Proc. 2nd Workshop on "Informationssysteme und Künstliche Intelligenz: Modellierung", Informatik–Fachberichte 303, pages 1–19, Berlin, 1992. Springer–Verlag

Concepts of Object–Orientation*†

Hans-Dieter Ehrich
Gunter Saake
Abteilung Datenbanken, Technische Universität, Postfach 3329
D–3300 Braunschweig, GERMANY

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

Abstract

An object is a unit of structure and behavior; it has an identity which persists through change; objects communicate with each other; they are classified by object types, collected into object classes, related by inheritance, and composed to form complex objects. In the first part of the paper, this rich world of concepts and constructions is explained in an informal but systematic way, independent of any language or system. In the second part, features of an object specification language are outline which incorporate most of these concepts and constructions.

1 Introduction

There are many languages, systems, methods and approaches in computing which call themselves "object–oriented", 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? Some common sense is given by Wegner [We89] who says:

*This work was partly supported by the EC under ESPRIT BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability), by DFG under Sa 465/1-1 and Sa 465/1-2, and by JNICT under PMCT/C/TIT/178/90 FAC3 contract.

†In: Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany). (R. Studer, ed.), Springer IFB 303, 1992 (1–19).

> *An object has a set of operations and a local shared state (data) that remembers the effect of operations. The value that an operation on an object returns can depend on the object's state as well as the operation's arguments. The state of an object serves as a local memory that is shared by operations on it. In particular, other previously executed operations can affect the value that a given operation returns. An object can learn from experience, storing the cumulative effect of its experience – its invocation history – in its state.*

Thus, an object 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.

But this is not all: 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, we have been working 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 this paper, we give an informal, easier–to–read account of these ideas.

In section 2, we explain what we mean by an *object template* as the common structure and behavior pattern of some kind of object. In section 3, we argue that we should carefully distinguish between objects and *aspects* of objects, and we introduce object *classes* as another kind of objects. In section 4, the basic object constructions around *inheritance* and *interaction* are explained. In section 5, we outline features of the language Troll ([JHSS91]) for specifying and designing object communities which is being developed at TU Braunschweig.

2 Templates – and How They Are Related

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, SEC90, 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 fo 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 $p_i : \texttt{twoqueue} \to \texttt{queue}, i = 1, 2$, 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 : `deque` $\rightarrow$ `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.

3 Objects and Classes

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

Definition 3.2 : Let $b{\bullet}t$ and $c{\bullet}u$ be two aspects, and let $h : t \rightarrow u$ be a template morphism. Then we call $h : b{\bullet}t \rightarrow c{\bullet}u$ 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 3.3 : An aspect morphism $h : b \bullet t \rightarrow c \bullet u$ 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 3.4 : 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` $\rightarrow$ `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` $\bullet$ `computer`	(`SUN` as a computer)	and
`SUN` $\bullet$ `el_dvice`	(`SUN` as an electronic device),	

related by the inheritance morphism h : `SUN` $\bullet$ `computer` $\longrightarrow$ `SUN` $\bullet$ `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 $f : \texttt{el\_dvice} \to \texttt{powsply}$ and $g : \texttt{computer} \to \texttt{cpu}$, respectively.

If `PXX` denotes a specific power supply and `CYY` denotes a specific cpu, we might have *interaction* morphisms $f' : \texttt{SUN}\bullet\texttt{el\_dvice} \to \texttt{PXX}\bullet\texttt{powsply}$ and, say, $g' : \texttt{SUN}\bullet\texttt{computer} \to \texttt{CYY}\bullet\texttt{cpu}$. 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 $\texttt{CBZ}\bullet\texttt{cable}$ which is part of both $\texttt{PXX}\bullet\texttt{powsply}$ and $\texttt{CYY}\bullet\texttt{cpu}$. This is expressed by a *sharing diagram*

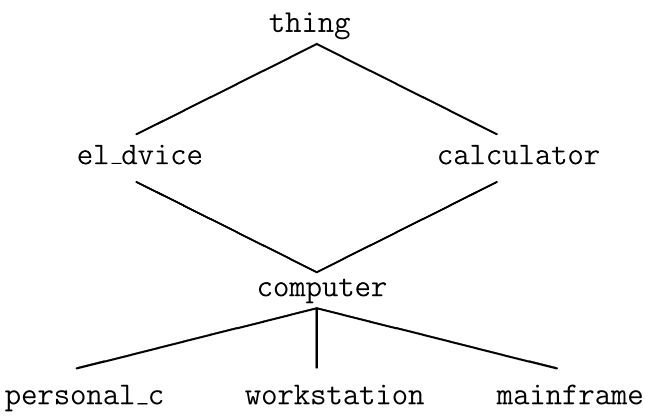
$$\texttt{CYY} \bullet \texttt{cpu} \longrightarrow \texttt{CBZ} \bullet \texttt{cable} \longleftarrow \texttt{PXX} \bullet \texttt{powsply}$$

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 $h : \texttt{computer} \to \texttt{el\_dvice}$ in order to express that each computer IS An electronic device, imposing that whenever we have an instance computer, say $\texttt{SUN}\bullet\texttt{computer}$, then it necessarily IS THE electronic device $\texttt{SUN}\bullet\texttt{el\_dvice}$ inherited by h as an aspect morphisms, $h : \texttt{SUN}\bullet\texttt{computer} \to \texttt{SUN}\bullet\texttt{el\_dvice}$.

Definition 3.5 : 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 3.6 : In the following inheritance schema, arrowheads are omitted: the morphisms go upward.



○

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 3.7 : 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 3.8 : An *object community* – or *community* for short – consists of an inheritance schema and an aspect community which is closed with respect to it.

Definition 3.9 : 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 3.10 : Consider an object community containing the inheritance schema in example 3.6, 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:

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

$\bigcirc$

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 3.11 : 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 3.12 : 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.

Objects rarely occur in isolation, they usually appear as members of *classes* – unless they are classes themselves. Indeed, we will see that a class is again an object, with a time–varying set of objects as members.

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 3.13 : Referring to the inheritance schema in example 3.6, 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 3.14 : 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 $\mathtt{ID}(C)$, the set of member identities which can be used in class C, and the notion of class type should comprise $\mathtt{ID}(C)$ along with the member template t.

Definition 3.15 : 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 3.16 : 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 3.13).

Definition 3.17 : 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.

4 Inheritance versus Interaction

When we build an object–oriented system, we must provide an *inheritance schema* (cf. definition 3.5). 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 4.1 : Referring to example 3.6, 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 event `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 3.6 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 4.2 : Referring to example 3.6 and assuming top–down construction, the template for `computer` is constructed by multiple specialization (multiple inheritance) from `el_dvice` and `calculator`. ◯

Example 4.3 : If we would have constructed the schema in definition 3.6 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 $c\bullet$`thing` aspect is a proper generalization of its $c\bullet$`el_dvice` and $c\bullet$`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 $p\bullet$`person` $\to$ $p\bullet$`contract_partner` or $c\bullet$`company` $\to$ $c\bullet$`contract_partner` on the instance level, but not as a proper object generalization.

When we build an object-oriented system, we must provide an *object community* (cf. definition 3.8). 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?

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! Please note that the part can also play the role of a communication *port* and that shared ports play the role of a communication *channel* (cf. example 3.4).

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 4.4 : Referring to example 3.4, the interaction morphisms

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

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 3.7), 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 4.5 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 $a \bullet t$, by relating it to $b_1 \bullet u_1, \ldots, b_n \bullet u_n$ simultaneously. In the case of incorporation, i. e. $h_i : a \bullet t \to b_i \bullet u_i$ for $i = 1, \ldots, n$, we have *aggregation* as mentioned above. In the case of interfacing, i. e. $h_i : b_i \bullet u_i \to a \bullet t$ for $i = 1, \ldots, n$, we have *synchronization* by sharing.

The latter was illustrated above in example 4.4 (cf. also example 3.4). An example for aggregation is the following.

Example 4.6 : Referring again to example 3.4, 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.

5 Specification and Design

After having discussed the fundamental concepts and constructions of object systems, we now present language features for formally describing objects and their aspects as well as the various kinds of interaction and inheritance between them. There are several specification languages for abstract objects allowing to describe object systems as they are presented in they previous chapters, among them the language **Oblog** (first described in [SSE87]) with its graphical presentation [SGCS91, SRGS91, SSGRG91, SGGSR91] and its textual counterpart TROLL presented in [JHSS91, JSH91, JSS91, JSHS91, SJ91] co-developed by the authors. Other proposals like CMSL [Wi90] and Object/Behaviour Diagrams [KS91] are examples from the area of information system design languages following a similar object concept and also fitting to the interpretation structures discussed earlier.

We will use the TROLL language to show some basic language features supporting specification of object systems as outlined in the first chapters. However, for a more complete language description of TROLL we refer to [JHSS91, JSS91]. The TROLL language and the semantic structures presented in this paper were developed in parallel. As a result of this development process, both base on the same concept of object but differ in specific notations and expressibility. For example, the TROLL language distinguishes between read and update events in object signatures in contrast to the presented theoretical framework.

Templates

Templates mainly describe processes built from observations and update actions which are called *attributes* and *events* in TROLL. These processes are based on a given set of abtract data types providing value sets for parameters and attribute values. In TROLL, a template specification is organized as follows :

template
  data types *imported data types;*
  attributes *attributes and their types;*
  events *events and their parameters;*
  constraints *integrity constraints;*
  valuation *attribute modification by events;*
  behavior *process specification;*

As example, we have the specification of a template modelling a book copy in a library. The attributes of `book_copy` are the boolean attribute `OnLoan` which indicates whether a copy has been borrowed, the return date `Due`, and the list of borrowers up to now (`Borrowers`). The events represent the acquisition of a copy (`GetCopy`), the lending and returning of a copy (`CheckOut` and `Return`, respectively) and the removal of the copy from the library (`ThrowAway`).

```
template book_copy
   data types bool, date, |BOOK|,|USER|,list(|USER|), nat;
   attributes
      constant Of:|BOOK|; OnLoan:bool; Due:date;
      Borrowers:list(|USER|);
   events
      birth GetCopy; death ThrowAway;
      CheckOut(|USER|, date, nat); Return;
   constraints
      length(Borrowers) = 0 implies not OnLoan;
   valuation
      variables U:|USER|, d:date, n:nat;
      [GetCopy] OnLoan = false;
      [GetCopy] Borrowers = emptylist();
      [CheckOut(U,d,n)] OnLoan = true;
      [CheckOut(U,d,n)] Due =  add_days_to_date(d,n);
      [CheckOut(U,d,n)] Borrowers = append(U,Borrowers);
      [Return] OnLoan = false;
   behavior
      permissions
         variables U:|USER|, d:date, n:nat;
         {OnLoan = false} CheckOut(U,d,n);
         {exists(U: |USER|, d:  date, n:  nat)
            sometime(after(CheckOut(U,d,n)))
               since last  (after(Return))} Return;
      obligations
         {exists(U:|USER|, d:date, n:nat) after(CheckOut(U,d,n))} ⇒ Return;
```

Template specifications describe event processes observed via attributes intuitively building the semantics domain for object aspects seen in isolation. We have the following specification sections in a template declaration :

- The import of *data types* after the key word **data types**. Special data types are object identities notated for example as |USER|. Data type constructors like **list** and **set** allow to build structured attribute domains.

- The definition of the template *signature* following the key words **attributes** and **events**. Constant attributes are marked by **constant**, whereas creation and deletion events are declared by **birth** and **death**.

- After **constraints** arbitrary *integrity constraints* on attribute values and their temporal evolution may be specified using temporal logic.

- The **valuation** rules define the effect of events on attributes by explicitly assigning a new value to attributes.

- To specify the *event process* resulting from the sequence of event occurences, TROLL offers several specification mechanisms (following a deontic style as in [KM89]).

  - After the key word **permissions**, several preconditions for the occurence of events can be specified. Permissions can refer to the past object evolution using temporal logic operators for the past.
  - **Obligations** state events which should appear in correct life cycles of templates (corresponding to liveness conditions from process theory).
  - Additionally, process patterns can be declared explicitly in a process definition language [JSH91, JHSS91] not presented here.

Objects, Classes and Object Identification

Having specified a template, we can create a single object `copy` fitting to this template simply by declaring

```
object copy;
   template book_copy ;
end object copy.
```

Such a single–standing object is not included in a class. To create an *object class*, we have to provide an *identification mechanism* providing identities attached to class members and a *member template*. In the language TROLL, the typed identity (or name) space of an object class is defined by the value set of an abstract data type. The declaration of this data type is notated in analogy to key attributes in semantic data models.

```
object class COPY
   data types nat;
   identification
      DocNo:nat;
   template book_copy;
end object class COPY.
```

Instead of using a predefined template, we can declare a template directly in a class definition. The result of an explicit TROLL class definition is always a homogeneous class. In analogy to templates, we can also define and reuse class templates.

Aspects and Inheritance

Based on single–standing objects and object classes, the TROLL language offers language features to build an inheritance schema for object templates as discussed in section 4. Because of the strong dynamic aspect of the TROLL object model, corresponding object aspects are called *phases* or *roles* of objects.

The following specification fragment defines the class `MANAGER` as a temporary phase of the object class `PERSON` :

```
object class PERSON
   identification
      Name:string; BirthDate:date;
   template
      attributes
         Salary:  money; ...
      events
         BecomeManager; ...
end object class PERSON.
object class MANAGER
   role of PERSON;
   template
      attributes
         OfficialCar:|CAR|;
      events
         birth PERSON.BecomeManager;
      constraints
         Salary ≥ 5.000;
end object class MANAGER.
```

A `PERSON` object may enter the temporary phase `MANAGER` due to the occurence of the event `BecomeManager`. Attributes and events of `PERSON` are inherited by the phase class; however, due to the locality principle of attribute modification, inherited attributes cannot be modified by new valuation rules. Semantically, this restriction requires retractions to model inheritance for phases.

The declaration of object roles enables an *event–driven* classification into object roles. In the current form, dynamic roles are not yet covered directly by the theory presented. The Troll language offers an additional **specialization** construct to classify objects based on constant attribute values. Both language constructs allow multiple inheritance inside the same inheritance schema. Naming conflicts can be solved by explicit renaming.

Complex Objects

The semantics base of building complex objects are *incorporation morphisms* as introduced in section 4. Troll allows to explicitly declare such an object inclusion as a retraction with possible renamings with the **including** construct [JSS91, JHSS91]. Such an incorporation is static in principle, i.e. it may not change during object lifetime.

In order to model the case of dynamically changing component objects, for example the `Trainer` and the `Players` of a `FootballTeam`, parametrized component constructors like **SET** and **LIST** for subobjects are included in Troll. Consider for example the object representing `TheCompany`, which is a complex object having a set of departments as component:

```
object TheCompany
   template
      components
         depts:  SET(DEPT);
      ...
end object TheCompany;
```

In order to model the dynamic changes of the component, special component manipulation events and attributes are implicitly generated with such a component declaration, for example `depts`.**INSERT**(|`DEPT`|) to add a department and `depts`.**COUNT** to count the current component instances. Additionally, the language TROLL offers the possibility to declare *local* subobjects to model non-shared object hierarchies ('disjoint complex objects') [JHSS91], for example the `Chapters` of a `document`.

Views as Interfaces

The basic concept of object encapsulates an internal state by defining an access interface consisting of attributes and events. Additionally encapsulated interfaces support the definition of restricted views on objects and classes allowing for access control und structuring like relational views in databases. Such interface definitions correspond to the abstraction and interface constructs discussed in section 4.

In TROLL, *interface* declarations can define a restriction of an object (class) signature. The first example is an interface to an object class `PERSON` defined for the use of the salary department or a subsystem handling the task of preparing the monthly salary report. Only attributes and events being of interest for this department are shown in the interface signature.

```
interface class SAL_EMPLOYEE
encapsulating PERSON
    attributes
        Name:string;
        IncomeInYear(integer):money;
        Salary:money;
    events
        ChangeSalary(money);
end interface class SAL_EMPLOYEE;
```

Additionally, we allow for defining derived attributes and events in interface definitions. An example for such a derivation is the following declaration of a derived attribute and a derived event.

```
    derivation
        derivation rules
            CurrentIncomePerYear = Salary *13.5;
        calling
            IncreaseSalary >> ChangeSalary(Salary *1.1);
```

Besides these encapsulation mechanisms on single object instances, we have a selection mechanism allowing to encapsulate part of a class population. The following interface class `RESEARCH EMPLOYEE` selects only those employees currently working for the research department.

```
interface class RESEARCH EMPLOYEE
encapsulating PERSON
selection where SELF.Dept = 'Research';
  ...
end interface class RESEARCH EMPLOYEE;
```

In the terminology presented earlier, one may implicitly define an *aggregation object* identified by the identification of its parts [EGS91, SE90] over each two objects using incorporation morphisms. Therefore, we can easily extend our mechanism to support *join views* following the database terminology:

```
interface class WORKS_FOR
encapsulating PERSON P, DEPT D
selection where P.oid in D.employees;
    attributes
      DeptName: string;
      PersonName: string;
    derivation
      derivation rules
        DeptName = D.id;
        PersonName = P.name;
end interface class WORKS_FOR;
```

In a certain sense, such join views correspond to *derived relationships.*

Interaction Relationships

In the TROLL language, interaction between otherwise unrelated objects is established by introducing *interaction relationships.* Such relationships realize an explicit concept of communication channels between objects. Semantically, they correspond to object aggregation using interaction morphisms. The main communication principle in TROLL is *event calling* realizing a synchronous, asymmetric communication between objects.

Consider e.g. the promotion of a person `P` identified by `P.oid` to become a manager of a department `D` identified by `D.oid`. The event `NewManager(P.oid)` of the department object calls the event `BecomeManager` of the corresponding person object:

```
relationship Promotion between PERSON P, DEPT D;
    DEPT(D.oid).NewManager(P.oid) >> PERSON(P.oid).BecomeManager;
```

6 Conclusions

The concept of an object seems to be very intuitive — at first sight! However, when trying to get the variety of object–oriented concepts and constructions into a systematic framewwork, things turn out to be not so simple. First steps towards a precise mathematical underpinning of the ideas presented here are given in [ES91, CSS91], exploring different semantic domains.

There is one fundamental issue of object–orientation which is not treated in this paper, namely *reification.* Reification requires a more general notion of process morphism, involving transactions in the place of events (see for instance [CSS91]). It remains to be investigated, however, which the most appropriate notion is, how it can be used to construct an implemetation on top of a given platform of objects, how this can be described by suitable language features, and what an appropriate notion of correctness is in this framework. Naturally, the issue of (hierarchic) transaction management comes in here, among others.

It should be pointed out that the reification relationship we have in mind is between objects and objects, not between object specifications and object specifications, and not between objects and object specifications either. That is, what we have in mind is software layers sitting on top of each other within running systems.

Features of the TROLL language as demonstrated in this paper should illustrate how the basic concepts and constructions can be put to work. A more comprehensive language description can be found in [JHSS91, JSS91]. TROLL is based on concepts from semantic modeling, algebraic specification and specification of reactive systems, combining the advantages of these approaches. TROLL offers a variety of structuring mechanisms for specification so that system specifications can be constructed from components that can be analysed locally. Additional language features not discussed in this paper give support for data type specification for attribute values, reusability using libraries of parameterized template and class type specifications, and support for modularization of system architecture.

Acknowledgements

Thanks to all IS-CORE colleagues who contributed to the development of the ideas presented here, and who took part in developing the TROLL specification language. In particular, Cristina Sernadas participated in discussing the basic ideas of objects and object descriptions. She, Ralf Jungclaus and Thorsten Hartmann were heavily involved in defining the TROLL language. Felix Costa's contribution to object semantics are greatfully acknowledged.

References

[At89] Atkinson et. al.: The Object–Oriented Database System Manifesto. 1st Int. Conf. on Deductive and Object–Oriented Databases, Kim, W. et. al. (eds.), 1989, 40–57

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

[BM91] Beeri, C.; Milo, T.: A Model for Active Object Oriented Database. Proc. 17th VLDB, Sernadas, A. (ed.), Barcelona 1991, to appear

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

[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

[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

[DRW89] Dignum, V.G.; van de Riet, R.P.; Wieringa, R.: Generalization and Specialization of Object Dynamics. Rapportnr. IR-204, Vrije Universiteit Amsterdam 1989

[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

[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. deBakker, J.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, Techn. 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

[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.

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

[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

[He88] Hennessy, M.: Algebraic Theory of Processes. The MIT Press, Cambridge, Mass. 1988

[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, Techn. 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), Springer 1991, 365–407

[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

[KM89] Khosla, S.; Maibaum, T.: The Prescription and Description of State-based Systems. In: B. Banieqbal, H. Barringer and A. Pnueli (eds) Temporal Logic in Specification, LNCS 398, Springer-Verlag 1989, 243-294

[KS91] Kappel, G.; Schrefl, M.: Object / Behavior Diagrams. Technical Report CD-TR 90/12, TU Wien, 1990. To appear in Proc. Int. Conf. on Data Engineering 1991.

[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

[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

[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

[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

[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

[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

[Wi90] Wieringa, R.J.: Equational Specification of Dynamic Objects. In: Meersman, R.; Kent, W. (eds.): Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4), Windermere (UK), 1990. North-Holland, Amsterdam. In print.