Formal Specification of Object Systems*

Ralf Jungclaus
Gunter Saake

Abt. Datenbanken, Techn. Universität Braunschweig
P.O. Box 3329, 3300 Braunschweig, FRG
E-mail: {jungclau/saake}@infbs.uucp

Cristina Sernadas

INESC, Apartado 10105
Rua Alves Redol 9, 1017 Lisboa Codex, Portugal
E-mail: css@inesc.ctt.pt

Abstract

The conceptual modeling of the Universe of Discourse (UoD) is an important phase for the development of information systems because the conceptual model is the basis for system development. Conceptual model specifications must be formal in order to be precise and unambiguous and to support consistency and completeness checks. The object-oriented paradigm is suitable for providing an integrated formal description of all relevant static and dynamic aspects of the UoD structured in objects. In this paper we introduce a formal concept of object suitable to represent the UoD by a collection of concurrent interacting objects. The **Oblog**$^+$-language for object-oriented UoD-specification based on this concept supports the integrated description of data about objects, the development of objects through time and of various relationships between objects taking into account static and dynamic aspects of object interaction.

1 Introduction

Information systems are data-intensive software systems which are capable of storing, manipulating and producing information about a Universe of Discourse (UoD). Thus, the basis for information systems development is a representation of the UoD in terms of all relevant structural and behavioral aspects [vGr82, SFSE89, Wie90a] and therefore the conceptual modeling phase is a very important phase in the development of information systems. The result of the conceptual modeling phase is an abstract formal representation of the relevant aspects of the UoD, which is called conceptual model specification

*This work was partially supported by the CEC under ESPRIT-2 BRA WG 3023 IS-CORE (Information Systems – COrrectness and REusability). The research work of Ralf Jungclaus is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1.

[BMS84, SS85, Wie90a, Wie90b] or requirements specification [Zav79]. This issue has been examined from different points of view in the knowledge base community [BM86, ST89], the database community [UD86, HK87, PM88], and the software engineering community [CHJ86, Par90].

The demands upon conceptual model specifications are manyfold [YZ80, BMS84, Par90]: On the one hand, specifications should be *readable* and *understandable* by both domain specialists and system developers, on the other hand specifications must support abstractions and conceptual integrity by a precise *formal framework and language.* Only formal specifications are compact and unambiguous and support consistency and completeness checks as well as systematic construction of prototypes and implementations. Furthermore, a conceptual specification framework must support *modifiability* of specifications (in case of changing requirements) and *liberality* of specifications, i.e. specifications should not enforce a single solution [LF82].

Traditional formalisms for conceptual model specification treat data and functions on data *separately.* In the design of databases, semantical data models stress data modeling [HK87, PM88], whereas in software engineering the emphasis seems to be more on functional models [DeM79, Jac83]. Some proposals allow separate description of static aspects and dynamic aspects in a uniform language framework (e.g. RML [GBM86], Taxis [MBW80]). In contrast, in the object-oriented approach data and functions local to an object are integrated in objects [Weg87, ABD$^+$89]. Thus, the relevant aspects of the UoD are structured in objects [KM90, Boo90]. An attempt is made to bridge the conceptual gap between specification and implementation since the artificial boundary between data and functions is overcome. Furthermore, the object-oriented paradigm seems to better reflect natural descriptions of UoD concepts because object descriptions may directly reflect UoD objects. In object systems, objects are encapsulated entities. Thus, the straightforward modeling of complex objects and complex relationships is supported.

An object-oriented approach supports modifiability of specifications. Only objects as encapsulated entities are changed. Thus, changes are local to objects and do not affect other parts of the specification if object interfaces remain unchanged.

Liberality in specifications can be achieved by *declarative* specification in which we only specify *what* has to be represented instead of *how* it should be represented.

Currently, most approaches to object-oriented conceptual modeling are rather informal [Boo90]. The modeling concepts rely on object-oriented programming and are therefore oriented towards implementations rather than requirement models. Formal approaches we are aware of are e.g. FOOPS [GM87] (which evolved from abstract datatype specification), CMSL [Wie90a] (which like our approach strongly emphasizes object dynamics) and Object/Behavior Diagrams [KS90] (a graphical approach supporting the modeling of static and dynamic aspects).

The approach described in this paper evolved from integrating work on formal program specification techniques like algebraic specification of abstract data types [BG77, EM85, FGJM85, ELG89], process modeling and process specification [Hoa85, BK86, Hen88, Mil89], abstract program specifications [Bal81, B$^+$85, Par90] and on conceptual modeling approaches for databases and information systems [BMS84, BM86, UD86, HK87, PM88, ST89] (although we only present a language instead of a graphical notation). The concept of object presented herein has been developed in [SSE87, SFSE89, SE90] accompanied by work on its semantics [ES89, EGS90]. The basic idea is to regard a UoD as a *collection of interacting objects.* In such objects, the description of structure (by means of properties) and dynamics (by means of processes over abstract events) is encapsulated. Collections of objects are structured using specialization, generalization, aggregation, and classification. System dynamics are described by interactions between dynamic objects. Based on the concept of object, work has been done concerning the semantics of object-oriented specifications [SS89, FSMS90, FM90]. We now present the **Oblog**$^+$-language for object-oriented conceptual model specification.

In the next section, we give a short introduction to the UoD at hand and try to point out the relevant static and dynamic aspects to be specified. In section 3, we formalize a simple model-theoretic concept of objects as observed processes and relationships between objects following [EGS90]. In sections 4 and 5 we introduce language constructs for the specification of simple objects and object societies, respectively. In these sections, we try to relate specification sections with object properties as described in section 3 since we specify the objects that represent relevant aspects of the UoD. We will close with a discussion of other approaches to conceptual modeling and some concluding remarks.

2 Description of the UoD

Using a simple example, we now want to point out requirements for formalisms suitable to describe UoD concepts. The UoD, which in part will be modeled in the following sections, consists of books in a library. In the library, only copies of books are present, books itself are regarded to be abstract objects only.

Each book is described by its title, the list of authors, the publisher, the edition, and the year of publication. Books are published and may then be acquired by the library. If a book is acquired, at least one copy is purchased. The book is entered into the library catalog. For each book, the copies are managed by the library.

For each copy, the list of borrowers up to now shall be maintained. Each copy is indicated either to be on loan or not. Additionally, for each copy we want to know when it has to be returned. Thus, copies after having been acquired can be borrowed and returned.

Users of the library are either students, who may borrow a book for at most 21 days or members of the university staff, who must return the book only if another request for the book is pending. For each user, the list of copies (s)he currently has borrowed is maintained. If a user wants to borrow a book, (s)he does not specify a particular copy but a particular book, of which (s)he wants to borrow a copy. If there are any copies available, (s)he borrows one.

Even using this rather small UoD, it is evident that a formalism for modeling the UoD has to make the integrated description of all structural and dynamic aspects possible. This has been claimed already in the conceptualization principles formulated in [vGr82].

A natural description of the UoD contains the relevant *objects* and descriptions of their *properties*, their *state*, their *behavior*, their *evolvement through time* as well as descriptions of *interactions* between objects. In the UoD described above, special relationships between objects like specializations (staff members and students are specializations of users) or roles (books in the role of books in the library) have to be modeled. Furthermore, one may easily find other concepts for relating objects like complex objects composed of several components (e.g. volumes of journals composed of numbers) or generalizations of objects (e.g. documents being either books or journals).

A major requirement for specifications of object systems is the possibility to describe the possible evolution of objects through time, i.e. the behavior of objects. In the usual definition of object-oriented data models [ABD$^+$89] or object-oriented programming languages [Weg87], this requirement is largely neglected because only explicit programming of methods to describe state transitions is possible. In most cases, it is not possible to describe the behavior through time.

At an abstract level, the behavior of a system as a whole can hardly be described locally. Thus, another requirement for UoD specification formalisms is the possibility to specify *global* assertions and interactions. During refinement, global constraints and interactions will be localized in objects.

3 A Concept of Object

In this paper, we introduce a model-oriented view of objects. The concept of object evolved from evaluating several approaches to system development towards their use for an integrated description of all relevant static and dynamic aspects of the UoD.

Arbitrarily structured values and type-specific operations have been investigated in the abstract data type approach (see e.g. [BG77, EM85, FGJM85]). However, the approach only describes values and thus does not include concepts of state, state transitions, temporal evolvement, and persistency.

In process theory, descriptions and models for concurrent systems, their behavior and temporal evolution have been investigated (see e.g. [ISO84, Hoa85, BK86, Mil89]). Most of these approaches cover solely the dynamic aspects of systems.

In contrast, semantic data models enable structured description of UoD entities and therefore only capture structural aspects of the UoD. Extensions of semantic data models allow the description of complex entities, abstractions, and structural relationships between entities (cf. [UD86, HK87, PM88]). Many of them fail, however, to provide adequate means to include state transitions and temporal evolution.

Proposals for object-oriented data models strive to overcome these deficiencies. They support object classification, the notion of state by providing immutable object identifiers, and they allow integrated description of local state information and operations on the state of an object (cf. [ABD$^+$89]). Operations alone, however, are still not suitable to describe temporal evolution of objects.

In object-oriented programming languages, some additional emphasis is put on communication between objects through message passing, encapsulation, and limited capabilities of representation of temporal evolution [Weg87, Mey88].

In our view, the most important benefit from the object-oriented approach is the encapsulated integration of local structure and behavior in objects.

In a formal description of dynamic object systems, objects are identified by names which are immutable for the lifetime of an object. The life of an object is described by a sequence of *snapshots* (the *life cycle*), where each snapshot is a set of *events* occurring simultaneously. Each life cycle starts with a snapshot including a special *birth event* describing the creation of the object and either ends with a death event in a snapshot or goes on forever. The overall behavior of an object is specified by a set of possible life cycles which is called a *process.* Properties of objects may be observed by typed *attributes.* The current attribute values depend on the internal state of an object, which itself may solely be changed by *events local to the object.* The sequence of snapshots executed by the object so far determines its local state and may depend on *interactions* with other objects. Interaction between objects is described by event occurrences which force a set of *called* events to occur simultaneously. Thus, objects in our view are *concurrent observable communicating processes.*

In the following definitions, our concept of object shall be formalized. Note that the presentation of the concept of object in this paper is simplified compared to the presentation in [EGS90, SE90].

We start with the definition of a snapshot which is an element of the powerset $\wp(X)$ of a set of events X. An empty snapshot is called a *silent event.*

Definition 3.1 *Let X be a set of events. A subset $S \in \wp(X)$ of X is called a* SNAPSHOT *over X. The events in a snapshop occur simultaneously.* □

Definition 3.2 *Let X be a set of events. Then $\wp(X)^*$ denotes the set of finite sequences of snapshots over X, and $\wp(X)^\omega$ denotes the set of infinite countable sequences of snapshots over X.* □

For objects, the set of events has to include at least one birth event and may include death events.

Definition 3.3 *An* OBJECT EVENT ALPHABET *X is a set of events where $B \subseteq X$ is a set of birth events such that $B \neq \emptyset$, $D \subset X$ is a set of death events such that $D \cap B = \emptyset$ and $U = X \setminus (B \cup D)$ is a set of update events.* □

We adopt a simple model of processes not taking into account non-deterministic behavior and parallel processes. Each non-empty life cycle is a sequence of snapshots starting with a snapshot that includes at least one birth-event. It may end by a snapshot including a death-event or go on infinitely.

Definition 3.4 *A* PROCESS *P is a tuple (X, Λ), where X is an object event alphabet and Λ is a non-empty set of life cycles over X such that*

$$\begin{aligned}\Lambda \subseteq\ & \{\epsilon\} \cup \\ & \{s_1 \tau s_n \mid (\exists b \in B : b \in s_1 \wedge s_1 \in \wp(B \cup U)),\ \tau \in \wp(U)^*, \\ & \qquad (\exists d \in D : d \in s_n \wedge s_n \in \wp(U \cup D))\} \cup \\ & \{s_1 \tau \mid (\exists b \in B : b \in s_1 \wedge s_1 \in \wp(B \cup U)),\ \tau \in \wp(U)^\omega\}\end{aligned}$$

and $\epsilon \in \Lambda$. □

The empty life cycle ϵ models an object which has not been created yet.

Definition 3.5 *The set of all finite prefixes of life cycles in Λ is denoted by $\Pi(\Lambda)$.* □

Let us now turn to attributes and attribute values. First, we define the set of all possible values of attributes.

Definition 3.6 *Let A be a set of attributes over data types* **type**(a) *for $a \in A$. Then*

$$obs(A) \subseteq \{(a,d) \mid a \in A, d \in \mathbf{type}(a)\}$$

is the set of POSSIBLE OBSERVATIONS *over A.* □

Now we define how a process may be observed using attributes and possible attribute values.

Definition 3.7 *An* OBSERVATION STRUCTURE *$V = (A, \alpha)$ over a process P is defined by a set A of attributes and an observation map*

$$\alpha : \Pi(\Lambda) \to obs(A)$$

such that $\alpha(\epsilon) = \emptyset$. □

Note that observations may be non-deterministic, i.e. α is a relation rather than a function. This is useful e.g. for attributes that have not been initialized when an object is created. We are not going to make any further investigations on non-deterministic observations.

Now, we are ready to define objects as observable processes.

Definition 3.8 *An* OBJECT MODEL *$ob = (P, V)$ is composed of a process P and an observation structure V over P.* □

Usually, objects in a representation of the UoD are classified. For this purpose, we introduce the notion of an object type.

Definition 3.9 *An* OBJECT TYPE *$ot = (I, OB)$ consists of a set I of object identifiers and a set OB of object models.* □

The set I is the carrier set of an arbitrary abstract data type. Note that the definition allows *heterogeneous types*, i.e. the intension of such a type consists of objects which not necessarily have the same model.

Object classes in our setting are sets of objects of a certain type. Thus, with each object class we associate one corresponding object type.

Definition 3.10 *An* OBJECT CLASS *$oc = (ot, O)$ consists of an object type ot and a set of instances O where $O \subseteq \{i.ob \mid i \in ot.I, ob \in ot.OB\}$.* □

Thus, an object class definition is extensional. Note that we use the "dot notation" to refer to components of structures.

For modeling an object society, the basic relationship of *object inclusion* has to be defined. For a more detailed description using a sheaf-theoretic approach see [EGS90].

First, we define the notions of restricted snapshots and life cycles. A snapshot over a set of events X restricted to a subset $Y \subset X$ is obtained deleting all events not in Y from the snapshot.

Definition 3.11 *Let S be a snapshot over a set of events X and $Y \subset X$. Then $S \downarrow Y = S \cap Y$ is the* SNAPSHOT RESTRICTION *to Y.* □

Note that snapshot restrictions can be silent events.

For life cycles, *life cycle restriction* only takes the non-empty snapshot restrictions into account, thus may not preserve the length of a life cycle. The life cycle restriction of a life cycle λ to Y is defined recursively:

Definition 3.12 *Let λ be a life cycle over X and $Y \subset X$. Then*

$$\lambda \downarrow\downarrow Y = \begin{cases} \epsilon & \text{if } \lambda = \epsilon \\ (s_1 \downarrow Y) \circ (\lambda' \downarrow\downarrow Y) & \text{if } \lambda = s_1\lambda' \text{ and } (s_1 \downarrow Y) \neq \emptyset \\ \lambda' \downarrow\downarrow Y & \text{if } \lambda = s_1\lambda' \text{ and } (s_1 \downarrow Y) = \emptyset \end{cases}$$

is the LIFE CYCLE RESTRICTION *of λ to Y.* □

Please note that for conveniance of notation we used the operator $\circ$ denoting a snapshot being a prefix of a life cycle.

Then, we introduce the notion of *event calling.* The calling relation $\gg$ is regarded to be a binary relation on a set of events. If an event x_1 calls an event x_2, then x_2 has to be in the same snapshot x_1 is in.

Definition 3.13 *Let X be a set of events, $S(X) \subseteq \wp(X)$ a set of snapshots over X and $\gg$ be a binary relation on X. Let x_1 and x_2 be events in X. Then $x_1 \gg x_2$ wrt $S(X)$ iff the following condition holds:*

$$\forall S \in S(X): \quad x_1 \in S \Rightarrow x_2 \in S$$

□

Thus, event calling is an asymmetric synchronization on event occurrences.

Now we define object inclusion, which is an encapsulated embedding of one object into another.

Definition 3.14 *Let $ob_1 = (P_1, V_1)$ and $ob_2 = (P_2, V_2)$ be object models. Then ob_2 is embedded into ob_1, $ob_2 \hookrightarrow ob_1$, iff*

$$X_2 \subseteq X_1$$

$$\forall \lambda_1 \in \Lambda_1 \exists \lambda_2 \in \Lambda_2 \colon \lambda_1 \downarrow\downarrow X_2 = \lambda_2$$

$$A_2 \subseteq A_1$$

$$\forall \tau_1 \in \Pi(\Lambda_1) \colon \alpha_1(\tau_1) \downarrow A_2 = \alpha_2(\tau_1 \downarrow\downarrow X_2)$$

□

That is, ob_1 is embedded within ob_2, all properties of ob_1 (in particular the set of life cycles of ob_1) are preserved and ob_2 events do not affect ob_1 attributes. This implies that the only way ob_2 can influence the behavior of ob_1 is by event calling as defined above. Note that all birth- and death-events of ob_1 are sent to update-events in ob_2.

For *symmetric* communication, we may identify events by defining a calling relation in both directions which is called *event sharing.*

Taking objects and object morphisms, a *category of objects* is established. We do not go into detail on that in this paper. For further details on its properties see [SE90, EGS90, FSMS90].

A category of objects represents the relevant aspects of the UoD. Since the relationships normally have an impact on the evolution of objects, we call a collection of interacting objects an *object society* (which is rather natural ...). In the next sections we will show how simple objects and object societies are specified using the **Oblog$^+$**-language.

4 Single Object Specification

A formal specification of objects includes the specification of the data types for attributes, event parameters, and identifier sorts, of attributes and events (which make up the interface of an encapsulated object), of the possible observations, of the effects of event occurrences on attribute values and of the set of possible life cycles. An object specification is a theory of which an object model introduced in the previous section is a model. In this paper, we are introducing the **Oblog$^+$** specification language rather informally using examples. A detailed investigation of specifications as theories in our setting can be found e.g. in [FSMS90, FM90] or [SS89].

A single object model is specified by an object name associated with a *template.* A template is structured as follows:

> **template**
>   **data types** *data types to be imported;*
>   **attributes** *attribute names and types;*
>   **events** *events;*
>   **constraints** *integrity constraints on states;*
>   **valuation** *effects of events on observations;*
>   *process specification;*

In the data type section, the data types to be imported for the underlying data universe are indicated. The data types may either be chosen from a set of predefined types (like 'nat' or 'string') or from a set of user-defined types specified using a specification language for abstract data type specification (e.g. ACT ONE [EM85]). Note that we also may import sets of object identifiers (also called surrogate spaces) which are defined by an abstract data type. Surrogate spaces are denoted by |OC| for an object class OC. The known type constructors (**list**, **set**, **record**, **bag**) may be applied to construct complex structured types.

The attribute and event sections of templates make up the *local signature* of object specifications. The local signature includes all attributes and events which are not inherited from embedded objects due to inclusion or embedding morphisms. The *signature* of an object specification includes the inherited attributes and events.

In the remaining part of a single template a set of logical formulae over the signature is given. A template then specifies a set of intended object models (as defined in definition 3.8) which satisfy the formulae. The set of formulae consists of three sections. In the process specification section, the set of possible life cycles Λ over the event signature is specified. The different ways to specify Λ will be explained in more detail later in this section. The set of possible observations (definition 3.6) as well as possible sequences of observations are specified in the **constraints** section using static and dynamic integrity constraints. Note that constraints are basically not necessary since observations depend on the life cycles, but they are desirable to support abstract descriptive specifications. *Valuation rules* specify the effect of event occurrences on attribute values, i.e. they link life cycles with observations and thus describe the observation map of an observation structure (definition 3.7), the second component of an object model (definition 3.8).

Single objects have a proper name and an associated template:

> **object** `object name`
>   **template** *template;*
> **end object** `object name`;

Objects normally are classified using object classes, e.g. copies of books belong to the class `COPY` in our UoD. The intension of a class is described by an object type which includes the surrogate space for identifiers and a set of templates for potential object instances (3.9). For the sake of simplicity, in this paper we only deal with homogeneous types, i.e. types with a single template only. A specification of an object type has the following structure:

> **object type** `type name`
>   **data types** *data types for surrogate space construction;*
>   **identification** *surrogate space specification;*
>   **template** *template;*

Surrogate spaces may be defined in the type specification as sets of (nested) records using a notation similar to the attribute specification and thus resemble primary keys in (nested) relational database schemas. Record selectors may be used like attributes in specifications.

An object class specification consists of a class name and an associated object type (3.10):

> **object class** `class name`
>   *type specification* or **type is** *type name;*
> **end object class** `class name`;

It is perfectly possible to have several classes with the same type. In order to keep object identifiers unique, the surrogate space of a class is an isomorphic copy of the surrogate space of the corresponding type.

Let us now consider an example. We specify the object class `BOOK`. The objects in this class describe the abstract properties of books without regarding "real" copies of it. Books in our UoD are static in the sense that their properties do not change after creation.

object class `BOOK`

```
   data types string;
   identification
      title: string;
      first_author: string;
   template
      data types nat, string, |PERSON|;
      attributes
         authors: list(|PERSON|);
         publisher: string;
         edition: nat;
         year: nat;
         derived no_authors: nat;
      events
         birth published;
         acquire;
      constraints
         static
            first_author = authors[1].name;
            year > 1500;
         derivation rules
            no_authors = length(authors);
end object class BOOK;
```

The data type associated to the surrogate space has a record structure:

|BOOK| := **record**(title: string, first_author: string).

The codomain of the attribute authors is the type **list**(|PERSON|), i.e. lists of identifiers for instances of type |PERSON|. The attribute no_authors is a derived attribute, i.e. its value depends on the values of other attributes. The derivation rules for derived attributes are specified in the constraint section.

The event published is the birth event for books, i.e. an occurrence of published brings a book object into existence. The event acquire denotes the acquisition of a book for the library. It has, however, no impact on the observable properties of books. Note that the the specification is not complete because we did not specify the initial attribute values. The initialization will be made by supplying the values as parameters of the birth-event.

As an example for specification of object dynamics consider the object class COPY which models the copies of books being in the library. Copies are identified by document numbers. Thus, the surrogate space consists of natural numbers. The constant of denotes the book of which a particular instance is a copy. Note that constants do not belong to the template since semantically they are part of the surrogate and thus may not be changed by attributes.

```
object class COPY
   data types nat, |BOOK|;
```

```
  identification
     doc_no: nat;
  constants
     of: |BOOK|;
  template
     data types bool, date, |USER|, list(|USER|), nat;
     attributes
        on_loan: bool;
        due: date;
        borrowers: list(|USER|);
     events
        birth get_copy;
        death throw_away;
        check_out(|USER|, date, nat);
        return;
     valuation
        variables U: |USER|, d: date, n: nat;
        [get_copy] on_loan = false;
        [get_copy] borrowers = emptylist();
        [check_out(U,d,n)] on_loan = true;
        [check_out(U,d,n)] due = add_days_to_date(d,n);
        [check_out(U,d,n)] borrowers = append(U,borrowers);
        [return] on_loan = false;
     safety
        variables U: |USER|, d: date, n: nat;
        {on_loan = false} check_out(U,d,n);
        {exists(U: |USER|, d: date, n: nat)
           sometime(after(check_out(U,d,n))) since last after(return)} return;
     obligations
        {exists(U: |USER|, d: date, n: nat) after(check_out(U,d,n)) } ⇒ return;
end object class COPY;
```

The attributes of the object class COPY are the boolean attribute on_loan 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 (get_copy), the lending and returning of a copy (check_out and return, respectively) and the removal of copies from the library (throw_away). The check_out-event has parameters for the borrower, the date the lending takes place, and the number of days the user is allowed to keep the copy. Event parameters allow for data to be exchanged during communication between objects and to define the effects of events on attribute values.

These effects are specified using valuation rules after the keyword **valuation**. The rules are universally quantified implicitly and invariant. A valuation rule has the following form (here simplified):

```
[event] attribute = data term;
```

Such a rule denotes that the attribute `attribute` after the occurrence of the event `event` will have the value of the term `data term`. The data term is evaluated in the state *before* the occurrence of the event and thus corresponds to an assignment in imperative programming languages. Note that we use an implicit frame rule stating that attribute values not being set by a valuation rule remain unchanged after an event has occurred. The following rule computes the return date after a book has been borrowed using a function of the data type 'date':

```
valuation
    variables U: |USER|, d: date, n: nat;
    [check_out(U,d,n)] due = add_days_to_date(d,n);
```

The set of possible life cycles Λ may be specified using several different formalisms. This is due to the fact that objects may represent a wide variety of UoD concepts which include typical passive objects like books and active objects like a calendar as well as abstract user interface objects and even objects performing query evaluation [JSS90].

Currently, **Oblog**$^+$ supports the following formalisms for process specification:

- *Safety rules* state preconditions for the occurrence of events.
- *Obligations* state completeness conditions for life cycles, i.e. events which are obliged to occur if a certain condition holds.
- A process can be specified explicitly using a CSP-like notation. We will not consider this further in this paper.

In the COPY-template we state two safety rules. The first rule simply states that a copy can only be borrowed if it is not on loan currently:

```
safety
    variables U: |USER|, d: date, n: nat;
    {on_loan = false} check_out(U,d,n);
```

The second rule states that a book can only be returned if it has been lend sometime in the past and has not been returned already:

```
safety
    {exists(U: |USER|, d: date, n: nat)
        sometime(after(check_out(U,d,n))) since last (after(return))} return;
```

Note that we refer to the history of an object. In safety rules, we thus may use a temporal logic in the past with operators like **always in the past** or **sometime ... since last ...**, which can be defined analogously to the known operators in temporal logics for the future (cf. [Ser80, Saa90]). The predicate **after** holds in each state immediately after the occurrence of the particular event.

In obligations, conditions for complete life cycles are stated. The conditions must hold before an object can be deleted. Simple obligations require events to occur, maybe depending on the current state. For copies of books we state that they have to be returned sometime after having been checked out:

```
    obligations
        {exists(U: |USER|, d: date, n: nat) after(check_out(U,d,n))) } ⇒ return;
```

After these considerations, let us take a brief look at the object model of the instances of class COPY. The event alphabet of the process component contains a family of check_out-events indexed by |USER| $\times$ date $\times$ nat. The set of life cycles is specified by the rules in the safety- and obligations-sections. In possible life cycles, an element of the family of check_out-events is only allowed to occur if an event setting the value of the attribute on_loan to **false** has occurred before. Each check_out-event occurrence has to be followed by one and only one occurrence of the event return. Associated with the prefix consisting only of the birth-event get_copy is the observation $\{(\texttt{on\_loan}, \textbf{false}), (\texttt{borrowers}, [])\}$, where [] denotes the empty list. Each prefix ending with an occurrence of return is mapped to an observation $\{(\texttt{on\_loan}, \textbf{false}), (\texttt{due}, d), (\texttt{borrowers}, b)\}$, where d and b are the values of due and borrowers in the state before the return-occurrence.

Due to the limited space, we are not able to go into more detail concerning object specifications. We now take the step from single isolated objects to object societies.

5 Object Society Specifications

Objects in a UoD may be related to each other in various ways – they may interact, they may be components of complex objects, and there may be generalization hierarchies of objects and object classes. The semantics of relationships between objects is defined using the object inclusion morphism introduced in section 3 (definition 3.14). Recall that this morphism preserves the properties of objects concerning locality of observation maps and possible life cycles. In this section, we show how relationships between objects may be specified.

The first type of relationship we consider are *roles* of objects (see also [Wie90a, Wie90b] for further considerations of the role concept). If an object plays a role, it is regarded as a temporay specialization of the base object. As a specialized object, it may have additional properties and a restricted behavior. Since it conceptually is not a new object, the surrogate space remains unchanged. A role of an object is created by the occurrence of events in the base object which are considered to be birth-events of the role.

Take for example the object class LIB_BOOK, which models books in the role of books being present in the library. The class is specified as follows:

```
object class LIB_BOOK
    view of BOOK;
    template
        data types nat;
        attributes
            no_available: nat;
        events
            birth acquire;
            dec_copies;
            inc_copies; ...
        valuation
            {no_available> 0} ⇒ [dec_copies] no_available = no_available − 1;
            [inc_copies] no_available = no_available + 1;
        ...
end object class LIB_BOOK;
```

A book starts to play the role of a library book by performing the event acquire which is the birth event of the role object. For each book in the library the number of copies currently not being on loan is noted in the attribute no_available, which is an additional property of library books only. The value of this counter is changed by the events dec_copies and inc_copies.

Formally, the signature of the BOOK-template is included in the LIB_BOOK-template, object instances of class BOOK are embedded in the corresponding instances of class LIB_BOOK using an object embedding morphism (definition 3.14), and the set of current LIB_BOOK-identifiers is included in the class of current BOOK-identifiers.

For roles, one base object is included implicitely. We may, however, specify the inclusion of objects explicitly, too. With the **inheriting**-construct in **Oblog**$^+$ we may import single objects as well as (sub-)classes of objects. Consider e.g. copies of library books to be subobjects of the corresponding library book:

```
object class LIB_BOOK
    view of BOOK;
    template
        inheriting C in COPY where C.of = SELF.id as COPIES;
        ...
```

Here, we include all those copies C of class COPY, whose constant of denotes the book itself. To support access to instances of the class of included copies, we may assign a name to it (COPIES).

The complete specification of the class LIB_BOOK is the following:

```
object class LIB_BOOK
    view of BOOK;
    template
        data types nat;
        inheriting C in COPY where C.of = SELF.id as COPIES;
        attributes
            no_available: nat;
        events
            birth acquire;
            dec_copies;
            inc_copies;
            hand_out(|USER|, |COPY|);
            request(|USER|);
            send_message(|STAFF_USER|);
        valuation
        {no_available> 0} ⇒ [dec_copies] no_available = no_available − 1;
        [inc_copies] no_available = no_available + 1;
        [aquire] no_available = 0;
    safety
        variables U: |USER|, C, C1: |COPIES|, SU: |STAFF_USER|
        {no_available > 0 and sometime(after(request(U)))} hand_out(U,C);
        {not(sometime(after(hand_out(U,C))) since last after(COPIES(C).return))}
            hand_out(U,C1);
        {no_available = 0 and
            sometime(after(request(U))) since last after(hand_out(U,C))}
                send_message(SU);
    interaction
        variables U: |USER|, C: |COPIES|, d: date, n: nat;
        calling
            COPIES(C).check_out(U,d,n) >> dec_copies;
            COPIES(C).return >> inc_copies;
            COPIES(C).get_copy >> inc_copies;
            hand_out(U,C) >> COPIES(C).check_out(U,d,n);
end object class LIB_BOOK;
```

The event hand_out models the delivery of a copy to a user, the event request represents a user request for a book. By the occurrence of the event send_message staff members are requested to return the copy they borrowed. The safety rules state that a copy of the book can only be given out to the user U if at least one copy is available and U issued a request for the book. Furthermore, a user is only allowed to borrow one copy of a particular book at a time. Messages to borrowers can only be sent if there is a request but no copy currently is available.

Note that interactions between an instance of type LIB_BOOK and the embedded instances of type COPY can only take place by event calling or event sharing. Here, we use event calling to update the no_available-counter everytime a copy is borrowed or returned:

```
    interaction
        variables U: |USER|, C: |COPIES|, d: date, n: nat;
        calling
            COPIES(C).check_out(U,d,n) >> dec_copies;
            COPIES(C).return >> inc_copies;
            COPIES(C).get_copy >> inc_copies;
```

If a copy is checked out, the counter has to be decremented, if a copy is returned, the counter is incremented. Since locality has to be preserved, an event of an object of class COPY is not allowed to directly have an effect on attributes of a LIB_BOOK-instance. Note that if we defined interactions using *event sharing* (i.e. calling in both directions), each time an event dec_copies of a LIB_BOOK-instance occurred in each included instance of class COPY a corresponding check_out-event would have occurred (which not exactly is how it should work ...).

Formally, those instances of class COPY whose constant of is set to the identifier of an instance of class LIB_BOOK are embedded in the corresponding LIB_BOOK-instance using object inclusion morphisms (definition 3.14).

To complete the conceptual model of the UoD, we specify the object class USER:

```
object class USER
   data types string;
   identification
      uid: string;
   constants
      copies_allowed: {3, 10};
   template
      data types |COPY|, date, nat, |LIB_BOOK|;
      attributes
         borrowed: set(|COPY|);
      events
         birth subscribe;
         death unsubscribe;
         borrow(|COPY|, date, nat);
         active return(|COPY|);
         active request(|LIB_BOOK|);
      valuation
         variables C: |COPY|, d: date, n: nat;
         [borrow(C,d,n)] borrowed = insert(C, borrowed);
         [return(C)] borrowed = remove(C, borrowed);
      safety
         variables C: |COPY|, d: date, n: nat;
         {not(in(C, borrowed)) and card(borrowed) < copies_allowed} borrow(C,d,n);
         {sometime(after(borrow(C,d,n)))} return(C);
         {empty(borrowed)} unsubscribe;
      initiatives
         variables C: |COPY|, d: date, n: nat;
         {after(borrow(C,d,n))} ⇒ return(C);
end object class USER;
```

Objects of class USER are active in the sense that they may *perform* events rather than only *suffer* events. The optional keyword **active** denotes that the occurrence of such events can be forced by the object itself. In the initiative-section, we may state goals which in contrast to liveness requirements are not obligatory to occur. Here, users may return copies on their own initiative once they borrowed that particular copy (which in reality is not always the case, though ...).

Depending on the constant copies_allowed we now define two specializations of user objects. In contrast to roles, specializations are a static concept, i.e. they depend on static properties of base objects.

Students are those users who may borrow at most three copies at a time. Moreover, they have to return a copy after 21 days, which is specified using an additional safety rule:

```
object class STUDENT_USER
    specializing from USER where copies_allowed = 3;
    template
        safety
            variables C: |COPY|, d: date, n: nat;
            { n ≤ 21 } borrow(C,d,n);
end object class STUDENT_USER;
```

Staff members do not have any time limit for returning book. They have, however, to return a copy on request (get_message):

```
object class STAFF_USER
    specializing from USER where copies_allowed = 10;
    template
        data types |LIB_BOOK|;
        events
            get_message(|LIB_BOOK|);
        obligations
            variables LB: |LIB_BOOK|, C: |COPY|;
            { after(get_message(LB))
                and in(C,borrowed) and C.of = LB }  ⇒  return(C);
end object class STAFF_USER;
```

Another relationship between object classes are *generalizations* of object classes. We observe that object instances with different models may belong to the generalized class. Thus, the associated type usually is a heterogeneous type with more than one element in the set of object models (definition 3.9). We do not consider generalizations further in this paper.

Additionally, we may specify *object interfaces* to explicitly provide particular views on objects and object classes [SJ90].

An object society specification has the following structure:

object society SocietyName
    data type specification;
    [*template and type specifications*;]
    object and object class specifications;
    global interaction *communication between autonomous objects*;
    global constraints *global integrity constraints*;
end object society SocietyName.

In the global part, we may state integrity constraints concerning several objects or object classes and communications between objects which are not related. Although this may be regarded as violation of locality, we consider it to be essential for readable and understandable abstract specifications. During refinement of specifications, global communication and global constraints will be localized in objects. As an example for global communication, we model the request for a library book issued by a user, the lending of a copy, and the sending of messages to users:

global interaction

```
  variables C: |COPY|, d: date, n: nat, U: |USER|, SU: |STAFF_USER|; LB: |LIB_BOOK|;
     USER(U).request(LB) >> LIB_BOOK(LB).request(U);
     COPY(C).check_out(U,d,n) >> USER(U).borrow(C,d,n);
     LIB_BOOK(LB).send_message(SU) >> STAFF_USER(SU).get_message(LB);
```

The concept of object described in section 3 does not support global constraints and interactions. Therefore, these formulae describe properties of a complex object being the aggregation of the related objects.

6 Discussion and Conclusions

In this paper, we introduced a formal concept of object. Objects are regarded to be dynamic entities encapsulating structure and dynamics which are able to communicate, i.e. we put strong emphasis on the specification of dynamic evolvement of systems. As a semantical basis for relationships between objects, we introduced the notions of event calling and morphisms between objects. Based on this semantical model, we proposed a formal language **Oblog$^+$** which supports the object-oriented specification of information systems. The language enables the integrated specification of object structure and behavior as well as the specification of object societies on an abstract level.

A lot of work towards UoD-modeling has been done in the past. One thread is the knowledge-based approach discussed e.g. in [BM86, ST89]. The approach stresses the idea that UoD modeling is in fact modeling of knowledge about the UoD. The results produced in this branch are concerned with ontological (concepts for representing UoD aspects) and epistomological (building blocks and structuring mechanisms for UoD models) aspects. An example for such a language is RML [GBM86].

Another thread emerges from the database community. The languages proposed aim at designing the implementation of an information system as the result of a requirements analysis. Examples for such proposals are Taxis [MBW80, Nix84], SDM [HM81], IFO [AH87], and the (variants of) the ER-model [Che76, HG88] (for a survey see [UD86, HK87, PM88]).

Characteristics of the two threads mentioned above are:

- Object/Entity descriptions are separated from dynamics descriptions (if dynamics can be described at all) and even constraints are separated from entity descriptions (in Taxis and RML). Thus, the view of objects being dynamic entities is not supported.

- The proposed formalisms cannot be applied to different levels of abstraction during the development process. This becomes evident e.g. in the DAIDA project [BJM$^+$89].

- The approaches do not support the explicit modeling of communication and cooperation between concurrent objects.

A third thread is concerned with tailoring programming languages with concepts for semantic modeling. Examples are Galileo [ACO85] and ADAPLEX [SFL83]. Their reliance on programming langugages makes them directed towards implementation and thus unsuitable for conceptual modeling.

Among the object-oriented approaches, only a few suited for UoD-modeling exist. The language FOOPS [GM87] evolved from integration the specification of abstract data types with object-oriented concepts. Due to the lack of process and role concepts, the emphasis is more on the description of static structure and state transition.

The language CMSL [Wie90a, Wie90b] is a language especially for formal specification of object-oriented conceptual models. In CMSL, objects are (like in **Oblog**$^+$) dynamic entities whose life is described by a process. Similar to our approach, the UoD is represented by a collection of interacting dynamic objects. The capabilities of **Oblog**$^+$ and CMSL are roughly the same.

A graphical notation based on a semantic data model (for describing the static aspects) and Petri Nets (for object dynamics) are Object/Behavior Diagrams [KS90]. In subsequent steps, the object types, the admitted life cycles for instances of object types and interactions between objects are modeled making the rationale of this approach similar to **Oblog**$^+$.

Further research issues on **Oblog**$^+$ concern the refinement and implementation of specifications. The idea is to define a set of transformations on theories to obtain an equivalent specification involving only simply structured objects. This specification may then be implemented using techniques for object reification as presented in [ES89, SE90].

Acknowledgements

Thanks to all IS-CORE colleagues, who contributed to the definition of the **Oblog**$^+$ specification language. In particular, Hans-Dieter Ehrich and Amílcar Sernadas developed basic ideas of objects and object descriptions.

References

[ABD$^+$89] Atkinson, M.; Bancilhon, F.; DeWitt, D.; Dittrich, K. R.; Maier, D.; Zdonik, S. B.: The Object-Oriented Database System Manifesto. In: Kim, W.; Nicolas, J.-M.; Nishio, S. (eds.): *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, Kyoto, Japan, December 1989. pp. 40–57.

[ACO85] Albano, A.; Cardelli, L.; Orsini, R.: Galileo: A Strongly Typed Interactive Conceptual Language. *ACM Transactions on Database Systems*, Vol. 10, 1985, pp. 230–260.

[AH87] Abiteboul, S.; Hull, R.: IFO – A Formal Semantic Database Model. *ACM Transactions on Database Systems*, Vol. 12, No. 4, 1987, pp. 525–565.

[B$^+$85] Bauer, F. L. et al.: *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, Berlin, 1985.

[Bal81] Balzer, R.: Final Report on GIST. Technical report, USC/ISI, Marina del Rey, CA, 1981.

[BG77] Burstall, R.; Goguen, J. A.: Putting Theories Together to Make Specifications. In: *Proc. 5th Int. Joint Conf. on Artificial Intelligence IJCAI'77*, Cambridge, MA, 1977. pp. 1045–1058.

[BJM+89] Borgida, A.; Jarke, M.; Mylopoulos, J.; Schmidt, J. W.; Vassiliou, Y.: The Software Development Environment as a Knowledge Base Management System. In: Schmidt, J. W.; Thanos, C. (eds.): *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989, pp. 411–439.

[BK86] Bergstra, J. A.; Klop, J. W.: Algebra of Communicating Processes. In: de Bakker, J. W.; Hazewinkel, M.; Lenstra, J. K. (eds.): *Mathematics and Computer Science*, CWI Monographs 1. North-Holland, Amsterdam, 1986, pp. 89–138.

[BM86] Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.

[BMS84] Brodie, M. L.; Mylopoulos, J.; Schmidt, J. W.: *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, Berlin, 1984.

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

[Che76] Chen, P.P.: The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, pp. 9–36.

[CHJ86] Cohen, B.; Harwood, W. T.; Jackson, M. I.: *The Specification of Complex Systems.* Addison-Wesley, Reading, MA, 1986.

[DeM79] DeMarco, T.: *Structured Analysis and System Specification.* Prentice-Hall, Englewood cliffs, NJ, 1979.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: *Proc. REX/FOOL Workshop*, Noordwijkerhood (NL), 1990. *To appear.*

[ELG89] Ehrich, H.-D.; Lipeck, U. W.; Gogolla, M.: *Algebraische Spezifikation abstrakter Datentypen.* Teubner Verlag, Stuttgart, 1989.

[EM85] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[ES89] Ehrich, H.-D.; Sernadas, A.: Algebraic Implementation of Objects over Objects. In: deRoever, W. (ed.): *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mood (NL), 1989. LNCS 394, Springer Verlag, Berlin, 1989, pp. 239–266.

[FGJM85] Futatsugi, K.; Goguen, J. A.; Jouannaud, J.-P.; Meseguer, J.: Principles of OBJ2. In: *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 1985. pp. 52–66.

[FM90] Fiadeiro, J.; Maibaum, T.: Describing, Structuring and Implementing Objects. In: *REX90: Foundations of Object-Oriented Languages*, Noordwijkerhood (NL), 1990. Springer-Verlag, Berlin. *To appear.*

[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.*

[GBM86] Greenspan, S.; Borgida, A. T.; Mylopoulos, J.: A Requirements Modelling Language and its Logic. In: Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Base Management Systems.* Springer-Verlag, Berlin, 1986, pp. 471–502.

[GM87] Goguen, J. A.; Meseguer, J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In: Shriver, B.; Wegner, P. (eds.): *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 417–477.

[vGr82] van Griethuysen, J.: Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[Hen88] Hennessy, M.: *Algebraic Theory of Processes.* The MIT Press, Cambridge, MA, 1988.

[HG88] Hohenstein, U.; Gogolla, M.: A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In: *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, Rome, 1988. North-Holland, Amsterdam, 1988.

[HK87] Hull, R.; King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol. 19, No. 3, 1987, pp. 201–260.

[HM81] Hammer, M. M.; McLeod, D. J.: Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, Vol. 6, No. 3, 1981, pp. 351–381.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[ISO84] ISO: Information Processing Systems, Definition of the Temporal Ordering Specification Language LOTOS. Report N1987, ISO/TC97/16, 1984.

[Jac83] Jackson, M. A.: *System Development.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[JSS90] Jungclaus, R.; Saake, G.; Sernadas, C.: Using Active Objects for Query Processing. 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.*

[KM90] Korson, T.; McGregor, J. D.: Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM*, Vol. 33, No. 9, 1990, pp. 40–60.

[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.*

[LF82] London, P.; Feather, M.: Implementing Specification Freedoms. *Science of Computer Programming*, Vol. 2, 1982, pp. 91–131.

[MBW80] Mylopoulos, J.; Bernstein, P. A.; Wong, H. K. T.: A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, Vol. 5, No. 2, 1980, pp. 185–207.

[Mey88] Meyer, B.: *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mil89] Milner, R.: *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.

[Nix84] Nixon, B. (ed.): TAXIS'84: Selected Papers. Technical Report CSRG-160, Dept. of CS, U. of Toronto, 1984.

[Par90] Partsch, H. A.: *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, Berlin, 1990.

[PM88] Peckham, J.; Maryanski, F.: Semantic Data Models. *ACM Computing Surveys*, Vol. 20, No. 3, 1988, pp. 153–189.

[Saa90] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 5, 1990. *In print.*

[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.*

[Ser80] Sernadas, A.: Temporal Aspects of Logical Procedure Definition. *Information Systems*, Vol. 5, 1980, pp. 167–187.

[SFL83] Smith, J. M.; Fox, S. A.; Landers, T.: ADAPLEX: Rationale and Reference Manual. Technical Report CCA-83-08, Computer Corp. of America, Cambridge, MA, 1983.

[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.

[SS85] Sernadas, A.; Sernadas, C.: Capturing Knowledge about the Organization Dynamics. In: Methlie, L.; Sprague, R. (eds.): *Knowledge Representation for Decision Support Systems*. North-Holland, Amsterdam, 1985, pp. 255–267.

[SS89] Sernadas, C.; Saake, G.: Formal Semantics of Object-Oriented Languages for Conceptual Modelling. IS-CORE Report, INESC, Lisbon, 1989.

[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.

[ST89] Schmidt, J. W.; Thanos, C. (eds.): *Foundations of Knowledge Base Management*. Springer-Verlag, Berlin, 1989.

[UD86] Urban, S. D.; Delcambre, L.: An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In: *Proc. Int. Conf. on Data Engineering*, Los Angeles, 1986. ACM, New York, 1986, pp. 382–387.

[Weg87] Wegner, P.: Dimensions of Object-Based Language Design. In: *OOPSLA Conference Proceedings*, Orlando, FL, 1987. ACM, New York, 1987, pp. 168–182. (Special Issue of SIGPLAN Notices, Vol. 22, No. 12, November 1987).

[Wie90a] Wieringa, R. J.: *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[Wie90b] 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.*

[YZ80] Yeh, R. T.; Zave, P.: Specifying Software Requirements. *Proc. IEEE*, Vol. 68, No. 9, 1980, pp. 1077–1085.

[Zav79] Zave, P.: A Comprehensive Approach to Requirements Problems. In: *Proc. COMPSAC'79*, Chicago, IL, 1979. pp. 117–127.