \rightarrow Alg(EXP_c)$ with the free functor $Free_s$: $Alg(IMP) \rightarrow Alg(BOD)$. Another possibility is a loose semantics, where only some pairs (A_I, A_{E_c}) are chosen. In the latter case, the semantics must be definable uniformly for all classes and must be closed with respect to Amalgamation [Ehrig and Mahr 1985].

Definition 3.3 (*Class*). A class $C = (C_{spec}, C_{impl})$ consists of a class specification C_{spec} and a class implementation C_{impl} such that $C_{impl} = A_B$ for some *BOD*-algebra A_B.

Example 3.4. The morphisms in our examples of class specifications are just inclusions. In the notation, we use the keywords **Parameter, Instance Interface, Class Interface, Import Interface, Body** to declare the subspecification to be added to the parts already defined. For example, since $PAR \subseteq EXP_i$, after the keyword **Instance Interface** only $EXP_i - PAR$ is listed. When a subspecification keyword is missing, the relative specification is just the closest subspecification in the diagram. In the instance interface the distinguished sort $pt(EXP_i)$ is indicated by the keyword *class sort*.

Next, the DSTACK class specification is defined. Such an abstract data type is a *double stack* which allows us to push and pop elements into/from two sides, denoted by front and rear side (see Fig. 1). Moreover, pushing an element into one side allows us to remove it from the other side, provided that there are no elements ahead, i.e., the front and rear operations are not totally independent. The DSTACK example also shows the role played by each component of the class specification. The instance interface describes the *abstract* properties, in the sense that the intended behavior is representation

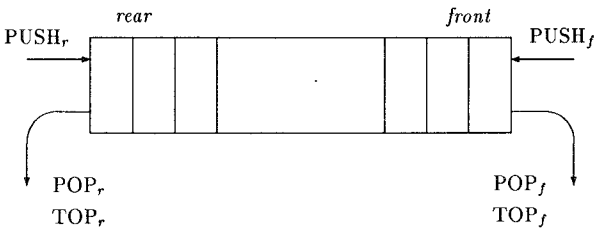

Fig. 1. The double stack abstract data type.

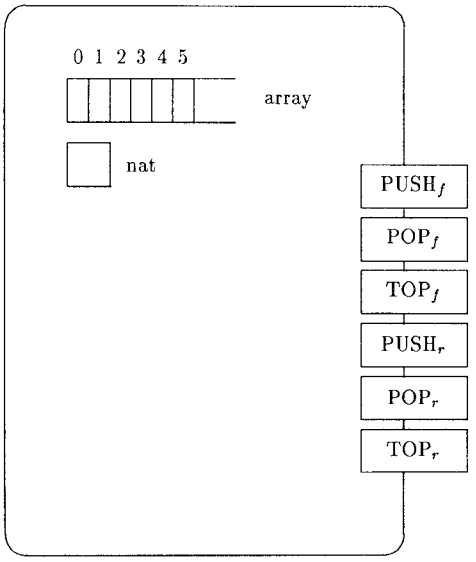

Fig. 2. The double stack.

independent. The body describes, in turn, how such data type is implemented (Fig. 2) in terms of what is required by the formal import. The constructor

$$\langle \_ , \_ \rangle : \text{array nat} \rightarrow \text{dstack}$$

describes the record consisting of the instance variables, and the double stack is realized by means of an array and a natural as a pointer. The equations in the body specify how the value of the *representation record* changes consistently with the export specification. Of course, if we wish to represent the stack in a different way, such a constructor will be defined accordingly, e.g., $\langle \_ \rangle$:list $\rightarrow$ dstack in case the representation is based on a linked list.

The operations SHIFT$_L$ and SHIFT$_R$ shift the array (required in the import) to the left and to the right, respectively, one position at time. The syntax uses the underscore as a placeholder for the arguments. For example, $\_[\_] := \_$:array nat data $\rightarrow$ array can be used on an array a with parameters n and x as in the term $a[n] := x$. The element $\perp : \rightarrow$ data is required since we need a unique representation for each stack, i.e., our semantical frame-

work does not allow us to implement the pop operations just by decreasing the pointer of the stack. This problem can be overcome by relying on our institution-independent formalism and adopting a behavioral semantics [Nivela and Orejas 1987].

DSTACK **is Class Specification**

Parameter

sort data

opns $\perp : \rightarrow$ data

Instance Interface

class sort dstack

opns EMPTY: $\rightarrow$ dstack

PUSH_f, PUSH_r:dstack data $\rightarrow$ dstack

POP_f, POP_r:dstack $\rightarrow$ dstack

TOP_f, TOP_r:dstack $\rightarrow$ data

eqns $\text{POP}_i(\text{PUSH}_i(s, x)) = s$ $\quad i \in \{r, f\}$

$\text{TOP}_i(\text{PUSH}_i(s, x)) = x$ $\quad i \in \{r, f\}$

$\text{POP}_i(\text{PUSH}_j(\text{EMPTY}, x)) = \text{EMPTY}$ $\quad i, j \in \{r, f\}$

$\text{TOP}_i(\text{PUSH}_j(\text{EMPTY}, x)) = x$ $\quad i, j \in \{r, f\}$

$\text{POP}_i(\text{PUSH}_j(\text{PUSH}_i(s, x_1), x_2)) = \text{PUSH}_j(s, x_2)$ $\quad i, j \in \{r, f\}, i \neq j$

$\text{TOP}_i(\text{PUSH}_j(\text{PUSH}_i(s, x_1), x_2)) = x_1$ $\quad i, j \in \{r, f\}, i \neq j$

Class Interface

sort nat

opns 0: $\rightarrow$ nat

$\_ + 1$:nat $\rightarrow$ nat

$\_$.HEAD:dstack $\rightarrow$ nat

eqns EMPTY.HEAD = 0

$(\text{PUSH}_i(s, x)).\text{HEAD} = s.\text{HEAD} + 1$ $\quad i \in \{r, f\}$

Import Interface

sorts array,nat

opns 0: $\rightarrow$ nat

$\_ + 1$:nat $\rightarrow$ nat

NIL: $\rightarrow$ array

$\_[\_] := \_$:array nat data $\rightarrow$ array

$\_[\_]$:array nat $\rightarrow$ data

SHIFT_L, SHIFT_R:array $\rightarrow$ array

eqns $\text{NIL}[i] = \perp$

$(a[i] := e)[j] =$ if $i = j$ then e else $a[j]$

$(a[i] := e_1)[i] = e_2 = a[i] := e_2$

$(\text{SHIFT}_L(a))[i] = a[i + 1]$

$(\text{SHIFT}_R(a))[i + 1] = a[i]$

Body

opns $\langle \_, \_ \rangle$:array nat: $\rightarrow$ dstack

eqns $\text{EMPTY} = \langle \text{NIL}, 0 \rangle$

$\text{PUSH}_f(\langle a, n \rangle, x) = \langle a[n] := x, n + 1 \rangle$

$\mathrm{TOP}_f(\langle a, n+1\rangle) = a[n]$
$\mathrm{POP}_f(\langle a, n+1\rangle) = \langle a[n] := \perp, n\rangle$
$\mathrm{PUSH}_r(\langle a, n\rangle, x) = \langle(\mathrm{SHIFT}_R(a))[0] := x, n+1\rangle$
$\mathrm{TOP}_r(\langle a, n\rangle) = a[0]$
$\mathrm{POP}_r(\langle a, n+1\rangle) = \langle \mathrm{SHIFT}_L(a), n\rangle$
$\langle a, n\rangle.\mathrm{HEAD} = n$

End DSTACK

Presenting this class model, we intend to cover a large number of class structures as they are defined in current object-oriented languages. We have focused on the importance of avoiding uncontrolled code reuse without any constraint, and therefore, in the proposed model, we provide an explicit import interface. None of the languages analyzed allows us to specify some requirements for the import, although some allow the direct importing of other existing classes, incorporating (with the *use* clause) a combination mechanism. The opportunity to hide some implementational aspects gives to a class designer the freedom to modify the implementation without affecting the clients of the instances of that class. All the languages analyzed but BETA have constructs for the protection of data representation. The set of all public operations of a class forms the external interface, which we call *instance interface*. Another form of protection is given to prevent another kind of client, the designer of a subclass, to access some variables. We have named this other interface, which contains the instance one, *class interface*. The C++, POOL, and Trellis/Owl languages have an explicit class interface, distinct from the instance interface. For instance, in the C++ language the instance interface consists of all public items which can be declared via a *public* clause, while the class interface includes both the public and the subclass-visible items, declared through a *protected* clause, accessible only to derived classes.

Encapsulation and inheritance are the major features of object-oriented methodology, but other techniques can as well enhance some quality factors. In Meyer [1986] an informal comparison between genericity and inheritance is presented. Genericity represents a good solution to achieve a good amount of flexibility with a static type system (untyped languages provide a great deal of flexibility, but the errors can be detected only at run-time). Many languages, such as Eiffel, Trellis/Owl, POOL, BETA, and OOZE, allow genericity, although with some differences. The only properties treated by these languages are signature properties; OBJ allows the specification of behavior with an equational language, by means of theories and views, while OOZE uses pre- and postconditions in the style of Z. All these languages supply an actualization mechanism in order to instantiate the generic classes.

In Table I we have indicated which of the components of our model of class are explicitly present in the notion of class in some of the analyzed languages. In this table we distinguish among unconstrained and constrained genericity (which are indicated by the *unconstr.* and *constr.* keywords, respectively).

Table I. Class Components

	BETA	C + +	Eiffel	POOL	Smalltalk	Trellis/Owl
Instance Interface	NO	YES	YES	YES	YES	YES
Class Interface	NO	YES	NO[a]	YES	NO[b]	YES
Genericity	constr.	NO	unconstr.[c]	constr.	NO	constr.
Import with requirements	NO[d]	NO	NO[d]	NO[d]	NO	NO[d]

[a]The class interface coincides with the implementation part.
[b]The class interface coincides with the instance interface.
[c]We refer to the version described in Meyer [1986; 1988].
[d]Although it is possible to import through actualization, there is no explicit import interface

The former kind of genericity does not allow operations on the generic type parameters while the latter does.

4. INHERITANCE

Inheritance is one of the main notions of the object-oriented paradigm. Its importance is widely recognized because it allows designers to reuse, extend, and combine abstractions in order to define other abstractions. It allows also the definition of new classes starting from the variables and methods of other classes. The usual terminology calls the former *subclasses* and the latter ones *superclasses*. Unfortunately, in the space of languages the notion of inheritance is not homogeneous since it ranges from functional specialization to the reuse of code without any constraint. We can consider inheritance as a technique for the implementation of an abstract data type and its use as a private decision of the designer of the inheriting class. The omission and/or shadowing of features can be reasonable. Through this mechanism we can arrange classes in hierarchies which describe how programs are structured: we call this technique *reusing inheritance*. On the other hand, we can consider inheritance as a technique for defining behaviorial specialization and its use as a public declaration of the designer that the instances of the subclass obey the semantics of the superclass. Thus each subclass instance is a special case of superclass instance: we call this kind of inheritance *specialization inheritance*. The hierarchies obtainable by means of the specialization inheritance contain two kinds of information. From one side they describe how code is distributed among classes and thus how programs are structured. On the other side they produce compatible assignment rules which are related with the subtyping relation: each context which expects a superclass instance can accept a subclass instance since the behavior of the subclass is at least that of the superclass.

We now present a formal distinction between these two different notions

Definition 4.1 (*Reusing Inheritance*). Let $C1 = (C1_{spec}, C1_{impl})$ and $C2 = (C2_{spec}, C2_{impl})$ be classes. Then

(1) $C2$ weakly reuses $C1$, notation $C2$ Wreuse $C1$, if there exists a morphism

$$f: EXP_{c1} \rightarrow BOD_2,$$

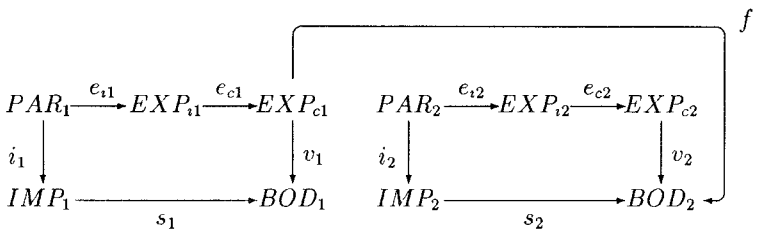

Fig. 3. Reusing inheritance.

(called reusing morphism) as in Figure 3, such that f is pointed, i.e.,

$$f^S(pt(EXP_{c1})) = v_2^S(pt(EXP_{c2}));$$

(2) $C2$ strongly reuses $C1$, notation $C2$ Sreuse $C1$, if, additionally,

$$V_f(C2_{impl}) = V_{v1}(C1_{impl}).$$

Interpretation. The morphism f from the export of $C1$ to the body of $C2$ indicates that the methods provided by $C1$ are used inside the class $C2$. The only constraint imposed is that the sort of interest of $C1$ coincides (in BOD_2) with the sort of interest of $C2$. There may or may not be a relationship between EXP_{c1} and EXP_{c2}, and all the "reused items" could be hidden and relabelled via v_2.

Example 4.2. In this example, we introduce a *counted stack*, i.e., an ordinary stack with an additional *count* operation which returns the number of elements present in the stack. The reusing inheritance allows the definition of such an abstract data type starting from DSTACK. Actually, we remove as visible attributes the rear operations, which remain anyway in the body of CSTACK, and introduce a new operation called *count*. Moreover, in the example, we show how the class interface visibility of DSTACK allows us to implement the count operation in the body of CSTACK. The front operations of DSTACK are renamed via the reusing morphism f:DSTACK.EXP_c → CSTACK.BOD, defined here.

f^S			f^{OP}		
			$\perp$	$\longmapsto$	$\perp$
			EMPTY	$\longmapsto$	EMPTY
			$PUSH_f$	$\longmapsto$	PUSH
data	$\longmapsto$	data	POP_f	$\longmapsto$	POP
dstack	$\longmapsto$	cstack	TOP_f	$\longmapsto$	TOP
nat	$\longmapsto$	nat	$PUSH_r$	$\longmapsto$	$PUSH_r$
			POP_r	$\longmapsto$	POP_r
			TOP_r	$\longmapsto$	TOP_r
			_.HEAD	$\longmapsto$	_.HEAD

Notice how the inherited features are defined from scratch in the body of the derived class while, for instance, the naturals are required in the formal import. The body describes how CSTACK is implemented. If the class interface did not have the *head* item we would be unable to implement the count

operation; in this case, we would have to introduce an extending representation record $\langle \_,\_ \rangle$:dstack nat → cstack and redefine EMPTY, PUSH, and POP in order to keep track of the inserted and removed elements on the additional natural. This procedure corresponds to adding an instance variable in a derived class.

CSTACK **is Class Specification**
Parameter
  sort data
  opns $\perp$: → data
Instance Interface
  class sort cstack
  sort nat
  opns 0: → nat
  _ + 1:nat → nat
  EMPTY: → cstack
  PUSH:cstack data → cstack
  POP:cstack → cstack
  TOP:cstack → nat
  COUNT:cstack → nat
  eqns POP(PUSH(s, x)) = s
  TOP(PUSH(s, x)) = x
  COUNT(EMPTY) = 0
  COUNT(PUSH(s, x)) = COUNT(s) + 1
Class Interface
  opns _.HEAD:stack → nat
  eqns EMPTY.HEAD = 0
  (PUSH(s, x)).HEAD = s.HEAD + 1
Import Interface
  sort nat
  opns _ + 1:nat → nat
  Body
  opns PUSH_r:cstack data → cstack
  POP_r:cstack → cstack
  TOP_r:cstack → nat
  _.HEAD:cstack → nat
  eqns $\text{POP}_r(\text{PUSH}_r(s, x)) = s$
  $\text{TOP}_r(\text{PUSH}_r(s, x)) = x$
  POP(PUSH_r(EMPTY, x) = EMPTY
  POP_r(PUSH(EMPTY, x)) = EMPTY
  TOP(PUSH_r(EMPTY, x)) = x
  TOP_r(PUSH(EMPTY, x)) = x
  POP(PUSH_r(PUSH(s, x_1), x_2)) = $\text{PUSH}_r(s, x_2)$
  POP_r(PUSH($\text{PUSH}_r(s, x_1, x_2)$) = PUSH($s$, x_2)
  TOP(PUSH_r(PUSH(s, x_1), x_2)) = x_1
  TOP_r(PUSH($\text{PUSH}_r(s, x_1), x_2$)) = x_1
  EMPTY.HEAD = 0
  (PUSH(s, x)).HEAD = s.HEAD + 1
  ($\text{PUSH}_r(s, x)$.HEAD = s.HEAD + 1
  COUNT(s) = s.HEAD
End CSTACK

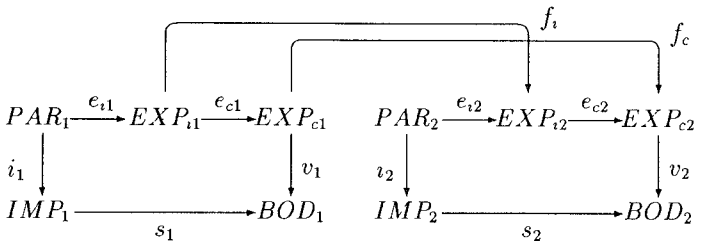

Fig. 4. Specialization inheritance.

The second notion of inheritance is that of specialization inheritance which allows the enrichment of the functionalities of a class and can be modeled by morphisms from the inherited class to the inheriting class in such a way that behavior is preserved.

Definition 4.3 (*Specialization Inheritance*). Let $C1 = (C1_{spec}, C1_{impl})$ and $C2 = (C2_{spec}, C2_{impl})$ be classes. Then

(1) $C2$ is a weak specialization of $C1$, notation $C2$ Wspec $C1$, if there exist pointed morphisms (called specialization morphisms)

$$f_i: EXP_{i1} \to EXP_{i2}, f_c: EXP_{c1} \to EXP_{c2},$$

such that $e_{c2} \circ f_i = f_c \circ e_{c1}$, as in the commutative Figure 4

(2) $C2$ is a strong specialization of $C1$, notation $C2$ Sspec $C1$, if, additionally,

$$V_{f_c}(V_{v2}(C2_{impl})) = V_{v1}(C1_{impl}).$$

Interpretation. The morphisms f_i and f_c indicate that the class and instance interfaces of the specialization $C2$ must contain all the methods—with the same properties—of $C1$ and possibly new auxiliary sorts, new methods on old sorts, and properties for both old and new methods. The sort of interest must be maintained. If f_i and f_c determine a renaming, it is required that it be done consistently.

Example 4.4. In order to illustrate the specialization inheritance we consider the following STACK which is implemented as the DSTACK by an array and a pointer

STACK **is Class Specification**
Parameter
  sort data
  opns $\perp : \to$ data
Instance Interface
  class sort stack
  opns EMPTY: $\to$ stack
  PUSH:stack data $\to$ stack
  POP:stack $\to$ stack
  TOP:stack $\to$ data

```
eqns      POP(PUSH(s, x)) = s
          TOP(PUSH(s, x)) = x

Class Interface
  sort    nat
  opns    0: → nat
          _ + 1:nat → nat
          _.HEAD:stack → nat
  eqns    EMPTY:HEAD = 0
          PUSH(s, x)).HEAD = s.HEAD + 1
Import Interface
  sorts   array,nat
  opns    0: → nat
          _ + 1:nat → nat
          NIL: → array
          _[_]:= _:array nat data → array
          _[_]:array nat → data
  eqns    NIL[k] = ⊥
          (a[i]:= e)[j] = if i = j then e else a[j]
          (a[i]:= e1)[i]:= e2 = a[i]:= e2
Body
  opns    ⟨_,_⟩:array nat: → stack
  eqns    EMPTY = ⟨NIL, 0⟩
          PUSH(⟨a, n⟩, x) = ⟨a[n]:= x, n + 1⟩
          TOP(⟨a, n + 1⟩) = a[n]
          POP(⟨a, n + 1⟩) = ⟨a[n] = ⊥ , n⟩
          ⟨a, n⟩.HEAD = n
End STACK
```

Now we can define a CSTACK (specializing STACK) as follows with the inclusions as specialization morphisms:

```
CSTACK is Class Specification
Parameter
  sort    data
  opns    ⊥ : → data
Instance Interface
  class sort cstack
  sort    nat
  opns    0: → nat
          _ + 1:nat → nat
          EMPTY: → cstack
          PUSH:cstack data → cstack
          POP:cstack → cstack
          TOP:cstack → nat
          COUNT:cstack → nat
  eqns    POP(PUSH(s, x)) = s
          TOP(PUSH(s, x)) = x
```

```
            COUNT(EMPTY) = 0
            COUNT(PUSH(s, x)) = COUNT(s) + 1
Import Interface
  sort      nat
  opns      _ + 1:nat → nat
Body
  opns      _.HEAD:stack → nat
  eqns      EMPTY.HEAD = 0
            (PUSH(s, x)).HEAD = s.HEAD + 1
            (PUSH_r(s, x)).HEAD = s.HEAD + 1
            COUNT(s) = s.HEAD
End CSTACK
```

Remark 4.5. In general, the inheritance at the specification level is twofold since it can be instantiated at the specification level and at the code level. For instance, in the specialization inheritance the incremental requirement of properties and/or operations in the subclass implies the refinement of the set of all models of the superclass. This is captured by the weak specialization and is mainly intended to support monotonic decision steps in the software design process. On the other hand, each of the models of the superclass may be related to a specific model of the subclass which is a subalgebra of the superclass model. Such a pointwise relation among algebras (abstract implementations) corresponds to the inheritance relation at the programming language level.

Although both are called inheritance, there is no confusion between the idea of code sharing and the notion of functional specialization. Notice how, in general, reusing inheritance does not create correctness problems when its use is intended: unfortunately, its use can be also unintended. Indeed, although the redefinition of methods may be useful to refine inherited operations, it can be, at the same time, dangerous: in fact, the redefinition can violate the invariants of the superclass obtaining something with an unexpected behavior. In other words, the intended specialization inheritance degenerates in reusing inheritance while the designer is expecting to use the subclass as a subtype, i.e., as something with a compatible behavior.

This is particularly true for C++ and Smalltalk but also for Eiffel, even if it allows the specification of the methods by pre- and postconditions. The BETA programming language does not allow the redefinition of methods and gives the possibility to extend in a subclass the old method defined in a superclass. The extension consists of a portion of code specified in the subclass. When a message is received by a class instance, the method executed is from the topmost superclass containing the message; the execution can, in turn, trigger another execution (by the imperative *inner*) from a subclass, and so on. Unfortunately there is no constraint in providing the methods extension, and this could violate the invariants of the superclass as well.

The two inheritance relations satisfy some properties formalized in the following propositions. The first one justifies our use of the adjectives strong and weak.

PROPOSITION 4.6. *The strong relations* Sreuse *and* Sspec *imply the weak relations* Wreuse *and* Wspec, *respectively.*

PROOF. It is straightforward from the definitions of the relations. □

The next one shows that the specialization relation is stronger than the corresponding reuse relation. It is easy to construct an example to show that the coverse in not true.

PROPOSITION 4.7. *Let* $C1 = (C1_{spec}, C1_{impl})$ *and* $C2 = (C2_{spec}, C2_{impl})$ *be classes. Then*

(1) *if* $C2$ Wspec $C1$ *then* $C2$ Wreuse $C1$
(2) *if* $C2$ Sspec $C1$ *then* $C2$ Sreuse $C1$

PROOF.

(1) By definition there exists a pointed morphism $f_c: EXP_{c1} \to EXP_{c2}$. Then $f = v_2 \circ f_c: EXP_{c1} \to BOD_2$ is the reusing morphism between $C2$ and $C1$ since

$$\begin{aligned} f^S(pt(EXP_{c1})) &= v_2^S(f_c^s pt(EXP_{c1})) \\ &= v_2^S(pt(EXP_{c2})). \end{aligned}$$

Thus $C2$ Wreuse $C1$.

(2) Since the Sspec relation implies the Wspec one, $C2$ Wreuse $C1$ by the previous point via the morphism $f = v_2 \circ f_c$. If, additionally, $V_{fc}(V_{v2}(C2_{impl}) = V_{v1}(C1_{impl})$ then

$$\begin{aligned} V_{v1}(C1_{impl}) &= (V_{fc} \circ V_{v2})(C2_{impl}) \\ &= V_{v2 \circ f_c}(C2_{impl}) \\ &= V_f(C2_{impl}). \end{aligned}$$ □

In the next proposition, we show that specialization is transitive and that reuse (obviously not transitive) satisfies a similar *approximate* property.

PROPOSITION 4.8. *Let* $C1 = (C1_{spec}, C1_{impl})$, $C2 = (C2_{spec}, C2_{impl})$ *and* $C3 = (C3_{spec}, C3_{impl})$ *be classes.*

(1) *Each of the two relations* Sspec *and* Wspec *is transitive*;
(2) *if* $C3$ Wreuse $C2$ *and* $C2$ Wspec $C1$ *then* $C3$ Wreuse $C1$;
(3) *if* $C3$ Sreuse $C2$ *and* $C2$ Sspec $C1$ *then* $C3$ Sreuse $C1$.

PROOF.

(1) If $C3$ Wspec $C2$ and $C2$ Wspec $C1$, then there exist morphisms

$$\begin{aligned} f_c': EXP_{c2} &\to EXP_{c3} \qquad & f_i': EXP_{i2} &\to EXP_{i3} \\ f_c'': EXP_{c1} &\to EXP_{c2} \qquad & f_i'': EXP_{i1} &\to EXP_{i2}. \end{aligned}$$

such that

$$e_{c3} \circ f'_{\iota} = f'_{c} \circ e_{c2}$$

and

$$e_{c2} \circ f''_{\iota} = f''_{c} \circ e_{c1}.$$

It is easy to check that $C3$ Wspec $C1$ via the morphisms $f_c = f'_c \circ f''_c$ and $f_\iota = f'_\iota \circ f''_\iota$. If $C3$ Sspec $C2$ and $C2$ Sspec $C1$, then, additionally,

$$V_{f'_c}\big(V_{v3}(C3_{\iota mpl})\big) = V_{v2}(C2_{\iota mpl})$$

and

$$V_{f''_c}\big(V_{v2}(C2_{\iota mpl})\big) = V_{v1}(C1_{\iota mpl}).$$

Then, with f_c and f_ι as above

$$\begin{aligned} V_{fc}\big(V_{v3}(C3_{\iota mpl})\big) &= V_{f''_c}\big((V_{f'_c}C3_{\iota mpl})\big) \\ &= V_{f''_c}\big(V_{v2}(C2_{\iota mpl})\big) \\ &= V_{v1}(C1_{\iota mpl}) \end{aligned}$$

and therefore $C3$ Sspec $C1$.

(2) If $C3$ Wreuse $C2$ and $C2$ Wspec $C1$ then there exist morphisms

$$f' : EXP_{c2} \to BOD_3$$
$$f''_c : EXP_{c1} \to EXP_{c2}.$$

The morphism $f = f' \circ f''_c$ is the reusing morphism between $C3$ and $C1$ since

$$\begin{aligned} f^S(pt(EXP_{c1})) &= f'^S\big(f''^S_c(pt(EXP_{c1}))\big) \\ &= f'^S(pt(EXP_{c2})) \\ &= v_3^S(pt(EXP_{c3})) \end{aligned}$$

Thus $C3$ Wreuse $C1$.

(3) Since Sreuse and Sspec imply Wreuse and Wspec, respectively, $C3$ Wreuse $C1$ by the previous point. Moreover

$$V_{f'}(C3_{\iota mpl}) = V_{v2}(C2_{\iota mpl})$$

and

$$V_{f''_\iota}\big(V_{v2}(C2_{\iota mpl})\big) = V_{v1}(C1_{\iota mpl}).$$

Then, with $f = f' \circ f''_c$ as above,

$$\begin{aligned} V_f(C3_{\iota mpl}) &= V_{f''_\iota}\big(V_{f'}(C3_{\iota mpl})\big) \\ &= V_{f''_c}\big(V_{v2}(C2_{\iota mpl})\big) \\ &= V_{v1}(C1_{\iota mpl}) \end{aligned}$$

and therefore $C3$ Sreuse $C1$ □

Another important notion is that of a *virtual class* which is formalized in the following definition. Virtual class is not intended in the same sense of the BETA language where it represents the parameter part of a class. It is virtual in the sense that it does not include an import part *IMP* and the implementation *BOD* (relative to *IMP*). It resembles the use of the word virtual in *C++*.

Definition 4.9. A *virtual class* V consists of a triple (PAR, EXP_i, EXP_c) of algebraic specifications, a specification morphisms $e_i: PAR \to EXP_i$, and a pointed specification morphism $e_c: EXP_i \to EXP_c$ as in Figure 5.

The following result shows that it suffices to consider specialization inheritance as the only relation generating a hierarchy of classes. The reuse relation can be obtained as the composition of the specialization relation and its (set-theoretic) inverse.

THEOREM 4.10. *Let* $C1 = (C1_{spec}, C1_{impl})$ *and* $C2 = (C2_{spec}, C2_{impl})$ *be classes. If* $C2$ Wreuse $C1$ *then there exists a virtual class* C *such that*

(1) $C1$ Wspec C;

(2) $C2$ Wspec C.

PROOF. Since $C2$ Wreuse $C1$, there exists a reusing morphism $f: EXP_{c1} \to BOD_2$. Then define PAR, EXP_i, and EXP_c as the specification obtained by the pullback of the morphisms f and v_2, $v_2 \circ e_{c2}$ and $f \circ e_{c1}$, and $v_2 \circ e_{c2} \circ e_{i2}$ and $f \circ e_{c1} \circ e_{i1}$, respectively, as in Figure 6. The morphisms $e_i: PAR \to EXP_i$ and $e_c: EXP_i \to EXP_c$ are uniquely defined by the universal property of pullbacks. The specifications PAR, EXP_i, and EXP_c and the morphisms e_i and e_c define a virtual class C. Moreover, by the construction of the pullbacks, there exist induced morphisms $f_i': EXP_i \to EXP_{i1}$, $f_i'': EXP_i \to EXP_{i2}$, $f_c': EXP_c \to EXP_{c1}$, and $f_c'': EXP_c \to EXP_{c2}$. Thus $C1$ Wspec C and $C2$ Wspec C. Notice that since

$$f^S(pt(EXP_{c1})) = v_2^S(pt(EXP_{c2}))$$

and the morphism $e_{i1}, e_{i2}, e_{c1}, e_{c2}$ are pointed, so are the specifications EXP_c, EXP_i, and PAR and the morphisms $f_c', f_i', f_p', f_c'', f_i'', f_p''$. □

Remark 4.11. In case $C2$ Sreuse $C1$, it is possible to construct a *class* C such that $C1$ Sspec C and $C2$ Sspec C. We can, in fact, consider BOD_2 and IMP_2 as body part and import interface, respectively, and take as implementation C_{impl} the algebra $C2_{impl}$. As a rule, there are several ways to *complete* the virtual class: for instance the parameter part and the class interface are other candidates, and they are minimal with respect to $C1$ and $C2$.

By using the specialization inheritance (which implies subtyping) it is possible to represent the implementation inheritance (which usually reflects the development over time of the system) and keep the correctness under control. On the other hand, the theorem allows us to consider only monotonic decision steps (via specialization inheritance) since omitting properties and/or operations, i.e., non monotonic steps, can be reduced to monotonic ones.

$PAR \xrightarrow{e_t} EXP_t \xrightarrow{e_c} EXP_c$ Fig. 5. Virtual class.

Example 4.12. We already know how the CSTACK can be implemented from DSTACK via reusing inheritance. The theorem assures the existence of a class, say VSTACK, which is (weakly) specialized by DSTACK and CSTACK

VSTACK **is Class Specification**
Parameter
  sort data
  opns $\perp : \rightarrow$ data
Instance Interface
  class sort vstack
  opns EMPTY: $\rightarrow$ vstack
  PUSH:vstack data $\rightarrow$ vstack
  POP:vstack $\rightarrow$ vstack
  TOP:vstack $\rightarrow$ data
  eqns POP(PUSH(s, x)) = s
  TOP(PUSH(s, x)) = x
End VSTACK

The specialization morphism and the morphisms f'_p and f''_p are inclusions.

As pointed out earlier, strong specialization should imply subtyping. Up to now, such a notion has not been formally introduced. As usual, the terminology in the literature is sometimes misleading although it is somehow an intuitive concept. For instance, Goguen, and later Breu [1991], calls subtyping what is called strong specialization here.

The subtyping relationship is a more general relation than strong specialization since the latter implies the former but not vice versa. Furthermore, subtyping may hold also without inheritance since the external behavior should be representation independent. Some languages like POOL and Trellis/Owl allow the explicit definition of subtyping statements even if the classes are defined in different inheritance hierarchies. This allows to define different classes for different representations of stacks or complex numbers, for instance, and consider them as being of the same type.

In Wegner and Zdonik [1988] different definitions of subtyping are given. They are mainly distinguished into a *subset compatibility*, where the set of subclass instances is a subset of the set of superclass instances, and *subcomplete compatibility*, which, additionally, requires that the operations in the subclass behave as the corresponding operations in the superclass restricted to the instances of the subclass. The latter notion leads to the subalgebra concept which, naturally, captures such a subtype notion. We consider the subalgebra-based subtyping better behaved than the subset-based one since it is operation closed and guarantees that the subtype has a predictable behavior. For instance, if we consider the naturals modulo 8 as (subset-based) subtype of naturals, it is possible to have division by 0 whenever there is an attempt at dividing a natural by the successor of 7, which is 8 for the naturals but 0 for the considered subtype.

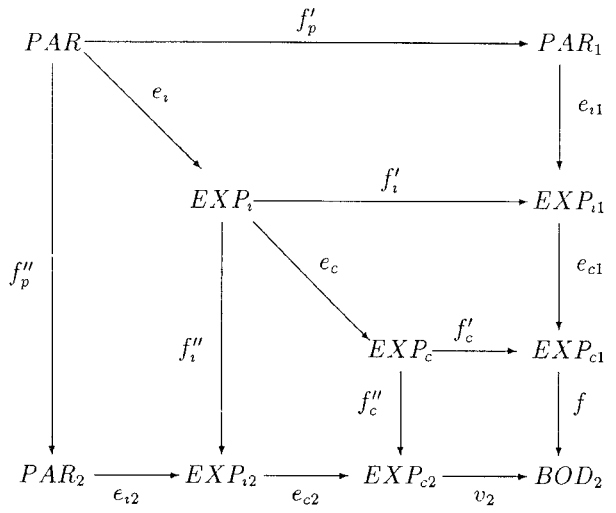

Fig. 6. Theorem diagram.

As a rule, the subtyping relation should guarantee that the carrier set of the subclass sort is an operation-closed subset of the carrier set of the superclass sort while the other carriers are left unchanged. This definition would be too restrictive, not allowing other carrier sets to be reduced and therefore causing an incompatibility between actualization and subtyping. For instance, if we consider strings and strings without repetitions, the latter is a subtype of the former, and it is reasonable to consider a stack of strings without repetitions as a subtype of a stack of strings. In order to have such an induced subtyping, we need a more relaxed definition of subtyping where also other carrier sets can be restricted.

Definition 4.13. Let $C1$ and $C2$ be classes and $\sigma: Sig(EXP_{i1}) \rightarrow Sig(EXP_{i2})$ be a signature morphism, then $C2$ is a subtype of $C1$ if

$$V_\sigma\big(V_{e_{c2}}\big(V_{v2}(C2_{impl})\big)\big)_s \subseteq V_{e_{c1}}\big(V_{v1}(C1_{impl})\big)_s$$

for $s \in sorts(EXP_{i1})$.

Interpretation. The signature morphism σ requires that the instance interface of $C2$ contains all the methods of the instance interface of $C1$, although their properties may not be formally preserved.

The next result shows that one way to obtain a subtype is by specialization inheritance.

THEOREM 4.14. *Let* $C1 = (C1_{spec}, C1_{impl})$ *and* $C2 = (C2_{spec}, C2_{impl})$ *be classes. If* $C2$ Sspec $C1$ *then* $C2$ *is a subtype of* $C1$.

PROOF. If $C2$ Sspec $C1$, then

$$V_{f_c}\big(V_{v2}(C2_{impl})\big) = V_{v1}(C1_{impl})$$

for f_c and f_t such that $e_{c2} \circ f_t = f_c \circ e_{c1}$. Let σ be the specification morphism f_t restricted to $Sig(EXP_{t1})$. Then

$$V_\sigma\left(V_{e_{c2}}\left(V_{v2}(C2_{impl})\right)\right) = V_{e_{c1}}\left(V_{fc}\left(V_{v2}(C2_{impl})\right)\right)$$
$$= V_{e_{c1}}\left(V_{v1}(C1_{impl})\right)$$

and in particular

$$V_\sigma\left(V_{e_{c2}}\left(V_{v2}(C2_{impl})\right)\right)_s \subseteq V_{e_{c1}}\left(V_{v1}(C1_{impl})\right)_s$$

for all s. □

5. STRUCTURED INHERITANCE

In this section, we define two additional relations between class specifications and between classes (weak and strong relations, respectively) which describe mechanisms for interconnecting class specifications and define new classes, which inherit from the old ones. The two relations correspond to instantiating the parameter part of a (generic) class and to replacing the import interface of a (*semivirtual*) class with the class export of another class.

Definition 5.1 (*Actualizable*). Given classes $C1 = (C1_{spec}, C1_{impl})$ and $C2 = (C2_{spec}, C2_{impl})$ with $PAR_2 = IMP_2$

(1) $C1$ is weakly actualizable by $C2$, denoted by $C1$ Wact $C2$, if there exists a specification morphism $f: PAR_1 \rightarrow EXP_{t2}$

(2) $C1$ is strongly actualizable by $C2$, denoted by $C1$ Sact $C2$, if $C1$ Wact $C2$ with f and $V_f(V_{e_{c2}}(V_{v2}(C2_{impl}))) = V_{e_{c1}}(V_{e_{c1}}(V_{v1}(C1_{impl})))$.

Interpretation. The binary relation Wact indicates that the parameter part of the class $C1$ can be replaced by the instance interface of the class $C2$, i.e., that the instances of class $C2$ satisfy the constraints of the parameter of $C1$. The distinction between weak and strong again separates the class specification from the class. In $C1$ Wact $C2$, a realization of the class specification $C2_{spec}$ *can* be used for the parameter part of a realization of the class specification $C1_{spec}$. On the other hand, $C1$ Sact $C2$ indicates that the realizations $C1_{impl}$ and $C2_{impl}$ of the class specification *coincide* on their PAR_1 part.

The restriction imposed on the class $C2$ that $PAR_2 = IMP_2$ is both technical and methodological. On the technical side, it makes the construction of the resulting class specification after replacing the parameter part cleaner. On the methodological side, it requires that only *complete* classes, i.e., classes which do not rely on other classes for their completion of the import specification, be considered as actual parameters of generic classes. By allowing the actualizing class to have a nonempty parameter, we do not restrict the actualizing steps to be bottom up, but we are able to construct, for example, *strings of*(*sets of*(*bags of nat*)) from *strings of data* actualized with *sets of elements* and then actualizing the result with *bagsofnat*, with more flexibility on the strategy.

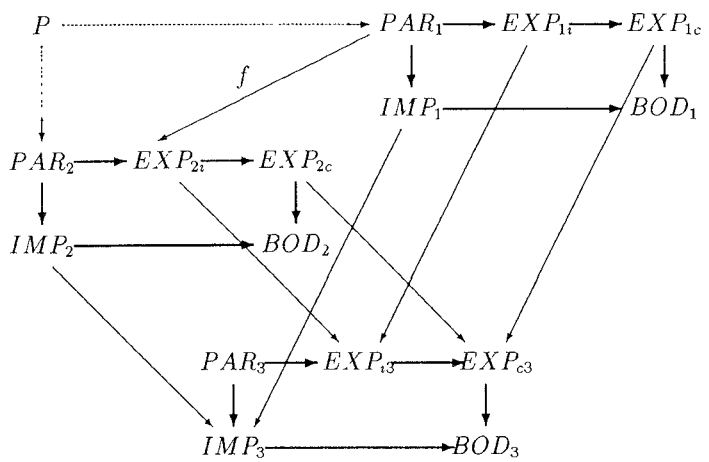

Fig. 7. Actualization.

We now describe the result of actualizing the parameters part PAR_1 of $C1$ by $C2$. The categorical constructions and the detailed proofs of syntactical and semantical properties are also provided.

Definition 5.2 (*Parameter Passing*). Let $Ci = (Ci_{spec}, Ci_{impl})$, $i = 1, 2$ be classes with $C1$ Wact $C2$ via $f: PAR_1 \rightarrow EXP_{i2}$. The actualization of $C1$ by $C2$, denoted by $ACT(C1, f, C2)$, is the class specification $C3_{spec}$ as in Figure 7 (where the pushout morphisms of BOD are omitted for clarity) with

—the parameter part PAR_3 is the parameter part PAR_2 of $C2$;
—the instance interface EXP_{i3} is the union of EXP_{i1} and EXP_{i2} (i.e., the pushout of f and e_{i1});
—the class interface EXP_{c3} is the union of EXP_{c1} and EXP_{c2} (i.e., the pushout of $f \circ e_{c2}$ and $e_{i1} \circ e_{c1}$);
—the import interface IMP_3 is the union of IMP_1 and PAR_2 (i.e., the pushout of the two with respect to the intersection of PAR_1 and PAR_2);
—the implementation part BOD_3 is the union of BOD_1 and BOD_2 with respect to PAR_1.

The specification morphisms are induced by the universal properties of the pushout objects. The distinguished sorts of EXP_{i3}, EXP_{c3}, and BOD_3 are the ones inherited from EXP_{i1}, EXP_{c1}, and BOD_1, respectively, in the union construction.

Interpretation. The new class specification $ACT(C1, f, C2)$ is obtained by *replacing* the parameter part PAR_1 in EXP_{i1} and EXP_{i2} with the instance and class interface, respectively, of $C2_{spec}$, and in BOD_1 with the implementation part BOD_2. The new parameter part is just the parameter PAR_2 of $C2$, which is also added to IMP_1 to obtain the new import interface.

Now we show why actualization has been called a clean operation: the semantics of the result can be expressed explicitly by using the semantics of the actualizing class $C2$. The proof, left to the reader, is an application of the Amalgamation Lemma in the Appendix.

THEOREM 5.3 (INDUCED SEMANTICS). *The semantics of the class specification $ACT(C1, f, C2)$ is the set of all pairs $(A_{I_3}, A_{E_{c3}})$ such that:*

—$A_{I_3} = A_{I_2} +_{A_P} A_{I_1}$
—$A_{E_{c3}} = A_{E_{c2}} +_{A_{P1}} A_{E_{c1}}$
—$(A_{I_1}, A_{E_{c1}}) \in SEM(C1_{spec})$
—$(A_{I_2}, A_{E_{c2}}) \in SEM(C2_{spec})$
—$V_{e_{i1}}(V_{e_{c1}}(A_{E_{c1}})) = V_f(V_{e_{c2}}(A_{E_{c2}})) = A_{P_1}$
—$V_{p_2}(V_{i2}(A_{I_2})) = V_{p_1}(V_{i1}(A_{I_1})) = A_P$

for some $A_{I_j} \in Alg(IMP_j)$ and $A_{E_{cj}} \in Alg(EXP_{cj})$.

The next two theorems show (as expected) that the result of actualizing a class $C1$ is a new class which inherits by specialization from $C1$.

THEOREM 5.4 (INDUCED INHERITANCE). *If $C1$* Wact *$C2$ via f, then $ACT(C1, f, C2)$* Wspec *$C1$.*

PROOF. The result follows from the existence of the induced morphisms $f_{i1}: EXP_{i1} \to EXP_{i3}$, $f_{c1}: EXP_{c1} \to EXP_{c3}$ with the appropriate commutativity guaranteed by the pushout properties in the category *CATSPEC*. Notice that both morphisms must be pointed. □

If the relation between $C1$ and $C2$ is strong, then it is possible to use the realization of $C1$ and $C2$ to construct a realization of $ACT(C1, f, C2)$ as the next result shows.

THEOREM 5.5 (INDUCED STRONG INHERITANCE). *If $C1$* Sact *$C2$ via f, then there exists an algebra $C3_{impl} \in Alg(BOD_3)$ such that*

(1) *$(ACT(C1, f, C2), C3_{impl})$ is a class (still denoted by $ACT(C1, f, C2)$)*
(2) *$ACT(C1, f, C2)$* Sspec *$C1$.*

PROOF. $C1$ Sact $C2$ implies that $C1$ Wact $C2$, then $ACT(C1, f, C2)$ Wspec $C1$ by Theorem 5.4. The induced morphisms $f_1: BOD_1 \to BOD_3$, $f_2: BOD_2 \to BOD_3$, $f_{c_1}: EXP_{c1} \to EXP_{c3}$ and $f_{c_2}: EXP_{c_2} \to EXP_{c3}$ satisfy $v_3 \circ f_{c2} = f_2 \circ v_2$ and $v_3 \circ f_{c_1} = f_1 \circ v_1$ by the universal property of pushouts. By definition, BOD_3 is the specification obtained from the pushout of

$$v_2 \circ e_{c2} \circ f: PAR_1 \to BOD_1$$

and

$$v_1 \circ e_{c2} \circ e_{i1}: PAR_1 \to BOD_1.$$

Since by assumption $C1$ Sact $C2$ via f, then for some PAR_1-algebra A

$$V_f\big(V_{e_{c2}}\big(V_{v_2}(C2_{impl})\big)\big) = A$$
$$= V_{e_{i1}}\big(V_{e_{c1}}\big(V_{v_1}(C1_{impl})\big)\big).$$

Define $C3_{impl}$ as the amalgamation sum of $C1_{impl}$ and $C2_{impl}$ with respect to A. By definition, $C3_{impl}$ is a BOD_3-algebra. Furthermore

$$\begin{aligned} V_{f_{c1}}\big(V_{v_3}(C3_{impl})\big) &= V_{v_3 \circ f_{c1}}(C3_{impl}) \\ &= V_{f_1 \circ v_1}(C3_{impl}) \\ &= V_{v_1}\big(V_{f_1}(C3_{impl})\big) \\ &= V_{v_1}(C1_{impl}). \end{aligned}$$

Hence $C3$ Sspec $C1$. □

In Meyer [1986] it is shown how inheritance can simulate genericity, by allowing generic classes in an object-oriented language. But the price to pay for this simulation is recognized to be high due to some difficulties with static type checking; our model allows us to specify some of the properties of the generic type parameters, while in most of the languages analyzed only signatures can be specified.

Example 5.6. OBJ3 allows the specification of behavior through equational logics, and the parameter properties are formalized in a theory. The actualization of a parametric class consists of providing a view, which represents a signature morphism, between the theory and the actual parameter. If the properties of the theory (parameter part) are derivable from those of the formal parameter, up to the renaming determined by the provided view, the result of actualization becomes the pushout object between the theory and the actual parameter.

We are now able to prove that our notion of subtyping is compatible with the mechanism of actualization. In particular we show that if $C2$ is a subtype of $C1$, then actualizing C with $C2$ yields a subtype of the actualization of C with $C1$.

THEOREM 5.7 (SUBTYPING COMPATIBILITY). *Let C, $C1$, and $C2$ be classes. If $C2$ is a subtype of $C1$ via σ, C* Sact *$C2$ via f_2, C* Sact *$C1$ via f_1 and $f_2 = \sigma \circ f_1$ (as signature morphisms), then $ACT(C, f_2, C2)$ is a subtype of $ACT(C, f_1, C1)$.*

PROOF. For notational convenience, let $C3 = ACT(C, f_1, C1)$ and $C4 = ACT(C, f_2, C2)$. From the universal property (see Appendix) of the pushouts used in the construction of $C3$ and $C4$, there exists a unique signature morphism $\tau: Sig(EXP_{\iota 3}) \to Sig(EXP_{\iota 4})$ with the usual commutativity properties.

We need to show that for $s \in sorts(EXP_{\iota 3})$

$$V_\tau\big(V_{e_{c4}}\big(V_{v4}(C4_{impl})\big)\big)_s \subseteq V_{e_{c3}}\big(V_{v3}(C3_{impl})\big)_s.$$

By Theorem 5.3 and the fact that C is strongly actualized by both $C1$ and $C2$, we have for the same $A \in Alg(PAR)$

$$C3_{impl} = C_{impl} +_A C1_{impl} \quad \text{and} \quad C4_{impl} = C_{impl} +_A C2_{impl}.$$

By the uniqueness of Amalgamation (see Appendix)

$$V_{e_{c3}}\big(V_{v3}(C3_{impl})\big) = V_{e_c}\big(V_v(C_{impl})\big) +_A V_{e_{c1}}\big(V_{v1}(C1_{impl})\big)$$

and

$$V_{e_{c4}}\big(V_{v4}(C4_{impl})\big) = V_{e_c}\big(V_v(C_{impl})\big) +_A V_{e_{c2}}\big(V_{v2}(C2_{impl})\big).$$

Now, by definition of $C3$, if $s \in sorts(EXP_{i3})$, then either $s \in sorts(EXP_i)$ or $s \in sorts(EXP_{i1})$ (or both). In the former case, what we need to prove reduces to

$$V_{e_c}\big(V_v(C_{impl})\big)_s \subseteq V_{e_c}\big(V_v(C_{impl})\big)_s$$

which holds trivially; in the latter case, it reduces to

$$V_\sigma\big(V_{e_{c2}}\big(V_{v2}(C2_{impl})\big)\big)_s \subseteq V_{e_{c1}}\big(V_{v1}(C1_{impl})\big)_s$$

which holds since by assumption $C2$ is a subtype of $C1$. □

The other relation that we are going to introduce relates a class $C2$, viewed as *producer* of its class interface, with another class $C1$, viewed as *consumer* of its import interface. Again we distinguish between a potential producer (weak notion) and a factual producer (strong notion).

Definition 5.8 (*Combinable*). Given classes $C1 = (C1_{spec}, C1_{impl})$ and $C2 = (C2_{spec}, C2_{impl})$

(1) $C1$ is weakly combinable with $C2$, denoted by $C1$ Wcomb $C2$, if there exists a specification morphism $h: IMP_1 \to EXP_{c2}$;
(2) $C1$ is strongly combinable with $C2$, denoted by $C1$ Scomb $C2$, if $C1$ Wcomb $C2$ via h and

$$V_h\big(V_{v_2}(C2_{impl})\big) = V_s(C1_{impl}).$$

Interpretation. The relation Wcomb indicates that $C2$ can provide, through its class interface, the data and operations needed in the import interface of $C1$. The strong counterpart Scomb indicates that the specific realization of the class $C2$ provides exactly the import part of the chosen realization of $C1$.

The availability of a class $C2$ which can be combined to a class $C1$ determines an interconnection of classes equivalent to a single class which we are going to define now. The construction is similar to the composition of module specifications in Blum et al. [1987].

Definition 5.9 (*Import Passing*). Let $Ci = (Ci_{spec}, Ci_{impl})$, $i = 1, 2$, be classes with $C1$ Wcomb $C2$ via $h: IMP_1 \to EXP_{c2}$. The combination of $C1$ and $C2$, denoted by $COMB(C1, h, C2)$, is the class specification $C3_{spec}$ as in Figure 8,

—EXP_{i3} and EXP_{c3} are just EXP_{i1} and EXP_{c1}, respectively.
—IMP_3 is just IMP_2.

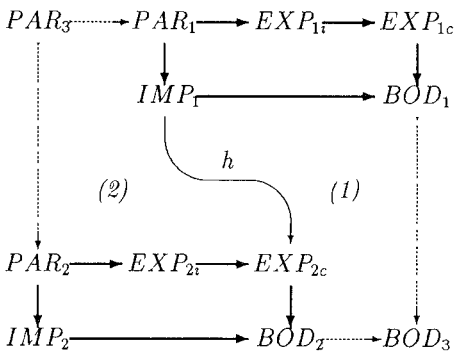

Fig. 8. Combination.

—the new implementation part is the union (pushout) of BOD_1 and BOD_2 with respect to IMP_1, with distinguished sort the one inherited from BOD_1.

—the new parameter part PAR_3 is the intersection (pullback) of PAR_1 and PAR_2 in (with respect to) EXP_{c2}.

The specification morphisms are the appropriate compositions deducible from the diagram.

Interpretation. The new class specification $COMB(C1, h, C2)$ is obtained by *replacing* the import interface of $C1$ with the *product* of $C2$. The new implementation part is that of $C1$ where the yet-unimplemented part IMP_1 is replaced by the implemented part of EXP_{c2}, via the *fitting morphism* h. The specification IMP_1 is no longer the import, having been provided by EXP_{c2}, which in turns *needs* IMP_2, which becomes the overall import interface. The new parameter PAR_3, part of the instance interface EXP_{i3}, is no longer PAR_1: the reason is that the composition $h \circ i_1$ associates EXP_{c2} to the items of PAR_1, and thus the only elements still "generic" are those not determined in EXP_{c2}, i.e., those *also* in PAR_2. Hence the definition of PAR_3 as the intersection of PAR_1 and PAR_2 in EXP_{c2}.

The next theorem again justifies the term *clean* for the combination. By viewing the semantics of a class as a relation (a set of pairs of algebras), the semantics of the result of the combination of $C1$ and $C2$ can be seen as the composition (possibly via the renaming determined by h) of the semantics of $C1$ and $C2$. Again the proof is a direct application of the Amalgamation Lemma.

THEOREM 5.10 (INDUCED SEMANTICS). *The semantics of the class specification $COMB(C1, h, C2)$ is the set of all pairs $(A_{I_3}, A_{E_{c3}})$ such that*

—$(A_{I_3}, A_{E_{c2}}) \in SEM(C2_{spec})$
—$(A_{I_1}, A_{E_{c3}}) \in SEM(C1_{spec})$
—$A_{I_1} = V_h(A_{E_{c2}})$

for some $A_{Ec2} \in Alg(EXP_{c2})$ and $A_{I1} \in Alg(IMP_1)$.

The next two theorems show that combination is one way to obtain a new class by reusing an old one. This reuse is *predictable* since the semantics of the new class can be expressed using those of the combining classes.

THEOREM 5.11 (INDUCED INHERITANCE). *If* $C1$ Wcomb $C2$ *via* h, *then* $COMB(C1, h, C2)$ Wspec $C1$. *If, additionally,* $s'_S(pt(BOD_2)) = pt(BOD_3)$ *for* $s': BOD_2 \to BOD_3$, *then* $COMB(C1, h, C2)$ Wreuse $C2$.

PROOF. By definition $COMB(C1, h, C2)$ and $C1$ have the same instance and class interfaces; the specialization morphisms are identities over EXP_{i1} and EXP_{c1}. Thus $COMB(C1, h, C2)$ Wspec $C1$. Let $s': BOD_2 \to BOD_3$ be the induced morphism and define $f = s' \circ v_2$. If $s'_S(pt(BOD_2)) = pt(BOD_3)$, then f is a pointed morphism and therefore a reusing morphism. Thus $COMB(C1, h, C2)$ Wreuse $C2$. □

As in the case of actualization, if the relation between $C1$ and $C2$ is strong, then it is possible to construct a realization of $COMB(C1, h, C2)$ using those of $C1$ and $C2$. The proof of this result is based on standard properties of amalgamation [Ehrig and Mahr 1985].

THEOREM 5.12 (INDUCED STRONG INHERITANCE). *If* $C1$ Scomb $C2$ *via* h, *then there exists an algebra* $C3_{impl} \in Alg(BOD_3)$ *such that*

(1) $(COMB(C1, h, C2), C3_{impl})$ *is a class (still denoted by* $COMB(C1, h, C2)$*)*
(2) $COMB(C1, h, C2)$ Sspec $C1$
(3) *If, additionally,* $s'_S(pt(BOD_2)) = pt(BOD_3)$ *then* $COMB(C1, h, C2)$ Sreuse $C2$.

PROOF.

(1) $C1$ Scomb $C2$ implies that $C1$ Wcomb $C2$, then $COMB(C1, h, C2)$ Wspec $C1$ by Theorem 5.11. By definition, BOD_3 is the specification obtained from the pushout of

$$s_1: IMP_1 \to BOD_1 \quad \text{and} \quad v_2 \circ h: IMP_1 \to BOD_2.$$

Let $s': BOD_2 \to BOD_3$ and $v': BOD_1 \to BOD_3$ be the induced morphisms. Since $C1$ Scomb $C2$ via h, we have

$$V_{s_1}(C1_{impl}) = V_h\big(V_{v_2}(C2_{impl})\big)$$

call this IMP_1-algebra A. Define $C3_{impl}$ as the amalgamated sum of $C1_{impl}$ and $C2_{impl}$ with respect to A. By definition, $C3_{impl}$ is a BOD_3-algebra, and therefore $C3 = (C3_{spec}, C3_{impl})$, with $COMB(C1, h, C2) = C3_{spec}$, is a class.

(2) With $v_3: EXP_{c3} \to EXP_{c1} \to BOD_3$ given by $v' \circ v_1$, we have

$$\begin{aligned} V_{id}(C3_{impl}) &= V_{v_2}\big(V_{s'}(C3_{impl})\big) \\ &= V_{v_2}(C2_{impl}) \end{aligned}$$

and thus $C3$ Sreuse $C2$. □

It is possible, following the lines Ehrig and Mahr [1990] and Parisi-Presicce [1987a] to show that combination and actualization are associative, and thus

they do not require that the classes be built in a specified order. There are also several connections among these two relations and the interconnection mechanism. Furthermore, it is possible to extend the notion of Combination and Actualization to the case where one class provides only part of what another class needs [Parisi-Presicce 1987b]. We can define in these cases notions of partial actualization and combination and obtain results similar to the weak and strong induced inheritance mentioned above.

Most of the current languages allow importing from separately written classes. But they do not provide any combination mechanism since the importing procedure requires the name of the class from which we wish to import and produces an implicit combination of the imported code with what is being developed. The only way to require something, and then provide a supplier for it, is by means of genericity and the actualization mechanism. Of course this does not represent a satisfactory solution because it is just a special case of combination.

6. CONCLUDING REMARKS

The algebraic theory of class specifications presented here is general enough to model all the features in the analyzed languages. Not all languages provide all the features of our model, some identifying the instance and the class interface, some the class interface and the implementation, none providing an explicit import interface without an explicit reference to an existing (virtual or not) class. The class specification allows a designer to distinguish formally between two common forms of inheritance, the reusing one and the specialization one. In fact, as shown, the reusing inheritance can be viewed as syntactic sugar, since it can be expressed in terms of the relation of specialization inheritance (and its inverse), but without any control on the correctness or the preservation of semantics. Other two relations between classes have been proposed: in both cases (but with different intentions and use) they relate a class in need to be completed (either by choosing a generic part *PAR* or by implementing a virtual part *IMP*) with another class able to produce what is needed. For both relations, new hierarchies of classes can be constructed, representing the usability of certain classes. These hierarchies are closely related to inheritance hierarchies, as shown in the Section 5: the producer C of an interface which satisfies the constraints of the parameter or of the import part of a class C' determines a new class which inherits from it by reusing and from C' by specialization. We believe this to be very important since its provides a restricted reuse of code where we can predict the behavior of the outcome. We have presented here definitions and theorems using the simplest form of algebraic specifications, but both the formalizations and the results can immediately be extended to other frameworks based on institutions [Goguen and Burstall 1983] other than the equational one and on different specification logics [Ehrig et al. 1991]. In particular, we plan to consider different semantics for the classes, such as the behavioral one.

We have outlined how the reusing inheritance is not useful since it can be simulated by the specialization. Moreover, some problems arise when the intended use of specialization degenerates in reusing inheritance. Actually, in

current object-oriented languages it is not possible to refine a method without redefinition: a shadowing method can violate the invariants of the superclass because it does not assure the capatibility of related semantics. Refinement, as realized in current object-oriented programming, can therefore be considered harmful.

Different techniques can be used in order to enhance the quality of software. Among them, inheritance and genericity play an important role. An informal comparison between these two techniques can be found in Meyer [1986] where the author claims that inheritance is more powerful than genericity. Section 5 provides a formal proof of this result through Theorems 5.4 and 5.5.

Some formalizations of inheritance have been provided recently. In Breu [1991] the specification language OS is introduced in which the class specification is defined following the CIP-approach without distinguishing the different roles played by the subspecifications in the class specification. The formalization of inheritance is distinguished in inheritance and subtyping, which correspond to our weak and strong specialization, respectively, without formalizing the notion of reusing inheritance. Moreover, OS is extended through environment algebras to take into account identities and in order to give a formal semantics to an object-oriented language defined in Breu [1991]. In Clerici and Orejas [1988] an algebraic specification language with inheritance operators is proposed. The inheritance is used to complete incomplete specifications in the software design process and to restrict superclasses in order to obtain subtypes. Reusing inheritance is not considered; the class specification is flat, and parameterization is implemented via inheritance as done in Meyer [1986]. In Gaudel and Moineau [1988] a theory of software reuse is provided where programs are models for the specification. Moreover, there is a distinction between reuse and efficient reuse: the former is obtained essentially via a forgetful functor, while the latter requires also the restriction of the forgotton model to the generated part of the algebra. These notions could be related with our reusing inheritance and specialization inheritance although the framework is not concerned directly with the object paradigm. The reusing specialization implies the efficient reuse, while specialization is just reuse.

The relations determing potential interconnections between classes can also be generalized to their partial version, where the producer provides only part of the parameter or of the import interface [Parisi-Presicce 1987a; 1987b]. The properties of such partial interconnections are related to the problem of multiple inheritance, which is currently under investigation. Further work will be devoted to the analysis of *simulation* of a class, by means of another class and related morphisms, as a relation of subtyping.

APPENDIX

Some Basic Notions of Category Theory and Algebraic Specifications

In this section, we review briefly some basic notions on algbraic specifications; details can be found in Ehrig and Mahr [1985] and Wirsing [1991]. A

signature Σ is a pair (S, OP) where S is a set of *sorts* and OP a set of *constant* and *function symbols*; constant symbols are referred to as operation symbols of arity 0. The set S is sometimes denoted by $sorts(\Sigma)$ and the set OP by $opns(\Sigma)$. Each operator symbol $N \in OP$ has associated a *signature* $s_1 \cdots s_n \to s$ for $s_1 \cdots s_n \in S^+$ and $s \in S$ (for constant symbols it is denoted by $\to s$). A *pointed signature* is a signature $\Sigma = (S, OP)$ with a distinguished element of the set S of sorts denoted by $pt(\Sigma)$. By a Σ-algebra $A = (S_A, OP_A)$ of a signature $\Sigma = (S, OP)$ we mean two families $S_A = (A_s)_{s \in S}$ and $OP_A = (N_A)_{N \in OP}$, where A_s are sets for all $s \in S$, which are called *domains* of A, and $N_A: A_{s_1} \times \cdots \times A_{s_n} \to A_s$ are functions for all operator symbol $N: s_1 \cdots s_n \to s$ and all $s_1 \cdots s_n \in S^+$, $s \in S$ (for constant symbols $N: \to s$, $N_A \in A_s$). The set of all Σ-algebras is denoted by $Alg(\Sigma)$.

If $\Sigma_1 = (S_1, OP_1)$ and $\Sigma_2 = (S_2, OP_2)$ are signatures, a *signature morphism* $h: \Sigma_1 \to \Sigma_2$ is a pair of functions $(h^S: S_1 \to S_2, h^{OP}: OP_1 \to OP_2)$ such that for each $N: s_1 \cdots s_n \to s$ in OP_1 and $n \geq 0$ we have $h^{OP}(N): h^S(s_1) \cdots h^S(s_n) \to h^S(s)$ in OP_2. A signature morphism $h: \Sigma_1 \to \Sigma_2$ induces a *forgetful functor* $V_h: Alg(\Sigma_2) \to Alg(\Sigma_1)$ defined, for each Σ_2-algebra A'', by $V_h(A'') = A' \in Alg(\Sigma_1)$ with $A'_s = A''_{h s(s)}$ for each $s \in S_1$, $N_{A'} = h^{OP}(N)_{A''}$ for each $N \in OP_1$. A *pointed signature morphism* is a signature morphism $h: \Sigma_1 \to \Sigma_2$ such that $h^S(pt(\Sigma_1)) = pt(\Sigma_2)$. It is easy to check that pointed signature morphisms are closed under composition.

By an *algebraic specification* $SPEC = (\Sigma, E)$ we intend a pair consisting of a signature Σ and a set E of (positive conditional) equations. For convenience, $sorts(\Sigma)$ in this case is also denoted by $sorts(SPEC)$. If $SPEC_1 = (\Sigma_1, E_1)$ and $SPEC_2 = (\Sigma_2, E_2)$ are two algebraic specifications, a *specification morphism* $f: SPEC_1 \to SPEC_2$ is a signature morphism $f: \Sigma_1 \to \Sigma_2$ such that the translation $f^{\#}(E_1)$ of the equations of $SPEC_1$ is contained in E_2. A *pointed algebraic* specification *is an algebraic specification with a pointed signature. A* pointed specification morphism *between pointed specifications is a pointed signature morphism f such that* $f^{\#}(E_1) \subseteq E_2$.

For notional convenience, when $SPEC = (\Sigma, E)$ is a pointed specification the distinguished sort $pt(\Sigma)$ will be also denoted by $pt(SPEC)$.

The algebraic specifications and the specification morphisms form the category CATSPEC of algebraic specifications [Ehrig and Mahr 1985].

Definition 4.1 (*Pushout*). Given two specification morphisms $g_1: S \to S_1$ and $g_2: S \to S_2$ in CATSPEC, the *pushout* of g_1 and g_2 consists of S^*, $h_1: S_1 \to S^*$, $h_2: S_2 \to S^*$ such that as in Figure 9:

—$h_1 \circ g_1 = h_2 \circ g_2$ (commutativity) and
—for all specification S' and specification morphisms $k_1: S_1 \to S'$ and $k_2: S_2 \to S'$, such that $k_1 \circ g_1 = k_2 \circ g_2$ there is a unique specification morphism $k: S^* \to S'$ with $k \circ h_1 = k_1$ and $k \circ h_2 = k_2$ (universal property).

Given two specification morphisms $g_1: S \to S_1$ and $g_2: S \to S_2$ we denote the pushout object by $S_1 +_S S_2$ if the context is sufficient to avoid ambiguity. The dual notion of pushout is the pullback.

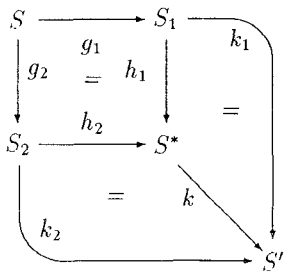

Fig. 9. Pushout diagram.

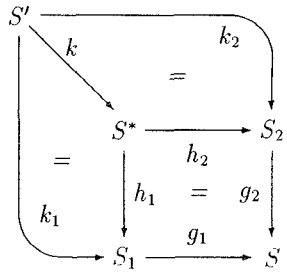

Fig. 10. Pullback diagram.

Definition A.2 (Pullback). Given two specification morphisms $g_1: S_1 \to S$ and $g_2: S_2 \to S$ in CATSPEC, the *pullback* of g_1 and g_2 consists of S^*, $h_1: S^* \to S_1$, $h_2: S^* \to S_2$ such that as in Figure 10:

—$g_1 \circ h_1 = g_2 \circ h_2$ (commutativity) and

—for all specification S' and specification morphisms $k_1: S' \to S_1$ and $k_2: S' \to S_2$, such that $g_1 \circ k_1 = g_2 \circ k_2$ there is a unique specification morphism $k: S' \to S^*$ with $h_1 \circ k = k_1$ and $h_2 \circ k = k_2$ (universal property).

THEOREM A.3. *The CATSPEC category is closed with respect to pushout and pullbacks.*

PROOF. We are just going to show how a pushout is constructed: details on the similar construction for pullbacks and on the verification of the desired properties can be found in Ehrig and Mahr [1985]. Given $g_1: S_1 \to S$ and $g_2: S_2 \to S$, the specification S^* is contructed as follows:

—$sorts(S^*)$ is the set $sorts(S_1) + sorts(S_2)/\equiv_S$ of equivalence classes of the disjoint union $sorts(S_1) + sorts(S_2)$ by the equivalence relation $\equiv s$ generated by $g_1(s) = g_2(s)$ for $s \in sorts(S)$

—$opns(S^*)$ is similarly the quotient set $opns(S_1) + opns(S_2)/\equiv_{OP}$

—$eqns(S^*) = g_1^{\#}(eqns(S_1)) \cup g_2^{\#}(eqns(S_2))$ where $g_i^{\#}(eqns(S_i))$ is the translation of the equations of S_i to $(sorts(S^*), opns(S^*))$

The morphisms h_1 and h_2 map each element (sort or operation symbol) into its equivalence class. □

Definition A.4 (Amalgamation). Given a pushout diagram as in Figure 9, and the algebras $A_1 \in Alg(S_1)$, $A_2 \in Alg(S_2)$, and $A \in Alg(S)$ with

$$V_{g_1}(A_1) = A = V_{g_2}(A_2)$$

the *amalgamated sum*, or short *amalgamation*, of A_1 and A_2 with respect to A, written

$$A_1 +_A A_2$$

is the S^*-algebra A^* defined for all $s \in sorts(S_1) \cup sorts(S_2)$, $N \in opns(S_1) \cup opns(S_2)$

$$A^*_s = \text{if } s \in sorts(S_2) \text{ then } (A_2)_s \text{ else } (A_1)_s$$

$$N_{A^*} = \text{if } N \in opns(S_2) \text{ then } N_{A_2} \text{ else } N_{A_1}.$$

LEMMA A.5 (AMALGAMATION LEMMA). *Given a pushout diagram as in Figure 9, the amalgamated sum $A_1 +_A A_2$ has the following properties*

—*given algebras A_1, A_2, and A as in Definition A.4 the amalgamated sum $A_1 +_A A_2$ is the unique S^*-algebra A^* satisfying*

$$V_{h_1}(A^*) = A_1 \quad \textit{and} \quad V_{h_2}(A^*) = A_2.$$

Vice versa each S^*-algebra A^* has a unique representation

$$A_1 +_A A_2$$

where $(V_{h_1}(A^*) = A_1$, $V_{h_2}(A^*) = A_2$, and $V_{g_1}(A_1) = A = V_{g_2}(A_2)$.

—*An S^*-algebra A^* is isomorphic to the amalgamated sum $A_1 +_A A_2$, if and only if*

$$V_{h_1}(A^*) = A_1 \quad \textit{and} \quad V_{h_2}(A^*) = A_2.$$

A similar property holds for morphisms.

REFERENCES

ALENCAR, A. J., AND GOGUEN, J. A. 1991. OOZE: An object-oriented *Z* environment. In *Proceedings of ECOOP91.* Lecture Notes in Computer Science, vol. 512. Springer-Verlag, Berlin, 180–199.

AMERICA, P. H M. 1990. Designing an object-oriented programming language with behavioral subtyping. In *Proceedings of REX / FOOL.* Lecture Notes in Computer Science, vol. 489. Springer-Verlag, Berlin, 60–90.

AMERICA, P. H. M. 1987. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings ECOOP87.* Lecture Notes in Computer Science, vol. 276. Springer-Verlag, Berlin, 234–242.

AMERICA, P. H. M., AND VAN DER LINDEN, F. 1990. A parallel object-oriented language with inheritance and sybtyping. In *Proceedings of ECOOP / OOPSLA.* ACM, New York, 161–168.

BLUM, E., EHRIG, H., AND PARISI-PRESICCE, F. 1987 Algebraic specifications of modules and their basic interconnections. *J. Comput. Syst. Science 34*, 2/3, 293–339.

BREU, R. 1991. Algebraic specification techniques in object oriented programming environments. In *Lecture Notes in Computer Science* vol. 562. Springer-Verlag, Berlin.

CLERICI, S., AND OREJAS, F. 1988. GSBL: An algebraic specification language based on inheritance. In *Proceedings of ECOOP88.* Lecture Notes in Computer Science, vol. 322. Springer-Verlag, Berlin, 78–92.

EHRIG, H., AND MAHR, B. 1985. Fundamentals of algebraic specification 1 Equations and initial semantics. In *EATCS Monograph in Computer Science.* Vol. 6, Springer-Verlag, Berlin.

EHRIG, H., AND MAHR, B. 1990. Fundamentals of algebraic specification 2. Module specifications and constraints., *EATCS Monograph in Computer Science.* Vol. 21. Springer-Verlag, Berlin.

EHRIG, H., AND WEBER, H. 1985. Algebraic specification of modules. In *Formal Models in Programming*. North-Holland, Amsterdam.

EHRIG, H., BALDAMUS M. AND OREJAS, F. 1991. New concepts for amalgamation and extension in the framework of the specification logic. TUB Tech. Rep.

GABRIEL, R. P., WHITE, J. L., AND BOBROW, D. G. 1991. CLOS integrating object oriented and functional programming. *Commun. ACM 34*, 9 (Sept.), 28–38.

GAUDEL M. C., AND MOINEAU, T. 1988. A theory of software reusability. In *Proceedings of ESOP88*. Lecture Notes in Computer Science, vol. 300. Springer-Verlag, Berlin, 115–130.

GOGUEN, J. A., AND BURSTALL, R. 1984. Introducing institutions. In *Proceedings of Logics of Programming Workshop*. Lecture Notes in Computer Science, vol. 164. Springer-Verlag, Berlin, 221–256.

KRISTENSEN, B. B., MADSEN, O. L., MOLLER-PEDERSEN, B., AND NYGAARD, K. 1987. The BETA programming language. In *Research Directions in Object Oriented Programming*. MIT Press, Cambridge, Mass, 7–48.

MADSEN, O. L., AND MOLLER-PEDERSEN, B. 1989. Virtual classes: The powerful mechanism in object-oriented programming., In *Proceedings of OOPSLA89*. ACM, New York, 397–406.

MADSEN, O. L., MAGNUSSON, B., AND MOLLER-PEDERSEN, B. 1990. Strong typing of object oriented languages revised., In *Proceedings ECOOP / OOPSLA*. ACM, New York, 140–149.

MEYER, B. 1988. *Object-Oriented Software Construction*. Prentice-Hall, Englwood Cliffs, N.J.

MEYER, B. 1986. Genericity versus inheritance. In *Proceedings of OOPSLA86*. ACM, New York, 391–405.

NELSON, N. L. 1991. An object oriented tower of Babel. *OOPS Mess. 2*, 3(July).

NIVELA, M. P., AND OREJAS, F. 1987. Initial behavior semantics for algebraic specifications. In *Recent Trends in Data Type Specification*. In Lecture Notes in Computer Science, vol. 332 Springer-Verlag, Berlin, 184–207.

PARISI-PRESICCE, F. 1988. Product and iteration of module specifications. In *Proceedings of CAAP88*. Lecture Notes in Computer Science, vol. 299. Springer-Verlag, Berlin, 149–164

PARISI-PRESICCE, F. 1987a. Union and actualization of module specifications: Some compatibility results. *J. Comput. Syst. Science 35*, 1, 72–95.

PARISI-PRESICCE, F. 1987b. Partial composition and recursion of module specifications. In *Proceedings of CAAP87*. Lecture Notes in Computer Science, vol. 249. Springer-Verlag, Berlin, 217–231.

PARISI-PRESICCE, F., AND PIERANTONIO, A. 1991. An algebraic view of inheritance and subtyping in object oriented programming., In *Proceedings of ESEC91*. Lecture Notes in Computer Science, vol. 550. Springer-Verlag, Berlin, 364–379.

O'BRIEN, P. D., HALBERT, D. C., AND KILLIAN, M. 1987. The Trellis programming environment. In *Proceedings of OOPSLA87*. ACM, New York, 91–102.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILLIAN, M., AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In *Proceedings of OOPSLA86*. ACM, New York, 9–16.

SNYDER, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA86*. ACM, New York, 38–45.

STROUSTRUP, B. 1986. *The C + + Programming Language*. Addison-Wesley, Reading, Mass.

WEGNER, P. Dimensions of Object-Based Language Design. Proceedings OOPSLA87 (1987) 168–182. Also as special issue of SIGPLAN N. 22, 12 (Dec. 1887).

WEGNER, P., AND ZDONIK, B. 1988. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings ECOOP88*. Lecture Notes in Computer Science, vol. 322. Springer-Verlag, Berlin, 55–77.

WIRFS-BORCK, R. J., AND JOHNSON, R. E. 1990. Surveying current research in object-oriented design. *Commun. ACM, 33*, 9 (Sept.).

WIRSING, M. 1991. Algebraic specification. In *Handbook of Theoretical Computer Science*. Vol. B. North-Holland, Amsterdam, 677–788.

Received August 1992; revised March 1993; accepted April 1994