Object-Oriented Specification and Stepwise Refinement*†

Gunter Saake
Ralf Jungclaus
Hans-Dieter Ehrich

Abt. Datenbanken, Techn. Universität Braunschweig

Postfach 3329, 3300 Braunschweig, Germany

Abstract

A basic concept in object-oriented approaches is the notion of object as integrated unit of structure and behavior. Conceptually, objects are modeled as processes of which certain dynamic characteristics of their internal state can be observed using attributes. Objects are the basic units of design. Systems are composed from objects that interact to provide the desired services. In the semantics domain, concepts related to the object-oriented paradigm like interaction, inheritance and object aggregation can be uniformily modelled by object morphisms.

In this paper we introduce the language TROLL to conceptually specify dynamic information systems. TROLL supports abstract description of temporal evolvement of objects, classification of objects, complex objects, active objects and specialization hierarchies. Additionally, the integration of concepts like modularization support, and component reusability into the TROLL framework is briefly discussed.

1 Introduction

The modeling of complex systems requires the formal description of a large number of features and properties that are different in nature. Established formal methods generally focus on certain aspects of system design (e.g. data structures or functionality or dynamics) and thus do not support an integrated description of the system to be developed. Describing a complex dynamic software system may e.g. require the modeling of database structures, interactive interfaces, tasks (or scripts), active components (like system clocks) or the explicit specification of temporal requirements. In general, complex systems may conceptually be regarded as *collections of interacting subsystems*.

Aspects like those mentioned above should be supported by a formalism to specify complex open systems. Additionally, such a formalism should meet requirements like

*To appear in: Proc. IFIP TC6 Int'l Workshop on Open Distributed Processing, Berlin 1991, published by North-Holland.

†This work was partly supported by CEC under ESPRIT BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability) and by Deutsche Forschungsgemeinschaft under grant no. Sa 465/1-1.

modularization support and component reusability. Approaches to object-oriented design of programs and databases provide useful concepts but yet lack a formal foundation which is essential for the design of complex and safe systems.

A basic concept in object-oriented approaches is the notion of object as integrated unit of structure and behavior. In our approach, objects are modeled as processes of which certain dynamic characteristics of their internal state can be observed using attributes. Objects are the basic units of design. Systems are composed from objects that interact to provide the desired services.

The current research in object-oriented databases [ZM89, Kim90] and active databases [Day88] leads to the notion of *object bases* containing both passive and dynamic/active objects as basic building blocks of an application system [SSE87, SFSE89, Wie90, KS90]. This new approach to system design has its roots in several fields of research:

- The *object-oriented databases* now under development and formalization at several sites mainly base on new database structuring techniques which firstly were required by non-standard applications [Dit88, Bee90]. They usually support object-centered clustering of data, object identy and inheritance of object properties. Object specific methods (for updates) are realized in some prototypes but are outside the current state of most data model formalization approaches [Bee90].

- *Object-oriented design* focuses on the integrated design of structure and dynamics encapsulated in object units [Boo90] but lacks up to now a formal semantics.

- *Object-oriented programming languages* like Simula, Smalltalk, Eiffel or C++ concentrate on the computational aspects of objects as independent units of computation having a local memory and communicating solely through method calling/message passing. The basic elements of computation are methods manipulating the local object variables in a conventional operational way.

The idea of *dynamic object bases* combines these approaches into one framework supporting structured and persistent database objects as well as object dynamics in terms of update methods.

This paper is organized as follows: After discussing the concept of object-orientation in a general sense, we present a framework for modelling object systems. We introduce the language TROLL to conceptually specify dynamic object systems. The specification language TROLL supports abstract description of temporal evolvement of objects, classification of objects, complex objects, active objects, specialization hierarchies, and modularization. The remainder of this contribution gives some ideas towards supporting the design of large object systems by modularization and refinement techniques.

2 Object-Orientation

In the last years, the object-oriented paradigm has attracted much attention in different fields of computer science, among them object-oriented design (for a survey see e.g. [MK90]), object-oriented programming languages [Weg90] and object-oriented databases [ZM89]. Resulting from this diversity, several different opinions exist what makes up the very essence of object-orientation. Different application areas naturally lead to different views on objects and their properties, and it is not easy to give a definition of a concept of object which a majority agrees on.

In this section, we will try to identify some basic features of object-oriented techniques which must be supported by an object-oriented conceptual modeling framework. In the conceptual modeling phase, we have to abstract from implementation-related details and techniques which play a major role in object-oriented programming languages and databases.

- The basic concept of object-oriented design is the concept of *objects as units of structure and behavior*. Objects integrate the structural and behavioral aspects of some 'system entities' into inseparable design units. An object has an *encapsulated internal state* which can be observed and manipulated exclusively through an object 'interface'. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e., methods), object-oriented database approaches put emphasis on the observable structure of objects (through attributes). We propose to support both views in an equal manner for the design of objects, i.e. object structure may be observed through attributes and object behavior may be manipulated through events, which are abstractions of methods.

- Objects are the *units* of *design*. Following this perspective, uniform modeling of the system *and* the environment (in which it will be embedded) is made possible. Thus, we achieve clean interfaces between components that are part of the environment and components that are computerized later on. This approach results in having higher levels of modularity and abstraction during early system design.

- Dynamic objects somehow *communicate* with each other. In object-oriented programming, this may be realized by method calling (i.e., procedure call) or by process communication mechanisms. For conceptual modelling, the concepts of *event sharing* and *event calling* serve as abstractions of those implementation-related techniques.

- Objects are *classified* into *object classes and object types*. The term object class denotes a dynamic collection of existing objects wich are conceptually treated as objects of the same kind (extensional classification), whereas an object type describes the possible instances of an object description (intensional description). Both concepts are commonly used in an integrated way, i.e. an object type defines the structure of instances of a corresponding object class and vice versa.

- Objects (and object classes) are often embedded into a *class hierarchy with inheritance*. Even if the concept of inheritance in object-oriented techniques seems to be relevant for most researchers, there is still a lot of confusion which kinds of inheritance should be supported by an object-oriented approach. We distinguish two sorts of inheritance relations which are sometimes confused in the literature. *Syntactic inheritance* denotes inheritance of *structure or method definitions* and is therefore related to reuse of code (and to overriding of code for inherited methods). *Semantic inheritance* denotes inheritance of *object semantics*, i.e. of the objects themselves. This kind of inheritance is known from semantic data models, where it is used to model one object that appears in several roles in an application. In this paper we will only address *semantic inheritance*.

- A concept which has attracted a lot of attention in object-oriented data models is the *composition of complex objects from objects*. In object-oriented design, the composition has to obey both structural *and* behavioral aspects. Along with the strict concept of object encapsulation, object composition can be modeled by *safe object import* using object incorporation morphisms.

- The last relevant aspect which must be supported by an object-oriented specification technique is a formal notion of *object refinement and object implementation*. These concepts are mandatory for structured design of object-oriented systems and enable e.g. top-down refinement techniques and abstract object descriptions in early design phases. Technically, object refinement and implementation can be modeled by explicit object interfaces for complex objects and by a mechanism to call sequences of events as atomic units of behavior.

In the following section, we will characterize basic concepts and constructs of such object systems and show how the specification of such systems is supported by the TROLL-language.

3 Fundamental Object Concepts

The aim of this section is to sketch a semantical framework dealing with objects, object classes, inheritance, interaction and other concepts related to the object-oriented approach. Conceptually, objects can be treated as *communicating processes with observable attributes* [SE90]. Based on single objects as processes, we have to formalize object systems as collections of objects related in manyfold ways.

What is an **object**? An object has structure and behavior, but there may be many objects with the same structure and behavior while they are different as objects. That is, an object has an *identity*, and it is an instance of a structure and behavior *template* which it may share with many other objects. Only if we distinguish clearly between individual objects and their templates is it possible to treat object concepts like inheritance and interaction in a clean and satisfactory way: interaction is a relationship between *different individual* objects, while inheritance relates *aspects* of the *same individual* object — which may be expressed by a corresponding inheritance schema relationship among anonymous templates.

There is a notorious confusion around the object–oriented notions *type* and *class*: some people would probably use one of these words for what we call template. But all three notions are different, as we will see. By *template* we mean an object's structure and behavior pattern without individual identity. Formally, a template can be modelled as a process [ES91].

Inheritance and interaction relationships among (aspects of) objects are based on corresponding relationships among their templates. In [ES91], a general notion of *template morphism*, i. e. a structure and behavior preserving map among templates, is described which captures inheritance as well as interaction relationships. In this paper, we will only use the special case of a template *projection*: it projects a template to a portion of it, where the portion might represent an abstraction or a physical part. For instance, a computer template may be projected to that of an electronic device (because any computer is an electronic device) or to the template of a computer cpu (describing a part–of relationship), cf. example 3.1 below.

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. In the TROLL language later on presented in this contribution, object identities are modelled as values of an arbitrary abstract data type.

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!

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

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 3.1 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`.

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 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*

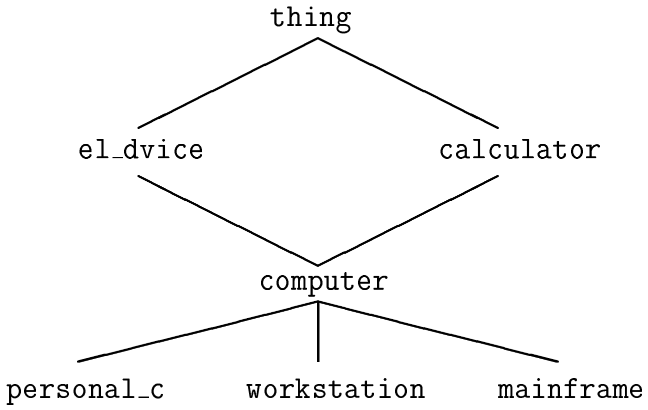
$$\texttt{CYY} \bullet \texttt{cpu} \longrightarrow \texttt{CBZ} \bullet \texttt{cable} \longleftarrow \texttt{PXX} \bullet \texttt{powsply}$$

The main effect of an object morphism is to include part of an object (aspect) into another object. This inclusion of properties can be characterized as semantic inheritance between object (aspects) in contrast to syntactical inheritance of signature between object descriptions.

As we have seen, objects may appear in different aspects, all with the same identity but with different 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, $h : \texttt{SUN}\bullet\texttt{computer} \to \texttt{SUN}\bullet\texttt{el\_dvice}$.

Template morphisms intended to prescribe inheritance are called *inheritance schema morphisms*. An *inheritance schema* is a diagram consisting of a collection of templates related by inheritance schema morphisms.

Example 3.2 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 Δ.

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 community of aspects*. Of course, the community 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 community.

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 in the previous section, 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 3.3 Referring to the inheritance schema in example 3.2, 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. This member template is called its *type*. But, since each aspect of an object 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 items 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 items (actions and observations) are provided implicitly, they need not be specified by the user.

Since classes are objects or aspects of objects, there is no difficulty in constructing meta–classes, i. e. classes of classes of .... See [ES91] for further details.

When we build an object–oriented system, we must provide an **inheritance** schema, as explained above. Now 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 surjective in the sense that all items of both partners are involved in an inheritance relationship. We refer to [ES91] for further details.

Example 3.4 Referring to example 3.2, consider the inheritance schema morphism h : `computer` $\rightarrow$ `el_dvice` expressing that each computer is an electronic device. Let `el_dvice` have the actions `switch_on` , `switch_off` and the observations `is_on` , `is_off`. By inheritance, `computer` has corresponding items `switch_on_c`, `switch_off_c`, etc. h maps `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 items.

Concerning the behaviors 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. ○

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

- if t is already in Δ, we create u and connect it to the schema via $h : t \to u$,
- if u is already in Δ, 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 3.2 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 3.5 Referring to example 3.2 and assuming top–down construction, the template for `computer` is constructed by multiple specialization (multiple inheritance) from `el_dvice` and `calculator`. ○

Example 3.6 If we would have constructed the schema in example 3.2 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`. ○

When we build an object–oriented system, we must provide an *object community*, i. e. a collection of **interacting** objects. Now 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, it seems that also interaction morphisms are surjective in all meaningful cases (cf. [ES91] for further details).

Example 3.7 Referring to example 3.1, 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 items 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. ○

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

- if $a{\bullet}t$ is already in Γ, 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 Γ, 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 Γ, we have to close it with respect to Δ, 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 3.8 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! $\bigcirc$

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 3.7 (cf. also example 3.1). An example for aggregation is the following.

Example 3.9 Referring again to example 3.1, 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 morphisms f : `SUN•computer` $\to$ `PXX•powsply` and g : `SUN•computer` $\to$ `CYY•cpu` showing the relationships to the parts. $\bigcirc$

After this brief discussion of the conceptual framework, we present in the next section specification language features based on this framework.

4 Language Features for Object Specification

A specification language for object systems has to offer language features which allow to describe the conceptual schema of a system using abstract and implementation–independent concepts. All aspects of a complete system must be covered by the description. As an example for designing complex systems, consider the case of describing an information system containing information about a given Universe of Discourse. This task puts some requirements upon a description language to be used for specifying objects:

- Objects must be described in an *abstract* way, i.e. independently from their later implementation. This holds both for objects from the UoD and for objects used for system functions.

- The conceptual schema must be complete in terms of restrictions on objects, be it constraints on object properties, restrictions on object updates or restrictions on long-term object evolution, because it is the central description of the object system being the reference both for the implementers as well as for application developers.

- As mentioned already, a formal semantics is an indispensible property of a language for specifying a conceptual model, because otherwise transformations to lower levels and their verification cannot be formalized.

We will use the presented object model as semantical framework for conceptual schema descriptions. Let us now show how objects and object classes can be specified in the TROLL-language. The language TROLL is presented in [JHSS91]. Therefore, we will recapitulate some basic features of TROLL only. We present the basic language features using a typical information system example, even if the language itself can be used to specify other system types as well. As an example, consider the description of an object class representing departments:

```
object class DEPT
  identification id: string;
  data types date,|PERSON|,set(|PERSON|);
  template
    attributes
      est_date: date;
      manager: |PERSON|;
      employees: set(|PERSON|);
    events
      birth establishment(date);
      death closure;
      new_manager(|PERSON|); assign_official_car(|CAR|,|PERSON|);
      hire(|PERSON|); fire(|PERSON|);
    valuation
      variables P: |PERSON|; d: date;
      [establishment(d)]est_date = d;
      [new_manager(P)]manager = P;
      [hire(P)]employees = insert(P,employees);
      [fire(P)]employees = remove(P,employees);
    permissions
      variables P: |PERSON|;
      { sometime(after(hire(P)))} fire(P);
      { for all(P: |PERSON|): sometime(P in employees) ⇒
         sometime(after(fire(P))) } closure;
end object class DEPT;
```

After the **identification** keyword, the abstract data type for object identifiers is explicitly declared analogously to database keys. Attributs and events define the access interface forming the object *signature*.

The **valuation** rules describe the evolvement of the attribute observations. They describe the effect of event terms on attribute values in terms of a data-valued term evaluated before the event occurence which determines the new attribute value. The **permissions** describe permitted sequences of events and thus restrict the set of possible sequences over the alphabet of events to admissible sequences. Additional features of the language not described here include arbitrary constraints on attributes, liveness requirements (i.e. goals to be achieved by the object in an active way) and activity, i.e. events that may occur on the object's own intitiative whenever their occurrence is possible. A more elaborated description of TROLL language features can be found in [SJ91, JSS91, JHSS91, JSH91].

An object in TROLL is thus a unit encapsulating data and evolvement. *Encapsulation* means that local data can solely be manipulated by local events which can be seen as basic operations on the state of an object and that the internal state can be observed by attributes only. In contrast to the view of encapsulation in object-oriented programming, we allow attribute values to be read by other objects, i.e. they are part of the object's interface. In a certain sense, we can see attributes as the read-only interface for queries and events as the manipulation interface offered by an object. We may, however, hide more information by explicitly defining interfaces to objects (see section 5.1).

TROLL also supports the specification of aspects. As stated in section 3, this involves on the one hand inheritance of templates which essentially means the reuse of specification texts. On the other hand, aspects involve the incorporation of instances. That is, attributes of the incorporated instance may not be changed by events not local to the incorporated instance. Events local to the incorporated instance may, however, be triggered from the outside.

In our language, we distinguish between *specializations* and *phases*. Specialization is static, i.e. the specialized object is born as a special kind of the base object and remains of this special kind for it's entire life (consider e.g. a woman as specialization of a person). An object being a special kind just for a part of it's life has special properties only when it is in this role (consider e.g. a manager as a special kind of person). Note that both concepts are based on semantic inheritance, i.e. on inheritance schema morphisms.

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

```
object class PERSON
  identification
    name: string;
    birthdate: date;
  template
    attributes
      ...
    events
      ...
      become_manager;
end object class PERSON;

object class MANAGER
  view of PERSON;
  template
    attributes
      OfficialCar : |CAR|;
    events
      birth PERSON.become_manager;
      ...
    constraints
      static Salary ≥ 5.000;
    ...
```

end object class `MANAGER`;

In TROLL, complex objects may be constructed using *aggregation*, i.e. composition of objects from sets or lists of objects as well as from single objects. The concept of aggregation is based on semantic incorporation: the components of a complex object are included in a property-preserving way.

Consider for example the object representing `TheCompany`, which is a complex object having a list of departments as component:

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

An object society is a (possibly large) collection of objects that *interact*. Interaction is e.g. performed by *event calling*. To call an event means to force *synchronous* occurrence of the called event by the occurrence of the calling event.

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

```
global interactions
   variables P: |PERSON|; D: |DEPT|;
      DEPT(D).new_manager(P) >> PERSON(P).become_manager;
```

Note that event calling is also used for interactions with incorporated objects. An important extension of event calling is *transaction calling* enabling an event to call a finite sequence of other events treated as a transaction unit [SE90]. We will see the usefulness of this concept during the discussion of formal object implementation in section 5.2.

5 Object Interfaces and Formal Implementation

Our aim is to formally describe large object systems based on the TROLL language. One important aspect of object-oriented design concepts is that object-orientation allows to leave the strict hierarchical system structure enforced by other design approaches. To allow a liberal system design, we have to support two basic structuring principles:

- We need a general *interface* mechanism to allow combination of subsystems while hiding details of these subsystems. This interface mechanism should support control and data flow in both directions.
- Subsystems should be designed on the basis of existing components in a stepwise refinement process starting with an abstract conceptual description and ending with executable descriptions close to the implementation platform. Therefore we need the concept of *formal implementation of objects over objects* supported by language features.

The following subsections discuss the language primitives for supporting these tasks as they are included in the TROLL language.

5.1 Object Interfaces

The basic idea of object interface definition is to give an access interface to existing objects. That is, we do not define new objects by defining interfaces. In terms of the presented approach, we define *new interfaces* to existing objects by defining attributes, events and components (interfaces for derived objects). Thus, external object interfaces are very similar to conceptual schema objects, but their internal semantics is given by a transformation of the signature components to an implementation in terms of conceptual objects. In terms of the concepts discussed in section 3, interfaces are language features to describe *interfacing* by interaction morphisms and abstractions.

As mentioned before, an *object (class) interface* is mainly a mechanism for controlling access to objects in an object base. Access control for a single object is achieved by defining a restricted interface for the object, e.g. by performing a projection on the attributes and events of the object. Furthermore, we allow a restricted form of deriving new attribute values and events. This projection can be defined for single objects as well as for all instances of an object class. The following examples show the principles of defining object interfaces.

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;
```

The semantics of this interface definition is a restriction of the *access* to `PERSON` objects as it is done in relational databases by a projection view. Please note that the projection definition does not only restrict the observation of attributes but also the possible modification events being offered to the view users. This kind of projection can be used to restrict the access to complex object components, too.

The object's identifiers (for `PERSON`s the values of `name` and `birthday`) are not generally preserved by an interface definition as visible attributes, but the internal object identity is preserved since we do not derive new objects.

As an example for an interface with *derived attributes and events* we have the interface class `SAL_EMPLOYEE2` where the additional attribute `CurrentIncomePerYear` is derived from the value of the attribute `Salary` and only a restricted way of changing salaries is offered.

```
interface class SAL_EMPLOYEE2
encapsulating PERSON
      attributes
```

```
            Name: string;
            derived CurrentIncomePerYear: money;
            Salary: money;
        events
            derived IncreaseSalary;
        derivation
            derivation rules
                CurrentIncomePerYear = Salary *13.5;
            calling
                IncreaseSalary >> ChangeSalary(Salary *1.1);
end interface class SAL_EMPLOYEE2;
```

The derivation part is usually hidden to the users. For the derivation of attribute values we may use an object query language enabling value retrieval from object states. We use an object query algebra presented in [SJ90, SJS91]. This algebra resembles well known concepts of database query algebras handling values (not objects !). Algebra terms are evaluated locally to the encapsulated object. For the derivation of events we can use arbitrary *process calling* [SE90]. Thus, the derived event may be evaluated by a finite process defined over the local events of the encapsulated object.

Furthermore, object class interfaces allow the selection of a subpopulation of an object class. To define a selection, we allow arbitrary query terms of sort **boolean**. A variable SELF denotes the currently observed instance. The following interface class RESEARCH EMPLOYEE selects only those persons working for the research department.

```
interface class RESEARCH EMPLOYEE
encapsulating PERSON
selection where SELF.Dept = 'Research';
        attributes
            Name: string;
            Salary: money;
        events
            ChangeSalary(money);
end interface class RESEARCH EMPLOYEE;
```

In the terminology presented in section 3, one may implicitly define an *aggregation object* identified by the identification of its parts [EGS90, SE90] over each two objects using incorporation morphisms. Therefore, we can easily extend our mechanism to support *join views*:

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

```
        PersonName = P.name;
end interface class WORKS_FOR;
```

In a join view, we typically introduce variables to identify the participating object instances (which is, however, not necessary in our example). These variables can be used in the (optional) selection clause and in derivation rules. As said before, the internal object identity is preserved by the view and therefore even derived updates can be offered in the view definition without semantical difficulties.

The purpose of interfaces is in principle mainly an *authorization* for *restricted object manipulation* and *attribute value retrieval*. Interfaces have nothing to do with object copies — they are only a restricted view on existing objects rather than an object-preserving operation. We can regard interface definitions as a mechanism to select objects for manipulation. The manipulations themselves are encapsulated within the objects, therefore we do not have to describe the concrete manipulations but the restrictions for using them correctly.

5.2 Formal Implementation

The basic idea of formal object implementation is taken from the area of abstract data types: Starting with already specified base objects, we define an abstract object in terms of the attributes and events of these base objects. The concrete implementation is hidden in the abstract object which then can be used abstracting from its concrete implementation. An implementation consists of three parts, namely

- the declaration of the base objects,
- the aggregation of the base objects and the implementation of the new events and attributes in terms of the base object signature,
- and the hiding of the implementation details using an interface definition.

As an example, we implement the object class EMPLOYEE on top of an object emp_rel describing a database relation of a relational database. The object class EMPLOYEE is very simple to keep the implementation example small. EMPLOYEE objects are identified (like persons) by their name (EmpName) and birthday (EmpBirth) and have a current salary (Salary). As update events we have hire and fire of an employee (HireEmployee and FireEmployee) and a change of their salary (IncreaseSalary). The implementation is done using one object managing a set of tuples storing the data. This example was chosen because it shows some critical aspects of formal object implementation, among them the sharing of base objects by several abstract objects.

We start with defining the base object emp_rel.

```
object emp_rel
template
  data types string, date, integer;
  attributes
    Emps : set(tuple(ename:string, ebirth:date, esalary:integer));
  events
    birth CreateEmpRel;
```

```
      UpdateSalary(string, date, integer);
      InsertEmpl(string, date, integer);
      DeleteEmpl(string, date);
      death CloseEmpRel;
   valuation
      variables n:string, b:date, s:integer;
      [CreateEmpRel] Emps = {};
      [InsertEmp(n,b,s)] Emps = insert(Emps, tuple(n,b,s));
      {in(Emps,tuple(n,b,s))} ⇒
         [DeleteEmp(n,b)] Emps = delete(Emps, tuple(n,b,s));
   permissions
      variables n:string, b:date, s:integer;
      { exists(s1:integer) in(Emps,tuple(n,b,s1))} UpdateSalary(n,b,s);
      { Emps = {} } CloseEmpRel;
   interaction
      variables n:string, b:date, s:integer;
      ChangeSalary(n,b,s) >> ⟨ DeleteEmp(n,b); InsertEmp(n,b,s) ⟩;
end object emp_rel;
```

The operators **in**, **insert** and **delete** are operations defined for the parametrized data type constructor **set**. The keyword **tuple** is used for the data type constructor (the 'record of' construct of programming languages) as well as for the tuple creation operation.

The interfaces (i.e., the object signature) of such implementation objects can be derived automatically from a given relational schema. For example, the semantics of update operations are semantically modelled by a sequence consisting of an insert and delete operation in a set of tuples under the requirement to satisfy the key constraints. In general, there are a number of update events generated from a given relational schema. It should be mentioned that this relation object itself may be implemented for example by another object using a B-tree or a hash table access method.

To define the relation between `emp_rel` and `EMPLOYEES`, we define as a next step the implementation of an object *class* `EMPL_IMPL` on top of the *single object* `emp_rel` representing the relational implementation. This way, we are able to express the sharing of a resource by a number of 'client' objects.

```
object class EMPL_IMPL
identification
      data types date, string;
      EmpName : string;
      EmpBirth : date;
template
   inheriting emp_rel as employees;
   attributes
      derived Salary;
   events
      birth HireEmployee;
      derived IncreaseSalary(integer);
      death FireEmployee;
   constraints
      derivation rules
         Salary =
```

```
            count(project[esalary]
                    (select[ename = EmpName and ebirth = EmpBirth] (employees)))
    interaction
        variables n:integer;
        HireEmployee >> employees.InsertEmpl(self.EmpName, self.Empbirth, 0);
        FireEmployee >> employees.DeleteEmpl(self.EmpName, self.Empbirth);
        IncreaseSalary(n) >>
            employees.UpdateSalary(self.EmpName, self.Empbirth, self.Salary + n);
end object class EMPL_IMPL;
```

The object class EMPL_IMPL realizes the implementation of 'employee' instances onto the single object realization. The last step is to hide implementation details (encapsulation) by defining an interface class EMPL for EMPL_IMPL.

```
interface class EMPL
encapsulating EMPL_IMPL
        attributes
            EmpName: string;
            EmpBirth: date;
            Salary: integer;
        events
            IncreaseSalary(integer);
            HireEmployee;
            FireEmployee;
end interface class EMPL;
```

To show the correctness of our implementation, we have to prove that all properties of the original EMPLOYEE specification can be derived from EMPL, too. It is outside the scope of this paper to present a proof theory for formal object implementation. Basic concepts for a proof theory for object specifications can be found in [FSMS90, FM91].

An implementation may use several base objects to implement an abstract object, for example if normalization of relations becomes necessary to realize nested structures. In general, each formal implementation consists of the following steps :

1. First of all, a complex object is constructed consisting of all base objects needed for the implementation.

2. An object (class) interface defines the encapsulation of implementation details.

Formal implementation of objects over objects enables consistency checks and formal verification because it is done in the same formal framework [FSMS90].

6 Schema Architecture and Modularization

In this section, we will briefly discuss some ideas towards the development large systems using modularization principles for object systems. After several facets of modularization object systems are informally discussed, we propose the use of a layered schema architecture for object bases modules.

6.1 Modularization of Object Systems

The main motivation for introducing additional modularization constructs to the presented object specification language is the problem of managing large object system descriptions. Objects are encapsulated modules of small granularity, but for really large systems we need further modularization constructs supporting specification in the large. Another motivation is the safe access to existing system components encapsulated by an object-oriented access interface during the specification process of components to be developed. Other important topics are the support of cooperative work by system modularization, modules as units for access rights, and local consistency checking.

In the area of object system specification, we can distinguish several kinds of modularization principles occuring in design documents:

- The first structuring principle for large specification documents is the use of *object specification libraries* to support reusability of object descriptions. We call this principle *syntactical reuse* of specifications text.

  We will not discuss syntactical reuse and related aspects like parametrization of object descriptions in this paper. For a proposal to integrate syntactical reuse in a framework for object specification we refer to [SRGS91].

- The classical case for modularization is the support of a *hierarchical system structure* as known from modern programming languages. The main aspect of this approach for system design is the support of divide-and-conquer methods in the design process.

  An interesting special case is the support of *module refinement by formal implementation steps* where one (more abstract) module is implemented in terms of dependent other modules, where the control flow follows the hierarchy.

- The more general case are object system modules as communicating subsystems. In our framework, we call such systems *communicating object societies*. This idea corresponds to horizontal composition of independent object systems in contrast to the strict hierarchical composition of dependent subsystems mentioned before.

In contrast to the first case, we can characterize the cases of hierarchical and horizontal composition as *semantical (re)use* of object system modules by other object systems. Typical examples for this general kind of system modularization are

- the case of shared databases, where different subsystems have access using application-specific view interfaces only,

- a shared system clock or calendar, where we have both read access to the current time or date as well as an active triggering mechanism for time-dependent system activities,

- and as general case we have cooperating object systems as connected autonomous systems realizing cooperating applications, where we have again both passive access and active communication.

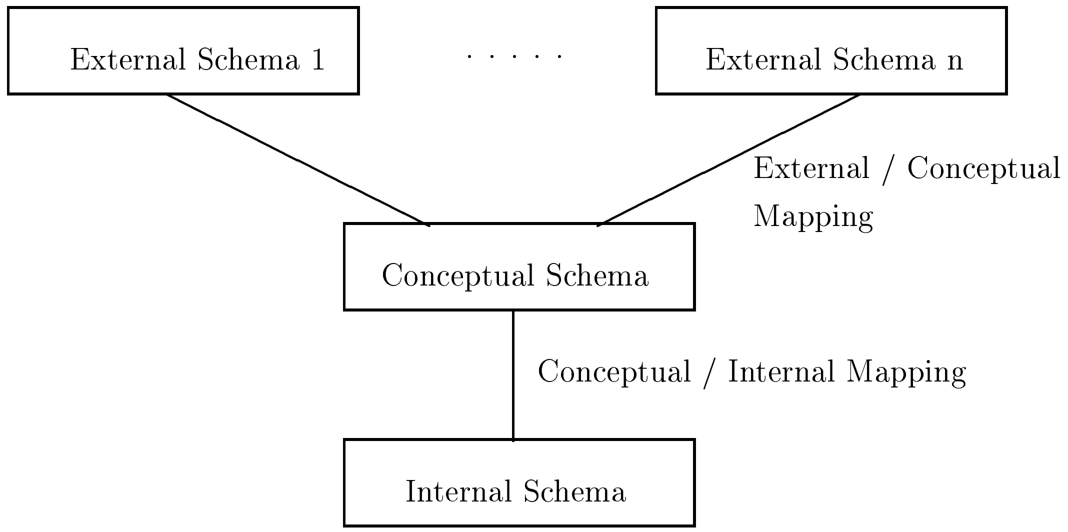


Figure 1: Schema structure of the three-level architecture.

What unit of modularization can support such different system architectures ? A single module must be expressed by an arbitrary object society to allow adequate module specifications. We need a concept of *society interface* structured like usual object societies but hiding module realization details. Of course, we can define such societies as collections of object interfaces as discussed in section 5.1.

Furthermore, we need language features for import and export of object society interfaces. Society interfaces can be passive or active interfaces; the passive case corresponds to hierarchical dependent subsystems and the active case to horizontal composition of independent systems. We need *several different export interfaces* for one module for modelling a controlled communication of autonomous subsystems.

As a summary of these considerations, we need a module concept supporting both the classical hierarchical refinement step for single modules and the general case of interacting modules with controlled module access using different export interfaces. Therefore, we propose to adopt the three-level schema architecture developed for database applications as structuring principle for object system modules.

6.2 Three-level Schema Architecture for Modules

To organize the description of an object subsystem, we propose the three-level schema architecture successfully introduced for structuring database applications [DAFTG86]. This three-level schema architecture for database applications can easily be explained using figure 1. This architecture organizes the schema in three different levels :

- The central *conceptual* schema represents the conceptual view on the complete database using an abstract data model schema. 'Abstract' data model means that the conceptual schema abstracts from implementation details.

- The *internal* schema describes the implementation details in terms of the used implementation platform (the used DBMS). This may require a change of the used data model. Typical description issues for classical DBMSs are access paths and their realization.

- The *external schemata* describe several views on the conceptual schema specific to particular applications or user groups.

We propose to adapt this three-level schema architecture for our abstract concept of dynamic objects (i.e., objects evolving over time). To realize such a schema architecture, we have to make use of several formalisms and languages :

- We need a *formal concept of objects* to define the semantics of an object base both on the conceptual and implementation level.

- Conceptual object base schemata must be defined using a language that enables an *implementation-independent abstract description* of object bases.

- A formal concept of *object interfaces* is required to define external schemata as views on an object base.

- Finally, we need a formal concept of *object implementation* to define the relation between conceptual schema and internal schema.

As this list shows compared with the topics of this contribution, the TROLL language and the theory behind it is well suited for using this architecture for object system modules.

Having modules organized following this architecture proposal, arbitrary systems can be built by connecting object system modules using society interface import. The implementation of single modules is hidden to the outside allowing the integration of already existing subsystems, too. The possibility of defining several external schemata as export interfaces allows to include access control and security mechanisms already on the system specification level.

7 Conclusions

In this paper, we have introduced a framework that attemps to overcome the gap between the description of static and dynamic properties of conceptual models. Object descriptions are the units of design and encapsulate all static and dynamic aspects local to an object. Object systems are collections of objects related by the notions of class membership, interaction, inheritance and object aggregation. We gave a formalization of our concept of object and introduced the language TROLL for the abstract description of object-oriented conceptual models. Besides the underlying framework, TROLL is based on concepts from semantic data modeling, from algebraic specification and from the specification of reactive systems and tries to combine the advantages of these approaches. TROLL offers a variety of structuring mechanisms for specifications, among them roles, complex objects and global interactions. With TROLL, system specifications may be constructed from components that can be analyzed locally.

Further work will focus on finding subsets of TROLL which are sufficient for the description of models on lower levels of abstraction. Using formal transformations, abstract specifications shall be refined and implemented. Other topics are modularization issues, in-the-large specification, reusability and graphical notations for TROLL.

Acknowledgements

Thanks to all IS-CORE colleagues, who contributed to the development of the concepts presented and in the definition of the TROLL specification language. In particular, Amílcar Sernadas and Cristina Sernadas participated in developing basic ideas of objects and object descriptions. Cristina Sernadas and Thorsten Hartmann were heavily involved in defining the TROLL language.

References

[Bee90] Beeri, C.: A Formal Approach to Object Oriented Databases. *Data & Knowledge Engineering*, Vol. 5, No. 4, 1990, pp. 353–382.

[Boo90] Booch, G.: *Object-Oriented Design*. Benjamin/Cummings, Menlo Park, CA, 1990.

[DAFTG86] Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group: Reference Model for DBMS Standardization. *ACM SIGMOD Records*, Vol. 15, No. 1, 1986, pp. 19–58.

[Day88] Dayal, U.: Active Database Management Systems. In: *Proc. Conf. on Data and Knowledge Bases*, Jerusalem, 1988. pp. 150–169.

[Dit88] Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems*. Lecture Notes in Comp. Sc. 334. Springer Verlag, Berlin, 1988.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Foundations of Object-Oriented Languages (Proc. REX School/Workshop)*, Noordwijkerhood (NL), 1990. LNCS 489, Springer-Verlag, Berlin, 1991, pp. 203–228.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Information Systems — Correctness and Reusability*, Technical Report 91-03, Technische Universität Braunschweig, 1991. pp. 1–24.

[FM91] Fiadeiro, J.; Maibaum, T.: Towards Object Calculi. In: Saake, G.; Sernadas, A. (eds.): *Information Systems — Correctness and Reusability*, Technical Report 91-03, Technische Universität Braunschweig, 1991. pp. 129–178.

[FSMS90] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. 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*.

[JHSS91] Jungclaus, R.; Hartmann, T.; Saake, G.; Sernadas, C.: Introduction to TROLL — Textual Language for Object-oriented Specification. In: Saake, G.; Sernadas, A. (eds.): *Information Systems — Correctness and Reusability*, Technical Report 91-03, Technische Universität Braunschweig, 1991. pp. 97–128.

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T. (ed.): *Proc. ER'91*, 1991. to appear.

[JSS91] Jungclaus, R.; Saake, G.; Sernadas, C.: Formal Specification of Object Systems. In: Abramsky, S.; Maibaum, T. (eds.): *Proc. TAPSOFT'91*, Brighton (UK), 1991. LNCS 494, Springer-Verlag, Berlin, pp. 60–82.

[Kim90] Kim, W.: Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, 1990, pp. 327–341.

[KS90] 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.*

[MK90] McGregor, J. D.; Korson, T. (Guest editors): Special Issue on Object-Oriented Design. *Communications of the ACM*, Vol. 33, No. 9, 1990.

[SE90] Sernadas, A.; Ehrich, H.-D.: What Is an Object, After All? 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.*

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, pp. 225–246.

[SJ90] Saake, G.; Jungclaus, R.: Information about Objects versus Derived Objects. In: Göers, J.; Heuer, A. (eds.): *Second Workshop on Foundations and Languages for Data and Objects*, Aigen (A), 1990. Informatik-Bericht 90/3, Technische Universität Clausthal, pp. 59–70.

[SJ91] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL Language. In: Harper, D. (ed.): *Proc. Workshop on Specification of Database Systems, SODS'91*, Glasgow, 1991. Springer-Verlag, Berlin. to appear.

[SJS91] Saake, G.; Jungclaus, R.; Sernadas, C.: Abstract Data Type Semantics for Many-Sorted Object Query Algebras. In: Thalheim, B. (ed.): *Proc. 3rd Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems MFDBS-91*, Rostock, 1991. Springer-Verlag, Berlin, 1991.

[SRGS91] Sernadas, C.; Resende, P.; Gouveia, P.; Sernadas, A.: In-the-large Object-Oriented Design of Information Systems. In: Van Assche, F.; Moulin, B.; Rolland, C. (eds.): *Proc. The Object-Oriented Approach in Information Systems.* North Holland, 1991. to appear.

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Hammerslay, P. (ed.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107–116.

[Weg90] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, Vol. 1, No. 1, 1990, pp. 7–87.

[Wie90] 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.*

[ZM89] Zdonik, S. B.; Maier, D. (eds.): *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann, Palo Alto, CA, 1989.