Towards a classification of inheritance relations*

Davide Ancona, Egidio Astesiano and Elena Zucca
Dipartimento di Informatica e Scienze dell'Informazione
Viale Benedetto XV, 3
16132 Genova (Italy)

Introduction

The central issue of this paper is to provide an attempt at classifying inheritance relations, by means of a new formal semantic model for classes and methods.

Admittely, there are quite different ways of defining models for classes and inheritance ; in the last section we will recall some of them.

The view we take here is "roughly" to consider object systems as data types with state; more precisely, in Wegner's words: "Objects are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations. Variables representing the internal state of an object are called *instance variables* and its operations are called *methods*. Its collection of methods determines its *interface* and its *behavior*" [Weg87].

Thus we model a class as a dynamic object system, called d-oid, where a configuration of the system is a collection of sets, the sets of values and objects existing at that time, with operations over them (formally an algebra, called an instant algebra); the state of an object in a configuration is given by an element in a set and its relation with the structure (the operations of the algebra); a call of a method in a configuration and with some actual parameters is modelled by a transformation of that configuration into another one, where the identity of the objects is taken care by a map, that we call tracking map and which accounts for a very abstract view of object identity.

*This work has been partially supported by ESPRIT-BRA WG ISCORE, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R. and MURST-40% Modelli e Specifiche di Sistemi Concorrenti

D-oids can be related by a notion of structure preserving map (a morphism of d-oids) which gives them an appropriate mathematical structure (a category). By exploiting the d-oid structure, we can make an attempt at analyzing different kinds of inheritance relations, depending on different levels of freedom in redefining methods.

In this paper we essentially distinguish three kinds of inheritance in a hierarchical order, but our analysis shows that, if we need, a much finer distinction is possible. *Minimal* inheritance corresponds to total freedom; *regular* to recover from the heir a semantic structure similar to the parent (the *parent view of the heir*, with possibly different behaviour of the methods); *conservative* to redefine methods in a way that they behave as before on the parent view.

It should be clear that our modelization concerns a non-concurrent view of objects, corresponding to possible semantic models for, say, Eiffel, Smalltalk, C++. By a different perspective, we provide an updating to the dynamic case of the classical static data type approach by many-sorted algebras.

The first section presents our view of classes as d-oids, first by an introductory example and then by a paradigmatic one that will be used throughout the following sections; the second part presents our proposed classification and in the conclusion we report on some other results and related work.

1 Modelling object systems by d-oids

In this section we present our formalization of a dynamically evolving system of objects by means of a structure that we call *d-oid*. In the following we assume for simplicity a fixed algebraic framework for static data types; actually our work can be framed in a parameterized way on the top of any chosen algebraic framework (e.g. total algebras, partial algebras, order-sorted algebras and so on). In the formal definitions we refer to the standard total approach, whose definitions and notations are listed in the appendix.

1.1 An introductory example

Our starting point is the sentence, that can be found in many papers about the object oriented approach, that "an object is a data type with state". Hence we illustrate our view of an object (object system) starting from a comparison with the notion of (static) data type. As an example of static data type, let $STACK$ denote the algebra of stacks of natural numbers, over the signature

sig Σ_{STACK} = **enrich** Σ_{NAT} **by**
  sorts *stack*, *nat*, *bool*
  opns
    $emptystack\colon \to stack$
    $top\colon stack \to nat$

$push$: $stack\ nat \to nat$
pop: $stack \to stack$

where Σ_{NAT} denotes the signature of the algebra NAT of natural numbers.

For the purpose of the following discussion, we can see a static data type (formally modelled at an abstract level by a many-sorted algebra) consisting of:

- an operational interface to the outside, whose formal abstract counterpart is the signature;
- an implementation part, whose formal abstract counterpart is the interpretation of operation symbols in the algebra.

From the programming point of view, an example of software module implementing a static data type like $STACK$ is an Ada package. The package specification gives the operational interface and contains the headings of Ada functions for evaluating *emptystack*, *top*, *push*, *pop*. The package body gives the implementation part and contains the bodies of the above functions. This software module can be used by another module throughout its interface (e.g. by calling *emptystack*). What we want to stress is that in this case many different calls of a function, e.g. *emptystack*, within the user module give the same result, i.e. in the example an empty stack. In other words, *the interpretation of the operations is constant.*

This modelization of stacks using the data type notion is very well-known and useful in many respects. However, what seems missing in that view is a formal counterpart of the "dynamic nature" of a stack. In other words, the carrier of sort *stack* in the data type above looks in some sense as the set of all the possible configurations of stacks, but there is no notion of a stack as a dynamically evolving entity which can have different configurations at different times, while remaining the same. A classical way of modelling this dynamic view of a stack is to treat it as a process; here we remain closer to the data type view (see the conclusion). A single stack of naturals can be seen as an object on which what we can do is observing which is the top element, adding a new element on the top (push) and removing the top element (pop). As a first try, we can formalize that by what we call a *dynamic signature*, as follows.

dynsig $D\Sigma_{ST}$ = **enrich** Σ_{NAT} **by**
  opns
    top: $\to nat$
  dynopns
    $push$: $nat \Rightarrow$
    pop: $\Rightarrow$

Note that the functionality of the operations has changed: indeed, since we are now considering a single stack object, we do not need any more a sort *stack*.

In our approach an object (data type with state), which will be formally modelled by what we call a *d-oid*, consists of:

- an operational interface to the outside, whose formal counterpart is a dynamic signature. In this operational interface we have now usual operations as before, which do not have side-effects on the object, like *top*, but also what we call *dynamic operations*, whose effect is to change the configuration of the object;
- an implementation part, whose formal abstract counterpart will be the interpretation of operation symbols in the d-oid.

From the programming point of view, an example of software module implementing an object like a stack is again an Ada package. Indeed Ada is considered an object based language in the Wegner's classification in [Weg87], since it supports the object notion (by means of the package construct). The package specification gives the operational interface and contains the headings of an Ada function for evaluating *top*, and two Ada procedures for performing *push*, *pop*. The package body gives the implementation part and contains the bodies of the above subprograms. This software module can be used by another module throughout its interface (e.g. by calling *top*). However what is different in this case is that now many different calls of a function, e.g. *top*, within the user module give different result, i.e. in the example the current top of the stack. In other words, *the interpretation of the operations is not constant*: in a situation in which the stack contains, e.g. , the sequence 1, 2, 3, the interpretation of *top* is 1; after performing e.g. a pop operation, the interpretation of *top* becomes 2.

Thus we are looking for a natural extension of the algebraic approach to the formalization of data types, basically adding a new dimension, which is dynamics. That means that in our view an object (object system) has both an *horizontal* and a *vertical structure*, that we are going to illustrate in the following. But, in order to prevent misunderstandings, note already that in the case of many stack objects, we should indicate to which stack the operations are referred, hence we would need to introduce a sort *stack* and correspondingly we would have, say, $push\colon stack\ nat \Rightarrow .$)

The sort *stack* in the arguments is to indicate the stack on which we perform the operation, while note that there is still no corresponding target sort (contrary to the static data type case), since the result is just a change of state and no value is returned (the distinction the one between procedures and functions).

1.2 Object identity and sharing

In this subsection we illustrate our approach, by means of some examples of systems in which many objects exist. That allows us also to show how object identity and sharing are handled in a very natural way.

As a first example of system with many objects, let us consider a universe in which at each instant of time a finite number of "rectangles" exist, where rectangles are instances of the following class definition.

We now show how we build a semantic model of as a structure, called *d-oid*, consisting of a *horizontal* (instant) and a *vertical* (dynamic) part.

Horizontal structure. In each intermediate configuration in the life of the system, there is a finite number of existing rectangles, each one with a given length and width (two natural numbers). However, many different rectangles with the same couple of dimensions may exist in the system. In order to express that, we model a configuration of the system as an algebra over the following instant signature, where we have an explicit sort *rect* for rectangles: ($\mathcal{R}$ denotes the d-oid structure that we are going to define)

sig $\Sigma_{\mathcal{R}}$ = **enrich** Σ_{NAT} **by**
  sorts *rect*
  opns
    $length\colon rect \to nat$
    $width\colon rect \to nat$

We have various possibilities in defining the configurations; our first choice is a very abstract one: possible configurations of the system are $\Sigma_{\mathcal{R}}$-algebras R such that:

(*) $R_{|\Sigma_{NAT}} = NAT$, R_{rect} finite.

We denote in the following by $IA_{\mathcal{R}}$ a class of such algebras, called *instant algebras*.

Note that we do not fix which is the carrier of sort *rect* in an instant algebra, since we do not want to define a particular representation of rectangles; the only requirement is that we can observe which is the length and which is the width of a rectangle; moreover we require that R_{rect} is a finite set since we want that only a finite number of rectangles exist in each configuration.

For example, a configuration in which two rectangles exist, say x with length 2, width 1 and y with length 3, width 1, is modelled by a $\Sigma_{\mathcal{R}}$-algebra $\overline{R}$ in $IA_{\mathcal{R}}$ defined by (*) and $\overline{R}_{rect} = \{x, y\}$, $length^{\overline{R}}(x) = 2$, $width^{\overline{R}}(x) = 1$, $length^{\overline{R}}(y) = 3$, $width^{\overline{R}}(y) = 1$. Note that it does not matter what x, y are.

It is of primary importance to note that the state of a single object in a configuration (e.g. of a rectangle) is modelled not just by an element in a carrier of the corresponding instant algebra (e.g. x) but also by the current interpretation of the operations, giving the relationship with other objects (e.g. *length*, *width*). Note moreover that in this approach usual values (e.g. natural numbers) can be viewed as "constant" objects, i.e. objects which always exist and never change.

Vertical structure. We consider now the dynamic (vertical) dimension of the system. Accordingly with the method definitions given above, the dynamic evolution of the system of rectangles consists in four possible transformations from a configuration to another: changing the length of some existing rectangle, changing the width of some existing rectangle, doubling the dimensions of some existing rectangle, creating a new rectangle in the system (which is initialized to the "pointwise" rectangle with both dimensions equal to zero).

We formalize that by the following dynamic operations:

$$\begin{array}{l} setLength\colon rect\ nat \Rightarrow \\ setWidth\colon rect\ nat \Rightarrow \\ double\colon rect \Rightarrow \\ new\colon \Rightarrow rect \end{array}$$

Note that for example *setLength* has no target sort, since the result is a change of configuration and not a value; while *new* gives, together with a new configuration, also a new element of sort *rect*; the distinction corresponds to the usual one between procedures and functions, say in Pascal.

Let us define the interpretation of, e.g. , *setLength* in the d-oid $\mathcal{R}$ modelling the system of rectangles. This interpretation will be denoted by $setLength^{\mathcal{R}}$, and associates with each triple $<R, r, l>$ where $R \in IA_{\mathcal{R}}$, $r \in R_{rect}$, $l \in R_{nat}$ (we call such a triple an *instant tuple in* $IA_{\mathcal{R}}$ *of arity rect nat*):

- a new instant algebra in $IA_{\mathcal{R}}$, say R', modelling the configuration of the system resulting by performing the dynamic operation *setLength*;

- a tracking map $f\colon |R| \to |R'|$ showing how the elements of R are transformed when R becomes R'.

Formally, we write

$$setLength^{\mathcal{R}}_{<R,r,l>} = f\colon R \Rightarrow R',$$

where $setLength^{\mathcal{R}}_{<R,r,l>}$ denotes the application of the dynamic operation *setLength* interpreted in $\mathcal{R}$ to the instant tuple $<R, r, l>$.

For example, assume that we want to model the effect of applying the dynamic operation *setLength* to the instant tuple $<\overline{R}, x, 3>$, i.e. we change to 3 the length of the rectangle x in $\overline{R}$. We have:

- a new instant algebra $\overline{R}' \in IA_{\mathcal{R}}$;
- a tracking map $f\colon |\overline{R}| \Rightarrow |\overline{R}'|$

such that

$f_{nat} = id_{\mathrm{N}}$
f_{rect} injective
$\overline{R}'_{rect} = \{f(r) \mid r \in \overline{R}_{rect}\}$
$length^{\overline{R}'}(f(x)) = 3$, $length^{\overline{R}'}(f(r)) = length^{\overline{R}}(r)$, $\forall r \in \overline{R}_{rect}$, $r \neq x$
$width^{\overline{R}'}(f(r)) = width^{\overline{R}}(r)$, $\forall r \in \overline{R}_{rect}$.

Note that in this way object identity is modelled in a very abstract way by means of the tracking map (there is no need of an explicit notion of "name"). In the sequel we will show some particular case of the above model in which we use explicit names. The tracking map keeps trace of object identity, allowing to recognize different states of a single object during the evolution of the system. Different informal assumptions on the notion of object identity correspond to different formal assumptions on the tracking map. For example, allowing creation, deletion and equation ("the murderer is the butler") of objects corresponds to non-surjective, partial and non-injective tracking maps, respectively. As an example of non-surjective tracking map, let us show the interpretation of *new* in $\mathcal{R}$. For each instant algebra R in $IA_{\mathcal{R}}$,

$$new^{\mathcal{R}}_{<R>} = <f\colon R \Rightarrow R', z>$$

where:

$z \notin R_{rect}$;
$R'_{rect} = \{f(r) \mid r \in R_{rect}\} \cup \{z\}$;
$length^{R'}(z) = width^{R'}(z) = 0$;
$length^{R'}(f(r)) = length^{R}(r)$, $\forall r \neq z$;
$width^{R'}$ analogously.

Finally, let us introduce *constant dynamic operations*, which intuitively correspond to initial configurations of the system. A constant dynamic operation symbol has an optional sort: we write dop:$[s]$ and its interpretation is a couple $<B[, b]>$. For example let us assume in the dynamic signature $D\Sigma_{\mathcal{R}}$ of our running example also a constant dynamic operation symbol, *empty*:. Its interpretation in the d-oid $\mathcal{R}$ can be the defined as the $\Sigma_{\mathcal{R}}$-algebra $R \in IA_{\mathcal{R}}$ s.t. $R_{rect} = \emptyset$.

Summarizing our model of a system of rectangles, what we have defined is:

- a dynamic signature $D\Sigma_{\mathcal{R}}$ which is a couple $(\Sigma_{\mathcal{R}}, DOP_{\mathcal{R}})$ where $\Sigma_{\mathcal{R}}$ is the instant signature and $DOP_{\mathcal{R}}$ is the family of the dynamic operation symbols (*setLength*, *setWidth*, *double*, *new*, *empty*).
- a d-oid, which is a couple $\mathcal{R} = (IA_{\mathcal{R}}, \{dop^{\mathcal{R}}\}_{dop \in DOP_{\mathcal{R}}})$ where $IA_{\mathcal{R}}$ is the class of the instant algebras and, for each *dop* dynamic operation symbol in $DOP_{\mathcal{R}}$, $dop^{\mathcal{R}}$ is the interpretation of *dop* in $\mathcal{R}$.

We recall that in the d-oid $\mathcal{R}$ described until now $IA_{\mathcal{R}}$ may be any class of $\Sigma_{\mathcal{R}}$-algebras satisfying (*) and closed w.r.t. dynamic operations. We show now two choices of $IA_{\mathcal{R}}$ leading to two more concrete models particularly interesting from the intuitive point of view.

Rectangles as couples with name. In the first model, we define $IA_{\mathcal{R}}$, the class of the instant algebras, as the class of all the $\Sigma_{\mathcal{R}}$-algebras R such that (*) holds and moreover

$$R_{rect} \in \mathcal{P}_{fin}(Name \times \mathbb{N} \times \mathbb{N}),$$
$$\forall <n, l, w> \in R_{rect},\ length^R(<n, l, w>) = l,\ width^R(<n, l, w>) = w,$$
$$(**)\ \text{if } <n, l, w>, <n, l', w'> \in R_{rect}, \text{ then } l = l',\ w = w'$$

where *Name* is some infinite denumerable set of names. This model corresponds to the intuitive view of a rectangle as a triple consisting of the two dimensions and of a name which must be unique in the system (as guaranteed by the assumption (**) above). It is easy to adapt the above definitions to this particular case; for example, in the interpretation of the dynamic operation *setLength* applied to $<\overline{R}, x, 3>$, with $x = <n, 2, 1>$, the tracking map f becomes as follows:

f maps $x = <n, 2, 1>$ into $<n, 3, 1>$,
f is the identity elsewhere.

With this choice, all the states of the same rectangle in different configurations of the system keep the same name.

Rectangles as pure names. Note that in the preceding model the information modelled by l and w in the triple $<n, l, w>$ is "redundant" since the fact that the rectangle whose name is n has length l and width w is already modelled by the current interpretation of the operations in R. Hence a different model can be defined fixing $IA_{\mathcal{R}}$ as the class of all the $\Sigma_{\mathcal{R}}$-algebras R such that (*) holds and moreover $R_{rect} \in \mathcal{P}_{fin}(Name)$.

This model correspond to the intuitive view of a rectangle just as a "name", while the information about the two dimensions is "stored" in the structure. Other definitions must change consistently; for example, in the interpretation of the dynamic operation *setLength*, the tracking map becomes the identity.

These two models (and all the other particular cases obtained by fixing the choice of $IA_{\mathcal{R}}$ and the tracking maps) look equivalent, and indeed they can be shown to be *isomorphic* w.r.t. to a morphism notion which will be given in the following.

Sharing. Before summarizing the formal definitions, we present one more example, whose aim is to illustrate how objects with object subcomponents and object sharing can be handled in this framework in a very natural way.

Let us turn again to the stack example of the beginning, but allowing existence of many stacks in the system, as for rectangles. In other words, we consider the following class definition:

...

...

...

In this case, an object in the system (a stack) has an object subcomponent (its rest, which is in turn a stack). We can model the system of stacks by a d-oid defined analogously to $\mathcal{R}$ for rectangles. A configuration in which, for example, two stacks x and y have top 1, 2 respectively and share the same stack z which contains 3, 4 as rest can be modelled by an instant algebra ST as follows

$ST_{stack} = \{x, y, z, u\}$;
$top^{ST}(x) = 1$, $top^{ST}(y) = 2$, $top^{ST}(z) = 3$, $top^{ST}(u) = 4$;
$rest^{ST}(x) = z$, $rest^{ST}(y) = z$, $rest^{ST}(z) = u$, $rest^{ST}(u)$ is undefined.

The effect of applying, e.g. , *push* to the instant tuple $<ST, x, 0>$, is the transformation:

$$f: ST \Rightarrow ST'$$

where:

$ST'_{stack} = \{f(x), f(y), f(z), f(u), w\}$;
$top^{ST'}(f(x)) = 0$, $top^{ST'}(f(y)) = 2$, $top^{ST'}(f(z)) = 1$, $top^{ST'}(f(u)) = 3$,
 $top^{ST'}(w) = 4$;
$rest^{ST'}(f(x)) = f(z)$, $rest^{ST'}(f(y)) = f(z)$, $rest^{ST'}(f(z)) = f(u)$,
 $rest^{ST'}(f(u)) = w$,$rest^{ST'}(w)$ is undefined.

Note that, informally speaking, changing the stack x has the side effect of changing also the stack y.

1.3 Formalization

In this subsection we give the formal definitions of dynamic signature and d-oid, informally presented above, together with a definition of d-oid morphism which allows to give a category structure to d-oids and will be used in the following section for modelling inheritance.
If A is a set, then $[A]$ denotes $A \cup \{\bullet\}$, for some $\bullet \notin A$, which is called *null*.

Def. 1.1 A *dynamic signature* is a couple $D\Sigma = (\Sigma, DOP)$ where

- Σ is a signature (called *instant signature*), with sorts in S;
- DOP is a $(S^\star \times [S]) \cup [S]$-family of symbols called *dynamic operation symbols*; if $dop \in DOP_{w,s}$, then w, s are called *arity* and *sort* of dop, respectively, and for $w = s_1 \ldots s_n$, we write

  $dop: s_1 \ldots s_n \Rightarrow s$, if s is non-null
  $dop: s_1 \ldots s_n \Rightarrow$, if s is null;

  if $dop \in DOP_s$, then dop is called a *constant* dynamic operation symbol, and s is called *sort* of dop; we write $dop: s$, if s is non-null, and $dop:$, if s is null. □

Let in what follows $D\Sigma = (\Sigma, DOP)$ be a dynamic signature with sorts in S. Assume $IA \subseteq Alg(\Sigma)$, and, for every $w = s_1 \ldots s_n \in S^\star$, set

$$IA_w = \{<A, a_1, \ldots, a_n> \mid A \in IA, a_i \in A_{s_i}, \forall i = 1, \ldots, n\}.$$

We also assume, for simplifying the notation, that dynamic signatures have no overloading, i.e. , for every dynamic signature $D\Sigma = (\Sigma, DOP)$, if $dop \in DOP_{w,s}$, and $dop \in DOP_{w',s'}$, then $w = w'$ and $s = s'$.

Def. 1.2 For every $IA \subseteq Alg(\Sigma)$ and for every $A \in IA$, a *transformation of A (into B)* is a triple $<A, f, B>$ where $B \in IA$, $f : |A| \to |B|$, denoted by $f: A \Rightarrow B$. The map $f : |A| \to |B|$ is called *tracking map*.

For every $IA \subseteq Alg(\Sigma)$, $w = s_1 \ldots s_n \in S^\star$, and $s \in [S]$, a *(non-constant) method (or dynamic operation) m over IA, of arity $s_1 \ldots s_n$ and sort s*, written $m: IA_{s_1 \ldots s_n} \Rightarrow IA_{[s]}$ (we write simply IA for $IA_\bullet$), is a function which associates with every $<A, a_1, \ldots, a_n> \in IA_w$ a transformation of A, say $f: A \Rightarrow B$, and, if s is non-null, a value $b \in B_s$. The result of applying m to $<A, a_1, \ldots, a_n>$ is denoted by $m_{<A,a_1,\ldots,a_n>}$, and we write $m_{<A,a_1,\ldots,a_n>} = <f: A \Rightarrow B[, b]>$.

For every $IA \subseteq Alg(\Sigma)$, $s \in [S]$, a *constant method* (or *constant dynamic operation*) m over IA, of sort s, written $m: IA_{[s]}$ (we write simply IA for $IA_\bullet$), consists of an algebra $B \in IA$, and, if s is non-null, a value $b \in B_s$; we write $m = <B[, b]>$. □

Def. 1.3 A *d-oid over* $D\Sigma = (\Sigma, DOP)$ is a couple $\mathcal{D} = (IA, \{dop^{\mathcal{D}}\}_{dop \in DOP})$ where:

- $IA \subseteq Alg(\Sigma)$ (the algebras in IA are called *instant algebras*);
- $dop^{\mathcal{D}}: IA_{s_1 \ldots s_n} \Rightarrow IA_{[s]}$, for $dop: s_1 \ldots s_n \Rightarrow [s]$;
- $dop^{\mathcal{D}}: IA_{[s]}$, for $dop: [s]$. □

The definition of morphism ϕ from a d-oid $\mathcal{D}$ into a d-oid $\mathcal{D}'$ is rather straightforward (though spelling out the details seems lengthy): ϕ consists of a family of usual morphisms between Σ-algebras $\phi_A: A \to A'$, for every $A \in IA$ (called an *instant morphism*), which is moreover compatible with dynamic operations; the compatibility consists as usual in the invariance of the order of applying dynamic operations and morphisms (roughly we could write $dop^{\mathcal{D}'} \circ \phi = \phi \circ dop^{\mathcal{D}}$) and is expressed by the commutativity of a diagram.

Def. 1.4 Given two classes IA, IA' of instant algebras over Σ, an *instant morphism ϕ from IA into IA', written* $\phi: IA \to IA'$, is a function which associates with each instant algebra $A \in IA$, a Σ-homomorphism $\phi_A: A \to A'$, with $A' \in IA'$. □

Def. 1.5 Given two d-oids $\mathcal{D}$ and $\mathcal{D}'$ over $D\Sigma$,

$$\mathcal{D} = (IA, \{dop^{\mathcal{D}}\}_{dop \in DOP}),$$

$$\mathcal{D}' = (IA', \{dop^{\mathcal{D}'}\}_{dop \in DOP}),$$

a *$D\Sigma$-morphism ϕ from $\mathcal{D}$ into $\mathcal{D}'$*, written $\phi: \mathcal{D} \to \mathcal{D}'$, is an instant morphism from IA into IA' such that the following conditions hold:

- for every $dop: [s]$,
  if $dop^{\mathcal{D}} = <A[,a]>$, $dop^{\mathcal{D}'} = <A'[,a']>$
  then $\phi_A: A \to A'$, $[\phi_A(a) = a']$;
- for every $dop: s_1 \ldots s_n \Rightarrow [s]$, $<A, a_1, \ldots, a_n> \in IA_{s_1 \ldots s_n}$,

  if
  $dop^{\mathcal{D}}_{<A,a_1,\ldots,a_n>} = <f: A \Rightarrow B[,b]>$,
  $\phi_A: A \to A'$,
  $\phi_A(a_1) = a'_1, \ldots, \phi_A(a_n) = a'_n$,
  $dop^{\mathcal{D}'}_{<A',a'_1,\ldots,a'_n>} = <f': A' \Rightarrow B'[,b']>$
  then
  $\phi_B: B \to B'$,
  $\phi_B(f(a)) = f'(\phi_A(a))$, for every $a \in A$;
  $[\phi_B(b) = b']$. □

The second condition can be expressed by the commutativity of a diagram[1]:

$$\begin{tikzcd}
<A, a_1, \ldots, a_n> \arrow[r, "{\phi_A}^{s_1 \ldots s_n}"] \arrow[d, "dop^{\mathcal{D}}"'] & <A', a'_1, \ldots, a'_n> \arrow[d, "dop^{\mathcal{D}'}"'] \\
<B[, b]> \arrow[r, "{\phi_B}^{[s]}"] & <B'[, b']>
\end{tikzcd}$$

where ${\phi_A}^{s_1 \ldots s_n}\colon IA_{s_1 \ldots s_n} \to IA_{s_1 \ldots s_n}$ is defined by:

$${\phi_A}^{s_1 \ldots s_n}(<A, a_1, \ldots, a_n>) = <A', \phi_A(a_1), \ldots, \phi_A(a_n)> \text{ for } \phi_A\colon A \to A'.$$

Fact. 1.6 *D-oids over $D\Sigma$ and their morphisms form a category, that we call* **Doid**$(D\Sigma)$. □

2 An analysis of inheritance relations

Inheritance is essentially a mechanism for incremental programming and sharing behaviors [Coo89]; in general, in object oriented programming, inheritance allows modification of existing modules (classes) in order to define a new module which shares some characteristics with parent modules. What changes among object oriented languages is the liberty degree in modifying classes.

Our goal is to study several possible kinds of inheritance relations that may exist between two d-oids modelling, respectively, a parent and a heir class.

In our analysis we start with a very general notion of inheritance and we refine it step by step, in order to get more specialized form of inheritance until obtaining a very strict inheritance relation.

2.1 Minimal inheritance

A distinction between two primary meanings of inheritance has been recommended by many authors (see for example [JWB90] from the design point of view):

- *interface inheritance* (or *interface hierarchy*) denotes a classification of objects based on common external interfaces: an object provides a superset of the services provided by another object;
- *implementation inheritance* denotes a mechanism by which object implementations can be organized to share descriptions.

[1]This diagram has been produced using Paul Taylor's package.

From the formal point of view, interface inheritance corresponds to the minimal requirement satisfied by any inheritance relation, i.e. a syntactic constraint: the syntactic structure of the heir must include the one of the parent. We call the modelization of this relation in our approach *minimal inheritance*.

Example 2.1 Consider the following class definition:

$\square$

With respect to the class defined in the preceding section, this class adds a new attribute ( ) and a new method ( ), and redefines two of the old methods (,).

A dynamic signature corresponding to this class definition is as follows:

dynsig $D\Sigma_{\mathcal{C}}$ = **enrich** Σ_{NAT} **by**
  sorts cub
  opns
    $length\colon cub \to nat$
    $width\colon cub \to nat$
    $height\colon cub \to nat$
  dynopns
    $setLength\colon cub \Rightarrow$
    $setWidth\colon cub \Rightarrow$
    $double\colon cub \Rightarrow$
    $new\colon \Rightarrow cub$
    $setHeight\colon cub \Rightarrow$

Comparing this dynamic signature with the dynamic signature $D\Sigma_{\mathcal{R}}$ of the d-oid $\mathcal{R}$ modelling rectangles defined in the preceding section, we can see that in this case the syntactic constraint is satisfied: the old signature is still present in the new one, the only difference is that the sort *rect* has been replaced by the sort *cub*. Note that the arity of the static operations *length* and *width* has been changed consistently, otherwise our syntactic requirement should not hold.

From the programming point of view, that means that the services offered to the external users by a rectangle (e.g. the *double* operation) are still offered by a cuboid, no matter which is the actual implementation of the service inside the two classes. Formally, we can say that in every inheritance relation there is a *morphism of dynamic signatures* which is a functionality preserving family of maps (possibly partial) sending sorts into sorts, and (static and dynamic) operation symbols into symbols of the corresponding functionality.

Def. 2.2 Let $D\Sigma = (\Sigma, DOP), D\Sigma' = (\Sigma', DOP')$ be two dynamic signatures; a *dynamic signature morphism* σ *from* $D\Sigma$ *into* $D\Sigma'$, written $\sigma\colon D\Sigma \to D\Sigma'$, is a couple $\sigma = (\sigma^I, \sigma^D)$ where:

- $\sigma^I\colon \Sigma \to \Sigma'$ is a signature morphism; I stands for "instant";
- $\sigma^D\colon DOP \to DOP'$ is a family of maps (D stands for "dynamic") such that

$$\begin{array}{l} \sigma^D(dop\colon s_1 \ldots s_n \Rightarrow [s]) : \sigma^I(s_1) \ldots \sigma^I(s_n) \Rightarrow [\sigma^I(s)], \\ \sigma^D(dop\colon [s]) : [\sigma^I(s)]. \end{array}$$ $\square$

Example 2.3 Considering the two dynamic signatures $D\Sigma_{\mathcal{R}}$ and $D\Sigma_{\mathcal{C}}$, we can define a dynamic signature morphism $\overline{\sigma}\colon D\Sigma_{\mathcal{R}} \to D\Sigma_{\mathcal{C}}$ in this way:

$\overline{\sigma}^I(rect) = cub$, the identity elsewhere;
$\overline{\sigma}^D$ the identity. $\square$

Usually we expect σ to be injective but not to be surjective and the (main) sort of the heir d-oid to be mapped into the one of the parent. Besides, σ does not need to be a total function: if there exists the possibility of hiding methods, for instance, then dynamic signature morphisms may also be partial. Assume to hide the method *double* in the d-oid class . Hence we have a dynamic signature morphism $\overline{\sigma}'$ equal to $\overline{\sigma}$ except that ${\overline{\sigma}'}^D(double)$ is not defined. Note that in this case the informal assumption of having common external interfaces holds in a restricted way. However in the rest of the paper we deal with total morphisms, which are the usual case.

Consider now what happens at the semantic level, in the case of minimal inheritance. We start by giving an example of a possible reasonable model for the class .

Example 2.4 Let $\mathcal{C}1 = (IA_{\mathcal{C}1}, \{dop^{\mathcal{C}1}\}_{dop \in DOP_{\mathcal{C}}})$ be the d-oid over $D\Sigma_{\mathcal{C}} = (\Sigma_{\mathcal{C}}, DOP_{\mathcal{C}})$ defined as follows:

- $$\begin{array}{lll} IA_{\mathcal{C}1} = \{C \mid & C_{|\Sigma_{NAT}} = NAT, C_{cub} \in \mathcal{P}_{fin}(Name \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}), & \\ & \forall <c,l,w,h> \in C_{cub} & length^C(<c,l,w,h>) = l, \\ & & width^C(<c,l,w,h>) = w, \\ & & height^C(<c,l,w,h>) = h. \\ & <c,l,w,h>, <c,l',w',h'> \in C_{cub} \supset l = l', w = w', h = h'\}; & \end{array}$$

- $double^{\mathcal{C}1}_{<C[,c]>} = f : C \Rightarrow C'$
  where $C' = C$ except that $length^{C'}(c) = length^{C}(c) \cdot height^{C}(c)$
  $width^{C'}(c) = width^{C}(c) \cdot height^{C}(c)$
  f is the identity.
- $\cdots\cdots$ $\square$

Note that the representation chosen for cuboids is the analogous of the one given in the previous section for rectangles (rectangles as triples). From now on we will refer to that d-oid by $\mathcal{R}$. We have omitted the definitions of the other dynamic operations of $\mathcal{C}1$, because of their similarity with the ones of $\mathcal{R}$.

At this point we can associate with every instant algebra $C \in IA_{\mathcal{C}1}$ over the instant signature $\Sigma_{\mathcal{C}}$, an instant algebra $C_{|\overline{\sigma}}$ over the instant signature $\Sigma_{\mathcal{R}}$, called *the reduct via* $\overline{\sigma}$ *of* C (see the appendix for a formal definition), whose carriers are the carriers of C of the corresponding sorts and whose operations are derived by those in C; hence

$$(C_{|\overline{\sigma}})_{rect} = C_{cub} \qquad\qquad (C_{|\overline{\sigma}})_{nat} = C_{nat}$$

and the operations of $C_{|\overline{\sigma}}$ are as in C.
We can summarize now the above discussion by the following formal definition.

Def. 2.5 Given two d-oids $\mathcal{D} = (IA, \{dop^{\mathcal{D}}\}_{dop \in DOP}), \mathcal{D}' = (IA', \{dop^{\mathcal{D}'}\}_{dop \in DOP})$ over $D\Sigma$ and $D\Sigma'$ respectively, a morphism of dynamic signatures $\sigma : D\Sigma \to D\Sigma'$, we say that $\mathcal{D}'$ *minimally inherits* $\mathcal{D}$ *via* σ, and that there is a *minimal inheritance relation from* $\mathcal{D}'$ *into* $\mathcal{D}$.
We denote by μ_σ the induced reduct mapping s.t.

$$\mu_\sigma(A') = A'_{|\sigma}, \text{for } A' \in IA'$$

and by $IA'_{|\sigma} = \{\mu_\sigma(A') \mid A' \in IA'\}$ the class of reducts via σ. $\square$

Fact. 2.6 *The minimal inheritance relationship is transitive.* $\square$

Notice that, as seen in the example, the usual syntactic construct of inheritance canonically defines a morphism of dynamic signatures mapping the main sort of the parent into the main sort of the heir.
What is missing until now is an association from $IA'_{|\sigma}$ into the class of instant algebras IA of the parent d-oid, which will be discussed in the following subsection.

2.2 Regular inheritance

In the previous example we have seen that a minimal inheritance relation exists between $\mathcal{R}$ and $\mathcal{C}1$ regardless the semantic structure of $\mathcal{C}1$, provided that there exists a signature morphism σ which permits to recover the old syntactic structure of the parent $\mathcal{R}$ in the new one of the heir $\mathcal{C}1$. But what about the semantics of $\mathcal{R}$? In the general case inheritance is a mechanism which does not preserve the semantics of modules, since, by using redefinition, the behavior of the redefined methods may be different from the behavior of the corresponding old methods.

But what does it mean comparing the behavior of two methods which act on objects of different classes? In the following we try to formalize this, showing that there are also situations in which the comparison does not even make sense.

In the next examples the symbol $\mathcal{R}$ will denote the d-oid of rectangles as triples defined in the previous section.

Example 2.7 Consider the following class definition:

This class definition can be modelled by a d-oid $\mathcal{C}2$ defined analogously to $\mathcal{C}1$ above, except that the interpretation of *double* is as follows.

$double^{\mathcal{C}2}_{<C[,c]>} = f\colon C \Rightarrow C'$, where
  $C' = C$ except that $height^{C'}(c) = 2 \cdot height^{C}(c)$,
  f is the identity. $\square$

In this case it is clear from the intuitive point of view that the behavior of the defined method *double* is different from the behavior of the corresponding old method. More precisely, what allows us to do the comparison is that:

(i) it is possible to associate with each cuboid its "rectangle part";

(ii) it is possible to associate with the new method $double^{\mathcal{C}2}$ (which act on cuboids) its "rectangle version" (which acts on rectagles): it is a method which has no effect at all and leaves unchanged the dimensions of the rectangle upon which was invoked.

At this point we are able to compare the dynamic operation $double^{\mathcal{R}}$ with the rectangle version of $double^{\mathcal{C}2}$ and to conclude that the behavior of $double^{\mathcal{C}2}$ on rectangles is different from the behavior of $double^{\mathcal{R}}$.

We formalize now the above requirements.

- The minimal inheritance relation always guarantees the existence of the map $\mu_\sigma: IA' \to IA'_{|\sigma}$, saying that we can recover from the heir an instant structure $IA'_{|\sigma}$ over the parent signature. What we need now is to map $IA'_{|\sigma}$ into the parent instant structure. This is done, as usual, admitting the existence of a family of structure preserving maps (formally, Σ-homomorphisms, see the appendix), i.e. what we call an instant morphism:

  $$\varphi: IA'_{|\sigma} \to IA$$

  (we recall that φ associates with each instant algebra in $IA'_{|\sigma}$ a homomorphic mapping into an instant algebra in IA).
  Note that in this case there exists a function

  $$\mu: IA' \to IA$$

  defined by $\mu = \varphi \circ \_{|\sigma}$, where φ denotes, by abuse of notation, the function

  $$\varphi: IA'_{|\sigma} \to IA$$

  defined by $\varphi(A'_{|\sigma}) = A$ if $\varphi_{A'_{|\sigma}}: A'_{|\sigma} \to A$.

  In the last example, the function φ can be defined in a straightforward way.

  Let $\psi: Name \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to Name \times \mathbb{N} \times \mathbb{N}$ be the function defined by
  $\psi(<n, l, w, h>) = <n, l, w>$,
  then, for each $C \in IA_{\mathcal{C}1}$
  $\varphi_{C_{|\sigma}}: C_{|\sigma} \to R$
  where
  $$\begin{array}{rcl} R_{rect} & = & \psi(C_{cub}) \\ width^R(\psi(c)) & = & width^C(c) \\ length^R(\psi(c)) & = & length^C(c) \end{array}$$

  Intuitively, μ transforms every configuration in $\mathcal{C}2$ in a configuration in $\mathcal{R}$ mapping each existing cuboid in a rectangle having the same length and width. It can be easily shown that in this case φ is bijective, but in general we will not expect this. (Note also that μ turns to be a total function; this is not always the case.)

- We are now able to express in a more precise way the requirement (ii) informally stated above.

  Essentially what we ask is that, if $\mathcal{D}'$ is an heir of $\mathcal{D}$, then every method in $\mathcal{D}'$ satisfies the following *regularity condition*: if we apply $dop^{\mathcal{D}'}$ to two instant tuples in IA' which have the same correspondent (formally, the same image via μ), then we get two instant couples in IA' which still have the same correspondent in IA

  For example, in the case of the definition of $double^{\mathcal{C}2}$, the regularity condition is satisfied. Indeed if two instant couples $<C_1, c_1>, <C_2, c_2>$ have the same image via μ, then C_1 and C_2 have the same Σ_{NAT}-part and the carriers C_{1cub}, C_{2cub} consists of elements which are the same as for what concerns the rectangle part; moreover, c_1 and c_2 must have the same rectangle part. Thus, since the effect of $double^{\mathcal{C}2}$ is local, i.e. a change of the argument cuboid, then the results in the two cases are two configurations still with the same rectangle view, and the two changed cuboids have still the same rectangle part.
  Notice, on the contrary, that $\mathcal{C}1$ clearly does not meet this regularity condition.

In this case, then we say that there is a *regular inheritance relation from $\mathcal{D}'$ into $\mathcal{D}$*, as formally defined below.

Def. 2.8 Let $\mathcal{D} = (IA, \{dop^{\mathcal{D}}\}_{dop \in DOP})$ be a d-oid over $D\Sigma = (\Sigma, DOP)$, $\mathcal{D}' = (IA', \{dop^{\mathcal{D}'}\}_{dop \in DOP})$ be a d-oid over $D\Sigma' = (\Sigma', DOP')$, and assume that there exist $\sigma: D\Sigma \to D\Sigma'$ dynamic signature morphism s.t. the following conditions hold:

- $\sigma: D\Sigma \to D\Sigma'$ is a minimal inheritance relation from $\mathcal{D}'$ into $\mathcal{D}$;
- $\varphi: IA'_{|\sigma} \to IA$ is an instant morphism;
- for every $dop \in DOP$ and for every $A', B' \in IA'$

  if
  $$\begin{array}{l} \sigma(dop)^{\mathcal{D}'}_{<A',\bar{a}'>} = <f': A' \Rightarrow C'[, c']> \\ \sigma(dop)^{\mathcal{D}'}_{<B',\bar{b}'>} = <g': B' \Rightarrow D'[, d']> \\ \varphi(A'_{|\sigma}) = \varphi(B'_{|\sigma}) \\ \varphi_{A'_{|\sigma}}(\bar{a}') = \varphi_{B'_{|\sigma}}(\bar{b}') \end{array}$$
  then

  * $\varphi(C'_{|\sigma}) = \varphi(D'_{|\sigma})$

  ** $\forall a' \in A'_{|\sigma}, b' \in B'_{|\sigma}, \varphi_{A'_{|\sigma}}(a') = \varphi_{B'_{|\sigma}}(b') \supset \varphi_{C'_{|\sigma}}(f'_{|\sigma}(a')) = \varphi_{D'_{|\sigma}}(g'_{|\sigma}(b'))$

*** $\varphi_{C'|_\sigma}(c') = \varphi_{D'|_\sigma}(d')$,

then we say that $\mathcal{D}'$ *regularly inherits* $\mathcal{D}$ *via* σ *and* φ and that there is a *regular inheritance relation from* $\mathcal{D}'$ *into* $\mathcal{D}$. □

Prop. 2.9 If $\mathcal{D}' = (IA', \{dop^{\mathcal{D}'}\}_{dop \in DOP})$ regularly inherits $\mathcal{D}'$ via σ and φ, then we can define a d-oid $\bar{\mathcal{D}} = (\overline{IA}, \{dop^{\bar{\mathcal{D}}}\}_{dop \in DOP})$ over $D\Sigma$, called the $\mathcal{D}$ *view of* $\mathcal{D}'$ *via* σ *and* φ, in this way:

- $\overline{IA} = \{\overline{A} \mid \overline{A} = \varphi_{A'|_\sigma}(A'|_\sigma), \varphi_{A'|_\sigma} \in \varphi\}$

- if $\sigma(dop)^{\mathcal{D}'}_{<A', \bar{a}'>} = <f' \colon A' \Rightarrow B'[, b']>$
  then $dop^{\bar{\mathcal{D}}}{}_{<\varphi(A'), \varphi_{A'|_\sigma}(\bar{a}')>} = <f \colon \varphi(A') \Rightarrow \varphi(B')[, \varphi_{B'|_\sigma}(b')]>$
  where f is defined as follow:
  $f(\varphi_{A'|_\sigma}(a')) = \varphi_{B'|_\sigma}(f'|_\sigma(a'))$ □

Note that the view of the heir d-oid in the parent d-oid is different from the parent d-oid since the interpretation of methods are different. Referring to the preceding example of $\mathcal{R}$ and $\mathcal{C}2$, in the $\mathcal{R}$ view of $\mathcal{C}2$ the interpretation of *double* is a method which leaves a rectangle unchanged, while in $\mathcal{R}$ it is a method that doubles the two dimensions.

In other words, it does not happen that methods in the parent view of the heir and methods in the parent have "analogous" effects; from the formal point of view, that corresponds to the fact that the instant morphism φ does not respect methods, i.e. is not required to be a morphism of d-oids.

We will illustrate in next subsection the case in which this further condition is required, which corresponds to what we call *conservative inheritance*.

2.3 Conservative inheritance

In the preceding subsection we have shown an example of regular inheritance relation which was not *conservative*, in the sense that the behavior of a redefined method over the parent view of the heir was different from the old one.
We show now an example of conservative inheritance.

Example 2.10 Consider the following class declaration

height

...

...

□

Let us fix our attention on the usual method `double`: this dynamic operation has the effect of doubling all the three dimensions of the cuboid upon which is invoked. It is quite evident that the methods `double` is regular: the attributes inherited from the class of rectangles `length` and `width` do not depend on the new attribute `height`, so we can forget the new attribute and obtain a method on the class which is, exactly, the same method defined in . It is easy to check that also the other redefined methods of are regular and, when reduced to the class , they coincide with the old ones.

From the formal point of view we have that, if $\mathcal{C}3 = (IA_{\mathcal{C}3}, \{dop^{\mathcal{C}3}\}_{dop \in DOP_{C3}})$ is a possible model of the class where $IA_{\mathcal{C}3} = IA_{\mathcal{C}1}$, then the $\mathcal{R}$ view of $\mathcal{C}3$ is exactly $\mathcal{R}$, because of the particular correspondence between $IA_{\mathcal{R}}$ and $IA_{\mathcal{C}1}$; in general, what we expect in the case of a conservative inheritance relation, is something less strict: for example, there may exists some form of restriction on the class of instant algebras of $\mathcal{C}3$ s.t. not all the instant algebras of $\mathcal{R}$ have a counterpart in $\mathcal{C}3$ and not all the rectangles of an instant algebra of $\mathcal{R}$ are the image of a cuboid in the corresponding instant algebra.
Formally what we ask is that the $\mathcal{R}$ view of $\mathcal{C}3$ be a sub-d-oid of $\mathcal{R}$ (formally, in the category sense of sub-object, i.e. there is an injective morphism from the sub-d-oid into $\mathcal{R}$).

Def. 2.11 Let $\mathcal{D} = (IA, \{dop^{\mathcal{D}}\}_{dop \in DOP})$ be a d-oid over $D\Sigma = (\Sigma, DOP)$, $\mathcal{D}' = (IA', \{dop^{\mathcal{D}'}\}_{dop \in DOP})$ be a d-oid over $D\Sigma' = (\Sigma', DOP')$; if $\mathcal{D}'$ regularly inherits $\mathcal{D}$ via σ and φ and the $\mathcal{D}$ view of $\mathcal{D}'$ via σ and φ is a sub-d-oid of $\mathcal{D}$, then we say that $\mathcal{D}'$ *conservatively inherits* $\mathcal{D}$ *via* σ *and* φ and that there is a *conservative inheritance relation from* $\mathcal{D}'$ *into* $\mathcal{D}$. □

Related work and conclusion

First of all we say cleraly and loudly, in all modesty, that we do not pretend by this approach to present an ideal and comprehensive model of classes and inheritance. The aim is much narrower: to upgrade the classical static data type approach to a rather natural treatment of dynamic entities, in the case that dynamics is performed by method invokation and not by interaction among the objects, say by communication and synchronization mechanisms. For example, it is well known that we could model a stack object as a process; this is the approach taken in the Edinburgh CCS and π-calculus school, in the dutch process algebra school and, in the context of algebraic specification of concurrency, by the dynamic specification/SMoLCS approach (see [AGRZ89] and [EGS92], also for references to other work).

The d-oid formalism has been introduced by two of the authors in [AZ92a], where it is shown that d-oids can provide a semantic model for a kernel language for defining methods. The mathematical foundations of the theory of d-oids, as dynamic data types, is presented in [AZ92b].

The idea of modelling each configuration as an algebra is not new: it has been used in the COLD language (see [WB87], part II), for long time within the structural inductive semantics (see [Ast90]) and also much emphasized in the evolving algebra approach (see e.g. [Gur91]). The novelty of the proposed approach is twofold. First we embody the dynamics (the evolution) into the structure: a d-oid is an overall structure which covers both static and dynamic aspects in a uniform way, extending the usual data type concept. The second novelty lies in the way we formalize method calls as transformations with the notion of tracking map, which plays a fundamental role in the definition of semantics.

What we do here is in the spirit of the "languages for data directed design", discussed in [Wag91]; from this viewpoint classes are a way of defining data and inheritance is a tool for building classes incrementally.

Semantics of incremental definition of classes is ouside of the scope of this paper and will be the subject of a forthcoming one. Then the problem of method redefinition and its link with the semantics of "self" is handled by assigning a class a (compositional) semantics, consisting of a family of d-oids parameterized over the semantics of the parent classand of the methods that can be redefined. This parameterization is, in a very different and purely applicative setting, also at the root of the Cook's approach (see [Coo89], [Weg87]). Clearly our model departs dramatically from the applicative approach, since we want to deal with states and we feel that without this it is difficult to capture the essence of objecst and methods as they are understood in most real life settings.

Some very interesting work in the line of the abstract data type approach has been presented in [GM87] and more recently in [Bre91]. The difference is that we have an abstract notion of state and the dynamics, via the notion of methods, is incorporated in the structure; while in those approaches everything was modelled by a reduction to a classical static data type view. In the field of the object oriented data bases, the work of Beeri (see [Bee91]) presents a very interesting abstract data type approach to model query languages; we feel that d-oids could provide a key for extending his work to treat also update languages and related issues.

The fact that we are not treating objects as processes does not mean that we underestimate the important issue of concurrency (see indeed the work by some of us, e.g. in [AGRZ89]); simply the d-oid model has different target, the case without communication and explicit parallelism (as in Eiffel, C++ and Smalltalk). One of the topics for future work will be to relate d-oids to the process view taken by many authors in a setting still closed to the abstract data type approach, but with objects as processes, whose state is represented by a term in the context of an overall algebraic specification. Among them, the dynamic specifications (or algebraic transition systems) of [AGRZ89], with their evolution as processes with identity and sharing (the entity algebras of [Reg91]) and the very much related approach with a rewriting flavour of rewriting logic in [Mes90].

Finally, a much broader and very interesting view of classes and inheritance is taken in [EGS92], where objects are seen as processes but related to a class by means of object templates. The relationship with that work is an interesting but demanding issue, that we plane to pursue in the context of the ESPRIT-BRA WG ISCORE.

References

[AGRZ89] E. Astesiano, A. Giovini, G. Reggio, and E. Zucca. *An integrated algebraic approach to the specification of data types, processes and objects*, pages 91–116. Number 394 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1989.

[Ast90] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal description of programming concepts*, Berlin, 1990. Springer Verlag.

[AZ92a] E. Astesiano and E. Zucca. A semantic model for dynamic systems. In U.W. Lipeck and B. Thalheim, editors, *Fourth International Workshop on Foundations of Models and Languages for Data and Objects - Modelling Database Dynamics*, Volkse (Germany), October 1992.

[AZ92b] E. Astesiano and E. Zucca. Towards a theory of dynamic data types. In *9th ADT Workshop*, Caldes de Malavella, October 1992. full paper in preparation.

[Bee91] C. Beeri. Theoretical foundations for oodb's - a personal perspective. *Database Engineering*, 1991. to appear.

[Bre91] R. Breu. *Algebraic specification techniques in object oriented programming environment.* PhD thesis, Universität Passau - TU München, Berlin, 1991.

[Coo89] W. Cook. *A denotational semantics of inheritance*. PhD thesis, Brown University, 1989.

[EGS92] H. D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their specification. In C. Choppy M. Bidoit, editor, *Recent Trends in Data*

Type Specification, number 655 in Lecture Notes in Computer Science, Berlin, 1992. Springer Verlag.

[GM87] J.A. Goguen and J. Meseguer. *Unifying functional, object-oriented and relational programming with logical semantics*, pages 417–477. MIT Press, 1987.

[Gur91] Y. Gurevich. Evolving algebras, a tutorial introduction. *Bulletin of the EATCS*, (43):264–284, 1991.

[JWB90] Ralph E. Johnson and Rebecca J. Wirfs-Brock. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, september 1990.

[Mes90] J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA '90 Conference on Object-Oriented Programming*, pages 101–115, Ottawa (Canada), october 1990. ACM.

[Reg91] G. Reggio. Entities: Institutions for dynamic systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *7th Workshop on Specification of Abstract Data Types*, number 534 in Lecture Notes in Computer Science, Berlin, 1991. Springer Verlag.

[Wag91] E. G. Wagner. Some mathematical thoughts on languages for data directed design. Technical Report RC 16686 (73950), IBM Research Division, 1991.

[WB87] M. Wirsing and J. A. Bergstra, editors. *Algebraic Methods: Theory, Tools and Applications.* Number 394 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1987.

[Weg87] P. Wegner. Dimensions of object based language design. In *Proc. OOPSLA '87*, pages 168–182, 1987.

Appendix

If A is a set, then id_A denotes the identity of A.

A *signature* is a couple $\Sigma = (S, OP)$, where S is a set of *sort symbols* and OP is a $S^\star \times S$-indexed family of *operation symbols*. If $op \in OP_{w,s}$, $w = s_1 \dots s_n$, then we write

$$op\colon s_1 \dots s_n \to s.$$

We assume, for simplifying the notation, that signatures have no overloading, i.e. , for every signature $\Sigma = (S, OP)$, if $op \in OP_{w,s}$, $op \in OP_{w',s'}$, then $w = w'$ and $s = s'$.

A *signature morphism* σ *from* Σ *into* Σ', *written* $\sigma: \Sigma \rightarrow \Sigma'$, is a couple $<\sigma^S, \sigma^O>$ where:

- $\sigma^S: S \rightarrow S'$;
- $\sigma^O: OP \rightarrow OP'$ is a family of functions s.t.

$$\sigma^O(op: s_1 \dots s_n \rightarrow s): \sigma^S(s_1) \dots \sigma^S(s_n) \rightarrow \sigma^S(s)$$

. $\square$

We will often write simply σ instead of σ^S, σ^O.

An *algebra over* $\Sigma = (S, OP)$, also called Σ-*algebra*, is a couple $A = (|A|, \{op^A\}_{op \in OP})$ where:

- $|A|$ is a S-indexed family of sets; for each $s \in S$, $|A|_s$ is called the *carrier of sort* s and denoted also by A_s;
- for each $op: s_1 \dots s_n \rightarrow s$, $op^A: A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$ is called *the interpretation of op in A*

$Alg(\Sigma)$ denotes the class of the algebras over Σ.

A Σ-*homomorphism from A into B, where A, B are algebras over* Σ, *written* $f: A \rightarrow B$, is a S-indexed family of functions s.t. :

- for each $s \in S$, $f_s: A_s \rightarrow B_s$,
- for each $op: s_1 \dots s_n \rightarrow s$, $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$,

$$f_s(op^A(a_1, \dots, a_n)) = op^B(f_{s_1}(a_1), \dots, f_{s_n}(a_n))$$

If $\sigma: \Sigma \rightarrow \Sigma'$ is a signature morphism, A' an algebra over Σ', then the *reduct of* A' w.r.t. σ, *written* $A'_{|\sigma}$, is the algebra A over Σ defined as follows:

- for each $s \in S$, $A_s = A'_{\sigma(s)}$;
- for each $op: s_1 \dots s_n \rightarrow s$ in Σ, $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$,

$$op^A(a_1, \dots, a_n) = \sigma(op)^{A'}(a_1, \dots, a_n).$$

In particular, if $\iota: \Sigma \rightarrow \Sigma'$ is the embedding, then we write also $A'_{|\Sigma}$ instead of $A'_{|\iota}$