# Introduction to TROLL –

## A Language for Object-Oriented Specification of Information Systems[‡]

Ralf Jungclaus[*]
Thorsten Hartmann[*]
Gunter Saake[*]
Cristina Sernadas[†]

**Abstract**

In this paper, we present 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 UoD is regarded to be a collection of interacting objects. An object is modeled as a process with an observable state. The language TROLL itself allows for the integrated description of structure and behavior of objects. We explain the abstraction mechanisms provided by TROLL, namely roles, specialization, generalization 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 information 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 reactive systems [Pnu77, MP89] (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

[*]Abt. Datenbanken, Techn. Universität Braunschweig, Postfach 3329, D–3300 Braunschweig, Germany (`{jungclau,hartmann,saake}@infbs.uucp`)

[†]Departamento de Matemática – Instituto Superior Técnico, Av. Rovisco Pais, 1096 Lisboa Codex, Portugal (`css@inesc.inesc.pt`)

[‡]This work was partially supported by CEC under ESPRIT-II Basic Research Action Working Group No. 3023 IS-CORE (Information Systems – COrrectness and REusability). The work of Ralf Jungclaus and Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1 and Sa 465/1-2.

with collecting and representing *knowledge* about the UoD, which covers all relevant static and dynamic aspects [Gri82]. 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 produced as early as possible in the design process of information systems. A desirable feature of specification languages is a logical representation with a *proof theory* for reasoning about specifications based on a suitable logical framework. A proof theory allows to *derive knowledge* from specifications in order to support assessment of solutions and to formally verify implementations against specifications.

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 subobjects) 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 [EM85, EGL89] and databases [Ehr86, EDG86], process specification [Hoa85, Mil89], the specification of reactive systems [Ser80, MP89, Saa91], conceptual modeling [Che76, MBW80, BMS84, Bor85, HG88, EGH+90] and knowledge representation [BM86, ST89, MB89]. The concept of object used as a basis for TROLL has been developed in [SSE87, SFSE89, SE90] accompanied by work on a categorical semantics [ES89, EGS90, ES91]. Based on this formal concept, work has been done towards a logical framework of structured theories over a suitable logical calculus [FSMS90, FM91]. First versions of the language introduced in this paper have appeared in [JSS91, SJ91a, SJ91b, JSH91].

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 attracting 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. [GR83, Mey88]) to design (for a survey see [MK90]) and has already entered analysis (e.g. [CY89, RBP+90]). In the database community, object-oriented databases have been very popular in recent years [DD86, Dit88, ABD+89, KL89]. According to traditional research issues, each community puts special emphasis on certain aspects of object-orientation [Ver91]

## 2.1 Conceptual Object-Oriented Modeling

Traditionally, many notations for conceptual modeling have been *entity-based* in the sense that they look at the world consisting of interrelated entities [Che76]. 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 [TN89]. Temporal aspects in system specification have been addressed by approaches to use *temporal logic* (see e.g. [MP89]) in conceptual modeling of databases and information systems [Ser80, Lip89, Saa91].

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. It is also useful to distinguish *disjoint* from *non-disjoint* complex objects. In non-disjoint complex objects, components may be shared with other complex objects, whereas in non-disjoint complex objects only belong to one complex object. This implies that the existence of components of disjoint complex objects is strongly dependent on the existence of the aggregation. For non-disjoint complex objects, components may exist independently from aggregations. Non-disjoint complex objects can be defined based on static properties (*static aggregation*) or operations to alter the composition (*dynamic aggregation*).

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.

## 2.2 Basic Ideas of Object Models

After having discussed the concepts for conceptual object-oriented modeling, we will take a short look at the mathematical structures being possible formal models for object systems. As mentioned already, an object can be regarded as an observable communicating process encapsulated by an access interface. Observations consist of reading its attribute values, object changes over time are driven by occurrences of its events. The event occurrence may be initiated by the object itself (active object) or as a result of a communication with another object (event calling). Both the values of attributes and the possible parameters of events are *data values* in the sense of abstract data type values.

The basis for a formalization of such an object concept are values structured using the concept of abstract data types (ADT's) [EM85, EGL89]. Data values are elements of the carrier set of some ADT. An ADT is defined by a carrier set for data values along with data type specific functions on this set.

For the formalization of object signatures, we use the notion of a signature from the ADT framework. An object signature consists of attribute and event symbols. Attributes are null-ary functions into a certain domain (a data type), events are functions with parameters and a special event sort as domain. The object signature defines the alphabet for an object specification in a formal framework.

The next step is the formalization of object evolutions using linear processes in a process framework. The reason we only consider linear processes is that we do not want to have intra-object concurrency, i.e. our objects are the units of concurrency. A possible choice is to adopt the life cycle model which defines a process as the set of possible snapshots of events. A snapshot is a set of concurrently occurring events. Special attention must be paid to events that create and destroy an object. Therefore we require the first snapshot in a life cycle to include at least one *birth* event. If the life cycle is finite, a *death* event is included in the last snapshot. Note that death events are not required, thus there may be objects that exist infinitely.

The last point to formalize for the specification of simple objects is the *observation of current object properties.* This is done by introducing an observation structure fixing attribute values for each reachable state of an object. A reachable object state is given by any finite prefix of possible object life cycles. The observation structure can be described as a mapping from object states (life cycle prefixes) to attribute-value relations [SSE87, ESS90].

Up to now we have only sketched a formalization of single objects. Another topic is the definition of object composition. Component relation between objects is modeled using structure preserving mappings between objects, the so-called object morphisms [ESS90, EGS90], which can be be compared to process combination known from process theory. As a special case we have inclusion morphisms where the mapping is supposed to be injective. Inclusion morphisms can be used to explain the embedding of subobjects into objects, i.e. aggregation of objects. The embedded objects are regarded as subprocesses in the enclosing composite object, where only events local to the subobject may have effects on attribute observations (of the subobject attributes).

The step from single objects to sets of objects is done by introducing an identification mechanism for object instances. Again we use ADT's to describe object identifiers. A class type defines a set of identifiers along with a prototype object model (an object template). An object class then consists of a class type and a set of object instances together with a mapping from a set of object identifiers to actual object instances. Note that it is possible to extend the notion of (*homogeneous*) class types by introducing a *set* of object templates for one class type, i.e. we may have *heterogeneous* class types [SSE87].

A formalization of these ideas can be found in [JSS91, SJ91b, JSH91]. For recent developments towards a more sophisticated semantic domain for object systems see [ES91].

# 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*
    **commitments** *state-dependent short-term goals*
    **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 predefined. We assume data types in general to be specified independently from object specifications in a suitable algebraic framework (e.g. [EM85]). 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 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 components 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 may e.g. state the following:

```
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;
```

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 [Ser80, Lip89, Saa91]. Dynamic constraints describe how the values of attributes may evolve in the future. Consider the initial constraint

```
initially ((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 [FS90]. 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) and (Red = true);
  { Red } => [withdrawal(m)]Balance = Balance + m;
  { Red and (m >= Balance)} =>
    [deposit(m)](Balance = m - Balance) and (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`. Please note that 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 conjunction:

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

Please note that 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);
  commitments
    variables t:bool; m:money;
    { after(accept_update(deposit,m)) } => deposit(m);
    { after(accept_update(withdraw,m)) and
      (m <= MaxWithdrawal) } => withdrawal(m);
```

Basically, we provide four 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 [Saa91]. 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;
```

The intuitive semantics of *commitments* is a bit more subtle. Commitments describe internal activity of objects. In our example, a request to update the account should be processed actively as soon as possible. As an example the following formula states that after a request for an update of type deposit was accepted the corresponding `deposit`-event instance should occur:

```
{ after(accept_update(t,m)) and t = deposit } => deposit(m);
```

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) and (Red = true);
    { Red } => [withdrawal(m)]Balance = Balance + m;
    { Red and (m >= Balance)} =>
      [deposit(m)](Balance = m - Balance) and (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);
    commitments
      variables m:money;
      { after(accept_update(deposit,m)) } => deposit(m);
      { after(accept_update(withdraw,m)) 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 [Hoa85] and

LOTOS [ISO84]. To illustrate the use of the process sublanguage consider the description of automatic teller machines (ATMs):

```
object class ATM
  identification
    data types nat;
    IdentNo:nat;
  template
    data types nat,money,bool,|CashCard|;
    attributes
      CashOnHand:money;
      derived Dispensed:bool;
    events
      birth set_up;
      death remove;
      ready; cancel;
      refill(in Amount:money);
      read_card(in C:|CashCard|);
      check_card_w_bank(in Acct:nat:, in PIN:nat);
      card_accepted; bad_PIN_msg; bad_account_msg;
      issue_TA(in Acct:nat,in Amount:money);
      TA_failed_msg; eject_card;
      dispense_cash(in Amount:money);
    constraints
      initially CashOnHand = 10000;
    derivation
      Dispensed = (CashOnHand < 100);
    valuation
      variables m:money;
      [refill(m)]CashOnHand = CashOnHand + m;
      [dispense_cash(m)]CashOnHand = CashOnHand - m;
    behavior
      permissions
        variables n:nat; m:money; C:|CashCard|;
        { not Dispensed } read_card(C);
        { m <= CashOnHand } issue_TA(n,m);
      patterns
        variables n,p:nat; m:money;
        process ATM_USAGE = read_card(C) -> choice
                                              cancel;
                                              check_card_w_bank(n,p)
                                                   -> GO_ON
                                            end choice -> eject -> ready
        end process
```

```
        where process GO_ON = choice
                                 bad_account_msg;
                                 bad_PIN_msg;
                                 card_accepted -> choice
                                                     cancel;
                                                     DO_IT;
                                                  end choice;
                              end choice
        end process
        where process DO_IT = issue_TA(n,m) -> choice
                                                  dispense_cash(m);
                                                  TA_failed_msg
                                               end choice
        end process
end object class ATM;
```

ATMs are identified by ident numbers. The observable state consists of the amount of cash on hand in the ATM. The events denote the start of a service-session (`read_card`), the request for checking a cash card (`check_card_w_bank`), various messages to the user of the ATM (`card_accepted`, `bad_PIN_msg`, `bad_account_msg`), the issuing of a transaction request to the bank (`issue_TA`), the eject of the inserted card (`eject`), the dispense of cash (`dispense`) and the refill of cash (`refill`).

The behavior of an ATM during a service-session is quite complex. An ATM only accepts a cash card if the cash on hand exceeds 100. After the card has been read, the session may either be canceled or it may go on with the corresponding `check_card_w_bank`-event. Please note that through the variables we are able to use the values read by the `read_card`-event, i.e. variables are not universally quantified but chosen by the environment. The keyword **choice** denotes an *external choice*, i.e. the decision for one of the alternatives is left to the environment. The `GO_ON`-pattern starts with another external choice which depends on whether the inserted card is valid or not (a decision which is not made by the ATM itself). If the card is accepted, the process may be canceled again or may go on with the launch of a transaction that can either be terminated with the dispense of cash or a failure message. That decision is again not made by the ATM but the environment. In any case, the service session with an ATM ends by the eject of the cashcard. Please note that an external choice requires communication with the environment that decides on how to proceed. This kind of communication is explained in section 5.

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 epistimologic concepts for the structuring of specifications.

# 4 Abstractions

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

## 4.1 Roles, Specialization and Generalization

The three concepts are related in the sense that they describe *is_a* relationships between object descriptions, i.e. each instance of a role/specialization/generalization 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 [Wie90]. 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

Let us now consider a special case of a role which is called *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 (static) 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).

### 4.1.3 Generalization

Generalization is used to construct a uniform view on (several) different base classes. Consider for example the very popular example of legal persons being either 'real' persons or companies. Basically, generalization is similar to the second is-a concept specialization. The difference is that it goes in the other direction: Using specialization, one defines a number of subclasses for a base class whereas by generalization, one defines a common superclass for a number of subclasses. The issue of specialization vs. generalization has been discussed in the semantic data modeling community for some time [SS77, UD86, HK87, PM88].

Consider the following example. Suppose the bank wants to treat persons being customers and companies being customers in a uniform way. We may then define a class `Customer` that generalizes the classes `BankCustomer` and `Company` (which is not defined in this paper).

```
object class Customer
  generalizing BankCustomer, Company:
    template
      data types nat,set(nat),string,|Customer|;
      attributes
        derived Accts:set(nat);
        derived Address:string;
      events
        derived open_acct(out Acct:nat);
        derived close_acct(in Acct:nat);
```

```
derivation
  variables C:|Customer|; n:nat
  { in-class(C,BankCustomer) } =>
      Accts = BankCustomer(C).Accts and
      Address = BankCustomer(C).Address and
      open_acct(n) == BankCustomer(C).open_account(n) and
      close_account(n) == BankCustomer(C).close_account(n);
  { in-class(C,Company) } =>
      Accts = Company(C).Accounts and
      Address = Company(C).Location and
      open_acct(n) == Company(C).new_acct(n) and
      close_acct(n) == Company(C).delete_acct(n);
end object class Customer;
```

Please note that *all* attributes and events of the generalized class `Customer` are derived from the corresponding attributes and events of the base classes. The predefined predicate **in-class** evaluates to true if the instance identified by an identifier is a member of the indicated class. For the derivation of events, we use event sharing since we want the derived events to be identified with their corresponding base events.

## 4.2 Complex Objects

Using the aggregation concept, we may construct objects from components. In the database community, this is also known as constructing *complex objects*. Basically, we can identify two kinds of complex objects [BB84]:

- *Disjoint* complex objects do not share any components. This implies that components cannot exist outside the complex object, they are strongly dependent and the components are *local* to the complex object. The composition is always static.
- *Non-disjoint* complex objects may share components. Thus, components are autonomous objects. In TROLL, we distinguish two kinds of non-disjoint complex objects: Dynamic complex objects may alter their composition through events whereas the composition of static complex objects is described through static predicates.

For disjoint and non-disjoint complex 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 complex object and its components or between the components must be done by communication. Let us now continue with the concepts of non-disjoint complex objects, that is static and dynamic aggregation.

### 4.2.1 Static Aggregation

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

In section 2 we sketched the concept of embedding subobjects in objects. Object embedding can be used directly to describe static aggregation. All properties of the embedded objects are preserved. Attribute values of the embedded objects can only be altered by events local to the subobjects. Interaction between embedded objects or embedded objects and the complex object must be achieved by synchronous communication. These interactions may not violate the life cycle specification of the communicating objects.

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 Bank1
  template
    including A in Account;
    data types |Account|,set(|Account|),nat,UpdateType,money;
    attributes
      TheAccounts:set(|Account|);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      process_TA(in Acct:nat,in Type:UpdateType,in Amount:money);
      TA_failed(in n: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);
        { not sometime after(process_TA(n,t,m)) since last
          (after(TA_failed(n)) or after(TA_OK(n)))} process_TA(n,t1,m1);
        /* Each Transaction must be completed before */
        /* the next for account n can start */
      interactions
        variables t:UpdateType; m:money; n:nat;
        open_account(n) >> Account(n).open;
        close_account(n) >> Account(n).close;
        process_TA(n,t,m) >> Account(n).accept_TA(t,m);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).withdrawal(m) >> TA_OK(n);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).deposit(m) >> TA_OK(n);
        { sometime after(process_TA(n,t,m) } =>
            Account(n).update_failed >> TA_failed(n)
end object Bank1;
```

In the interaction section communication between the composite object `Bank1` 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 `Bank1` 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 `Bank1` 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 `Bank1` object is obtained by disjoint union of the (local) signatures of the `Bank1` 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 complex 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. For observations of the complex object projected to the attributes of the component we must obtain the same observation as for applying the observation mapping of the component to a life cycle of the composite object restricted to the events of the component.

### 4.2.2 Dynamic Aggregation

The use of dynamic complex 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. The components of the dynamic complex object are behaviorally independent from the aggregation. They have a life of their own and may be shared by other objects as well.

With the specification of a dynamic complex 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 complex object construction can be found in the upcoming language report.

This view of complex 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 Bank
  template
    data types nat,|ATM|,UpdateType,money,|CashCard|;
    components
      Accounts:SET(CheckingAccount)
    events
```

```
  birth establish; death close_down;
  open_account(in No:nat);
  close_account(in No:nat);
  verify_card(in Acct:nat,in PIN:nat,in ATM:|ATM|);
  no_such_account(out ATM:|ATM|);
  bad_PIN(out ATM:|ATM|);
  card_OK(out ATM:|ATM|);
  process_TA
    (in Acct:nat,in Type:UpdateType,in Amount:money,in ATM:|ATM|);
  TA_failed(out ATM:|ATM|);
  TA_OK(out ATM:|ATM|);
  card_request(in AcctNo:nat);
  card_return(in C:|CashCard|);
behavior
  permissions
    variables n,p,p1:nat; m,m1:money; t,t1:UpdateType; atm:|ATM|;
    { not Accounts.IN(CheckingAccount(n)) } open_account(n);
    { sometime after(open_account(n)) } close_account(n);
    { not sometime after(process_TA(n,t,m,atm)) since last
      (after(TA_failed(n,atm)) or after(TA_OK(n,atm)))
      and Accounts.IN(CheckingAccount(n))} process_TA(n,t1,m1,atm1);
    /* Each Transaction must be completed before */
    /* the next for account n can start */
    { not sometime after(verify_card(n,p,atm)) since last
      (after(bad_PIN(atm)) or after(no_such_account(atm))
      or after(card_OK(atm)))} verify_card(n,p,atm);
    /* Each verification for a particular account must */
    /* be completed before the next can start */
    { sometime after(verify_card(n,p,atm)) and
    Accounts.IN(CheckingAccount(n)) and
    (Accounts(CheckingAccount(n)).PIN = p)} card_OK(atm);
    { sometime after(verify_card(n,p,atm)) and
      not Accounts.IN(CheckingAccount(n))} no_such_account(atm);
    { sometime after(verify_card(n,p,atm)) and
    Accounts.IN(CheckingAccount(n)) and
    not(Accounts(CheckingAccount(n)).PIN = p)} bad_PIN(atm);
    commitments
      {after(verify_card(n,p,atm))} =>
        (bad_PIN(atm) or no_such_account(atm) or card_OK(atm));
  interaction
    variables t:UpdateType; m:money; n:nat; C:|CashCard|;
    open_account(n) >> Accounts.INSERT(CheckingAccount(n));
    open_account(n) >> Account(CheckingAccount(n)).open;
    close_account(n) >> Accounts.REMOVE(CheckingAccount(n));
    close_account(n) >> Account(CheckingAccount(n)).close;
    { Accounts.IN(CheckingAccount(n)) and
      not in(C,CheckingAccount(n).Cards) } =>
      card_request(n) >> Accounts(CheckingAccount(n)).assign_card(C);
    card_return(C) >>
```

```
      Accounts(CheckingAccount(CashCard(C).ForAccount)).cancel_card(C);
    process_TA(n,t,m,atm) >>
      Accounts(CheckingAccount(n)).accept_TA(t,m);
    { sometime after(process_TA(n,t,m,atm) } =>
      Accounts(CheckingAccount(n)).withdrawal(m) >> TA_OK(n,atm);
    { sometime after(process_TA(n,t,m,atm) } =>
      Accounts(CheckingAccount(n)).deposit(m) >> TA_OK(n,atm);
    { sometime after(process_TA(n,t,m,atm) } =>
      Accounts(CheckingAccount(n)).update_failed >> TA_failed(n,atm)
end object Bank;
```

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

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

```
open_account(n) >> Accounts.INSERT(CheckingAccount(n));
open_account(n) >> Account(CheckingAccount(n)).open;
```

which state, that every time an account identified by the natural number n is opened, it becomes a member of the set of components and the event open is called in the corresponding object CheckingAccount(n) which is inherited from the more general account specification (see page 9). Note that the event open_account in the Bank object may only occur if the expression **not** Accounts.IN(|CheckingAccount|) evaluates to true. As an example for a conditional calling in the opposite direction consider the following expression:

```
{ sometime(after(process_TA(n,t,m,atm))) } =>
    Accounts(CheckingAccount(n)).update_failed >> TA_failed(n,atm)
```

This clause states, that sometime after the event process_TA(...) occurred in the Bank object, the effect of the event update_failed in the component object CheckingAccount(n) is the calling of event TA_failed(...) in the Bank object. Intuitively the calling only takes place if the specified condition holds.

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 complex 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 complex object. Whenever the composition changes, a new complex 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.

Consider for example the Bank object containing a set of CheckingAccounts. Just to explain the idea, we may construct an *object class* ComplexBank identified by instances of **set**(|CheckingAccount|). For each object of this class its structure is defined by embedding all CheckingAccounts that are elements of its (external) identifier. Changing the composition thus yields a new object instance. We require this new instance to have the same observable state as the old instance. The old object can be destroyed. This view implies a change of object identity whenever the composition of the complex object changes. However this change is not visible at specification level [KC86].

### 4.2.3 Disjoint Complex Objects

Disjoint complex objects may not share components with other objects. The parts of a disjoint complex object are strongly dependent on the composition. They cannot exist independently from the aggregation. In TROLL, disjoint complex objects are described using *subtemplates*. Subtemplates are a means for structuring the specification. Conceptually, subtemplates describe local subobjects not usable outside the complex object.

For example let us specify a class of banks each containing a set of divisions. The first step is to model the Division template:

```
template Division
  data types |Person|,...;
  attributes
    Manager:|Person|;
    ...
  events
    birth openDivision;
    death closeDivision;
    setManager(|Person|);
    ...
  valuation
    variables p:|Person|;
    [setManager(p)] Manager = p;
    ...
end template Division;
```

This specification fragment shows a division of a bank having an attribute Manager which can be changed using the event setManager(|Person|). Divisions can be created with the birth event openDivision and destroyed with the death event closeDivision. In the second step the template specification of divisions can be used as subtemplates in a possible bank object:

```
object class Bank2
  identification
    data types nat;
    no:nat;
  template
    data types string,|Person|,...;
    subtemplates
      Divisions(string):Division;
    ...
    events
      newDivisionManager(string,|Person|);
      ...
```

```
    interaction
      variables d:string, p:|Person|;
      newDivisionManager(d,p) >> Division(d).setManager(p);
      ...
end object class Bank2;
```

Inside the Bank template we have a possible Division template for each value of the internal name space, in this case for each possible string. We can use the components specified in the subtemplate by supplying the subtemplate name and its internal identification. For example the clause:

```
  newDivisionManager(d,p) >> Division(d).setManager(p);
```

states, that an event denoting the advancement of a person p to be the new division manager for division d calls for the event setManager(p) in the subobject Division(d).

Note that different Bank instances can have the same names for their divisions. The names of the subobjects are local to the enclosing object as are the objects themselves. They may not be shared with other bank instances.

One way to explain the semantics is static aggregation again. Objects specified using subtemplates may be transformed to simply structured objects. Therefore we may generate (internal) object class descriptions employing the identification space of the original object together with the internal template identification. The template of this new object class is obtained from the subtemplate specification. For example the Bank2 object may be transformed to the object class Division_Internal:

```
object class Division_Internal
  identification
    data types |Bank2|,string;
    Bank2:|Bank2|;
    internalID:string;
  template Division
end object class Division_Internal;
```

and the modified object class Bank2 including all instances of Division_Internal belonging to the current instance:

```
object class Bank2
  identification
    data types nat;
    no:nat;
  template
    including Division_Internal D where D.id.Bank2 = SELF.id;
  ...
end object class Bank2;
```

Note that the last two specifications are not visible to the designer. Each object instance of the class Division_Internal may be shared only by exactly one Bank2 class object. This property must be guaranteed with the including condition in the modified Bank2 object. The condition is also an example for using predicates to describe the aggregation of static complex objects.

An alternative semantics for disjoint complex objects can be obtained by *flattening* the specification. In this case, the internal identification mechanism of subtemplates will be used to index its attribute and event symbols. These indexed symbols are then added to the signature of the enclosing complex object. Thus in the `Bank` object we have events `openDivision(string)`, `setDivision(string)`, ... and attributes `Manager(string)`.

# 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 may 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 [MBW80]). As usual, communication is described using event calling and event sharing.

As an example consider the description of interactions between an ATM and the bank. The relationship describes all communications that are necessary to carry out remote bank transactions from an ATM:

```
relationship RemoteTransaction between Bank, ATM;
  data types |ATM|,nat,money,UpdateType;
  interaction
    variables atm:|ATM|; n,p:nat; m:money;
    /* Card checking business */
    ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
    Bank.bad_PIN(atm) >> ATM(atm).bad_PIN_msg;
    Bank.card_OK(atm) >> ATM(atm).card_accepted;
    /* bank transaction business */
    ATM(atm).issue_TA(n,m) >> Bank.process_TA(n,withdraw,m,atm);
    Bank.TA_failed(atm) >> ATM(atm).TA_failed_msg;
    Bank.TA_OK(atm,m) >> ATM(atm).dispense_cash(m);
  end relationship;
```

From a process point of view, such a relationship describes how the involved processes *synchronize*. For the business of checking a cashcard inserted into an ATM, we may specify the first four clauses. The event `check_card_w_bank` occurring in an ATM denotes a request to the bank to verify the inserted cashcard. Please note that we have to use event calling here since we do not want to identify the `verify_card` event of the bank with the `check_card_w_bank` event of *each* ATM. The other three clauses concern the result of the card checking at the bank which must be transmitted to the corresponding ATM. The bank transaction business is described in an analogous way.

In interaction specifications, we may want to refer to the history of events in the connected objects. Consider the interaction between an ATM customer (of which the specification may become the description of a user interface later) and an ATM. Here, we must put precedence rules into conditions for interactions to model the process of communication:

```
relationship UseATM between ATMCustomer,ATM;
  data types |ATMCustomer|,|ATM|,|CashCard|,nat,money;
  interaction
    variables C:|ATMCustomer|; atm:|ATM|; CC:|CashCard|; p:nat, m:money;
    Customer(C).insert_card(CC,atm) >> ATM(atm).read_card(CC);
    { sometime after(Customer(C).insert_card(CC,atm)} =>
      Customer(C).enter_PIN(p,atm) >>
        ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
    Customer(C).enter_cancel(atm) >> ATM(atm).cancel;
    { sometime after(Customer(C).enter_PIN(p,atm))} =>
      ATM(atm).card_OK >> Customer(C).card_accepted(atm);
    { sometime after(Customer(C).insert_card(CC,atm))} =>
      Customer(C).enter_amount(m,atm) >>
        ATM(atm).issue_TA(CashCard(CC).ForAcct,m,atm);
    { sometime after(Customer(C).enter_amount(m,atm)) } =>
      ATM(atm).dispense_cash(m) >> Customer(C).cash_dispensed;
    Customer(C).take_cash(atm) >> ATM(atm).eject_card;
    { sometime after(Customer(C).insert_card(CC,atm))} =>
      ATM(atm).eject_card >> Customer(C).take_card(atm);
    Customer(C).take_card(atm) >> ATM(atm).ready;
end relationship;
```

For precedence rules, we may use the **after** predicate. Take e.g. the following clause:

```
{ sometime after(Customer(C).insert_card(CC,atm)} =>
    Customer(C).enter_PIN(p,atm) >>
      ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
```

It states that once a particular ATM customer inserted a cashcard, the input of the personal id number (PIN) calls for the `check_card_w_bank` event in the ATM (which itself calls for the `verify_card` event in the `Bank`).

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). 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 be 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 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 [SJ91b].

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. 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 [SE90, ES89] 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) -> issue_TA(n,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 two bank transactions will occur. The first one `issue_TA(n,m)` to withdraw the amount of money requested by the user. The second one `issue_TA(n,extraCharge)` denotes the extra charge. The derivation of events is done using arbitrary process calling, which again is part of a formal implementation step hidden from the view user. In this case the `->` between the two events denote sequential composition of events. The angle brackets denote a transactional requirement: either both events may occur or none of them may occur. If for example the second event cannot take place because there is no money left, the event `issue_extra_TA(n,m)` will be rejected.

Please note that 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), short-term initiatives (commitments) or even 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 tries to combine 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. TROLL is a semi-formal language. Syntactic sugar has been added in quite large amounts to make the language more user-friendly than a pure logical calculus.

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.

This work is accompanied by work on a logical calculus being the formal background for TROLL specifications.

# Acknowledgements

For many fruitful discussions on the language we are grateful to all members of IS-CORE, especially to Amilcar Sernadas, Hans-Dieter Ehrich, Jose Fiadeiro, and Egon Verharen.

# 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.

[BB84] Batory, B.; Buchmann, A.: Molecular Objects, Abstract Data Types and Data Models: A Framework. In: *Proc. VLDB'84*, Singapore, 1984. pp. 172–184.

[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.

[Bor85] Borgida, A.: Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, Vol. 2, No. 1, 1985, pp. 63–73.

[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.

[CY89] Coad, P.; Yourdon, E.: *Object-Oriented Analysis.* Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1989.

[DD86] Dittrich, K. R.; Dayal, U. (eds.): *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986. IEEE Computer Society Press, Washington, 1986.

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

[EDG86] Ehrich, H.-D.; Drosten, K.; Gogolla, M.: Towards an Algebraic Semantics for Database Specification. In: *Proc. 2nd IFIP WG 2.6 Working Conf. on Knowledge and Data (DS-2)*, Albufeira (Portugal), 1986. North-Holland, 1988, pp. 119–135.

[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.

[EGL89] Ehrich, H.-D.; Gogolla, M.; Lipeck, U. W.: *Algebraische Spezifikation abstrakter Datentypen.* Teubner Verlag, Stuttgart, 1989.

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

[Ehr86] Ehrich, H.-D.: Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. In: Pitt, D. et al. (eds.): *Proc. Workshop on Category Theory and Computer Programming.* Springer Verlag, Berlin, 1986, pp. 412–433.

[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.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: *This volume.*

[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.

[FM91] Fiadeiro, J.; Maibaum, T. S. E.: Towards Object Calculi. In: *This volume.*

[FS90] Fiadeiro, J.; Sernadas, A.: Logics of Modal Terms for System Specification. *Journal of Logic and Computation*, Vol. 1, No. 2, 1990, pp. 187–227.

[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.*

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

[Gri82] van Griethuysen, J.: Concepts and Terminology for the Conceptual Schema and the Information Base. Report N695, ISO/TC97/SC5, 1982.

[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.

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

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

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T. (ed.): *Proc. 10th Int. Conf. on the ER-Approach*, San Mateo (CA), 1991. To appear.

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

[KC86] Khoshafian, S.N.; Copeland, G.P.: Object identity. In: *Proc. OOPSLA Conference*, Portland, OR, 1986. ACM, New York, 1986, pp. 406–416. (Special Issue of SIGPLAN Notices, Vol. 21, No. 11, November 1986).

[KL89] Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989.

[Lip89] Lipeck, U. W.: *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung.* Informatik-Fachbericht 209. Springer-Verlag, Berlin, 1989.

[MB89] Mylopoulos, J.; Brodie, M. (eds.): *Readings in Artificial Intelligence & Databases.* Morgan Kaufmann Publ. San Mateo, 1989.

[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.

[MP89] Manna, Z.; Pnueli, A.: The Anchored Version of the Temporal Framework. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.* LNCS 354, Springer-Verlag, Berlin, 1989, pp. 201–284.

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

[Pnu77] Pnueli, A.: The Temporal Logic of Programs. In: *Proc. 18th IEEE Symp. Found. of Computer Science*, 1977, pp. 46–57.

[RBP+90] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991, pp. 47–73.

[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.

[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.

[SJ91a] Saake, G.; Jungclaus, R.: Konzeptionelle Modellierung von Objektgesellschaften. In: Appelrath, H.-J. (ed.): *Proc. Datenbanksysteme für Büro, Technik und Wissenschaft BTW'91*, Kaiserslautern, 1991. IFB 270, Springer-Verlag, Berlin, 1991, pp. 327–343.

[SJ91b] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL Language. In: Harper, D. (ed.): *Proc. Int. Workshop on the Specification of Database Systems*, Glasgow, 1991. Springer-Verlag, London, 1991. *In print.*

[SS77] Smith, J.M.; Smith, D.C.P.: Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, Vol. 2, No. 2, 1977, pp. 105–173.

[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.

[TN89] Tsichritzis, D. C.; Nierstrasz, O. M.: Directions in Object-Oriented Research. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 523–536.

[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.

[Ver91] Verharen, E.: Object-Oriented System Development – An Overview. In: *This volume.*

[Wie90] Wieringa, R. J.: *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.