Proc. 8th Workshop on Abstract Data Types (ADT'91)
M. Bidoit, C. Choppy (Eds.), Springer, Berlin, LNCS 655, pp. 40-65

Objects and Their Specification *

Hans-Dieter Ehrich
Martin Gogolla
Abteilung Datenbanken, Technische Universität, Postfach 3329
W–3300 Braunschweig, GERMANY

Amilcar Sernadas
Computer Science Group, INESC, Apartado 10105
1017 Lisbon Codex, PORTUGAL

Abstract

Object–oriented concepts and constructions are explained in an informal and language–independent way. Various algebraic approaches for dealing with objects and their specification are examined, ADT–based ones as well as process–based ones. The conclusion is that the process view of objects seems to be more appropriate than the data type view.

1 Introduction

There is a peculiar confusion around the notions of object and abstract data type in practice: while the latter has been made mathematically precise as an isomorphism class of algebras, practitioners tend to view an abstract data type as an encapsulated module exporting a set of procedures through which its data can be accessed and manipulated. The bridge to the theory is to look at the module's set of internal states as a carrier, and its procedures as operations of an algebra.

The problem with this view is that it blurs the distinction between data and objects.

The integers constitute a basic example of a data type. Another popular example is a stack. Pragmatically, however, integers and stacks are quite different. The integer data type provides a supply of values which can be used, say, as actual parameters in procedure calls. A stack, on the other hand, is a unit of structure and behavior: its states are not meant to be used as actual parameters in procedure calls. Rather, the stack *as a whole* is subject to operations. An integer is added to another integer to give a result which is a third integer. A stack entry, however, is pushed onto a stack, and we still look at the latter as being the same stack, although its state has changed.

*This work was partly supported by the EC under ESPRIT BRA WG 3264 COMPASS, under ESPRIT BRA WG 3023 IS-CORE and by JNICT under PMCT/C/TIT/178/90 FAC3 contract.

That is, a stack is an *object*, not a data type: it has an *identity* which persists through change. It seems that, for the most part, the practical success of abstract data types is one of object–orientation.

There are many languages, systems, methods and approaches in computing which call themselves "object–oriented" by now, among them object–oriented programming languages like SmallTalk [GR83], C++ [St86] and Eiffel [Me88], object–oriented database systems like GemStone [BOS91], O_2 [De91], IRIS [Fi87] and ORION [Ki88], and object–oriented system development methods like GOOD [SS86], MOOD [Ke88] and HOOD [Hei88].

High–level system specification languages and design methodologies are evolving which are based on object–oriented concepts and techniques. [Ve91] gives an overview of recent work in this area. We are cooperating in the ESPRIT BRA Working Group IS-CORE where a graphical language [SGCS91, SRGS91, SSGRG91, SGGSR91] and a textual counterpart [JSS90, JHSS91, JSS91, JSHS91, SJ91] for designing, specifying and implementing object communities are being developed.

But what precisely **is** an object? As we have seen, it has an internal state and a certain behavior reflected by its operations: it is a *unit of structure and behavior* — and it has an *identity* which persists through change. Moreover, dynamic objects somehow *communicate* with each other, they are classified by object *types*, collected into object *classes*, related by various forms of *inheritance*, and composed to form *complex* objects.

This rich world of concepts and constructions seems to be very fertile: An enormous amount of work is being invested in developing object–oriented techniques for software engineering. Evidently, there is much hope that software production and maintenance can be made more effective, more productive, more adequate, and more reliable this way. Indeed, object–oriented languages and systems as well as design and implementation methods are invading all disciplines of software engineering.

With all these practical developments, it is amazing that theoretical foundations for object–oriented concepts and constructions do not have found so wide attention yet. Matters are changing slowly: there are formal approaches to object–oriented programming language semantics [CP89], database concepts [Be91, GKS91], and specification languages [GW90]. Besides this, also language– and system–independent discussions of fundamental object–oriented issues are evolving [Cu91, HC89, LP90].

In the IS-CORE working group, the first and third authors have been cooperating in the latter direction. Recent contributions to semantic fundamentals are [ESS90, ES90, EGS91, CS91, CSS91, SE90, SEC90, SFSE89], emphasizing the process view of objects. In cooperation, logic fundamentals of object specification have been developed [FM91a, FM91b, FS91, FSMS90]. A first result harmonizing logics and semantics of object specification can be found in [FCSM91].

A systematic formalization of basic object–oriented concepts and constructions in terms of this theory has been published in [ES91] and, together with features of an object–oriented specification language and methodology, in [SJE91].

In the second section of this paper, we give an informal account of these ideas. In the third section, we examine various approaches for dealing with objects and their specification, ADT–based ones as well as process–based ones. The conclusion is that the process view of objects seems to be more appropriate than the data type view.

2 Object–Oriented Concepts and Constructions

In this section, we give an informal account of our view of object–oriented concepts and constructions. The purpose is to establish requirements for formalization of object–orientation, not a formalization itself: we declare the intuitive basis on which to evaluate formal approaches. First steps towards formalization are discussed in section 3.

We feel that it is necessary to discuss the intuitive background at some length, because of the confusion and disagreement on fundamental terms in this area. We took some pain to make the definitions precise enough to serve their purpose, but they are not meant to be a formalization yet.

In order to be short and concise, we present one particular view rather than giving a review of alternative opinions and philosophies. Whenever appropriate, however, we drop a hint at other viewpoints.

2.1 Templates

In natural language, we refer to objects by substantives, but we use the same substantive in two different ways: with the definite article **the** (or words like **this** or **that**) for referring to specific individual objects, and with the indefinite article **a** for referring to generic terms.

The distinction between individual objects and generic terms is somewhat sloppy in natural language. Consider, for example, the sentence

- *This computer is a SUN workstation; it works quite well.*

Does the speaker want to say that the specific SUN workstation referred to by *this* works quite well, or does she want to say that SUN workstations *in general* work quite well? The first meaning is probably more obvious, but you can hear people talk like this with the second meaning in mind.

In computing, we have to be very specific about this distinction. For generic terms, the word *type* is often used, but this word is overloaded with too many connotations. We avoid it and prefer to speak of *object templates* if we mean generic objects without individual identity. The notion of an *object class* is easily confused with this, but it means something else, namely a time–varying collection of objects! We will be back to this.

An object template represents the common structure and behavior pattern of some kind of object.

The basic ingredients of structure and behavior are *observations* and *actions*. For instance, for a queue object with integer entries, we might have

- observations `front=7` , `rear=2` , `size=3` , ...
- actions `create` , `enter(7)` , `leave` , ...

with obvious meanings. It is essential that we are general enough to allow for *concurrency*. In a nonempty queue, for instance, `enter`(4) and `leave` may occur simultaneously, and this simultaneous occurrence can be considered as *one* composite action `enter`(4)‖`leave`. Moreover, also actions and observations can occur at the same time, giving rise to expressions like `enter`(4)‖`front`=7 or `size`=3‖`rear`=2‖`leave`.

But what do the latter expressions mean? They are neither pure actions nor pure observations. In order to make sense of this, we introduce the concept of *event* as a generalization of actions and observations:

- *an event is anything which can occur in one instant of time.*

Let E be a given set of events including atomic actions, atomic observations, and simultaneous combinations of finitely many of these. Obviously, we have a binary operation $\|$ on E which should be associative, commutative and idempotent, and it should satisfy the cancellation law. Adding, for completeness and convenience, a neutral element 1 standing for *nothing happens* (or, rather, nothing *visible* happens, i.e., it might represent a *hidden* event), we obtain an

- *event monoid* $\bar{E} = (E, \|, 1)$

with the above properties as the basic event structure to work with.

An object template has an event monoid associated with it, representing the events which can possibly happen to this kind of object. But, of course, it is not sufficient to know *which* events can happen, we need to know *how* they can happen in sequence or concurrently.

A *process* is a well known concept modelling precisely this. There are many process models and languages in the literature, including CSP, CCS, ACP, Petri nets, labelled transition systems, various trace models, etc. We cannot go into process theory here, and fortunately we need not: we are ready to accept any sufficiently powerful process model for modelling object template behavior.

In order to help the reader's intuition, however, she might envisage labelled transition systems (lts) as an example process model: an object template has states, and it can move (non–deterministically) from state to state by transitions labelled by events (only actions will actually change state, observations will leave it fixed). A mathematical elaboration of this model can be found in [CSS91], and a more abstract denotational model is outlined in [ES91]. Other appropriate process categories are currently being investigated, denotational and operational ones [CS91, CSS91], and also logic–based ones [FM91a, FSMS90]. An interesting unifying approach can be found in [Me91].

Templates in isolation, however, are not enough for capturing the relevant object–oriented concepts: for studying inheritance and interaction, we have to deal with suitable *relationships between* templates. In this respect, process theory offers only rudimentary help. We found it necessary to develop a general and powerful notion of *process morphism* as some kind of "behavior preserving map" between processes [ES91, ESS90, ES90, SE90, SFSE89, SSE87]. Amazingly, one single concept turned out to be sufficient for dealing with inheritance as well as interaction!

Templates and template morphisms constitute a well known mathematical structure called a *category*. We have been able to find instances of process categories where not only the morphisms are appropriate for modelling inheritance and interaction, but where also fundamental process operations are reflected by basic categorial constructions: parallel composition by limits, and internal choice by colimits [ES91, CS91].

In what follows, we will need one special case of template morphism in particular, namely *projection*: a template is projected to a part or an aspect of itself by mapping all "global" states to their "local" part or aspect, and correspondingly for transitions. The events relevant for the part or aspect are maintained, while the remaining events are "hidden" by mapping them to 1. Please note that nondeterminism might be introduced this way.

For example, let `twoqueue` be the template for objects consisting of two queues working in parallel, without any interaction between them. The states of `twoqueue` are all pairs (q_1, q_2) of states of the two component queues, whereas the events (labels) are given by the product of the two event monoids, i.e., all events of the form $e_1 \parallel e_2$ where e_1 is an event of the first queue and e_2 is one of the second. Let `queue` be the template for just one such queue. Then we have two obvious projections $p_i : \texttt{twoqueue} \rightarrow \texttt{queue}, i = 1, 2$, where p_i leaves the $i-$th queue fixed and "forgets" the other one by mapping all actions and all observations to 1.

For another example, let `queue` be as above, and let `deque` (double–ended queue) be like a queue, but with actions to enter and to leave at both ends: there is an obvious "abstraction" $a : \texttt{deque} \rightarrow \texttt{queue}$ leaving the states fixed but "forgetting" the additional actions.

These examples demonstrate the twofold use of template morphisms: for restricting to a constituent part and for abstracting an aspect.

2.2 Objects

What is an object ? Its behavior is a process, but an object is more than its behavior: there may be many objects with the same behavior, but they are different as objects. That is, an object has an *identity* while a process has not. Only if we can distinguish clearly between individual objects is it possible to treat object concepts like inheritance and interaction in a clean and satisfactory way: interaction is a relationship between *different* objects, while inheritance relates aspects of the *same* object.

Object identities are atomic items whose principle purpose is to characterize objects uniquely. Thus, the most important properties of identities are the following: we should know which of them are equal and which are not, and we should have enough of them around to give all objects of interest a separate identity. Identities are associated with templates to represent individual objects — or, rather, *aspects* of objects, as we will see.

Given templates and identities, we may combine them to pairs $b{\bullet}t$ (to be read "*b as t*"), expressing that object b has behavior pattern t. But there are objects with several behavior patterns! For instance, a given person may be looked at as an employee, a patient, a car driver, a person as such, or a combination of all these aspects. Indeed, this is at the heart of inheritance: *$b{\bullet}t$ denotes just one aspect of an object* – there may be others with the same identity!

Definition 2.1 : An *object aspect* – or *aspect* for short – is a pair $b{\bullet}t$ where b is an identity and t is a template.

Definition 2.2 : Let $b{\bullet}t$ and $c{\bullet}u$ be two aspects, and let $h : t \to u$ be a template morphism. Then we call $h : b{\bullet}t \to c{\bullet}u$ an *aspect morphism*.

Aspect morphisms are nothing else but template morphisms with identities attached. The identities, however, are not just decoration: they give us the possibility to make a fundamental distinction between the following two kinds of aspect morphisms.

Definition 2.3 : An aspect morphism $h : b{\bullet}t \to c{\bullet}u$ is called an *inheritance morphisms* iff $b = c$. Otherwise, it is called an *interaction morphism*.

The following example illustrates the notions introduced so far.

Example 2.4 : Let `el_dvice` be a behavior template for electronic devices, and let `computer` be a template for computers. Assuming that each computer IS An electronic device, there is a template morphism h : `computer` $\to$ `el_dvice` (roughly speaking, the `el_dvice` part in `computer` is left fixed, while the rest of `computer` is projected to 1).

If `SUN` denotes a particular computer, it has the aspects

`SUN.computer`	(`SUN` as a computer)	and
`SUN.el_dvice`	(`SUN` as an electronic device),	

related by the inheritance morphism h : `SUN.computer` $\longrightarrow$ `SUN.el_dvice` .

Let `powsply` and `cpu` be templates for power supplies and central processing units, respectively. Assuming that each electronic device HAS A power supply and each computer HAS A cpu, we have *template* morphisms f : `el_dvice` $\to$ `powsply` and g : `computer` $\to$ `cpu`, respectively. If `PXX` denotes a specific power supply and `CYY` denotes a specific cpu, we might have *interaction* morphisms f' : `SUN.el_dvice` $\to$ `PXX.powsply` and, say, g' : `SUN.computer` $\to$ `CYY.cpu`. f' expresses that the `SUN` computer – as an electronic device – HAS THE `PXX` power supply, and g' expresses that the `SUN` computer HAS THE cpu `CYY`.

These examples show special forms of interaction, namely between objects (aspects) and their *parts*. More general forms of interaction are established via *shared parts*. For example, if the interaction between `SUN`'s power supply and cpu is some specific cable `CBZ`, we can naively view the cable as an object `CBZ.cable` which is part of both `PXX.powsply` and `CYY.cpu`. This is expressed by a *sharing diagram*

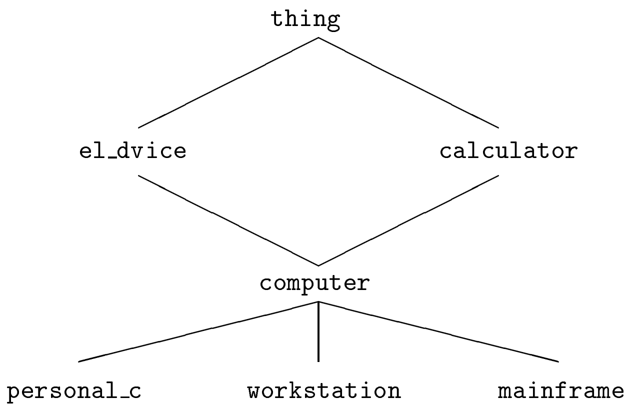
$$\texttt{CYY.cpu} \longrightarrow \texttt{CBZ.cable} \longleftarrow \texttt{PXX.powsply}$$

A more realistic way of modeling this would consider the cable as a separate object not contained in the cpu and not in the power supply either. Rather, the cable would share contacts with both. ∘

This shows that objects may appear in different aspects, all with the same identity but with different behavior templates, related by inheritance morphisms. The information which aspects are related by inheritance morphisms is usually given by *template* morphisms *prescribing* inheritance. For example, we specify h : `computer` $\to$ `el_dvice` in order to express that each computer IS An electronic device, imposing that whenever we have an instance computer, say `SUN.computer`, then it necessarily IS THE electronic device `SUN.el_dvice` inherited by h as an aspect morphisms, h : `SUN.computer` $\to$ `SUN.el_dvice` .

Definition 2.5 : Template morphisms intended to prescribe inheritance are called *inheritance schema morphisms.* An *inheritance schema* is a collection of templates related by inheritance schema morphisms.

Example 2.6 : In the following inheritance schema, arrowheads are omitted: the morphisms go upward.



$\circ$

Practically speaking, we create an object by providing an identity b and a template t. Then this object $b{\bullet}t$ has *all* aspects obtained by relating the same identity b to all "derived" aspects t' for which there is an inheritance schema morphisms $t \to t'$ in the schema.

Thus, an object is an aspect together with all its derived aspects. All aspects of one object have the same identity – and *no other* aspect should have this identity!

But the latter statement is not meaningful unless we say which aspects are there, i.e., we can only talk about objects *within a given collection of aspects.* Of course, the collection will also contain aspect morphisms expressing how its members interact, we will be back to this. And if an aspect is given, all its derived aspects with respect to a given inheritance schema should also be in the collection.

Definition 2.7 : An *aspect community* is a collection of aspects and interaction morphisms. It is said to be *closed* with respect to a given inheritance schema iff, whenever an aspect $a{\bullet}t$ is in the community and an inheritance morphism $t \to t'$ is in the schema, then we also have $a{\bullet}t'$ in the community.

Definition 2.8 : An *object community* – or *community* for short – consists of an inheritance schema and an aspect community which is closed with respect to it.

Definition 2.9 : Let an object community be given, and let a be an object identity. The *aspect graph* of a in that community is the graph consisting of all aspects in the community with the identity a as nodes, and all inheritance morphisms lifted from the schema as edges.

By lifting we mean that whenever $a{\bullet}t$ and $a{\bullet}t'$ are in the aspect graph of object a and $t \to t'$ is in the schema, then also $a{\bullet}t \to a{\bullet}t'$ is in the aspect graph.

One could argue that the object with identity a is the aspect graph of a. Intuitively, an object with identity a is a collection of consistent aspects. But, as we will see, we take a simpler and more practical approach.

Example 2.10 : Consider an object community containing the inheritance schema in Example 2.6, a particular workstation named `SUN`, and a particular calculator named `UPN`. By inheritance, `SUN` automatically is a computer, an electronic device, a calculator, and a thing. Since `UPN` is a calculator, it is also a thing, etc. So we have the following object diagrams:

$$
\begin{tikzcd}
 & \texttt{SUN.thing} \arrow[dl, dash] \arrow[dr, dash] & & \texttt{UPN.thing} \arrow[d, dash] \\
\texttt{SUN.el\_dvice} \arrow[dr, dash] & & \texttt{SUN.calculator} \arrow[dl, dash] & \texttt{UPN.calculator} \\
 & \texttt{SUN.computer} \arrow[d, dash] & & \\
 & \texttt{SUN.workstation} & &
\end{tikzcd}
$$

$\circ$

In a given community, an object is usually constructed by picking a specific identity a and associating it with a specific template t, yielding a *core* aspect $a{\bullet}t$ for this object. Then the object diagram of a is determined by all aspects $a{\bullet}t'$ where t' is related to t by an inheritance schema morphism $t \to t'$. Consequently, object diagrams have an inheritance morphism from the core aspect to any other aspect.

Definition 2.11 : An object community is called *regular* iff each aspect graph of an object in it has a core aspect.

While the notion of object should probably be taken as that of its aspect graph in general, we can make things easier and closer to popular use in a regular community: here it is safe to identify objects with their core aspects.

Definition 2.12 : In a regular community, an *object* b is the core aspect of its aspect graph.

Clearly, from the core aspect and the inheritance schema we can recover the entire aspect graph.

Since, according to this definition, objects are special aspects, we immediately have a notion of object morphism: it is an aspect morphism between objects.

2.3 Classes

Objects rarely occur in isolation, they usually appear as members of *classes*. A warning is in order: the notion of class is subject to considerable confusion! Essentially, there seem to be two different schools. The first one says that a class is a sort of abstraction of patterns, i.e., an intensional description of an invariant set of potential members. The other one says that a class is an extensional collection of objects which may vary from state to state. Both notions are needed. We capture the first one by *templates*, and we use the word class in the second sense.

That is, a class is a time–varying collection of objects as members – and this means that a class is a particular kind of object itself!

Or should we say *aspects* rather than objects? With the distinction between objects and aspects made above, we have to be careful with what can be a member of a given class, and whether a class is an aspect or an object. Let us first look at the member problem.

Example 2.13 : Referring to the inheritance schema in Example 2.6, let `CEQ` – the computer equipment – be a class of computers of some company `Z`. Let `MAC` be a specific personal computer in `Z`, and let `SUN` be a specific workstation in `Z`. The question is: are the objects `MAC.personal_c` and `SUN.workstation` members of `CEQ`, or rather their aspects `MAC.computer` and`SUN. computer`? ∘

It is easier to work with *homogeneous* classes where all members have the same template, so we formally adopt the second alternative: each class has a fixed member template. We call this member template the *type* of the class. But, since each aspect of an object carries its identity and thus determines the object uniquely, there is no objection to considering, for example, the `MAC.personal_c` a member of the class `CEQ`.

Therefore, while classes are formally homogeneous, they have a heterogeneous – or polymorphic – flavor when working with inheritance: each object with an appropriate aspect whose template is the type of the class can be a member of that class!

Classes can be specialized by inheritance. For example, if we define a club as a class of persons, we might subsequently define special classes like a football club, a motor club, and a chess club.

Therefore, we consider classes as aspects. The class events are actions like inserting and deleting members, and observations are attribute-value pairs with attributes like the current number of members and the current set of (identities of) members. In most object–oriented systems, standard class events are provided implicitly, they need not be specified by the user.

Definition 2.14 : Let t be a template. An *object class* – or *class* for short – of type t is an aspect $C = a_C {\bullet} t_C$ where a_C is the *class name* and $t_C = (E_C, P_C)$ is the *class template*. If `ID` is a given set of identities, the *class events* E_C contain

- actions `insert(ID)` , `delete(ID)`
- observations `population=set(ID)` , `#population=nat` .

The *class process* P_C describes the proper dynamic behavior in terms of the class events. ∘

In practice, we would probably have the information in the environment which member identities can go with which class, i.e., some typing of identities. In this case, the argument `ID` in the above definition should be replaced by `ID`(C), the set of member identities which can be used in class C, and the notion of class type should comprise `ID`(C) along with the member template t.

Definition 2.15 : Let $C = a_C \bullet t_C$ be a class of type t. An *aspect* $a \bullet t$ is called a *member* of C iff a is an element of the population of C. An *object* $b \bullet u$ is called a *member* of C iff it has an aspect $b \bullet t$ which is a member of C.

This definition justifies our calling a class an *object* class, not an *aspect* class: the members may be considered to be the objects having the relevant aspects, emphasizing the polymorphic viewpoint.

Since classes are objects or aspects of objects, there is no difficulty in constructing meta–classes, i.e., classes of classes of ...

Definition 2.16 : A class C is called a *meta–class* iff its type is a class template.

Since class templates tend to be homogeneous even if their types are not, a meta–class may have classes of different types as members. For example, we could define the class of all clubs in a given city without generalizing the club member templates so as to provide an abstract and uniform one for all clubs.

Sometimes, we might want to restrict the members of a meta–class to contain sub–populations of a given class. For example, we may devise classes `CEQ(D)` for the computer equipment of each department `D` of company `Z`, given the class `CEQ` of computers in the company (cf. Example 2.13).

Definition 2.17 : Let C_1 and C_2 be classes. C_1 is called a *meta–class of* C_2 iff (1) the type of C_1 is the template of C_2, and (2) each member of C_1 is a class whose population is a subset of that of C_2.

Since classes are aspects, we immediately have a notion of *class morphism*: it is just an aspect morphism between classes.

Please note that the dynamic evolution of a class is completely separated from the dynamic behavior of its members. Typically, a class changes its state by inserting and deleting members, i.e., by changing its population. If the population does not change, then the state of the class does not change, even if some of its members change their state! Formally, this is achieved by having only the member *identities* in the class.

2.4 Inheritance

When we build an object–oriented system, we must provide an *inheritance schema* (cf. Definition 2.5). Without it, the very notion of object does not make sense. In this section, we investigate how to construct such an inheritance schema: which are the inheritance morphisms of interest, and how are they used to grow the schema step by step?

The inheritance morphisms of interest seem to be special indeed: in all cases we found meaningful so far, the underlying event maps were *surjective*. Since they are total anyway, this means that *all* events of both partners are involved in an inheritance relationship. And this makes sense: if we take a template and add features, we have to define how the inherited features are affected; and if we take a template and hide features, we have to take into account how the hidden features affect those inherited.

For any reasonable process model, the template morphisms with surjective event maps will be the *epimorphisms*, i.e., those morphisms r having the property that whenever $r;p = r;q$, then $p = q$. We found a special case of epimorphism useful which reflects an especially well–behaved inheritance relationship where the smaller aspect is "protected" in a certain sense: retractions. A *retraction* is a morphism $r : t \to u$ for which there is a reverse morphism $q : u \to t$ such that $q; r = id_u$. Retractions are always epimorphisms.

Intuitively speaking, the target of a retraction, i.e., the smaller aspect, is not affected by events outside its scope, it is *encapsulated.* As a consequence, retractions maintain the degree of nondeterminism: if the bigger aspect is deterministic, so is the smaller one.

Example 2.18 : Referring to Example 2.6, consider the inheritance schema morphism h : `computer` $\to$ `el_dvice` expressing that each computer is an electronic device. Let `el_dvice` have the following events:

- actions `switch_on` , `switch_off`
- observations `is_on` , `is_off`

By inheritance, `computer` has corresponding events `switch_on_c`, `switch_off_c`, etc. h sends `switch_on_c` to `switch_on` expressing that the `switch_on_c` of the computer is the `switch_on` inherited from `el_dvice`, and similarly for the other events. But what about the other events of `computer`, i.e., the ones not inherited? For example, there might be

- actions `press_key` , `click_mouse` , ...
- observations `screen=dark` , ...

Well, all these events are mapped to 1 indicating that they are *hidden* when viewing a computer as an electronic device.

Concerning the processes of the templates, we would expect that a `computer`'s behavior "contains" that of an `el_dvice`: also a computer is bound to the protocol of switching on before being able to switch off, etc.

Naturally, the template morphism h : `computer` $\to$ `el_dvice` is a retraction: there is also an embedding g : `el_dvice` $\to$ `computer` such that $g; f$ is the identity on `el_dvice`. Intuitively, this means that the `el_dvice` aspect of a computer is protected in the sense that it cannot be influenced by `computer` events which are not also `el_dvice` events: a `computer` can only be switched off by its `el_dvice` switch.

This would not be so if we had a strange computer which, say, can be switched off by other means, not using the `el_dvice` switch (perhaps by a software option...). In this case, we would have *side effects* of the computer on its `el_dvice` aspect: the latter would change its state from `is_on` to `is_off`, but would not be able to observe the reason for it locally: its `switch_off` was not used. In this case, the morphism h would still be an epimorphism, but not a retraction. Please note how nondeterminism is introduced for the local `el_dvice` aspect. ∘

Let an inheritance schema be given. If we have a surjective inheritance morphism $h : t \to u$ not (yet) in the schema, we can use it in two ways to enlarge the schema:

- if t is already in the schema, we create u and connect it to the schema via $h : t \to u$,
- if u is already in the schema, we create t and connect it to the schema via $h : t \to u$.

The first construction step corresponds to *specialization*, the second one to *abstraction.*

The most popular object–oriented construction is *specialization*, constructing the inheritance schema in a top–down fashion, adding more and more details. For example, the inheritance schema in Example 2.6 was constructed this way, moving from `thing` to `el_dvice` and `calculator`, etc. By "inheritance", many people mean just specialization.

The reverse construction, however, makes sense, too: *abstraction* means to grow the inheritance schema upward, hiding details (but not forgetting them: beware of side effects!). Taking our example inheritance schema, if we find out later on that computers – among others – belong to the sensitive items in a company which require special safety measures, we might consider introducing a template `sensitive` as an abstraction of `computer`.

Both specialization and abstraction may occur in *multiple* versions: we have *several* templates, say $u_1, \ldots, u_n$, already in the schema and construct a new one, say t, by relating it to $u_1, \ldots, u_n$ simultaneously. In the case of specialization, i.e., $h_i : t \to u_i$ for $i = 1, \ldots, n$, it is common to speak of "multiple inheritance". In the case of abstraction, i.e., $h_i : u_i \to t$ for $i = 1, \ldots, n$, we may speak of *generalization.*

Example 2.19 : Referring to Example 2.6 and assuming top–down construction, the template for `computer` is constructed by multiple specialization (multiple inheritance) from `el_dvice` and `calculator`. ∘

Example 2.20 : If we would have constructed the schema in Definition 2.6 in a bottom–up way, we would have obtained `thing` as a generalization of `el_dvice` and `calculator`.

A less contrived example of generalization, however, is the following: if we have templates `person` and `company` in our schema, we might encounter the need to generalize both to `contract_partner`. ∘

We note in passing that, with respect to objects, we have two kinds of generalization. For a computer c, its c•`thing` aspect is a proper generalization of its c•`el_dvice` and c•`calculator` aspects. We would not expect to have an object, however, which is both a person and a company. Thus, the proper generalization `contract_partner` of `person` and `company` in the schema would only appear as single object abstractions p•`person` → p•`contract_partner` or c•`company` → c•`contract_partner` on the instance level, but not as a proper object generalization.

2.5 Interaction

When we build an object–oriented system, we must provide an *object community* (cf. Definition 2.8). Without it, the very notion of object does not make sense. In what follows, we investigate how to construct such an object community: which are the interaction morphisms of interest, and how are they used to grow the community step by step?

The basic mechanism of interaction in our approach is *sharing parts*, as illustrated in example 2.4. In the simplest case, the shared part may be a single action, modeling a "global" action which is seen by all objects sharing it. In this case, the "part" plays the role of a communication *port*, and the shared ports can be seen as a communication *channel*, allowing for symmetric and synchronous communication.

Other forms of communication (asymmetric, asynchronous,...) can be modeled and explained on these grounds, but we cannot go into detail here. Please note that shared parts playing the role of communication channels are objects themselves! Thus, they may have static structure (observable attributes) as well as dynamic behavior (the communication protocol).

As with inheritance morphisms, we found that interaction morphisms are *epimorphisms* in all meaningful cases. And this makes sense, too. An interaction morphism $h : a_{\bullet}t \to b_{\bullet}u$ tells that the aspect $a_{\bullet}t$ has the part $b_{\bullet}u$, and how this part is affected by its embedding into the whole: this has to be specified for *all* items in the part!

As with inheritance morphisms, we found that *retractions* model an especially meaningful case of part–of relationship, namely *encapsulated* parts which are not affected by events outside their scope.

Example 2.21 : Referring to Example 2.4, the interaction morphisms

$$\texttt{CYY}_{\bullet}\texttt{cpu} \longrightarrow \texttt{CBZ}_{\bullet}\texttt{cable} \longleftarrow \texttt{PXX}_{\bullet}\texttt{powsply}$$

express that the cable `CBZ` is a shared part of the cpu `CYY` and the power supply `PXX`.

Suppose the events relevant for cables are voltage level observation and switch–on/switch–off actions. The sharing expresses that, if the power supply is switched on, the cable and the cpu are switched on at the same time, etc. If the cable's voltage level depends only on the shared switch actions, the cable is an encapsulated part of both cpu and power supply, and the interaction morphisms are retractions. If, however, events from outside can influence the voltage level (say, by magnetic induction), then the sharing morphisms are just epimorphisms, no retractions. ∘

Let an object community be given. If we have a surjective interaction morphism $h : a_{\bullet}t \to b_{\bullet}u$ not (yet) in the community, we can use h in two ways to enlarge it:

- if $a_{\bullet}t$ is already in the community, we create $b_{\bullet}u$ and connect it to the community via $h : a_{\bullet}t \to b_{\bullet}u$,
- if $b_{\bullet}u$ is already in the community, we create $a_{\bullet}t$ and connect it to the community via $h : a_{\bullet}t \to b_{\bullet}u$.

After connecting the new morphism to the community, we have to close it with respect to the schema (cf. Definition 2.7), i.e., add all aspects derived from the new one by inheritance.

By *incorporation* we mean the construction step of taking a part and enlarging it by adding new items. Most often the *multiple* version of this is used, taking several parts and aggregating them. We will be back to this.

The reverse construction is also quite often used in the single version, we call it *interfacing*. Interfacing is like abstraction, but it creates an object with a new identity.

Example 2.22 Consider the construction of a database view on top of a database: this is interfacing. Please note that it is quite common to have non–encapsulated interaction: a non–updateable view would display many changes which cannot be explained from local actions! That is, the interaction morphism from the database to its view is not a retraction. ∘

Both incorporation and interfacing may occur in *multiple* versions: we have several objects, say $b_{1\bullet}u_1, \ldots, b_{n\bullet}u_n$, already in the community and construct a new one, say a•t, by relating it to b_1•$u_1, \ldots, b_n$•u_n simultaneously. In the case of incorporation, i.e., $h_i : a$•$t \to b_i$•u_i for $i = 1, \ldots, n$, we have *aggregation* as mentioned above. In the case of interfacing, i.e., $h_i : b_i$•$u_i \to a$•t for $i = 1, \ldots, n$, we have *synchronization* by sharing.

The latter was illustrated above in Example 2.18 (cf. also Example 2.4). An example for aggregation is the following.

Example 2.23 : Referring again to Example 2.4, suppose that `PXX.powsply` and `CYY.cpu` have been constructed and we want to assemble them (and other parts which we ignore here) to form our `SUN.computer`. Then we have to aggregate the parts and provide the epimorphisms (retractions in this case?) f: `SUN.computer` $\to$ `PXX.powsply` and g: `SUN.computer` $\to$ `CYY.cpu` showing the relationships to the parts. Please note that f sends the cpu items within the `SUN` to 1, while it sends the power supply items to themselves (modulo renaming). The same holds for g, with cpu taking the place of power supply. ∘

It is remarkable how much symmetry the inheritance and interaction constructions display. Their mathematical core is the same, namely epimorphisms between aspects. Taking the constructions in either direction and considering single and multiple versions, we arrive at the following table:

Object Constructs	*inheritance*	*interaction*
small-to-big/single	specialization	incorporation
small-to-big/multiple	mult. specialization	aggregation
big-to-small/single	abstraction	interfacing
big-to-small/multiple	generalization	synchronization

For each of these cases, we also have the encapsulated variant where the epimorphisms are retractions.

3 Object–Oriented Specification Approaches

3.1 FOOPS

FOOPS [GM87, GW90] (Functional Object-Oriented Programming System) is an object–oriented programming system with a declarative functional style. It is designed to preserve the essence of both functional and object–oriented programming: functional programming provides abstract data types for object attribute values, while object–oriented programming limits access to an object's state via methods associated with the specific object. In FOOPS, data elements are unchanging entities, while objects are persistent but changing entities.

Among many other concepts, FOOPS knows *functional* and *object* modules. Very roughly speaking, *functional modules* correspond to definitions of abstract data types in OBJ3 [GW88]. The underlying logic employs conditional equations and a form of sort inheritance via subsorting. The semantics is constructed by initial algebras. *Object modules* allow to define specific methods and attributes by means of conditional equations. Objects modules with a distinguished object sort correspond to classes. The state of an object – the attribues (or more generally the observable properties) – may change when certain methods are performed on it. Special methods, *new* and *remove* for object creation and deletion, are provided with any object module. Object modules are allowed to have sub–modules, thus providing a form of class inheritance. Three different kinds of semantics have been proposed for the object level:

- A *reflective semantics* representing the FOOPS database and all object manipulating methods as a functional module, i.e., as an abstract data type.
- A *direct algebraic semantics* employing hidden sorts to model states.
- A *sheaf semantics* modelling objects as sheaves and systems as collections of sheaves and sheaf morphisms.

As an example we consider a FOOPS specification describing integer cells in an object–oriented fashion. An attribute `get` and a method `set` are defined as follows

```
omod CELL                           \* object module     *\
  cl Cell                           \* class             *\
  pr INT                            \* protecting        *\
  at get : Cell -> Int              \* attribute         *\
  me set : Cell Int -> Cell         \* method            *\
  eq get(set(C,N)) = N              \* equation          *\
endo                                \* end object module *\
```

FOOPS has a sophisticated user–defined syntax and therefore it is possible to denote this specification in the more readable form given below. Additionally, an operation `declare_: Integer` for the declaration of variables is introduced. The definition of this operation employs the method `new` provided automatically with each class.

```
omod CELL                               \* object module     *\
  cl Cell                               \* class             *\
  pr INT                                \* protecting        *\
  at _ : Cell -> Int                    \* attribute         *\
  me _:=_ : Cell Int -> Cell            \* method            *\
  eq C := N = N                         \* equation          *\
  op (declare_: Integer) : Id -> Cell   \* operation         *\
  eq declare I : Integer = new(I)       \* equation          *\
endo                                    \* end object module *\
```

Using this definition of `CELL`, one can create new cell objects and apply methods to them in a convenient way. For example, FOOPS can evaluate the following lines.

```
declare x : Integer ;
declare y : Integer ;
x := 5 ;
y := x .
```

Generally speaking, object modules like those given above denote *theories* which specify "templates" as introduced in section 2. The power of specifying object dynamics, i.e. processes, is not quite clear, but there seem to be limitations, especially concerning the capability to specify infinite (non–terminating) behavior and full concurrency.

In the hidden–sorted algebra approach to semantics (cf. [Go91], object identities are associated with particular models of theories where each such model is a particular algebra of possible process states. The permissible algebras are given by observational equivalence.

Inheritance relationships are modeled at the syntactic level, by theory morphisms. The corresponding semantic relationship between aspects of the same object is not discussed in the FOOPS literature, but it seems that "cryptomorphisms" as introduced in [TBG89] (or, rather, their opposites) can do the job [Go92].

Using cryptomorphisms, it is also possible to model interaction along the lines outlined in section 2 [Go92]. So far, this point has not been emphasized in the FOOPS literature.

In conclusion, barring details which still have to be worked out, FOOPS has the potential of expressing all object–oriented concepts and constructions as outlined in section 2.

3.2 Entity Algebras

Entity algebras [Re91] provide models for data types, processes, and objects. They are special partial algebras with predicates [BW82] including the following features:

- The elements of certain sorts represent *dynamic elements*, called entities. Their dynamics is realized by allowing to perform labelled transitions.
- Entities may have *subcomponents*. The structure of these subcomponent entities is not fixed but can be defined by appropriate operations and axioms.
- Entities are given *identities* so that subcomponents can be retrieved and sharing of subcomponents can be described.

The entity algebra approach is based on previous work by the Genova and Passau Abstract Data Type groups (see for instance [AR87a, AR87b, AGRZ89, BZ89]). Another feature is that the approach distinguishes between *specifications–in–the–small* corresponding to usual specifications of dynamic systems as abstract data types and *specifications–in–the–large* suitable for expressing abstract properties of classes of dynamic systems.

Entity specifications distinguish between static sorts (like `int` or `bool`) and dynamic sorts (like `queue-with-state`). For each dynamic sort `s` providing the values of sort `s`, there are additionally sorts `ident(s)` and `ent(s)`. The elements in `ident(s)` are the identities for sort `s` and `ent(s)` represents the entities of sort `s`. Consequently, there is an operation

$$\_:\_ : \mathtt{ident(s)} \times \mathtt{s} \rightarrow \mathtt{ent(s)}$$

for the construction of entities. Dynamics is realized by providing a sort `lab(s)` where the possible transitions (or labels) for entities are collected (e.g., labels like `remove` or `put` for entity sort `queue-with-state`). The concrete behavior of entities is specified by axioms for predicates of the form

$$\_\ \texttt{---}\_\texttt{--->}\ \_ : \texttt{ent(s)} \times \texttt{lab(s)} \times \texttt{ent(s)}$$

Thus, the process model in this approach is labelled transition systems. Please note that the entities (or objects) are constructed by giving an identity and a current value for the object state. Therefore, elements in dynamic sorts are object states rather than (abstract) objects. The same holds for the object–oriented algebraic specification language OS introduced by Ruth Breu [Bre91].

The following specification defines an object–oriented version of data type queue. The entities of sort `ent(queue)` represent the objects of the class `queue`, the elements in `ident(queue)` are the identifiers for queues.

```
enrich INT by
  dsorts queue
  opns
    Empty : -> queue                                    total
    Get : queue -> ident(int)
    Remove : ent(queue) -> ent(queue)
    Put : ent(int) x queue -> queue
    A, B, ..., ABC, ... -> ident(queue)
    _.Empty : ident(queue) -> lab(queue)                total
    _=_.Get : ident(int) x ident(queue) -> lab(queue)   total
    _.Remove : ident(queue) -> lab(queue)               total
    _.Put(_) : ident(queue) x ent(int) -> lab(queue)    total
  preds
    _Diff_ : ident(queue) x ident(queue)
    Unused : ident(int) x queue
  axioms
    A Diff B ... A Diff ABC ...
    Get(Put(id:i,Empty)) = id
    DEF(Get(q)) => Get(Put(ei,q)) = Get(q)
    Remove(id:Put(ei,Empty)) = id:Empty
    Remove(id:q)=eq => Remove(id,Put(ei,q))=eq
    Unused(id,Empty)
    id Diff id' & Unused(id,q) => Unused(id,Put(id':i,q))
    Unused(id,q) => DEF(Put(id:i,q))
    id:q ---id.Empty---> id:Empty
    DEF(Put(ei,q)) => ( id:q ---id.Put(ei)---> id:Put(ei,q) )
    DEF(Remove(q)) => ( id:q ---id.Remove---> id:Remove(q) )
    ...
end enrich
```

Operations which have the sorts `queue` or `ent(queue)` as argument or result sorts – in this case `Empty`, `Put`, `Remove`, and `Get` – define the methods of class queue. The application of methods is specified by giving axioms for the transition predicate. Transitions are labelled by sending appropriate messages. If we now have the queue object `MYQUEUE:Put(INT_ENT1,Put(INT_ENT2,Empty))` (provided `int` objects `INT_ENT1` and `INT_ENT2 are given)`, the application of the methods `MYQUEUE.Remove` and `MYQUEUE.Empty` yield `MYQUEUE:Put(INT_ENT1,Empty)` and `MYQUEUE:Empty`, respectively.

Based on the algebraic semantics of processes as developed in [AR87a, AR87b], the semantics can be given by various observational equivalences, for example (generalized) bisimulation.

The literature on entity algebras does not address inheritance and interaction explicitly. A concept which compares with template (aspect) morphisms as described in section 2 is missing, but inheritance and interaction can probably be handled by different means and on different levels of abstraction. However, it would be interesting to see whether the entity algebra approach can be extended to include inheritance and interaction on the abstraction level of section 2.

Entity algebras do have a concept for subcomponents which can be shared, providing a means of symmetric and synchronous interaction as described in section 2. Together with the algebraic approach to concurrency invented in [AR87a], powerful means for expressing interaction of processes are available, with different kinds of parallelism and even with higher–order processes.

3.3 OBLOG

The languages OBLOG [CSS89] (OBject LOGic) and its successor TROLL [JHSS91] (Textual Representation of an Object Logic Language) were developed hand in hand with the conceptual framework outlined in section 2. Information systems are described as communities of interacting objects. Object templates specify processes and observations on these processes.

The languages employ features of *equational logic* for the specification of data, events, and identities and they use elements of *temporal*, *dynamic*, and *deontic logic* for system behavior and abstract system properties. Object templates introduce the objects' attributes characterising the objects' state and define the actions (called "events" in the language) modifying these attributes. The effect of performing actions is described by valuation rules. Additionally, safety and liveness properties can be specified.

As a specification example for the TROLL language, we consider a bank account. After importing the data types used, the attribute section determines the observable properties of accounts. The events (i.e., actions) specify the possible state transitions. Both, attribute values and state transitions have to satisfy the conditions given in the constraints part. The valuation section defines the effect of events, and the behavior part states permissions and obligations for events to occur, employing a deontic logic.

```
TEMPLATE account
  DATA TYPES |BankCustomer|, money, bool, UpdateType
  ATTRIBUTES
    CONSTANT Holder:|BankCustomer|
```

```
    CONSTANT Type:{checking,saving}
    Balance:money
    CreditLimit:money
    DERIVED MaxWithdrawal:money
  EVENTS
    BIRTH open(Holder:|BankCustomer|, Type:{checking,saving})
    DEATH close
    new_credit_limit(Amount:money)
    accept_update(Type:UpdateType, Amount:money)
    update_failed
    withdrawal(Amount:money)
    deposit(Amount:money)
  CONSTRAINTS
    INITIAL CreditLimit = 0
  DERIVATION
    MaxWithdrawal = Balance + CreditLimit
  VALUATION
    VARIABLES m:money
    [new_credit_limit(m)] CreditLimit = m
    [withdrawal(m)] Balance = Balance - m
    [deposit(m)] Balance = Balance + m
    ...
  BEHAVIOR
    PERMISSIONS
      VARIABLES t,t1:UpdateType; m,m1,m2:money
      { Balance = 0 } close
      { NOT SOMETIME(AFTER(accept_update(t1,m1)))
          SINCE LAST(AFTER(update_failed) OR
                     AFTER(deposit(m2)) OR
                     AFTER(withdrawal(m2))) } accept_update(t,m)
     ...
     OBLIGATIONS ...
END TEMPLATE account
```

The semantics of such specifications is not fully worked out yet, but there are semantic models available which have been designed for this purpose, meeting all requirements described in section 2.

Essentially, these models provide categorial process models which can be used for formalizing these conceps: processes formalizing templates, and process morphisms formalizing template morphisms. This way, there is a heavy emphasis on object *dynamics*.

In what follows, we briefly outline one particular such model (cf. [ES91] for more detail).

The basic event structure is an *event monoid* $\bar{E} = (E, \|, 1)$ which is associative, commutative, idempotent, and satisfies the cancellation law. Intuitively, $\|$ denotes simultaneous occurrence of events, and 1 denotes the empty event.

Event monoids $\bar{E}_i = (E_i, \|_i, 1_i)$, $i = 1, 2$, can be related by *event monoid mor-*

phisms $h : \bar{E}_1 \to \bar{E}_2$ which are structure preserving maps $h : E_1 \to E_2$ satisfying the properties $h(1_1) = 1_2$ and $h(e_1 \|_1 e_2) = h(e_1) \|_2 h(e_2)$. Henceforth, we will omit the indexes from $\|$ and 1 as long as there is no danger of confusion.

Let $\bar{E} = (E, \|, 1)$ be an event monoid. A subset $M \subseteq E$ is called a *menu*: it might appear on a screen as a selection of events (actions and observations) which may occur next. For a collection α of menues, $\bigcup \alpha$ denotes the union of these menues, i.e., $\bigcup \alpha = \{e \mid \exists M \,\epsilon\, \alpha : e \,\epsilon\, M\}$.

A *process* over an alphabet $\bar{E}$ is a map $\mu : E^* \to 2^{2^E}$ satisfying the following conditions for each $\tau \,\epsilon\, E^*$:

1. $\bigcup \mu(\tau) \,\epsilon\, \mu(\tau)$
2. $\forall X, Y \subseteq E \;:\; X \subseteq Y \subseteq \bigcup \mu(\tau) \;\wedge\; X \,\epsilon\, \mu(\tau) \;\Rightarrow\; Y \,\epsilon\, \mu(\tau)$

The intuition might help that each $M \,\epsilon\, \mu(\tau)$ corresponds to a state of knowledge about what might possibly occur next after the history τ of the system.

Let $\mu_i : E_i^* \to 2^{2^{E_i}}$, $i = 1, 2$, be processes over alphabets $\bar{E}_1$ and $\bar{E}_2$, respectively. Let $h : \bar{E}_1 \to \bar{E}_2$ be a monoid morphism. h is called a *process morphism* iff the following condition holds:

$$h(\mu_1(\tau)) \subseteq \mu_2(h(\tau)) \text{ for each } \; \tau \,\epsilon\, E_1^* \; .$$

In this notation, h is extended to traces as well as sets and families of sets by elementwise application. For example, $h(ef) = h(e)h(f)$ and $h(\{\{a\}, \{a, b\}\}) = \{\{h(a)\}, \{h(a), h(b)\}\}$.

It is not hard to prove that processes and process morphisms as defined above form a category. We call this category *NDM*, the *N*on-*D*eterministic *M*enu model.

We cannot develop the mathematics of *NDM* in this paper. We claim, however, that

- *NDM* is complete, and limits reflect parallel composition,
- *NDM* is cocomplete, and colimits reflect internal choice.

Therefore, this model is especially well suited to model the object–oriented concepts explained in section 2. The reader is referred to [ES91] for further details.

3.4 Processes as Abstract Data Types

There are various approaches for the specification of processes by algebraic methods, by Bergstra and Klop [BK86], by the Berlin Abstract Data Type group [EPPB$^+$90], by Kaplan [Kap89], and others. The main idea behind these approaches is to introduce sorts for actions and processes explicitly and to provide operations like prefixing, parallel composition, choice, etc., for the combination of processes. The logic employs (conditional) equations. On the semantic side, the approaches use continuous algebras, projection spaces, or some form of observational or behavioral semantics.

It is perhaps not fair to evaluate these approaches on the basis of how well they meet requirements of object–orientation, since most of them do not claim to do so.

But there are quite a few allusions – more or less vague – that these approaches might be useful for the theory of objects, so a few remarks are in order.

The general idea is to give an object–oriented flavor by adding identification to processes. For instance, Große–Rhode [GR91] considers the following concepts: *objects as states* (the state of the system is the aggregation of all local object states); *objects for local behavior* (local process specifications define the dynamic behavior of objects); *class definitions* (Berlin module specifications serve as class definitions which are similar to object templates in the sense of section 2); *class combinators* (class combinators are the usual Berlin module operations building large modules from smaller ones); *(strict) inheritance:* (inheritance corresponds to union, import actualization or extension of modules). Objects are defined only on the syntactic level, they are enrichments of the specification representing the state of the system. The semantics of state changes of processes consists of the syntactic modification of the underlying specification.

Berlin module specifications are also used by Parisi-Presicce and Pierantonio [PP91] in order to explain inheritance mechanisms in object–oriented programming.

Another effort following these lines is CMSL (Conceptual Model Specification Language) proposed by Wieringa [Wie91].

Although some object–oriented concepts can be expressed in these approaches, there are some problems with this approach. *Inheritance* is no problem, at least not on the theory (or template) level: like in the FOOPS approach, specification or theory morphisms do the job. On the instance level, however, an obvious concept for modeling inheritance or interaction is missing.

The main problem is with *identification*. The problems with inheritance and interaction are consequences of identification problems. The relevant question here is: *what is the unit identified?*

One obvious unit of identification is a specification (or theory) specifying some kind of object (aspect), i.e., a template. However, object identification is a little more demanding than just giving names to templates. Rather, what is needed is a *dynamic* mechanism for generating, maintaining, and destroying objects (aspects) as named units *"at runtime"*. Object classes are time–varying collections of objects (aspects) with *different* names but with the *same* template. This is an essential feature of object–orientation, it is one of the reasons why, for example, MODULA2 is not considered object–oriented.

In most of the algebraic approaches mentioned above, the way of handling this is to introduce a sort `ID` of identities and introduce `ID` as a parameter sort for the create operation(s). For example, instead of `create:→stack`, we have `create:ID→stack`. This way, however, we obtain what we might call *a set of named process states* instead of a set of process states within a named object. That is, the named unit is an *object state* rather than an object. Moreover, all states of all objects are grouped together in one structure (carrier of an algebra).

As a consequence, algebra morphisms preserving this structure do not appropriately reflect inheritance and interaction at the instance level. For example, it is not possible to express formally that a user interacts with one specific stack, not with all the others around, using algebra morphisms that have the carrier set of all states of all stacks as source or target, respectively.

4 Conclusions

Let us first summarize the object–oriented concepts and constructions of section 2. Templates are generic objects without individual identity. Identities are associated with templates to represent aspects of individual objects. Interaction is a relationship between different objects, while inheritance relates aspects of the same object. Objects and aspects appear as members of classes. A class is again an aspect with a time–varying set of aspects as members. Since classes are objects or aspects of objects, there is no difficulty in constructing meta–classes having classes of different types as members.

By means of an object–oriented approach, systems can be elegantly described as communities of interacting objects. As has been pointed out, the notion of an object only makes sense within an object community. Please remember that an object community must be closed with respect to inheritance and must provide unique identifiers for objects, among others.

The algebraic approaches to object specification and semantics investigated in this paper suggests that an appropriate theory of objects should be based on processes rather than data types. This is obvious in the approaches investigated in sections 3.2 and 3.3. At first sight, the FOOPS approach described in section 3.1 seems to contradict this conclusion, but a closer look reveals that it does not: by employing hidden sorts and observational equivalence, this approach is based on an algebraic state machine model of processes rather than abstract data types.

It should be mentioned that not everything relevant to object specification – i.e., specification of object communities – is addressed in this paper. A full-fledged language and system for specification and design must provide means for specifying data types, (types of) identities, inheritance schemata (e.g., for specialization and generalization), interaction schemata (e.g., for aggregation and synchronization patterns), generic modules and actualization, classes and instances, etc.

The important issue of object reification (or implementation or refinement) has not been addressed in this paper. A satisfactory treatment is still missing, but there are promising approaches borrowing ideas from abstract data type implementation and from process refinement.

Acknowledgements

Thanks to all colleagues who contributed to the development of ideas presented here. Special thanks are due to our COMPASS collegues Egidio Astesiano, Hartmut Ehrig, Martin Große–Rhode, and Gianna Reggio for stimulating discussions on algebraic process theory, and to the ISCORE partners who took part in developing object theory and specification language features. In particular, Cristina Sernadas participated in discussing the basic ideas of objects and object descriptions. She, Ralf Jungclaus, Thorsten Hartmann, and Gunter Saake were involved in defining the TROLL language. Thanks to Ralf Jungclaus and Thorsten Hartmann for providing the TROLL example. Felix Costa's contributions to object semantics are greatfully acknowledged.

Special thanks are due to Joseph Goguen and Egidio Astesiano for correcting

errors and misconceptions in an earlier draft of this paper. Of course, the authors are fully responsible for all errors that are still there.

References

[AR87a] Astesiano,A.;Reggio,G.: An outline of the SMoLCS approach. Proc. Advanced School on Mathematical Models for Parallelism (M.Venturini Zilli, ed.), LNCS 280, Springer–Verlag, Berlin 1987, 81–113

[AR87b] Astesiano,A.;Reggio,G.: SMoLCS–driven concurrent calculi. Proc. TAPSOFT'87 Vol.1, (H.Ehrig et al, eds.), LNCS 249, Springer–Verlag, Berlin 1987, 169–201

[AGRZ89] Astesiano,E.;Giovini,A.;Reggio,G.;Zucca,E.: An integrated algebraic approach to the specification of data types, processes, and objects. Proc. Algebraic Methods – Tools and Applications, LNCS 394, 1989, 91–116

[Be91] Beeri,C.: Theoretical Foundations for OODB's – a Personal Perspective. Database Engineering, to appear

[BK86] Bergstra,J.A.;Klop,J.W.: Algebra of communicating processes. CWI Monographs Series, Proc. of the CWI Symposium Mathmatics and Computer Science, North–Holland, Amsterdam 1986, 89–138

[BOS91] Butterworth,P.;Otis,A.;Stein,J.: The GemStone Object Database Management System. Comm. ACM 34 (1991), 64–77

[Bre91] Breu,R.: Algebraic specification techniques in object–oriented programming environments. Ph.D. Thesis, Passau University; also: LNCS, Springer 1991.

[BW82] Broy,M.;Wirsing,M.: Partial abstract types. Acta Informatica 18, 1982, 47–64

[BZ89] Breu,R.;Zucca,E.: An algebraic compositional semantics of an object–oriented notation with concurrency. Proc 9th Conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS 405, 1989, 131–142

[CP89] Cook,W.;Palsberg,J.: A Denotational Semantics of Inheritance and its Correctness. Proc. OOPSLA'89, ACM Press, 433–443

[CS91] Costa,J.-F.;Sernadas,A.: Process Models within a Categorial Framework. INESC Research Report, Lisbon 1991, submitted for publication

[CSS89] Costa,J.-F.;Sernadas,A.;Sernadas,C.: OBL–89 User's Manual – Version 2.3. Internal Report, INESC Lisbon 1989.

[CSS91] Costa,J.-F.;Sernadas,A.;Sernadas,C.: Objects as Non–Sequential Machines. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G.Saake, A.Sernadas, eds.), Informatik–Berichte 91–03, Tech. Univ. Braunschweig 1991, 25–60

[Cu91] Cusack,E.: Refinement, Conformance and Inheritance. Formal Aspects of Computing 3 (1991), 129–141

[De91] Deux,O. et al: The O_2 System. Comm. ACM 34 (1991), 35–48

[DMN67] Dahl,O.-J.;Myrhaug,B.;Nygaard,K.: SIMULA 67, Common Base Language. Norwegian Computer Center, Oslo 1967

[EBO91] Ehrig,H.;Baldamus,M.;Orejas,F.: New concepts for amalgamation and extension in the framework of specification logics. Technical Report No. 91/05, Computer Science Department, TU Berlin 1991.

[EGS91] Ehrich,H.-D.;Goguen,J.A.;Sernadas,A.: A Categorial Theory of Objects as Observed Processes. Proc. REX/FOOL School/Workshop, deBakker,J.W. et. al. (eds.), LNCS 489, Springer–Verlag, Berlin 1991, 203–228

[EPPB+90] Ehrig,H.; Parisi-Presicce,F.; Boehm,P.; Rieckhoff,C.; Dimitrovici,Ch.; Große-Rhode,M.: Combining data type specifications and recursive process specifications using projection algebras. Theoretical Computer Science 71 1990, 347–380

[ESS90] Ehrich,H.-D.;Sernadas,A.;Sernadas,C.: From Data Types to Object Types. Journal of Information Processing and Cybernetics EIK 26 (1990) 1/2, 33–48

[ES90] Ehrich,H.-D.;Sernadas,A.: Algebraic Implementation of Objects over Objects. Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness. deBakkerJ.W.;deRoever,W.-P.; Rozenberg,G. (eds.), LNCS 430, Springer–Verlag, Berlin 1990, 239–266

[ES91] Ehrich,H.-D.;Sernadas,A.: Fundamental Object Concepts and Constructions. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G.Saake, A.Sernadas, eds.), Informatik–Berichte 91–03, Tech. Univ. Braunschweig 1991, 1–24

[FCSM91] Fiadeiro,J.;Costa,J.-F.;Sernadas,A.;Maibaum,T.: (Terminal) Process Semantics of Temporal Logic Specification. Unpublished draft, Dept. of Computing, Imperial College, London 1991

[Fi87] Fishman,D. et al: IRIS: An Object–Oriented Database Management System. ACM Trans. Off. Inf. Sys. 5 (1987)

[FM91a] Fiadeiro,J.;Maibaum,T.: Describing, Structuring and Implementing Objects. Proc. REX/FOOL School/Workshop, deBakker,J.W. et. al. (eds.), LNCS 489, Springer–Verlag, Berlin 1991

[FM91b] Fiadeiro,J.;Maibaum,T.: Temporal Theories as Modularisation Units for Concurrent System Specification, to appear in Formal Aspects of Computing

[FS91] Fiadeiro,J.;Sernadas,A.: Logics of Modal Terms for System Specification. Journal of Logic and Computation 1 (1991), 357–395

[FSMS90] Fiadeiro,J.;Sernadas,C.;Maibaum,T.;Saake,G.: Proof–Theoretic Semantics of Object–Oriented Specification Constructs. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[GKS91] Gottlob,G.;Kappel,G.;Schrefl,M.: Semantics of Object–Oriented Data Models — The Evolving Algebra Approach. Proc. Int. Workshop on Information Systems for the 90's, Schmidt,J.W. (ed.), Springer LNCS 1991

[GM87] Goguen,J.A.;Meseguer,J.: Unifying functional, object–oriented and relational programming with logical semantics. Research Direction in Object-Oriented Programming, B.Shriver,P.Wegner (eds.), MIT Press 1987, 417–477

[Go73] Goguen,J.: Categorical Foundations for General Systems Theory. Advances in Cybernetics and Systems Research, Transcripta Books, 1973, 121–130

[Go75] Goguen,J.: Objects. International Journal of General Systems, 1 (1975), 237–243

[Go89] Goguen,J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989. To appear in Mathematical Structures in Computer Science.

[Go90] Goguen,J.: Sheaf Semantics of Concurrent Interacting Objects, 1990. To appear in Mathematical Structures in Computer Science.

[Go91] Goguen,J.: Types as Theories. Proc. Conf. on Topology and Category Theory in Computer Science, Oxford University Press 1991, 357–390

[Go92] Goguen,J.: personal communication

[GR83] Goldberg,A.;Robson,D.: Smalltalk 80: The Language and its Implementation. Addison–Wesley, New York 1983

[GR91] Große–Rhode,M.: Towards object–oriented algebraic specifications. Proc. 7th Workshop on Specification of Abstract Data Types, Wusterhausen (Dosse), H. Ehrig, K.P. Jantke, F. Orejas, H. Reichel (Eds.), LNCS 534, 1991

[GW88] Goguen,J.A.;Winkler,T.: Introducing OBJ3. SRI International, Technical Report SRI-CSL-88-9, 1988.

[GW90] Goguen,J.;Wolfram,D.: On Types and FOOPS. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[HC89] Hayes,F.;Coleman,D.: Objects and Inheritance: An Algebraic View. Technical Memo, HP Labs, Information Management Lab, Bristol 1989

[Hei88] Heitz,M.: HOOD: A Hierarchical Object–Oriented Design Method. Proc. 3rd German Ada Users Congress, Munich 1988, 12-1 – 12-9

[JHSS91] Jungclaus,R.;Hartmann,T.;Saake,G.;Sernadas,C.: Introduction to TROLL — A Language for Object–Oriented Specification of Information Systems. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G.Saake, A.Sernadas, eds.), Informatik–Berichte 91–03, Tech. Univ. Braunschweig 1991, 97–128

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.:Language Features for Object-Oriented Conceptual Modeling. In:Proc. 10^{th} Int. Conf. on the ER-approach (T.J. Teorey,ed.), San Mateo, E/R Institute 1991, 309–324.

[JSHS91] Jungclaus,R.;Saake,G.;Hartmann,T.;Sernadas,C.: Object–Oriented Specification of Information Systems: The TROLL Language. Informatik–Bericht, TU Braunschweig 1991. To appear

[JSS90] Jungclaus,R.;Saake,G.;Sernadas,C.: Using Active Objects for Query Processing. Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[JSS91] Jungclaus,R.;Saake,G.;Sernadas,C.: Formal Specification of Object Systems. Proc. TAPSOFT'91, Abramsky,S.;Maibaum,T.S.E. (eds.), Brighton (UK) 1991

[Kap89] Kaplan,S.: Algebraic specification of concurrent systems. Theoretical Computer Science, 1989.

[Ke88] Kerth,N,: MOOD: A Methodology for Structured Object–Oriented Design. Tutorial presented at OOPSLA'88, San Diego 1988

[Ki88] Kim,W. et al: Features of the ORION Object–Oriented DBMS. In Object–Oriented Concepts, Databases, and Applications, Kim,W. and Lochovsky,E.H. (eds.), Addison–Wesley 1988

[Ki90] Kim,W.: Object–Oriented Databases: Definition and Research Directions. IEEE Transactions on Knowledge and Data Engineering 2 (1990), 327–341

[LP90] Lin,H.;Pong,M.: Modelling Multiple Inheritance with Colimits. Formal Aspects of Computing 2 (1990), 301–311

[Me88] Meyer,B.: Object–Oriented Software Construction. Prentice–Hall, Englewood Cliffs 1988

[Me91] Meseguer,J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Technical Report SRI–CSL–91–05, Computer Science Laboratory, SRI International, Menlo Park 1991

[PP91] Parisi-Presicce,F.;Pierantonio,A.: Towards the algebraic specification of classes in object–oriented programming. Bulletin of the EATCS, Vol. 45,1991, 86-97

[Re91] Reggio,G.: Entities: Institutions for dynamic systems. Proc. 7th Workshop on Specification of Abstract Data Types, Wusterhausen (Dosse), H. Ehrig, K.P. Jantke, F. Orejas, H. Reichel (Eds.), LNCS 534, 1991

[SE90] Sernadas,A.;Ehrich,H.-D.: What is an object, after all ? Proc. IFIP 2.6 Working Conference DS-4, Meersman,R.;Kent,W. (eds.), North–Holland, Amsterdam 1991

[SEC90] Sernadas,A.;Ehrich,H.-D.;Costa,J.-F.: From Processes to Objects. The INESC Journal of Research and Development 1 (1990), 7–27

[SGGSR91] Sernadas, C.; Gouveia, P.; Gouveia, J.; Sernadas, A.; Resende, P.: The Reification Dimension in Object–oriented Database Design. Proc. Int. Workshop on Specification of Database Systems, Glasgow 1991, Springer–Verlag, to appear

[SFSE89] Sernadas,A.;Fiadeiro,J.;Sernadas,C.;Ehrich,H.-D.: The Basic Building Blocks of Information Systems. Proc. IFIP 8.1 Working Conference, Falkenberg,E.; Lindgreen,P. (eds.), North–Holland, Amsterdam 1989, 225–246

[SGCS91] Sernadas,C.;Gouveia,P.;Costa,J.-F.;Sernadas,A.: Graph–theoretic Semantics of Oblog – Diagrammatic Language for Object–oriented Specifications. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G.Saake, A.Sernadas, eds.), Informatik–Berichte 91–03, Tech. Univ. Braunschweig 1991, 61–96

[SJ91] Saake,G.;Jungclaus,R.: Specification of Database Applications in the TROLL Language. Proc. Int. Workshop on Specification of Database Systems, Glasgow 1991 , Springer–Verlag, to appear

[SJE91] Saake,G.;Jungclaus,R.;Ehrich,H.-D.: Object–Oriented Specification and Stepwise Refinement. Proc. IFIP TC6 Int'l Workshop on Open Distributed Processing, Berlin 1991, to be published by North–Holland

[SRGS91] Sernadas, C.; Resende, P.; Gouveia, P.; Sernadas, A.: In–the–large Object–oriented Design of Information Systems. Proc IFIP 8.1 Working Conference on the Object–oriented Approach in Information Systems, van Assche, F.; Moulin, B.; Rolland, C. (eds.), Quebec City (Canada) 1991, North Holland, to appear

[SS86] Seidewitz,E.;Stark,M.: General Object–Oriented Software Development. Document No. SEL–86–002, NASA Goddard Space Flight Center, Greenbelt, Maryland 1986

[SSE87] Sernadas,A.;Sernadas,C.;Ehrich,H.-D.: Object–Oriented Specification of Databases: An Algebraic Approach. Proc. 13th VLDB, Stocker,P.M.; Kent,W. (eds.), Morgan–Kaufmann Publ. Inc., Los Altos 1987, 107–116

[SSGRG91] Sernadas,A.;Sernadas,C.;Gouveia,P.;Resende,P.;Gouveia,J.: Oblog – An Informal Introduction, INESC Lisbon, 1991.

[St86] Stroustrup,B.: The C++ Programming Language. Addison Wesley, Reading, Mass. 1986

[TBG89] Tarlecki,A.; Burstall,R.; Goguen,J.: Indexed Categories as a Tool for the Semantics of Computation. Technical Monograph PRG-77, August 1989, Oxford University Computing Laboratory.

[Ve91] Verharen, E.M.: Object–oriented System Development: An Overview. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G.Saake, A.Sernadas, eds.), Informatik–Berichte 91–03, Tech. Univ. Braunschweig 1991, 202–234

[We89] Wegner,P.: Learning the Language. Byte 14 (1989), 245–253

[Wie91] Wieringa,R.: A formalization of objects using equational dynamic logic. Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, Munich 1991