Object-Oriented Design of Information Systems: TROLL Language Features*

Gunter Saake
Thorsten Hartmann
Ralf Jungclaus
Hans-Dieter Ehrich
Abt. Datenbanken, Techn. Universität Braunschweig
Postfach 3329, D–38023 Braunschweig, Germany
email: {saake | hartmann | jungclau | ehrich}@idb.cs.tu-bs.de

Abstract

We present features of the language TROLL for the abstract specification of information systems. Information systems are regarded to be reactive systems with a large database. Before we present the constructs of TROLL, we briefly explain the basic ideas on which the language relies. The Universe of Discourse is regarded to be a collection of interacting objects. An object is modeled as a process with an observable state. The language TROLL allows for the integrated description of structure and behavior of objects. We explain the abstraction mechanisms provided by TROLL, namely roles, specialization, and aggregation. To support the description of systems composed from objects, the concepts of relationships and interfaces may be used.

1 Introduction

Information systems represent the relevant aspects of a portion of the real world (referred to as the Universe of Discourse (*UoD*) in the sequel) that are to be computerized. As such, an information system is capable of storing, processing and producing data about the UoD and thus is embedded in the UoD. The stored information changes over time according to interactions with the environment or predefined internal functions. Thus, information systems are *dynamic* in the sense that they may be regarded to be

*This work was partially supported by CEC under ESPRIT-II Basic Research Working Group 6071 IS-CORE II (Information Systems – COrrectness and REusability). The work of Ralf Jungclaus and Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3.

reactive systems [1] (note that we do not address the evolution of the schema). That is, an information system subsumes data, behavior and knowledge about both data and behavior. Recent trends in information systems research concern the distribution of information over (heterogeneous) systems and interoperability between cooperative systems, as information systems are increasingly used in a decentralized manner.

Information systems represent increasingly large and complex UoD's. Thus, adequate means to support the *design* are becoming more and more important. The design starts with collecting and representing *knowledge* about the UoD, which covers the relevant static and dynamic aspects [2]. In this phase, it is highly irrelevant to know *how* these aspects are implemented, thus a modeling approach should support a *declarative* description. As in engineering, the design process should produce *models* of solutions that can be assessed formally before any concrete system is implemented. Thus, formal specifications of mathematical models should be made as early as possible in the development process of information systems.

In this paper, we give an introduction to the language TROLL. TROLL is a specification language suitable for the description of the UoD and the information system on a high level of abstraction. TROLL is a logic-based language to describe properties and behavior of dynamic (cooperative) systems in an object-oriented way. That is, the specification is structured in objects. As far as possible, knowledge is localized in objects. Objects may interact by synchronous communications. Thus, a system is regarded as a collection of interacting objects. In these objects, the description of structure (by means of properties and components) and behavior over time (by means of processes over abstract events) is integrated. Collections of objects are further structured using the concepts of classification, specialization, generalization, and aggregation. Interactions and global assertions can be specified apart from object specifications to describe system properties and the overall behavior of systems.

The approach evolved from integrating work on algebraic specification of data types [3, 4] and databases [5, 6], process specification [7, 8], the specification of reactive systems [9, 1, 10], conceptual modeling [11, 12, 13, 14, 15] and knowledge representation [16, 17, 18]. The concept of object used as a basis for TROLL has been developed in [19, 20, 21] accompanied by work on a categorical semantics [22, 23]. Based on this formal concept, work has been done towards a logical framework of structured theories over a suitable logical calculus [24, 25]. First versions of the language introduced in this paper have appeared in [26, 27, 28, 29]. A complete description can be found in [30].

The paper is structured as follows: In the next section, we explain the basic ideas behind TROLL. We give a motivation for using an object-oriented approach and introduce informally the concept of object that underlies our language. In section 3, we show how objects as the basic system components can be specified. In section 4, we introduce abstraction mechanisms to construct objects from objects. In section 5, mechanisms to relate objects to build systems are presented. In the last section, we summarize and briefly discuss further research issues.

2 Basic Ideas behind TROLL

TROLL tries to integrate ideas from conceptual modeling (in the tradition of the ER-approach) and the specification of reactive systems with the object-oriented paradigm. This paradigm has been attracted a lot attention in different fields of computer science. In the software engineering community, object-orientation has taken the way up from programming (e.g. [31, 32]) to design (for a survey see [33]) and has already entered analysis (e.g. [34, 35]). In the database community, object-oriented databases have been very popular in recent years [36, 37, 38, 39, 40]. According to traditional research issues, each community puts special emphasis on certain aspects of object-orientation [41]

Traditionally, many notations for conceptual modeling have been *entity-based* in the sense that they look at the world consisting of interrelated entities [11]. Whereas entity-based notations emphasize *structural* aggregation, abstraction and inheritance, most object-oriented notations being around currently emphasize *behavioral* aggregation and inheritance. In conceptual modeling, *both* structural and behavioral aspects should be paid equal attention. Additionally, *temporal* aspects like precedence relationships between state transitions or possible life cycles and global aspects are of interest [42]. Temporal aspects in system specification have been addressed by approaches to use *temporal logic* (see e.g. [1]) in conceptual modeling of databases and information systems [9, 43, 10].

The basic idea is to integrate all static and dynamic aspects local to an entity (or object) in an *object description*. Object descriptions are thus encapsulated units of structure and behavior description. An object instance has an internal state that can be observed and changed exclusively through an object *interface*. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e. *methods*), object-oriented databases put emphasis on the observable structure of objects (through *attributes*). We propose to support both views in an equal manner, i.e. the structural properties of objects may be observed through attributes and the behavior of objects may be manipulated through *events* which are abstractions of state changing operations.

The encapsulation of all local aspects in object descriptions implies that object descriptions are the *units of design*. Following this perspective, we may model the system *and* its environment in a uniform way. We achieve in having clean interfaces between components that are part of the environment and components that are to be computerized later on. This approach results in having higher levels of modularity and abstraction in the early phases of system design.

An object description usually is regarded as a description of possible instances of the same kind which is similar to the notion of *type* in semantic data modeling. In object-oriented programming, the notion of type is closely related to (and sometimes even mixed up with) the notion of *class*. In our view, a class defines a *collection* of instances of the same type. Objects can be composed from other objects (*aggregation*).

Aggregation of objects imposes a *part-of* relation on a collection of object-descriptions. This kind of inheritance is known from semantic data models where it is used to model objects that appear in several roles in an application. Object descriptions may also be embedded in a *specialization* hierarchy. Usually, specialization implies reuse of specification code and allows to treat instances both as instances of the base class and the specialization class. A related concept is *generalization* that allows to treat conceptually different instances uniformly as instances of the generalization class.

Besides the structuring mechanisms mentioned above, a means to describe the *interaction* of object instances is needed to specify system dynamics. For conceptual modeling, we have to abstract from implementation-related details that arise from using message-passing and process communication. Communication is modeled conceptually by *calling* or *identifying* events of interrelated objects.

We do not present a detailed discussion of the basic ideas and their formalization here but refer instead to the accompanying paper by Hans-Dieter Ehrich, Ralf Jungclaus, Grit Denker and Amílcar Sernadas in this volume [44].

3 Specification of Objects

Throughout the rest of this paper, we use fragments of the following example taken from commercial applications. A bank maintains a number of accounts for customers. It also owns a number of automatic teller machines (ATMs) that are operated remotely. Accounts have the usual properties such that they may not be overdrawn etc. Associated with a checking account is a number of cash cards that can be used to withdraw money at an ATM. An ATM accepts a cash card, communicates with the user and the bank to carry out the transaction and dispenses cash if the transaction was successful and the ATM is not empty. The bank coordinates the card verification requests and the bank transactions issued concurrently from the ATMs.

In this section, we introduce object descriptions. The body of an object description is called *template*. In a template, the *signature* (the interface) as well as the structure and behavior of an object is described. A simple template may include the following sections:

template [*template name*]
  data types *import of data type signatures*
  attributes *attribute name and type declarations*
  events *event name and parameter declaration*
  constraints *static and dynamic constraints on attribute values*
  derivation *rules for the derivation of attributes and events*
  valuation *effects of events on attributes*
  behavior
    permissions *enabling conditions for event occurrences*
    obligations *completeness requirements for life cycles*

patterns *transactions and scripts*
[**end template** *template name*]

A single object is defined by a proper name and a template. An object class is defined by a class name, a template and an identification mechanism. In TROLL, we declare *external identifiers*. External identifiers are elements of the carrier set of an abstract data type. Similar to primary keys in databases, external identifiers are tuples of atomic data values. The set of external identifiers and the template make up the *class type*. An external identifier along with the class name defines a unique identification for instances of that class. The set of identifiers for a class is called the *id space*. An id space is an isomorphic copy of the set of external identifiers of the associated class type. As a notational convention, we denote the id space of a class `C` with `|C|`. Please note that `|C|` is a data type. Associated with an id space is an operation that maps an external identifier to the corresponding element of the id space. As a convention, the name of this operation is the class name: `C`: *type of external identifier* `-> |C|`.

Consider now an example for the description of a class in TROLL, the specification of the class `Account`. For this and the following examples, we assume a simple enumeration data type `UpdateType={deposit,withdraw}` to be defined elsewhere. We assume data types in general to be specified independently from object specifications in a suitable algebraic framework (e.g. [3]). The signature of such datatypes is explicitly imported in each template.

The attribute and event declarations defined make up the *local signature* which is the alphabet for the template. For our account example, the signature is specified as follows:

```
attributes
  constant Holder:|BankCustomer|;
  constant Type:{checking,saving};
  Balance:money;
  Red:bool;
  CreditLimit:money;
  derived MaxWithdrawal:money;
events
  birth open(in Holder:|BankCustomer|,in Type:{checking,saving});
  death close;
  new_credit_limit(in Amount:money);
  accept_update(in Type:UpdateType, in Amount:money);
  withdrawal(in Amount:money);
  deposit(in Amount:money);
  update_failed;
```

The local signature defines the *interface* of instances since it introduces the names and parameters of all visible properties of an instance. In the `Account`-example, we

declared the attributes `Holder`, `Type`, `Balance`, `Red`, `CreditLimit` and `MaxWithdrawal` along with their codomains, i.e. attributes in TROLL are typed. `Holder` is a *constant* attribute, i.e. it will be instantiated at creation time of an instance and may not be altered throughout the lifetime of that instance. The value of the attribute `Holder` is an identifier of an instance of class `BankCustomer`, i.e. it is a reference to another object (which is, however, *not a component*). The attribute `Type` denotes whether the account is used as checking account or savings account. The value of the attribute `MaxWithdrawal` is derived from the values of the other attributes according to the rules given in the **derivation**-section of a template.

In the **events**-section, the event names and parameters are declared. At least one **birth**-event is required that denotes the creation of an instance. Optionally, we may declare **death**-events that denote the destruction of an instance but we may declare objects that live "forever". All other events denote a noteworthy change in the state of instances. Events may have formal parameters which allow to define the effects of events on attribute values or for data to be exchanged during communication. The keywords **in** and **out** are used to decide about the data-flow direction during communication.

In the **constraints**-section, we may impose restrictions on the observable states. For accounts, we state the following:

```
constraints
  initially Red = false;
  initially CreditLimit = 0;
  initially Balance = 0;
  initially (sometime (Balance > 100) before Red);
  Red => (Balance <= CreditLimit);
  Red => sometimef(not Red);
derivation
  { Red } ==> MaxWithdrawal = CreditLimit - Balance;
  { not Red } ==> MaxWithdrawal = Balance + CreditLimit;
```

Constraints with the keyword **initially** state conditions to be fulfilled with respect to the initial state after the **birth**-event occurred. For initial and ordinary constraints we admit *dynamic* constraints stated in future tense temporal logic [9, 43, 10]. Dynamic constraints describe how the values of attributes may evolve in the future. Consider the initial constraint

```
initially ((sometime (Balance > 100) before Red);
```

which says that after an account has been opened, the balance must have been more that 100 before it can be overdrawn. The formula

```
Red => sometimef(not Red);
```

states that if an account is in 'red condition', sometime in the future it has to leave this condition. Implicitly, constraints restrict the possible state transitions.

In the **derivation**-section, rules to compute the values of the derived attributes may be stated. For the `Account`-template, we have conditional expressions to compute the maximal amount of money that can be withdrawn in the current state depending on the value of the attribute `Red`.

The values of attributes may change with the occurrence of events. Thus, to describe the change of objects over time, we have to describe how the occurrence of events affect the values of attributes. *Valuation formulae* stated in the **valuation**-section of a template are based on a positional logic [45]. The **valuation**-section of our account example looks as follows:

```
valuation
  variables m:money;
  [new_credit_limit(m)]CreditLimit = m;
  { not Red and (m <= Balance)} ==> [withdrawal(m)]Balance = Balance-m;
  { not Red } ==> [deposit(m)]Balance = Balance+m;
  { not Red and (m > Balance)} ==>
    [withdrawal(m)](Balance = m-Balance), (Red = true);
  { Red } ==> [withdrawal(m)]Balance = Balance+m;
  { Red and (m >= Balance)} ==>
    [deposit(m)](Balance = m-Balance), (Red = false);
  { Red and (m < Balance)} ==> [deposit(m)]Balance = Balance-m;
```

Valuation formulae may be conditional like the following one:

```
{ not Red and (m <= Balance)} ==> [withdrawal(m)]Balance = Balance-m;
```

The rule will only be applied if the condition evaluates to **true**. The rule states that after the occurrence of the event `withdrawal` instantiated by a value `m` the attribute `Balance` will have the value of (`Balance-m`). The term on the right side of the equals sign is evaluated in the state *before* the event occurred. In the following rule, we state that the occurrence of an event has an effect on more than one attribute. In that case, we may use a list of effects:

```
{ not Red and (m > Balance)} ==>
        [withdrawal(m)](Balance = m-Balance), (Red = true);
```

We implicitly use a *frame rule* saying that attributes for which no effects of event occurrences are specified do not change their value after occurrences of such events.

A major part of an object description is the description of the behavior of instances. Let us first give the **behavior**-section of the `Account` template:

```
behavior
  permissions
    variables t,t1:UpdateType; m,m1,m2:money;
    { Balance = 0 } close;
    { not sometime(after(accept_update(t,m1)))
      since last(after(update_failed) or
                 after(deposit(m2)) or
                 after(withdrawal(m2)))} accept_update(t1,m);
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = withdraw and (m > MaxWithdrawal)} update_failed;
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = deposit } deposit(m);
    { sometime(after(accept_update(t,m)))
      since last after(accept_update(t1,m1)) and
      t = withdraw and (m <= MaxWithdrawal)} withdrawal(m);
```

Basically, we provide three sections. In the **permissions**-section, we may state enabling conditions for event occurrences. Events may only occur if the enabling condition is fulfilled. Thus, permissions state that something bad may never happen. The general form of permissions is

{ *(temporal) condition* } *event_term*;

Permissions may refer to the current observable state (*simple permissions*) or to the history of events that occurred in the life of an instance so far (*temporal permissions*). As an example for a simple permission look at the following rule that requires an account to be empty before it can be closed:

```
{ Balance = 0 } close;
```

For temporal permissions, we may state preconditions being formulae of a past tense temporal logic. It is defined analogously to the future tense temporal logic of [10, 46]. Besides the temporal quantifiers **sometime**, **always** and **previous** we may also use the bounded quantifiers **sometime ... since last ...** and **always ... since last ...**. The following rule for example states that after a transaction has been completed with the occurrence of one of the events `update_failed` or `deposit(m1)` or `withdrawal(m1)`, at most one event `accept_update` may occur (i.e. we do not allow to handle interleaved updates in an account):

```
{ not sometime(after(accept_update(t,m1)))
      since last(after(update_failed) or
                 after(deposit(m2)) or
                 after(withdrawal(m2))) } accept_update(t1,m);
```

In the **obligations**-section, we state completeness requirements for life cycles. These requirements must be fulfilled before the object is allowed to die. Usually, obligations depend on the history of the object. The following requirement states that once an event `accept_update(t,m)` occurs, this update must be completed eventually by an occurrence of one of the events `update_failed` or `deposit(m)` or `withdrawal(m)`:

```
{ after(accept_update(t,m)) } ==>
      deposit(m) or withdrawal(m) or update_failed;
```

A template makes up the body of a *class type definition*. A class then can be seen as a container for instances of the associated type. In a class type specification we define an *identification mechanism* along with the template. An identification mechanism describes the set of possible external identifiers of instances. Each external identifier will be mapped to an immutable internal surrogate.

Let us now give the specification of the class `Account` as a whole:

```
object class Account
  identification
    data types nat;
    No:  nat
  template
    data types |BankCustomer|,money,bool,UpdateType;
    attributes
      constant Holder:|BankCustomer|;
      constant Type:{checking, saving};
      Balance:money;
      Red:bool;
      CreditLimit:money;
      derived MaxWithdrawal:  money;
    events
      birth open(in Holder:|BankCustomer|,in Type:checking,saving);
      death close;
      new_credit_limit(in Amount:money);
      accept_update(in Type:UpdateType, in Amount:money);
      withdrawal(in Amount:money);
      deposit(in Amount:money);
      update_failed;
    constraints
      initially Red = false;
      initially CreditLimit = 0;
      initially Balance = 0;
      initially ((Balance > 100) before Red);
      Red => (Balance <= CreditLimit);
```

```
    Red => sometimef(not Red);
  derivation
    { Red } ==> MaxWithdrawal = CreditLimit - Balance;
    { not Red } ==> MaxWithdrawal = Balance + CreditLimit;
  valuation
    variables m:money;
    [new_credit_limit(m)]CreditLimit = m;
    { not Red and (m <= Balance)} ==>
      [withdrawal(m)]Balance = Balance-m;
    { not Red } ==> [deposit(m)]Balance = Balance+m;
    { not Red and (m > Balance)} ==>
      [withdrawal(m)](Balance = m-Balance), (Red = true);
    { Red } ==> [withdrawal(m)]Balance = Balance+m;
    { Red and (m >= Balance)} ==>
      [deposit(m)](Balance = m-Balance), (Red = false);
    { Red and (m < Balance)} ==> [deposit(m)]Balance = Balance-m;
  behavior
    permissions
      variables t,t1:UpdateType; m,m1,m2:money;
      { Balance = 0 } close;
      { not sometime(after(accept_update(t1,m1)))
        since last(after(update_failed) or
                  after(deposit(m2)) or
                  after(withdrawal(m2)))} accept_update(t,m);
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = withdraw and (m > MaxWithdrawal)} update_failed;
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = deposit } deposit(m);
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = withdraw and (m <= MaxWithdrawal)} withdrawal(m);
end object class Account;
```

There is another means to describe the behavior of objects. Parts of life cycles (behavior *patterns*) may be described using a process language that draws on CSP [7] and LOTOS [47]. Examples for the use of this process language can be found in [30].

The last remaining basic language construct is the declaration of event synchronization by *event calling*. When events e_1 and e_2 are related by calling (e_1 >> e_2), then the occurrence of e_1 forces e_2 to occur simultaneously. Thus, calling can be characterized as synchronous communication between the components of a complex object. Calling declarations are marked with the keyword **interaction**. For examples see the next

section. Using parameters, we may specify the exchange of values during interactions (*communication*). The unification of uninstantiated variables with instantiated ones constitutes the flow of information. Calling is further discussed in [48, 49, 50] and [51, Section 7.2].

Templates are the building blocks of system specifications. As a first concept to structure system descriptions, we introduced classification. In the following section we want to introduce more concepts for the structuring of specifications.

4 Abstractions

System descriptions in TROLL can be structured in several ways. The mechanisms presented in this section are *roles*, *specialization*, and *aggregation*.

4.1 Roles and Specialization

Both concepts are related in the sense that they describe *is_a* relationships between object descriptions, i.e. each instance of a role / specialization class may be referred to as an instance of the base class, too.

4.1.1 Roles

The concept of *role* describes a dynamic (temporary) specialization of objects, i.e. a special view of objects [52]. As an example consider the roles customer or employee of persons. When looking at an object playing a role, we may want to know things that are not relevant for the base object. Thus, a role has additional properties, it is a more detailed description of the base object from a certain point of view.

Consider now an example. Suppose we have specified a template describing persons, called `Person`. The `Person` template is assumed to have the usual attributes like `Name`, `FirstName`, `Address`, `Birthdate` etc. The dynamics only cover attribute updates. In our bank world, let us now look at persons being customers:

```
object class BankCustomer
  role of Person
  template
    data types nat,set(nat);
    attributes
      Accts:set(nat);
    events
      birth bc_bank_customer;
      death cancel;
      active open_account(out Acct:nat);
      active close_account(in Acct:nat);
```

```
    valuation
      variables n:nat;
      [bc_bank_customer]Accts = emptyset();
      [open_account(n)]Accts = insert(n,Accts);
      { in(n,Accts) } ==> [close_account(n)]Accts = remove(n,Accts);
    behavior
      permissions
        variables n:nat;
        { sometime(after(open_account(n))) } close_account(n);
  end object class BankCustomer;
```

In the template, we introduce new attribute and event symbols that extend the `Person` signature. Here, we have an additional attribute `Accts` that clearly makes sense only for bank customers.

A birth event for a role corresponds to an ordinary event of the base object and denotes the start of playing a role. Each object may play a role several times. A death-event of a role denotes that an object ceases to play a role (at least for that moment).

In the `BankCustomer`-template, two other events are declared. Both are marked **active**, which denotes that they may occur on the initiative of the `BankCustomer` whenever they are permitted to.

Semantically, we have to deal with both syntactical and semantical inheritance. Syntactically, the base template is included in the role template. The local specifications extend the base template. Semantically, each role instance includes the corresponding base instance. In our example this means that an instance of `BankCustomer` includes the instance of `Person` of which it is a role. This way, a role instance may access the base instance's attributes and may call the base instance's events.

4.1.2 Specialization

A special case of a role is the definition of a *specialization*. We decided to introduce extra language features for this concept because it arises frequently in a system specification. Specialization describes that an object plays a role throughout its entire lifetime. In conceptual modeling, this concept is known under the term *is-a* or *kind-of*. A specialization hierarchy describes a *taxonomy* on objects.

For specializations, we do not have to describe the birth of a specialized object explicitly since it corresponds to the birth event of the base instance. Thus, we do not have to specify a birth event for a specialization. In case of a specialization, we must provide a specialization condition, stating which objects belong to the specialized object class.

As an example, consider two specializations of our `Account`: A savings account (`SavingsAccount`) and a checking account (`CheckingAccount`). Both have special properties in addition to their common ones which are described in the base template

Account. A special aspect of savings accounts is that the bank (usually) pays interest for it. Furthermore, the balance of a savings account is always non-negative, i.e. the credit limit is 0. Every once in a while, interest is paid (computed by some interesting function) and is added to the balance. Please note that we may not update directly the attribute Balance of the Account – we have to *call* explicitly the event deposit of the Account to update the Balance. The specialization class SavingsAccount is specified as follows:

```
object class SavingsAccount
  specializing Account where Type = saving;
    template
      data types real,date;
      attributes
        constant interest_rate:real;
        last_interest_paid:date;
      events
        pay_interest(in date:date);
      constraints
        alwaysf (CreditLimit = 0);
        alwaysf not Red;
      valuation
        variables d:date;
        [pay_interest(d)]last_interest_paid = d;
      behavior
        permissions
          variables d:date;
          { days_between(d,last_interest_paid) > 30 } pay_interest(d);
      interactions
        variables d:date;
        pay_interest(d) >>
                deposit((days_between(d,last_interest_paid)/360) *
                    (Balance * interest_rate / 100)
                  );
end object class SavingsAccount;
```

Consider now the specialization class CheckingAccount. In our small UoD, we may assign cashcards to checking accounts. With each checking account, a constant personal identification number (PIN) is associated:

```
object class CheckingAccount
  specializing Account where Type = checking;
    template
      data types nat,|CashCard|,money;
```

```
    attributes
      constant PIN:nat;
      Cards:set(|CashCard|);
    events
      assign_card(in C:|CashCard|);
      cancel_card(in C:|CashCard|);
    constraints
      initially Cards = emptyset();
    valuation
      variables C:|CashCard|;
      [assign_card(C)]Cards = insert(C,Cards);
      { in(C,Cards)} ==> [cancel_card(C)]Cards = remove(C,Cards);
end object class CheckingAccount;
```

Note that we do not impose further restriction on the life cycles – cashcards may be assigned anytime, and for example the credit limit may be updated anytime due to the occurrence of the event `new_credit_limit` inherited from the base object (although this is not done on the initiative of a `CheckingAccount`-instance itself).

4.2 Composite Objects

Using the aggregation concept, we may construct objects from components. In the database community, this is also known as constructing *composite objects*. Basically, we can identify two kinds of composite objects [53]:

- *Disjoint* composite objects do not share any components. This implies that components cannot exist outside the composite object, they are strongly dependent and the components are *local* to the composite object. The composition is always static. Language features for defining disjoint composite objects in TROLL can be found in [30, 48, 50].
- *Non-disjoint* composite objects may share components. Thus, components are autonomous objects. In TROLL, we distinguish two kinds of non-disjoint composite objects: Dynamic composite objects may alter their composition through events whereas the composition of static composite objects is described through static predicates.

For disjoint and non-disjoint composite objects the components are encapsulated in the sense that their state may only be altered by events local to the components. Their attribute values, however, are visible. The coordination and synchronization between the composite object and its components or between the components must be done by communication. Let us now continue with the concepts of non-disjoint composite objects, that is static and dynamic aggregation.

4.2.1 Static Aggregation

The aggregation of static composite objects is described using predicates over identifiers and constants. In contrast to dynamic aggregation where the structure of the composite object may vary over time, the structure of a static composite object never changes. Specificationwise we describe the *possible* object composition not violating the constraints for aggregation. Possibly there are components belonging to the composite object that are not yet born.

For example let us consider the specification of an object `Bank1`. Suppose that we want to describe this `Bank1` object including all possible `Account` objects. The specification may look as follows:

```
object Bank
  template
    including Account;
    data types |Account|,set(|Account|),nat;
    attributes
      TheAccounts:set(|Account|);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat); ...
    valuation
      variables n:nat;
      [open_account(n)]TheAccounts = insert(Account(n),TheAccounts);
      [close_account(n)]TheAccounts = remove(Account(n),TheAccounts);
    behavior
      permissions
        variables n,p,p1:nat; m,m1:money; t,t1:UpdateType;
        { not in(Account(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
      interactions
        variables t:UpdateType; m:money; n:nat;
        open_account(n) >> Account(n).open;
        close_account(n) >> Account(n).close; ...
end object Bank;
```

In the interaction section communication between the composite object `Bank` and its components – the `Accounts` – is specified. Since the aggregated object is constructed from a set of accounts, we must use an operation `Account : nat` $\rightarrow$ `|Account|` to generate identifiers for included objects.

Identifiers are only *references* to objects, thus a second operation taking an object identifier and yielding the object itself is needed. Since there is no ambiguity – the

accounts are *subobjects* of the bank – we could leave out the second operation, assuming that `Account(n)` delivers an object instance:

```
open_account(n) >> Account(n).open;
```

In this case for example the `Bank` event `open_account(n)` calls the `Account` event `open` in component object `Account(n)`. Thus the creation and destruction of `Account` instances is triggered by the `Bank` object. Note that the conditional calling (intuitively) only occurs if the condition evaluates to true in the current state.

Let us now give a few words on the semantics. The signature of the `Bank` object is obtained by disjoint union of the (local) signatures of the `Bank` and the `accounts`. Since we included a set of objects we have to deal with indexed symbols. Indexing is denoted using the dot notation (for example `Account(n).open`). For the life cycle of the composite object we state that if we constrain a life cycle to the events of a component object, we have to obtain a valid life cycle of this component.

4.2.2 Dynamic Aggregation

The use of dynamic composite objects allows for a high level description of object composition. Components may be specified as *single components* as well as *sets* or *lists* of components. Again, properties of components must not be violated. Components have a life of their own and may be shared by other objects as well.

With the specification of a dynamic composite object denoted with the keyword **components** we have implicitly defined events to update the composition. Additionally we have implicit attributes to observe the composite object. For example for a *set* of component objects we have events to insert and delete objects and an attribute to observe set-membership. For *lists* of objects we have events to append and to remove objects and for *single* objects there are events to add and remove them as well as an attribute to test if the component is assigned. A complete list of implicitly generated events and attributes for each composite object construction can be found in the TROLL language report [30].

This view of composite objects is operational instead of declarative like the concept of static aggregation. Before we will say how dynamic aggregation fits in our semantic framework we will give an example :

```
object AnotherBank
  template
    data types nat,|ATM|,UpdateType,money,|CashCard|;
    components
      Depts:LIST(Department);
    events
      birth establish; death close_down;
      open_dept(in No:nat);
      close_dept(in No:nat);
```

```
    behavior
      permissions
        variables n,p,p1:nat;
        { not Depts.IN(Department(n)) } open_dept(n);
        { sometime after(open_dept(n)) } close_dept(n);
        ...
      interaction
        variables ...,n:nat;
        open_dept(n) >> Depts.INSERT(Department(n));
        open_dept(n) >> Department(n).open;
        close_dept(n) >> Depts.REMOVE(Department(n));
        close_dept(n) >> Department(n).close;
        ...
end object AnotherBank;
```

In this example, we partially model `AnotherBank` object with a component list of `Department` objects. Initially, `AnotherBank` has no component. To manipulate the set of component accounts, the events `Depts.INSERT(|Department|)` and `Depts.DELETE(|Department|)` are automatically added to the signature of the `AnotherBank` object. For set components a parameterized, bool-valued attribute, in this example `Depts.IN(|Department|)`:bool, is included.

For the behavior of the composite object the communication inside the composite object must be specified. Communication can take place between the component objects and between the composite object and the component objects. In this example only the latter case is used. See for example the clauses

```
open_dept(n) >> Department(n).open;
open_dept(n) >> Depts.INSERT(Department(n));
```

which state, that every time a department identified by the natural number `n` is opened, the event `open` is called in the corresponding object `Department(n)` and it becomes a member of the set of components. Note that the event `open_dept` in the `Bank` object may only occur if the expression **not** `Depts.IN(|Department|)` evaluates to true.

Let us now briefly look at the semantics of dynamic object aggregation. Since our concept of object only allows for static object composition, i.e. embedding the components into the composite object, we need to simulate dynamic composition using the concept of static aggregation. This is done in the following way: for each possible composition the corresponding template is obtained by including the object signatures of all objects that are part of this composite object. Whenever the composition changes, a new composite object is created in the same way and the old one is destroyed. The observable properties of the unchanged components remain the same in the new instance.

5 Specification of Systems

When it comes to describing systems of interacting objects, it is not sufficient to provide only the structuring mechanisms described in the previous section. In system specification, we have to deal with static and dynamic *relationships* between objects, with *interfaces*, and with *object societies*.

5.1 Relationships

Relationships connect objects that are specified independently. Basically, relationships are language constructs to describe how system components are connected in order to describe the whole system.

In TROLL, two types of relationships are supported:

- (global) *interactions* and
- (global) *constraints*.

5.1.1 Global Interactions

Global interactions describe communication between objects. We use the syntax for interactions inside complex objects. Global interactions along with the specifications of the connected objects describe patterns of communication between the connected objects (these patterns are called *scripts* elsewhere [12]). As usual, communication is described using event calling.

From a process point of view, relationships describe how the involved processes *synchronize*. In interaction specifications, we may want to refer to the history of events in the connected objects. Consider the interaction between an ATM and the Bank that maintains it. Here, we must put precedence rules into conditions for interactions to model the process of communication:

```
relationship RemoteTransaction between Bank,ATM
  interaction
    /* -- Card checking business */
    ATM(n).check_w_bank(a,p) >> Bank.check_card(a,p);
    {after(ATM(n).check_w_bank(a,p))} ==>
        Bank.card_OK >> ATM(n).card_OK;
    {after(ATM(n).check_w_bank(a,p))} ==>
        Bank.card_NOK >> ATM(n).card_not_OK;
    /* -- Bank transaction business */
    ATM(n).issue_TA(n,m) >> Bank.process_TA(n,m);
    {after(ATM(n).process_TA(a,m))} ==>
        Bank.TA_OK >> ATM(n).TA_ok;
    {after(ATM(n).process_TA(a,m))} ==>
```

```
        Bank.TA_failed >> ATM(n).TA_failed;
  end relationship RemoteTransaction;
```

For precedence rules, we may use the **after** predicate, which holds in a state immediately after the event being the argument occurred. The following clause e.g. states that an interaction induced by the calling of the `TA_failed` event in an ATM by the `TA_failed` event in the `Bank` may only take place if the event `process_TA(a,m)` occured immediatley before that in the ATM:

```
{after(ATM(n).process_TA(a,m))} ==>
    Bank.TA_failed >> ATM(n).TA_failed;
```

Please note that we use a very simple execution model. A chain of calls may only be carried out if all called events are permitted to occur (atomicity principle) [49]. We are aware of the limitations of our approach with respect to exceptions and long transactions and plan to work on a more sophisticated model of execution.

5.1.2 Global Constraints

When we model systems by putting together objects, we sometimes have to state constraints that are to be fulfilled by related but independently specified objects. Such *global constraints* set up a relationship between objects. Consider the following example. When modeling our banking world, there may be a regulation that one particular bank customer may only be holder of at most one checking account. Please note that this is an example for a relationship since it cannot be specified to be local to *one* instance of the class `CheckingAccount`. In TROLL, this would be specified as follows:

```
relationship IB1 between CheckingAccount C1,CheckingAccount C2;
  data types |BankCustomer|,nat;
  constraints
    (C1.Holder=C2.Holder) => (C1.No=C2.No);
end relationship IB1;
```

Global constraints are specified using the same syntax as local constraints.

If a relationship between object classes contains both interactions and constraints, both sections may be specified together.

5.2 Interfaces

An object interface or object class interface is first of all a mechanism to describe access control to objects. Interfaces in TROLL support the explicit encapsulation of object properties. Access control is achieved by projecting the attribute and event symbols to external visible symbols. Interfaces may be defined for single objects as well as for object classes. For object classes, we may additionally define selection

interfaces, thus restricting the visible population of the class to some proper subset. Selection interfaces resemble the well known views from relational databases [28].

The first example shows a simple class interface for ATM's seen by a customer:

```
interface class ATMToCustomer
  encapsulating ATM:
    data types bool,|CashCard|,nat,money;
    attributes
      dispensed:bool;
    events
      active ready;
      active read_card(in C:|CashCard|);
      card_accepted; bad_PIN_msg; bad_account_msg;
      active issue_TA(in Acct:nat,in Amount:money);
      active cancel;
      TA_failed_msg; eject_card;
      dispense_cash(in Amount:money);
end interface class ATMToCustomer;
```

The only observation for customers is the status of the ATM in terms of the bool-valued attribute `dispensed`. Information about the amount of money available inside the machine should (for obvious reason) not be public. An interface to dynamic objects must define also the possible operations visible at this level of system description. Here a customer should only be able to talk to the ATM at 'user level', i.e. (s)he must be able to insert cards, issue transactions, cancel the transaction and not at least dispense money. Customers must not be able to refill a machine or even remove it. Also they should not see details of the internal operations, for example the event `check_card_w_bank` is not relevant at 'user level'. The semantics of this simple kind of interface is just a signature restriction to the explicit noted event and attribute symbols.

Another kind of access restriction in contrast to the above mentioned projection interface is the selection interface. Suppose we only want customers to use ATM's identified by a natural number between 100 and 199:

```
interface class ATMToCustomer2
  encapsulating ATM
  selection
    where IdentNo >= 100 and IdentNo <= 199
  data types bool,...;
  attributes
    dispensed:bool;
  events
    ...
end interface class ATMToCustomer2;
```

Here the actual visible population is limited using a predicate over the external key. The semantics of this kind of interface definition is given by an object class specialization followed by a projection interface. The predicate used for the definition of the specialized class can be seen as a filter allowing only those instances to pass, that satisfy the selection condition evaluated locally to the object instances.

In general we do not only want to restrict the external object interface to some subset of events and attributes, but also be able to present derived properties of objects. We may specify views of an object where some information is explicitly computed from existing attributes. An example interface for the ATM class may be used for service personnel only. Suppose that we want to indicate machines that must be refilled to avoid a dispensed condition. Therefore this view defines a derived bool-valued attribute `please_refill` to be true for ATM's with `CashOnHand` below a threshold value of 1000:

```
interface class ATMToService
  encapsulating ATM
  data types bool,money;
  attributes
    please_refill:bool;
    dispensed:bool;
  events
    refill(in Amount:money);
    ...
  derivation
    please_refill = (CashOnHand <= 1000);
end interface class ATMToService;
```

Note that the derivation part is generally hidden from the view users. Technically, an interface with derived properties consists of a formal implementation step [21, 54] and an explicit projection interface. More general we can also look at specialized operations as a view on the dynamic part of objects. For example we may have users of the ATM that have to pay an extra charge for each bank transaction. Suppose that the ATM has an additional (may be constant) attribute `extraCharge:money`, denoting the amount of money to be withdrawn from the account:

```
interface class ATMToExtraUser
  encapsulating ATM
  data types money,bool,money;
  attributes
    extraCharge:money;
    dispensed:bool;
  events
    issue_TA_Extra(nat,money);
    ...
```

```
  derivation
    calling
      variables n:nat, m:money;
      issue_TA_Extra(n,m) >> issue_TA(n,m+extraCharge)
end interface class ATMToExtraUser;
```

The attribute `extraCharge` is seen from the specialized user, since he should know about the extra charge. Each time this user issues an `issue_TA_Extra(n,m)` at this ATM, an additional extra charge is withdrawn.

Object interfaces have nothing to do with object copies. Interfaces are just a means to specify different views on objects, that is, to select special object populations out of the existing set of instances and to restrict the use of objects with respect to their observation and operation interface.

6 Conclusions and Outlook

In this paper, we have introduced an abstract specification language for information systems. Specifications are structured in objects. An object description includes the specification of structural properties and the specification of behavioral properties. For simple objects, attributes are used to describe static aspects of the object's state and events are used to describe the basic state transitions. The admissible temporal ordering of events is described using (temporal) enabling conditions (permissions), conditions to be guaranteed by objects (obligations), and explicit patterns of behavior. The evolution of the object's state depending on the actual behavior over time is described by valuation rules that specify the effects of event occurrences on attribute values.

Object descriptions are the basic units of structure. In TROLL, we may apply a number of abstraction mechanisms to object descriptions. Roles describe temporal (dynamic) specializations of objects. An object may play several roles concurrently and may play each role more than once. Specializations are roles which are fixed for the lifetime of an object. Using generalization, we may collect different objects under a common (virtual) class. Furthermore, we may describe objects that are constructed from components. Disjoint complex objects have components that are strongly dependent on the base object – such components may be described using subtemplates. Static and dynamic aggregation describe objects with components that may be shared between objects.

Object descriptions and their abstractions are the components of systems. They have to be connected in order to provide the services of a system. For this purpose, TROLL provides the features of relationships and interfaces. Relationships describe constraints and interactions between objects that are specified independently. Interfaces describe explicit views on objects and may be used to control access to system components.

TROLL offers a large number of constructs that are especially suited for the conceptual modeling of information systems at a very high level of abstraction. It combines features of conceptual modeling approaches and object-oriented approaches with formal approaches to data and process modeling. Providing objects as units of design, TROLL allows to achieve higher levels of modularity with clean but complete interface descriptions. Thus, the boundaries between the information system and the environment as well as the boundaries between data and processes become transparent.

Further work on TROLL will cover in-the-large issues like reuse, modularization above the object level, and parameterization. We plan to put another language level above TROLL with constructs that enable the construction of system descriptions from components of various grain.

In another direction, we are working on a language kernel which include those TROLL concepts that are suitable to describe (distributed) implementation platforms like operating systems and databases in an abstract way. This kernel language is regarded as an interface to an implementation platform. We then want to investigate the transformation of TROLL specifications into this kernel language. In our group, a dialect called TROLL*light* [55] has been developed as a simplified language especially suited for verification purposes.

Acknowledgements

For many fruitful discussions on the language we are grateful to all members of ISCORE, especially to Amílcar Sernadas, Cristina Sernadas, José Fiadeiro, and Roel Wieringa.

References

1. Z. Manna and A. Pnueli. The Anchored Version of the Temporal Framework. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Tome and Partial Order in Logics and Models for Concurrency*, pages 201–284. LNCS 354, Springer-Verlag, Berlin, 1989.
2. J. J. van Griethuysen. Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.
3. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.
4. H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, Stuttgart, 1989.
5. H.-D. Ehrich. Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. In D. Pitt et al., editors, *Proc. Workshop on Category Theory and Computer Programming*, pages 412–433. Springer, Berlin, LNCS series, 1986.

6. H.-D. Ehrich, K. Drosten, and M. Gogolla. Towards an Algebraic Semantics for Database Specification. In R.A. Meersmann and A. Sernadas, editors, *Proc. 2nd IFIP WG 2.6 Working Conf. on Database Semantics "Data and Knowledge" (DS-2)*, pages 119–135, Albufeira (Portugal), 1988. North-Holland, Amsterdam.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.
8. R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.
9. A. Sernadas. Temporal Aspects of Logical Procedure Definition. *Information Systems*, 5:167–187, 1980.
10. G. Saake. Conceptual Modeling of Database Applications. In D. Karagiannis, editor, *Proc. 1st IS/KI Workshop, Ulm (Germany), 1990*, pages 213–232. Springer, Berlin, LNCS 474, 1991.
11. P. P. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
12. J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2):185–207, 1980.
13. M. Brodie, J. Mylopoulos, and J. W. Schmidt. *On Conceptual Modelling—Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, Berlin, 1984.
14. A. Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, 2(1):63–73, 1985.
15. G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, 9(2):157–204, 1992.
16. M. L. Brodie and J. Mylopoulos, editors. *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.
17. J. W. Schmidt and C. Thanos, editors. *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.
18. J. Mylopoulos and M. Brodie, editors. *Readings in Artificial Intelligence & Databases.* Morgan Kaufmann Publ. San Mateo, 1989.
19. A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.
20. A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. The Basic Building Blocks of Information Systems. In E. Falkenberg and P. Lindgreen, editors, *Information*

System Concepts: An In-Depth Analysis, pages 225–246, Namur (B), 1989. North-Holland, Amsterdam, 1989.

21. A. Sernadas and H.-D. Ehrich. What Is an Object, After All? In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 39–70, Amsterdam, 1991. North-Holland.
22. H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Proc. REX/FOOL Workshop*, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin.
23. H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability*, pages 1–24. TU Braunschweig, Informatik Bericht 91-03, 1991.
24. J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 243–284, Amsterdam, 1991. North-Holland.
25. J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.
26. R. Jungclaus, G. Saake, and C. Sernadas. Formal Specification of Object Systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91, Brighton*, pages 60–82. Springer, Berlin, LNCS 494, 1991.
27. G. Saake and R. Jungclaus. Konzeptioneller Entwurf von Objektgesellschaften. In H.-J. Appelrath, editor, *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft BTW'91*, pages 327–343. Informatik-Fachberichte IFB 270, Springer, Berlin, 1991.
28. G. Saake and R. Jungclaus. Specification of Database Applications in the TROLL-Language. In D. Harper and M. Norrie, editors, *Proc. Int. Workshop Specification of Database Systems, Glasgow, July 1991*, pages 228–245. Springer, London, 1992.
29. R. Jungclaus, G. Saake, and T. Hartmann. Language Features for Object-Oriented Conceptual Modeling. In T.J. Teory, editor, *Proc. 10th Int. Conf. on the ER-approach*, pages 309–324, San Mateo, 1991.
30. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.
31. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

32. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

33. J. D. McGregor and T. (Guest editors) Korson. Special Issue on Object-Oriented Design. *Communications of the ACM*, 33(9), 1990.

34. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1989.

35. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

36. K. R. Dittrich and U. Dayal, editors. *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986. IEEE Computer Society Press, Washington, 1986.

37. K. R. Dittrich, editor. *Advances in Object-Oriented Database Systems*. Lecture Notes in Comp. Sc. 334. Springer Verlag, Berlin, 1988.

38. M. Atkinson, F. Bancilhon, D. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. Int. Conf. on Deductive and Object-Oriented Database Systems*, pages 40–57, Kyoto, Japan, December 1989.

39. W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

40. A. Heuer. *Objektorientierte Datenbanken*. Addison-Wesley, Bonn (D), 1992. *In German*.

41. E. Verharen. Object-Oriented System Development—An Overview. In G. Saake and A. Sernadas, editors, *Information Systems—Correctness and Reusability*. Informatik-Bericht 91–03, TU Braunschweig, 1991.

42. D. C. Tsichritzis and O. M. Nierstrasz. Directions in Object-Oriented Research. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 523–536. ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

43. U. W. Lipeck. *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung*. Informatik-Fachbericht 209. Springer, Berlin, 1989.

44. H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-oriented design of information systems: Theoretical foundations, 1993. *This volume*.

45. J. Fiadeiro and A. Sernadas. Logics of Modal Terms for System Specification. *Journal of Logic and Computation*, 1(2):187–227, 1990.

46. S. Schwiderski, T. Hartmann, and G. Saake. Monitoring Temporal Preconditions in a Behaviour Oriented Object Model. Informatik-Bericht 93-07, TU Braunschweig, 1993.

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

48. T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O. Lehrmann Madsen, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP'92)*, pages 57–77. Springer, LNCS 615, Berlin, 1992.

49. T. Hartmann and G. Saake. Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 1993.

50. G. Saake. *Objektorientierte Spezifikation von Informationssystemen.* Teubner, Leipzig, 1993.

51. R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-Based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

52. R. J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

53. B. Batory and A. Buchmann. Molecular Objects, Abstract Data Types and Data Models: A Framework. In U. Dayal, G. Schlageter, and L. H. Seng, editors, *Proc. 10th Int. Conf. on Very Large Databases VLDB'84*, pages 172–184, Singapore, 1984.

54. H.-D. Ehrich and A. Sernadas. Algebraic Implementation of Objects over Objects. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*, pages 239–266. LNCS 430, Springer, Berlin, 1990.

55. S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.