Language Features for Object-Oriented Conceptual Modeling*†

Ralf Jungclaus
Gunter Saake
Thorsten Hartmann

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

Abstract

In this paper, an approach to formally specify a conceptual model of a system and its environment to be developed in a behavior-oriented way is presented. We identify those concepts of object-oriented approaches that have to be formalized. We then propose characteristics of behavior-oriented mathematical models representing a system as a collection of interacting objects. To specify such models, we present the language TROLL which is a formal object-oriented language for conceptual modeling. TROLL avoids separate description of static and dynamic aspects of the system to be developed. Features of the language are explained using an example of a library information system.

1 Introduction

A large number of software systems can be characterized as information systems. Information systems are capable of representing, storing, processing and producing information (i.e. data *and* knowledge) about a particular application domain, the Universe of Discourse (UoD). The early phases of information system development are concerned with the abstract description of the UoD and the information system. 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

*In: Proc. 10th Int. Conf. on the ER-Approach, San Mateo (CA), October 23–25, 1991

†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 and Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1 and Sa 465/1-2.

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].

Currently, problems in those phases arise from using different notations for describing structure and behavior on the same level of abstraction. The structure of data and information usually is described using a semantic data model (see [HK87, PM88] for a survey), which is, however, not capable to represent behavioral aspects. In contrast, notations like Structured Design and Structured Analysis [DeM79] or JSD [Jac83] describe the functionality of a system and are thus suitable to describe behavioral aspects.

Some language proposals for conceptual modeling 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.

Since information systems are increasingly used for applications where safety and reliability are of great importance (such as aircraft control), *formal languages* to describe *formal methods* are essential.

There are several demands on a formal *conceptual modeling language*:

- In this early phase of system design, the developer (or analysist) has to describe *what* are the problems the system to be developed should handle and not *how* this functionality shall be realized. An important feature of a specification is therefore *abstraction from implementation.*

- A specification formalizes the reactions and functionality of the developed systems in terms of a mathematical semantics. The *mathematical structures* used as semantic models of a specification should intuitively reflect the architecture of the specified system. Hence, they support both verification of implementations and intuitive understanding of the specification by customers and system developers.

- Additionally, a desirable feature of conceptual modeling languages is a *proof theory for reasoning about specifications* based on a suitable logical framework. A proof theory is necessary for formal consistency checks and verification of refinement steps.

- Other relevant features are *prototyping*, *modularization of design documents* and *support for incremental design.*

The approach described in this paper evolved from integrating work on formal program specification techniques like algebraic specification of abstract data types [BG77, EM85, EGL89], 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 textual 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 [FSMS90, FM90]. We now present the language TROLL (Textual Representation of an Object Logic Language) for object-oriented conceptual model specification.

In the next section, we give a motivation why an object oriented approach is used. Then, we sketch a formalization of an object and the basic relation between objects. In the main chapter, we introduce basic features of the object-oriented conceptual modeling language TROLL. Finally, we relate our approach to other selected ones and give some concluding remarks.

2 Motivation for an Object-Oriented Approach

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 [GR83, Mey88] and object-oriented databases [ABD$^+$89]. 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 in having 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 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 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 embedding morphisms (which in fact is semantic object inheritance).

- 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, we will characterize how models of such object systems are characterized and how the specification of a model is supported by the TROLL-language.

3 Formalization of Object-Oriented Concepts

In this section, we want to sketch how a model (in the mathematical sense) representing a system as a collection of interacting objects is characterized. In our approach, emphasis is put on the representation of the behavior of objects through time and of their evolvement through time depending on their behavior resulting in a complete representation of all structural and behavioral aspects of an object.

Let us first outline the charateristics of a mathematical model of single isolated objects (called an *object model* in the sequel).

An object is in a certain state at any moment in time. A state transition corresponds to an occurrence of an *event* (which is an abstraction of a method). A set of events may occur simultaneously, which is said to be a snapshot. A sequence of events starting from creating the object (*birth event*) and either ending with deleting the object (*death event*) or going on "forever" is called a *life cycle*. The possible behavior of an object thus is modeled by a set of admissible life cycles which is called a *process*. The object's state may be observed using *attributes* that represent relevant properties of an object. The values of the attributes (an *observation*) depend on the object's state which is fully determined by the object's history, i.e. it depends on the sequence of state transitions occurred so far. The history is a finite prefix of an admissible life cycle.

Let us now sketch the formalization of these characteristics. Let X be a set of events including at least one birth event. Let $\mathcal{L}_X$ be the set of all life cycles over X, i.e. those sequences of snapshots that start with a snapshot including at least one birth event and (in case they are finite) that end with a snapshot including at least one death event plus the empty sequence denoted by ϵ.

Definition 3.1 *A* PROCESS *P is a pair (X, Λ) where X is a set of events and $\Lambda \subseteq \mathcal{L}_X$ is the set of permitted life cycles over X such that $\epsilon \in \Lambda$.* ■

Such a process is the behavioral component of an object model, determining it's implicit state. The empty life cycle ϵ describes the fact that the object has not been created. We now formalize how a process determines the observations. Let A be a set of attributes. Associated with each attribute $a \in A$ is a data type $\mathtt{type}(a)$ that determines the range of the attribute. Let $\Pi(\Lambda)$ be the set of all finite prefixes of life cycles in Λ and $\mathtt{carrier}(\mathtt{type}(a))$ the carrier set for the data type $\mathtt{type}(a)$.

Definition 3.2 *An* OBSERVATION STRUCTURE *V over a process P is a pair (A, α) where A is a set of attributes and α is an observation mapping $\alpha\colon \Pi(\Lambda) \to obs(A) \subseteq \{(a, d) \mid a \in A \wedge d \in \mathtt{carrier}(\mathtt{type}(a))\}$. A given prefix $\pi \in \Pi(\lambda)$ is called* (POSSIBLE) STATE *of an object.* ■

Note that by definition we allow for non-deterministic observations.

An object model is defined by putting together a process and an observation structure over that process yielding a complete description of possible behavior and evolution through time.

Definition 3.3 *An* OBJECT MODEL *$ob = (P, V)$ is composed of a process P and an observation structure V over P.* ■

Thus, an object is an *observable process*. As a notational convention, the sequence of observations during the execution of a life cycle $\lambda \epsilon \Lambda$ is denoted by $\omega \epsilon \Omega$, the set of all possible sequences of observations.

Recall that a system is represented by a collection of interacting objects. Thus, we need a mechanism to relate object models. This mechanism is an *object morphism* [EGS90, ESS90]. A special case of object morphism is the *object embedding morphism*. It describes an embedding of an object ob_2 as an encapsulated entity into another object ob_1.

This implies that the set of life cycles of ob_2 must be preserved and ob_1-events must employ ob_2-events to alter ob_2's state. The latter requirement is captured by the concept of *calling*. If an event e_1 calls an event e_2, then whenever e_1 occurs, e_2 occurs simultaneously.

Let us now define an object embedding morphism.

Definition 3.4 *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*

$$X_2 \subseteq X_1$$

$$\forall \lambda_1 \epsilon \Lambda_1 \exists \lambda_2 \epsilon \Lambda_2 : \lambda_1 \downarrow\downarrow X_2 = \lambda_2$$

$$A_2 \subseteq A_1$$

$$\forall \tau_1 \epsilon \Pi(\Lambda_1) : \alpha_1(\tau_1) \downarrow A_2 = \alpha_2(\tau_1 \downarrow\downarrow X_2)$$

■

$\lambda_1 \downarrow\downarrow X_2$ denotes life cycle restriction. All events that are not in X_2 are eliminated from the snapshots in λ_1 and only the sequence of non-empty snapshots is taken into account. A single arrow on observations like in $\alpha_1 \downarrow A_2$ only takes into account those attributes that are in A_2.

Usually, objects are classified. For this purpose, we introduce the notion of an class type.

Definition 3.5 *A* Class Type *$ct = (I, ob)$ consists of a set I of object identifiers and a (prototype) object model ob.* ■

The set I is the carrier set of an arbitrary abstract data type. Following this definition, each object of a class has the same object model as structure definition. This definition can be generalized to heterogeneous classes as proposed in [SSE87] and [JSS91].

Object classes in our setting are sets of objects of a certain type fixed by the corresponding class type. Thus, with each object class we associate one corresponding class type. A class type describes possible extensions of a class definition in terms of object sets identified by identifier values $i \epsilon I$. Therefore, possible class populations are subsets of $\{\langle i, ob \rangle \mid i \epsilon ct.I \wedge ob = ct.ob\}$. Usually, the term *object class* is connected with the current state of a class variable, i.e. a concrete current object population with current states for included objects.

Definition 3.6 *A state of an* Object Class *$oc = (ct, O)$ consists of an class type ct and a set of instances O. Instances $\langle i, ob, \pi \rangle \epsilon O$ are described by an object identifier i, an object model $ob = ct.ob$ and a current state $\pi \epsilon \Pi(ob.\Lambda)$. The identifier i identifies instances in O uniquely.* ■

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

Taking object models as objects and object morphisms as arrows, a category of objects is established. Models representing the system to be developed can be constructed using category-theoretic operations. For details see [EGS90].

4 Object-Oriented Conceptual Modeling in TROLL

In this section we present the language TROLL to specify a model of the system to be developed of which the characteristics have been described in the previous section. That is, the intended models of TROLL-specifications are societies of objects that are connected by object morphisms. TROLL is based on the OBLOG language proposal presented in [SSE87].

Let us first consider the specification of single isolated objects. The structure and behavior of an object is specified in a *template* which is not the object itself. As an example, we want to specify a template describing an object representing a book:

```
template Book
   data types nat, string, |PERSON|, list(|PERSON|);
   attributes
      Authors: list(|PERSON|);
      Publisher: string;
      Edition: nat;
      Year: nat;
      derived No_authors: nat;
   events
      birth published;
   constraints
      Year > 1500;
   derivation
      No_authors = length(Authors);
```

In the **data types**-section, the signatures of the data types being used in the template specification are imported. The data types may either be chosen from a set of predefined types (like 'nat' or 'string') or from a set of user-defined data types that were specified using a specification language for abstract data types. Since object identifiers are values of a certain data type (cf. section 3), these data types may also be imported and are denoted by |OC| for an object class OC. Complex structured data types may be constructed using the type constructors **list**, **set**, **tuple** and **bag**. In the example at hand, the predefined data types 'nat' and 'string' are imported.

Attributes and events make up the *local signature* of a template. The signature of a template includes the data type signatures, the signatures of embedded objects and the local signature defined for the template and is the alphabet underlying the specification of objects.

In the Book-template, there are attributes Authors, Publisher, Edition, Year and No_authors which represent the relevant observable properties of books. Note that the codomain of the attribute `Authors` is the type **list**(|`PERSON`|), i.e. lists of identifiers of instances of class `PERSON`. The attribute `No_authors` is a derived attribute, i.e. its value depends on the values of other attributes. The rules for deriving the value of derived attributes can be found in the **derivation**-section of a template specification. Here, the value is the length of the list of authors and it is computed using an operation defined for the data type constructor **list**.

The only event in the `Book`-template is the event `published`, which is the birth-event for objects being described by the `Book`-template, i.e. an occurrence of `published` brings such an object into existence. Note that we specify books to be static – their properties do not change over time. In a fully worked-out example the initialization of the attribute values is made by supplying the values as parameters of the birth-event.

In the **constraints**-section, static and dynamic integrity constraints may be given. Static integrity constraints are formulae given in first-order logic that specify the permitted observations. For books, we state that the year of publication must always be greater than 1500 (which means we are not interested in older books). Dynamic integrity constraints state how the values of attributes may evolve over time. A popular dynamic integrity constraint (for employees) e.g. states that their salary may not decrease in the future. Dynamic integrity constraints are formulae given in a temporal logic [Ser80, Saa91].

We may use the `Book`-template to specify objects, class types and classes. Single objects have a proper name and an associated template:

object `Gone_with_the_wind`
  template `Book`
end object `Gone_with_the_Wind`;

The class of books is specified by the following specification:

object class `BOOK`
  identification
    data types string, |`PERSON`|;
      `Title`: string;
      `FirstAuthor`: |`PERSON`|;
  template `Book`
end object class `BOOK`

Note that we implicitly specify the class type `BOOK` with the specification of the object class `BOOK`. A class type is defined by a template along with a set of identifiers which is the carrier set of a data type. As a convention, we may define identification types as sets of (nested) tuples using a notation similar to the attribute specification. Thus, the data type associated with the type `BOOK` is the following:

|`BOOK`| := **tuple**(`Title`: string, `FirstAuthor`: |`PERSON`|).

Let us now look at a specification of objects that evolve over time. Consider the object class `COPY` that models copies of books being in the library. Copies are identified by document numbers. The constant `Of` denotes the identification of the book of which a particular instance is a copy.

class `COPY`

```
identification
    data types nat, |BOOK|;
        Doc_no: nat;
constants
        Of: |BOOK|;
template
    data types bool, date, |USER|, list(|USER|);
    attributes
        On_loan: bool;
        Due: date;
        Borrowers: list(|USER|);
    events
        birth get_copy;
        death throw_away;
        check_out(|USER|,date);
        return;
    valuation
        variables U:|USER|, d:date;
        [ get_copy ] On_loan = false;
        [ get_copy ] Borrowers = emptylist();
        [ check_out(U,d) ] On_loan = true;
        [ check_out(U,d) ] Due = add_days_to_date(d,21);
        [ check_out(U,d) ] Borrowers = append(U,Borrowers);
        [ return ] On_loan = false;
    behavior
        permissions
            variables U:|USER|, d:date;
            { On_loan = false } check_out(U,d);
            { ((exists U:|USER|, d:date)
                sometime(after(check_out(U,d)))since last(after(return))) } return;
        obligations
            { ((exists U:|USER|, d:date) after(check_out(U,d))) } ⇒ return;
end class COPY
```

The attributes of the template 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 and the date the lending takes place. 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 implicitly universally quantified 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;
> [`check_out(U,d)`] `Due` = **add_days_to_date**(`d,21`);

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, TROLL supports the following formalisms for process specification:

- *Permissions* 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:

> **permissions**
> **variables** `U`:|`USER`|, `d`:date;
> { `On_loan` = **false** } `check_out(U,d)`;

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:

> { ((**exists** `U`:|`USER`|, `d`:date)
> **sometime**(**after**(`check_out(U,d)`))**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, Saa91]). The predicate **after** holds in each state after the occurrence of the particular event.

In obligations, requirements for complete life cycles are stated. The requirements must be fulfilled 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) after(check_out(U,d))) } ⇒ return;
```

Objects in a system 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 first type of relationship we consider are *roles* of objects (see also [Wie90] for further considerations of the role concept). If an object plays a role, it is regarded as a *temporary 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 identifier 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:

```
class LIB_BOOK
  role of |BOOK|;
  template
    data types nat;
    attributes
      No_available: nat;
    events
      birth acquire(nat);
      death throw_away;
      get_new_copy;
      copy_is_checked_out;
      copy_is_returned;
    valuation
      variables N:nat;
      [ acquire(N) ] No_available = N;
      [ get_new_copy ] No_available = No_available + 1;
      { No_available > 0 } ⇒
         [ copy_is_checked_out ] No_available = No_available − 1;
      [ copy_is_returned ] No_available = No_available + 1;
end 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 copy_is_checked_out and copy_is_returned.

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 and the set of current LIB_BOOK-identifiers is included in the class of current BOOK-identifiers.

For the sake of completeness, let us give the specification for library users with only a few comments on it. Users are regarded as roles of persons, which are assumed to have the usual properties. The marking **active** of certain events denotes that they may be initiated whenever possible during the life of an instance by the instance itself.

```
class USER
    role of |PERSON|;
    template
        data types |COPY|, |LIB_BOOK|, set(|COPY|);
        attributes
            Borrowed: set(|COPY|);
        events
            birth subscribe;
            death unsubscribe;
            active return(|COPY|);
            active request(|LIB_BOOK|);
            active borrow(|COPY|);
        valuation
            variables C:|COPY|;
            [ subscribe ] Borrowed = emptyset();
            [ borrow(C) ] Borrowed = insert(C,Borrowed);
            [ return(C) ] Borrowed = remove(C,Borrowed);
        behavior
            permissions
                variables C:|COPY|;
                { not in(C,Borrowed) } borrow(C);
                { sometime(after(borrow(C))) } return(C);
                { empty(Borrowed) } unsubscribe;
    end class USER
```

Last but not least let us consider the library. We specify the library to be a complex object with the books in the library, the copies and the users being components of it. In our example, we have a component `Catalog` being a set of library books, a component `Depot` being the set of copies of books and a component `Users` being the set of users having subscribed. An additional component `Calendar` is used for accessing the `SYSTEM_CLOCK` which is not explicitely specified in this example. If a complex object has components, a number of additional properties and events are added implicitely to the signature. For set components, a parameterized boolean attribute for set membership is included, e.g. for the component `Catalog` an attribute `Catalog.`**IN**(|`LIB_BOOK`|):bool is defined. To manipulate set components, events to insert and delete elements are provided, e.g. for the component `Catalog` the events `Catalog.`**INSERT**(|`LIB_BOOK`|) and `Catalog.`**DELETE**(|`LIB_BOOK`|) are defined.

A crucial issue concerning complex objects are interactions between the components and the complex object and between components. Let us first give the example specification for a library object:

```
object Library
    template
        data types date, |LIB_BOOK|, |USER|, |COPY|;
        components
```

```
        Catalog:SET(LIB_BOOK);
        Depot:SET(COPY);
        Users:SET(USER);
        Calendar: SYSTEM_CLOCK;
    events
        birth open;
        death close;
        user_request(|USER|,|LIB_BOOK|);
        check_out(|COPY|,|USER|,date);
        return(|USER|,|COPY|);
    behavior
        permissions
            variables C:|COPY|, B:|LIB_BOOK|, U:|USER|, d:date;
            { Catalog.IN(B) } user_request(U,B);
            { sometime(after(user_request(U,B))) and C.of = B } check_out(C,U,d);
            { sometime(after(check_out(C,U,d)))since last(after(return(U,C))) } return(U,C);
    interaction
        variables C:|COPY|, B:|LIB_BOOK|, U:|USER|, d:date;
        { Users.IN(U) } ⇒ USER(U).request(B) ≫ user_request(U,B);
        { LIB_BOOK(B).No_available > 0 and (choose C:(Depot.IN(C) and COPY(C).of = B and
              COPY(C).On_loan = false)) } ⇒
                 user_request(U,B) ≫ check_out(C,U,Calendar.current_date);
        check_out(C,U,d) ≫ COPY(C).check_out(U,d);
        check_out(C,U,d) ≫ LIB_BOOK(C.of).copy_is_checked_out;
        USER(U).return(C) ≫ return(U,C);
        return(U,C) ≫ COPY(C).return;
        return(U,C) ≫ LIB_BOOK(COPY(C).of).copy_is_returned;
end object Library;
```

In the interaction-section, we have a number of calling-clauses. The first one,

```
{ Users.IN(U) } ⇒ USER(U).request(B) ≫ user_request(U,B);
```

specifies that if a user identified by `U` is a user of the library, then a `request`-event for a book `B` in that particular component object of the library implies the occurrence of an event `user_request(U,B)` in the complex object `Library`. The second calling-clause is slightly more complex:

```
{ LIB_BOOK(B).No_available > 0 and (choose C:(Depot.IN(C) and COPY(C).Of = B and
          COPY(C).On_loan = false)) } ⇒
             user_request(U,B) ≫ check_out(C,U,Calendar.current_date);
```

It states that only if a copy of the required book `B` is available and a nondeterministically chosen copy `C` of `B` is in the `Depot` and not on loan currently, then a user request for book `B` implies that this particular copy is checked out. The check out date is obtained from the `Calendar`-component, assuming that it has an attribute `current_date`, using the "dot notation". Please note that a formula quantified by **choose** evaluates to true if there exists a copy `C` for which the condition is true. As a side effect, the **choose** quantor nondeterministically chooses one of the values yielding true and binds it to the external variable `C`. This value is used in the calling clause.

If a `check_out`-event for a copy `C` occurs in the complex object `library` then the `check_out`-event occurs in the component object identified by `C` and in the corresponding `LIB_BOOK`-object of which `C` is a copy the event `copy_is_checked_out` is called. If a user returns a copy, then the last three calling clauses apply in an analogous way.

As an additional feature, in TROLL we may specify *object interfaces* to explicitly provide particular views on objects and object classes [SJ91]. Views in TROLL resemble the well-known views in relational databases. Views are very helpful for shared object bases (external level) and for information hiding during implementation of objects [ES89].

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. In our example society, whenever a `USER`-instance, a `COPY`-instance, or a `LIB_BOOK`-instance is created, this particular instance must become a member of the corresponding set of `Library`-components. Since conceptionally, at creation time the components-to-be are not included in the `Library`-object, this has to be done by global communication: The birth-events of the "components-to-be" call the manipulation-events in the `Library`-object.

global interactions
variables U:|USER|, C:|COPY|, L:|LIB_BOOK|;
USER(U).subscribe ≫ LIBRARY.Users.**INSERT**(U);
COPY(C).get_copy ≫ LIBRARY.Depot.**INSERT**(C);
LIB_BOOK(L).acquire ≫ LIBRARY.Catalog.**INSERT**(L);

With these considerations we want to close the discussion of TROLL-language concepts.

5 Discussion and Conclusion

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, GH91, EGH^{+}90] (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 formal UoD-modeling exist. The language FOOPS [GM87] evolved from integrating 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 [Wie90] is a language especially for formal specification of object-oriented conceptual models. In CMSL, objects are (like in TROLL) 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 TROLL 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 TROLL.

In this paper, we have introduced a model 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. We gave a formalization of our concept of object and introduced the language TROLL for the abstract description of object-oriented conceptual models. 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

We are very grateful to Cristina Sernadas who participated significantly in the development of the TROLL-language. Thanks also to all other IS-CORE colleagues, who contributed to the definition of the TROLL 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: Bakker, J. W. de; 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.; 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.* Benjamin/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, 1979.

[EGH+90] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. Informatik-Bericht 90-05, Technische Universität Braunschweig, 1990.

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

[EGL89] Ehrich, H.-D.; Gogolla, M.; Lipeck, U. W.: *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 (Proc. REX'89)*, Mood (NL), 1989. LNCS 394, Springer Verlag, Berlin, 1989, pp. 239–266.

[ESS90] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, Vol. 26, No. 1/2, 1990, pp. 33–48.

[FM90] Fiadeiro, J.; Maibaum, T.: Describing, Structuring and Implementing Objects. In: de Bakker, J.; de Roever, W.; Rozenberg, G. (eds): *Foundations of Object-Oriented Languages (Proc. REX School/Workshop)*, Noordwijkerhood (NL), 1990. LNCS 489, Springer-Verlag, Berlin, 1991, pp. 275-310.

[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.

[GH91] Gogolla, M.; Hohenstein, U.: Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems*, 1991. To appear.

[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.

[GR83] Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[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.

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

[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.

[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.

[MK90] McGregor, J. D.; Korson, T. (Guest editors): Special issue on object-oriented design. *Communications of the ACM*, Vol. 33, No. 9, 1990.

[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.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991.

[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.

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

[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).

[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.*

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