Revised Version of the Conceptual Modeling and Design Language TROLL*

(Extended Abstract)

Thorsten Hartmann‡
Jan Kusch†
Gunter Saake†
Peter Hartel‡

Abstract

Conceptual modeling of complex information systems requires the use of a formal design approach covering both static and dynamic aspects of the system and the modeled Universe of Discourse. A language for conceptual modeling is regarded as a means to write contracts between customers and system designers. Thus on the one hand a certain degree of *readability* of specifications is necessary for the domain specialist and on the other hand formality is necessary to derive information for further design, implementation, and prototyping issues.

In this extended abstract we present the modeling language TROLL (the successor version of the language introduced in [JSHS91, JHSS91]) that provides a notation close to object oriented programming languages but also incorporates declarative features for high level design documents. TROLL provides abstractions like classes (*collections*), specialization and roles (*is-a*), composite objects (*part-of*). Furthermore TROLL provides views on objects like projection (*hiding*) and selection (*class subsets*), and object relationships (*communication channels*) that together are used to structure a design document. The core of an object description is made up of attributes, events, and components specification together with a behaviour specification that separates constraints for the observable state using first order and future temporal logic assertions and process specification in a CSP like fashion. The former is mainly used in early specification phases, whereas the latter is used for a more explicit and operational life cycle description.

1 Introduction

1.1 Information System Development

Information systems represent the relevant aspects of a portion of the real world (*UoD*) 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 into the UoD. The information to be stored changes over time according to interactions with the environment or predefined internal actions. Thus, information systems are *dynamic* in the sense that they may be regarded to be reactive systems [Pnu77].

Nowadays, information systems represent increasingly large and complex UoD's. Thus, adequate means to support their *specification* are becoming more and more important. The specification starts with collecting and representing *knowledge* about the UoD, which covers 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. However, as in engineering, the design

*This work was partially supported by the CEC under ESPRIT-III BRA WG 6071 IS-CORE (Information Systems – COrrectness and REusability). The work of Thorsten Hartmann and Jan Kusch is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3. The work of Peter Hartel is supported by OBLOG Software SA, Lisboa, Portugal.

†Institut für Technische Informationssysteme, Universität Magdeburg, POBox 4120, D–39106 Magdeburg, E-mail {kusch|saake}@iti.cs.tu-magdeburg.de

‡Abteilung Datenbanken, Technische Universität Braunschweig, POBox 3329, D–38023 Braunschweig, E-mail {hartel|hartmann}@idb.cs.tu-bs.de

process should produce *models* of solutions that can be assessed *formally* and *experimentally* before any concrete system is implemented – the former by means of a suitable logic for consistency checking the latter by means of translating specifications into an executable form for prototyping.

In this paper, we give an overview of the revised version of the language TROLL introduced in [JSHS91, JHSS91]. The approach evolved from integrating work on algebraic specification of data types [EM85, EGL89] and databases [Ehr86, EDG88], process specification [Hoa85, Mil89], the specification of reactive systems [Pnu77, Ser80, MP92, Saa91], conceptual modeling [Che76, MBW80, BMS84, Bor85, HG88, EGH+92] and knowledge representation [BM86, ST89, MB89]. The concept of object used as a basis for TROLL has been developed in [SSE87, SFSE89, SE91] accompanied by work on a categorical semantics [ES90, EGS90, ES91]. More recently steps towards a logical semantics [Jun93] based on *OSL* [SSC92] and towards refinement of specifications have been done [EDS93].

Close to the language described here is the language TROLL*light* that is based on a similar object model but tailored towards *verification* [CGH92, GCH93, Con94]. Syntactically TROLL*light* resembles the former version of TROLL.

1.2 Development Process and Prototyping

In the process of constructing an information system the step following the requirements analysis and conceptual modeling phase is known as *design* phase. During design the internal structure of the system to be implemented is chosen so that the following implementation can be performed using traditional implementation techniques like programming languages, data definition/manipulation languages, existing resources like communication platforms etc.

Although this straightforward process – in the software engineering community called *waterfall model* [MR91] – is appealing due to clearly defined interfaces between different specification documents, it is generally accepted that such an approach is seldom practical because [Hum90]:

- it does not adequately address changes,
- usually the development steps are not in such an orderly sequence, and
- it does not provide possibilities to integrate prototyping.

Especially if the requirements for a system under construction are not clear at the start of a project, a prototyping approach is necessary to clearly discover the needs (explorative prototyping [BP92]). Moreover, prototyping can be employed to test parts of the system during the design process (experimental prototyping [BP92]). The language TROLL depicted here is suited to support the latter form: experimental prototyping. Thus it should be possible to design parts of a system, test its behaviour patterns by means of a simple prototype execution and go back or proceed in the design phase (or even again in the conceptual modeling phase).

It is desirable that such a backtracking only involves clearly defined parts of the systems specified/designed so far and that the reiteration only affects small amounts of specification text. A *modularization* in early phases of modeling is therefore needed and is supported by a suitable notion of *object*. TROLL supports this needs with a strong notion of locality for state changes and a separation of object descriptions from communication relationships that are defined between objects or views on objects.

2 The Object Concept

Before we introduce the language and the motivation for the revised version let us have a short look at the underlying ideas. For a detailed description the reader may refer to [JSHS91, JSS91, HJS92, HS93]. Objects are the inseparable design units. An object has an *internal state* of which certain properties can be observed. The state can be manipulated exclusively through an *event interface*.

In contrast to object-oriented programming languages that emphasize a functional manipulation interface (using methods), object-oriented database approaches put emphasis on the observable structure of objects (through attributes). For the task of specifying information systems, we propose to support both views in an equal manner for the design of objects, i.e. object structure may be observed through attributes and the object state may be manipulated through events.

The behaviour of an object is then defined as a linear process consisting of the set of possible traces of *event* occurrences. More particular we regard *sets of concurrent events* (*snapshots*). Snapshots are needed for the modeling of synchronous communication. In terms of this process definition the internal state of an object is defined as a finite prefix of a possible snapshot trace, whereas an observation is defined as a mapping from such traces to a set of attribute-value pairs. In this sense, objects are *observable processes*. An example event trace can be expressed pictorially as follows:

$$\underbrace{\overset{\substack{\textit{birth}\\ \downarrow}}{\langle b\rangle} \longrightarrow \overset{\substack{\textit{event}\\ \textit{snapshot}\\ \downarrow}}{\left\langle \begin{array}{c} e_1 \\ \dots \\ e_n \end{array} \right\rangle} \longrightarrow \cdots \longrightarrow \langle e_k \rangle}_{\textit{Life cycle prefix}} \longrightarrow \overset{\substack{\textit{death}\\ \downarrow}}{\langle d \rangle}$$

where the observations by means of attributes are located 'between' the snapshots, i.e. after each snapshot we may observe altered attribute values. Events in a snapshot may be either related through a triggering mechanism (*event calling*, see below), or be unrelated as occurring concurrently in an object.

Up to now we only talked about single objects. To come from single objects to object classes as collections of similar objects, we have to introduce *object names* or *identities*. For the time being identities are just *values* of an object *identifier data type*. They associated to each object instance and are used to distinguish between different objects.

For the modelling of real world entities, an important topic is the description of objects composed of part objects. Since objects are defined as observable processes, we must define object composition as a combination of processes and observations. Aggregated objects are regarded as subprocesses embedded in a composite process. The observation of a composite object is then the sum of the observations of the parts where observations of the parts calculated in isolation are maintained. In other words: local attribute changes are defined solely by local events (encapsulation).

Objects somehow communicate with each other. Whereas object-oriented programming languages realize communication with message passing, we use the more abstract concept of *event calling*, characterized as synchronous, directed communication between processes. As mentioned earlier, snapshots are constructed from events that are related by calling (i.e. synchronous triggering of events). For details on these issues see [HS93].

In the next two sections we will now proceed with language features of Troll. First the specification of classes, specialization, composite objects, views on objects, and relationships are introduced. Then we will show how objects themselves are specified with attributes, events, components and how possible life cycles of objects are described.

3 Abstraction Principles in Troll

In this section we will introduce the basic abstraction mechanisms of Troll [HSJ+94] that are used to model the relevant parts of real world objects in terms of a formal language. These principles are classes, roles, composite objects, views on objects and synchronization relationships between separately specified objects.

To talk about objects in some formal language (like Troll) we firstly have to introduce a suitable vocabulary. As for algebraic specification of data types [EM85], we adopt the notion of a signature to define a *set of symbols* that are used to describe various properties of objects later on. For a formal language to specify objects in the context of information systems we have to describe different kinds of abstractions, thus we need different kinds of symbols.

First of all objects in information systems have properties that can be *observed* from other objects. Symbols to describe this part of objects are usually called instance variables or state variables. In Troll such symbols are called *attributes*. To describe not only the observable part of objects, we have to specify the *state changes* an object can undergo during its life time. In Troll *event symbols* denote such state changes. Last but not least objects can be composed of other objects. For this case we introduce *component symbols* as a kind of reference to part objects that are to be distinguished from attributes. Attributes in Troll describe *simple properties* of objects whereas components describe close associations of objects in terms of *part-of relationships*.

In contrast to the previous language version [JSHS91] the specification of attribute, event, and component features is clustered around the *signature introduction*. In Section 4 we will introduce the detailed specification of attributes, events and components and their use in the *template specification* of an object.

3.1 Classes

Usually if we model objects in some formal language we abstract away differences between objects in the real world and group objects together that are related by a similar structure and behaviour. Object oriented languages introduce the notion of *classes* for this purpose. An object class in Troll is described as a *set of objects* that behaves like modelled in a common *template*.

To distinguish between objects of a class we introduce a *naming mechanism* that is based on observable object properties. For the time being let attributes be just data valued property descriptions. Tuples of attribute values (called an *identification*) are thus defined as *keys* to objects of a class that have a semantics in the real world context of the object being described formally.

Apart from such attribute values, each object of a given class has a unique *identity*. If we know a class and an identification of the corresponding real world object we can *refer* to the (representation of the) object. As an example we may specify the naming mechanism for the familiar class `Person` as follows:

```
object class Person
  identification
     ByName:(Name,BirthDate) ;
     BySocSecNo:(SocialSecNo)
  ...
end object class Person
```

Where the symbols `Name`, `BirthDate`, and `SocialSecNo`, denote attributes defined in the template of `Person` (introduced below and not shown here), and the tuples `ByName` and `BySocSecNo` defining (named) keys for persons. We have than available two mechanisms to *refer* to objects of class `Person`. Implicitly with a class specification a data type for *identities* is provided. In this example this data type is called `|Person|`. To refer to some property of instances of `Person` with identity `x` $\in$ `|Person|` we write:

```
Person(x).some property
```

Alternatively we can use the *key maps* `ByName` or `BySocSecNo` if we want to refer to persons in a real world context – for example if we want to refer to it during a prototype execution – we can use a tuple `(Name,BirthDate)` to refer to John Smith, born July 29, 1994 as:

```
Person(ByName(''John Smith'', July 29, 1994)).some property
```

This second reference is thus closer to the real world (interface) whereas the former is often used 'inside' a specification document to refer to some object instance with identity `x` $\in$ `|Person|`. Formally the key map `ByName` is an injective mapping from real world object names to identities. We call the former external the latter internal object names (with respect to the specification).

As an optional feature for class specification we can specify *single objects*. Therefore the keyword class as well as the identification declaration is omitted. We refer to single objects just by their name.

3.2 Object Roles and Specialization

The notion of classes as the only abstraction principle is not sufficient if we want to model real world entities since usually an abstraction in terms of similar structure and behaviour still yields object descriptions that are not tractable with respect to size and complexity. Object oriented programming languages introduce the notion of *inheritance* for structuring code and to support code reuse. Troll introduces the notion of *roles* to factor out structure and behaviour of objects according to different viewpoints or aspects [ESS92] and in different phases of their life (see also [Wd92]). A role differs from a specialization in that it is valid in certain phases of a life cycle only.

As in programming languages, role classes in Troll *inherit* the specification of their base classes. Moreover role class objects in Troll *contain* their base objects leading to an inheritance mechanism based on *delegation* [Ste87]. Objects of role classes may be born sometime during the lifetime of their parent objects due to occurrence of *events*. Similarly they leave a role with the occurrence of events.

The concept of roles and specialization resembles the *is-a* hierarchies known from semantic data models [HK87]. In Troll the *is-a* relationship is integrated with the behaviour of objects in that a role object *is-a* base object for a specific time span during an object life cycle. In contrast to a simple textual inheritance of specification code the objects themselves are *inherited*. Employing more technical, implementation related terms: in contrast to syntactic copying of specification code to subclasses (or role/specialization classes) requests to role/specialization properties have to be forwarded to the parent instances.

As an example we may specify a class `Employee` as a role of `Person` as follows:

```
object class Employee role of Person
  events
    becomeEmployee birth .
end object class Employee
```

where an *event* `becomeEmployee` is introduced as the *creation event* for `Employee` objects. For a more detailed description of events see the next section. The construction event `becomeEmployee` is specified in the role class *Employee* to stress the belonging of this event to the description of employees. From a conceptional view the event belongs to a real world person entity. From the specification viewpoint this real world knowledge is spread across the role/specialization hierarchy.

A role class *inherits* also the naming mechanism, that is, identities of *Person*s are also identities of *Employee*s and we can refer to `Employee`s via their `SocialSecNo` for example. Such a reference is only allowed however if the referred role is valid at the time of access, i.e. the object 'plays' the role at that time.

A special case of roles is a role that is 'played' for the *whole life of the base class* – a *derived* role class like `Woman` also called a specialization:

```
object class Woman
  role of Person P derived as P.Sex = 'female'
  ...
end object class Woman
```

Here a creation event must not be specified since it is *derived* for a role class via the derivation condition. The derivation condition is a predicate over the parent class properties that are inherited. The treatment of roles and specialization in one framework of roles is done for incorporating class changes even for specializations in case of derivation properties containing changeable attributes. Such a concept can be 'implemented' with generated birth and death events that are dependent on the changing of the properties used in the predicate.

We are aware of the limitations of this approach with respect to *counting* entities and e.g. simultaneous `Employee` roles of one `Person` as described in [Wd92]. The approach here is adopted to keep the language as simple and orthogonal as possible. The role concept will be a field of future development however.

3.3 Composite Objects

The use of classes and roles as abstractions is not sufficient yet to map real world entities to objects because nearly all objects which we observe are *composed of parts*. Basically we can distinguish two kinds of component objects, *dependent* and *sharable* components. The former are local to an object in that they cannot be parts of other objects. They are also dependent with respect to the life of the surrounding object. The latter are independent and may be part of *different* objects.

TROLL supports both views by means of components drawn from *local* or *global* classes. A local class is specified the same way as a global class. The uniqueness of identification is supported in the context of *one* object only. For example a company class can contain a local department class with objects identified by department names where different companies have departments with the same name. In TROLL the specification of local classes has to be done *inside* the specification of the surrounding object just to stress the close relationship between part and whole of dependent subobjects.

Orthogonally to dependent and sharable components TROLL supports the component constructors *set* and *list* that can be used to describe sets respectively lists of component objects taken from a given class. The composition of a composite object can be altered by events that are *generated* with a component specification. Such events are for example insert respectively remove events for a given component. As an example we specify a `Company` class containing a local class `Department` as sketched above:

```
object class Company
  identification ByName:(Name)
  local classes
    object class Department
      identification ByDepName:(CompName)
```

```
    ...
    end object class Department
  components
    Deps:Department set .
  ...
end object class Company
```

The Deps component is specified as a *set component*. Insert and delete events for such a component are implicitly available, for example an event Deps.Insert(x) where x denotes an identifier for Department objects. For more details how objects are referenced the interested reader may refer to [HSJ+94]. Note that the component symbols are part of the signature of objects and are used in constructing *path expressions* referencing objects chained together in *part hierarchies*.

Referencing part objects is written orthogonally to referencing objects in classes. Viewing classes as *collections* of objects is carried over to viewing (set) components as collections of subobjects. We therefore regard the (local) component name as a name for the collection of subobjects and may write

Deps(d).*some_property*

for a $d \in |\mathtt{Department}|$ or

Deps(ByName(``Sales'')).*some_property*

The reference is only valid if the referred objects have become subobjects by means of occurrences of Insert events inside the surrounding objects *before* such a reference takes place.

3.4 Views on Objects

The mechanisms sketched so far are used to introduce the core of an object society. When it comes to describing the relationships between objects we first have to introduce suitable *views* on objects. Views are used to clarify which parts of an object society are relevant for communication relationships between objects, in other words which parts are visible and necessary to describe interactions between objects.

TROLL introduces the view concept similar to views known from relational databases, that is, we support selection, and projection views. Join views are not supported.

Like in the relational database model views can contain *derived information*, i.e. *derived attributes* but also *derived events* describing information that can be calculated from existing information, resp. operations (events) that are defined in terms of existing events. Objects 'visible' in views can be referred to by keys defined in the view specification or directly via their identities. Identification cannot be encapsulated, but we have to specify *which* key maps should be visible. So we may decide that a specific view of Person only makes visible the ByName key and not the BySocSecNo key.

The view definition resembles the role specification in that the keyword role of is substituted by view of. A derived option is used for views that describes a *selection condition* much like a condition for a derived role specifies the objects that are members of a role class. In contrast to roles, a view specifies no new attributes, events, and components. It is only possible to specify *derived* attributes, events and components in terms of already existing properties.

As an example we provide a view of Company containing only companies with more than 5000 employees:

```
object class BigCompany
  view of Company C derived C.NoOfEmps >= 5000
  identification ByName:(Name)
  ...
end object class BigCompany
```

assuming that the Company specification defined an attribute NoOfEmps and an identification ByName. The attributes, events, and components parts of a view specification introduced in the next section list the symbols visible to other objects and possibly describes derivation rules for attributes and events (see below). In a more realistic example big companies may as well be specified as roles because usually there have to be specified additional rules for big companies in a society.

3.5 Communication Relationships

After having specified objects and suitable views on objects we may *relate* objects by means of communication. Although relationships in TROLL can be used to describe constraints between objects, i.e. integrity rules referring to the observable part of objects, their main purpose is relating events i.e. communication.

The concept of event calling is extended to objects not related by inheritance or components using *relationships* leading to a system description where relationships between loosely connected objects are not buried in the object descriptions themselves [JHS93]. As an example we may specify parts of a `User-ATM` interaction by relating the relevant objects and synchronizing on their events:

```
relationship User_ATM between User U, ATM
  interaction
    variables atm:|ATM| ;
    U.insertMyCardInto(atm) >> ATM(atm).readUserCard ;
    ...
end relationship User_ATM
```

where `insertMyCardInto` and `readUserCard` are events of the communicating objects. In a more realistic example such relationship will be defined not between classes `User` and `ATM` but perhaps between the class `User` which is a *role class* specification already tailored to events and attributes of a person as a bank customer and a *view* of `ATM`s especially tailored for such customers, e.g. a view `ATM_to_Customer`.

The latter example shows the close relationship between role classes, that describe objects from a certain aspect and view classes, that make visible a certain subset of properties of objects or a subset of objects of a class. Together with relationships, views and roles structure a society and the clearly defined interfaces between parts of a specification give rise to parts that can be refined separately in later design steps.

After having sketched the basic abstraction principles in TROLL we will now have a look at the template specification of TROLL objects.

4 Specifying Objects in TROLL

Specifications in the language TROLL are based on a number of sublanguages that are the basic formalisms underlying the language TROLL. These sublanguages are *data terms* for data values and expressions (involving signatures of constant symbols and operation symbols as well as terms over attribute signatures), *first order logic* for a variety of assertions that can be formulated for objects, *future directed temporal logic* for dynamic constraints on attribute evolutions, *past directed temporal logic* for enabling conditions for event occurrences referring to the history of objects, and a language for *process specification* to specify fragments of life cycles explicitly.

Data terms are used throughout a specification in many different places as for describing the change of attribute values, values of event parameters used for communication etc. For TROLL there exist several predefined data types like *integer*, *nat*, *string*, etc. as well as the type constructors *set*, *list*, *tuple* that can be used to define arbitrary nested data structures. TROLL provides no sublanguage for the *definition* of user defined data types. Such user defined types can be specified in some suitable framework for the specification of data types and then be *included* into the society specification.

The sublanguages are used for specifying features of the concepts *attributes*, *events*, and *components* that we will describe in the next sections. We will begin with the *structural* object specification (attributes, components, constraints) and continue with the *behavioural* object specification (events, processes).

4.1 Structure of Objects

The structural specification is build from *attributes* defining the observable state, *components* defining the composition of objects from parts already sketched as an abstraction mechanism in the previous section, and from *constraints* depicting restrictions on attribute values and their evolution. The latter already points towards behaviour specification.

4.1.1 Features of Attributes

Attributes in TROLL are specified with a name and type. Optionally attributes may have parameters, thus introducing *sets of attributes*. Attribute parameters are specified with a name, which is considered as a formal parameter declaration.

Each attribute has associated a set of *features* further specifying properties of objects. The possible features are restricted, constant, initialized, and derived:

- restricted: A restricted attribute may have only values defined by a constraint formulated in first order logic. Only *local restrictions* are allowed, in other words only constraints mentioning the attribute symbol, constants and data type operations.

- constant: A constant attribute is an attribute that can never change its value after creation of an object. As such it must be initialized on creation of an object.

- initialized: With an initialization we can define the initial value of attributes using data terms that are restricted the same way as with feature restricted.

- derived: Derived attributes are defined via a more general data term that may reference other attributes. Circular derivations are not allowed however.

As an example see the following attribute specifications taken from an artificial person object specification:

```
attributes
   PersonalID:nat constant .
   Age:nat
     restricted Age >= 0 and Age <= 150
     initialized 0 .
   IncomeInYear(Year:nat):money
     restricted Year > 1870  and Year <= 2030 .
   HasBooks:set(tuple(Author:list(string),Title:string))
     initialized emptyset .
   NoOfBooks:nat derived card(HasBooks) .
   ...
```

The attribute `PersonalID` is classified as constant and must be set on creation of the object. For the attribute *Age* we specified a restriction and an initial value. `IncomeInYear` is an attribute generator defining attributes for all values of the parameter `Year`, the restriction rule stating that only particular attributes can have a value different from `undefined`. The `HasBooks` attribute has a complex structured codomain describing a set of books. This attribute is provided for presentation purpose only. In a more realistic example we would specify books separately. The last attribute specified here is derived from the `HasBooks` attribute, the derivation rule calculates the cardinality of the set `HasBooks`. Card is predefined for the set constructor.

4.1.2 Features of Components

Component specification is a means to describe the *part-of* relationship between objects. Parts may be *local* to an object or *shared* between objects. The former can only be accessed in the context of the enclosing object and are thus closely related to the composite object, the latter can be components in different objects in a society. Since this is the only difference and we have already introduced an example in the previous section, we will only describe shared components here.

In TROLL we may describe *single* and *set valued* components. Both concepts are closely related to the concept of object classes and single objects. We provide also a construct to specify *list valued* components that are handled similar to set valued components plus additional access to objects at the head or tail of the list and indexed accesses.

The component specification has a similar notation as the attribute and event (see below) specification. But there is an important difference to attributes. Components do not describe *object valued attributes* that are often called *references* in popular object oriented (programming) languages. Attributes of an object describe *data values* whereas components describe *part objects* which define a stronger relationship than object references. The component objects being part of a composition can be restricted in their behaviour by the embedding object. For example we can specify conditions that forbid event occurrences in components. The *part-of* relationship is thus a *two-way* relationship whereas references are generally *one-way*.

Components were already introduced briefly in the last section. The *features* of components resemble attribute features with the difference that we have to handle *objects* and *object populations* in contrast to *data values*. For components we provide the features set, list, restricted, initialized, and derived:

- set, list: The features set and list define multi object components (set) and optionally ordered components (list).
- restricted: Restriction for components are described with first order formulae stating properties that objects must have to be incorporated and to remain parts in a composition. The formulae may refer to attributes of the component objects and to attributes of the embedding object.
- initialized: In contrast to attributes where initializations and derivations are specified with data terms, for components also formulae are used. An initialization formula describes properties that objects of the domain class of a component symbol must have to be initially in the composition (a very simple query against the population of a class).
- derived: Similarly to initialization a derivation is formulated as a formula selecting the subset of objects from a component class that qualify for a given condition.

To alter the composition of a composite object dynamically TROLL provides special events to insert and delete objects from components. For observing the current contents special attributes are generated. The interested reader may refer to [HSJ+94] for more details. As an example for components see the following component specifications taken from a bank example:

```
components
  Manager:Person .
  Acct:Accounts A set
    derived A.No>100000 and A.No<999999 .
  ServiceQueue:Person list
    restricted ServiceQueue.Length < 10 .
```

The `Manager` component is a single object component describing the person that is currently the manager of the bank. It can be changed using specially generated events (see below). The component `Acct` is set valued and derived as all accounts (specified elsewhere) with numbers in the given range. The component `ServiceQueue` denotes a list of objects that is assumed to be initialized to the empty list by default. An invariant constraint to be valid for the whole life cycle of the object at hand states that the length of the list has to be lower than ten.

4.1.3 Constraints

Constraints formulated in first order logic or future directed temporal logic are used to describe static restrictions of the observable object state and the evolution of attribute values. In contrast to restriction rules for attributes we may specify arbitrary relationships *between* attributes. For example we can specify account objects with attributes like `Balance`, denoting the balance of an account or `Red` that is true if the account is overdrawn. Constraints may look like the following examples:

```
constraints
   initially (Balance > 100 before Red) ;
   Red => sometime(not Red) ;
   not (Red and Balance > 10000) ;
```

stating that the Balance of an account must be at least 100 before it can be overdrawn (first rule), and that an account in 'red condition' must be filled up in the future (second rule), and that an account cannot be overdrawn more than 10000$. The keyword initially states that the constraint is relative to the creation of the object whereas the second and third rules are invariant.

Note that the use of future temporal logic constraints already is a specification of *behaviour*. This kind of behaviour specification is used in early modeling phases and should be refined to a behaviour specification that *respects* the future temporal logic formulae. We will now introduce event specification with enabling conditions local to events and (simple behaviour restrictions) and process specification (object behaviour).

4.2 Behaviour of Objects

The behavioural specification is build from *events* defining the state changing operations, and *processes* defining the possible traces of events of an object (life cycles) in an operational fashion.

4.2.1 Features of Events

The event signature in TROLL is specified in the same manner as attributes, e.g. introducing a name and parameters. Events define the state changing operations for objects. Event parameters are considered to be formal parameters specified with a name. A parameterized event thus describes a set of events one for each possible tuple of parameter values.

Each event has associated a set of features that specify its properties. Event features are birth, death, active, enabled, changing, calling, and binding:

- birth, death: Birth and death events create respectively destroy objects. Thus a birth event can only occur at the beginning of a life cycle whereas a death event can only occur at the end of a life cycle. Birth events are required for an object specification, death events are optional since we may want to model objects that are never destroyed.
- active: The notion of *active* event comes in consideration due to the requirement to abstract from causality. Usually an event must be triggered, i.e. caused by some other event. In modeling real world entities, we sometimes do not want to or cannot specify such a causality. Examples are representations of user objects that have their own *initiative*. The cause for their events is often not in the scope of such a specification.
- enabled: Events can be enabled or be forbidden in a given state of an object. Depending on current attribute values or the current history of an object we describe such conditions by the use of past directed temporal logic formulae as *enabling* conditions.
- changing: The `changing` event feature is used to describe the change of attribute values triggered by an event. With a *changing rule* specification we describe the assignments of new attribute values denoted by data terms referring to the current state and possible event parameters. An imperative style of specifying attribute updates is chosen.
- binding: Similar to attribute updates we can specify *locally bounded parameters* for events using *binding rules*. Such parameter values are determined by data terms referring to attributes of the object and parameters of the event. The name of such a parameter is marked with a preceding `!`. We will provide an example for such events delivering values back to the caller later on.
- calling: To describe more complex state changes we may *call for* or *trigger* other events of the object when an event occurs. The underlying execution model is a *synchronous* execution of all events called transitively: they are contained in one snapshot and define a composite state change [HS93].

See for example events that model arrival and service in the composite bank object sketched above:

```
events
  Arrival(P:|Person|)
    calling ServiceQueue.InsertLast(P) .
  Serviced(P:|Person|)
    calling ServiceQueue.RemoveFirst .
```

The events `Arrival` and `Serviced` denote events that occur if persons are arriving or if they are being served at a bank desk. Both events *call for* the insert and remove events generated for the list valued component `ServiceQueue`. The notation `|Person|` denotes the data type of *identifiers* for objects of class `Person`.

Calling as depicted by this example is one way to specify events that are triggered with the occurrence of another event. TROLL supports an additional feature for templates (interaction) that is used as for relationships but only *inside* composite objects. We refer the reader to [HSJ+94]. All events called this way (possibly also transitively) occur in *one* snapshot, that is *synchronously*. If one of these triggered events cannot take place, all of them cannot occur.

As an example for most of the features for attributes and events introduced briefly we specify the attributes and events for a stack object that stores natural numbers:

```
attributes
   Top:nat derived if Empty then undefined else Array(Pointer-1) fi .
   Empty:bool  derived (Pointer = 0) .
   Array(No:nat):nat initialized Array(No) = undefined .
   Pointer:nat initialized 0 .
events
   Create birth .
   Destroy death .
   Push(Elem:nat)
     changing
       Array(Pointer) := Elem;
       Pointer := Pointer + 1 .
   Pop
     enabled not Empty
     changing Pointer := Pointer - 1 .
```

In this example the event Push has a parameter Elem that denotes the element to be pushed on the stack. The changing rules define the new observable object state, here the value is stored in the Array attribute with index Pointer and the Pointer attribute is incremented. The Pop event is only enabled if the stack is not empty.

A more liberal specification of stacks can be described in the following way without a precondition for Pop events but an additional parameter delivering a status. The Pop event is no more disabled for a stack that is empty, but it has *no effect* on the observable state of the stack and returns false if the stack is empty:

```
events
  pop( ! Status:bool )
    changing  { not Empty } Pointer := Pointer - 1
    binding  Status = not Empty .
```

The condition in the changing rule states that a decrement of the Pointer variable is only performed if the stack is not empty. The value of the parameter Status must not be set by the caller of the event but is determined locally. Note that this example is only for presenting the concept of events that deliver values back to the caller. Generally the former description of stacks is more adequate since it disallows the pop event for empty stacks.

4.2.2 Process Specification

The specification of life cycles of an object is the main part of the overall behaviour specification and describes the possible *evolution* of objects.

For the behaviour in terms of allowed operations we have to formulate event sequencing patterns. An event description merely describes the conditions and effects local to event occurrences. To describe entire life cycles we have to talk about *event sequences* and *dependencies* among events that occur sequentially or concurrently.

Like for attributes, events and components we specify named processes that are further depicted with process features. The features are:

- initiative denoting a sequence of events that is executed by own activity (compared to a procedure that is executed in the context of an operating system process with cpu power),
- interleaving denoting which events may interleave the sequence of events specified,
- start describing when a specified sequence is valid, and
- weak denoting a process that can be safely ignored (i.e. a *normative* behaviour that can be violated if requested by the environment).

The process sequence is written down in a process language that mainly provides *sequencing*, *alternatives* described by a set of *guarded processes*, and *parallel events*. Recursion is introduced using process identifiers in the sequence.

We cannot go into detail in this paper and will provide as example a process defining the sequence of a user interacting with an automatic teller machine (ATM). We assume attributes `MyPIN` and `MyCard` and event `decisionToGetMoneyAtATM` be defined in the surrounding object specification:

```
process declaration
   variables A:money; Valid:bool;
   UserATMInteraction =
       arrival -> enterCard(MyCard) ->
       enterPINCode(MyPIN) -> checkPinCode(?Valid) ->
       (  {Valid}
            enterAmount(?A) -> throwOutCard ->
            removeCard -> throwOutMoney -> getMoney(A)
        |
          {not Valid} throwOutCard -> removeCard
       ) -> leave .
process
  UserATMInteraction
     interleaving normal
     start after(decisionToGetMoneyAtATM) .
```

After arrival at the machine a card (attribute `MyCard`) has to be inserted and after entering the PIN-code (attribute `MyPIN`) the code is checked. The parameter `Valid` is set somewhere else and is henceforth bound to a value of type `bool`. Depending on the value of `Valid` the usual process of interacting with the machine has to be executed or the card is removed and the user leaves the machine (a *choice* between different *alternatives*). Event parameters preceded with a `?` are bound to values during execution of the process and can be used in the sequel.

After declaring the process, the process `UserATMInteraction` is *defined* and then further specified more particularly with features to be a *pattern of possible behaviour* for the object specified. Interleaving normal means that all events used in the definition must respect the sequencing conditions, e.g. a `throwOutMoney` event must be preceded by a `removeCard` event etc. However events like `startSomethingElse` can be performed as long as the process is eventually terminated with event `leave`.

In this example the (sub)process starts in the state after the event `decisionToGetMoneyAtATM`, and the sequence *cannot* be exited without violating the specification (otherwise it has to be specified as weak). If the weak mode is missing, a process specification is assumed to be *strict* meaning that the process *must* be performed until its normal end (here event `leave`). For details on process specification and its features see [HSJ+94].

5 Conclusion

In this abstract we presented the conceptual modeling and design language Troll in its revised version introduced in [HSJ+94].

The former version was strongly influenced by the underlying logical foundations and ADT framework: after presenting the signature of an object template in terms of attribute, component, and event symbols, different kinds of formulae like constraints, enabling conditions, valuation rules, interaction rules, etc. had to be written down. Already simple object specifications became unreadable since the specification of e.g. an event was spread over the specification text. Clearly, this was an undesirable feature for a language that should also serve as *documentation* vehicle. A conceptual modeling language should be readable at least for trained domain specialist. The latter point is also supported by a parallel development of a graphical notation on top of Troll [JWH+94].

The revised version of the language presented here clusters the *specification* of a *concept* like e.g. an event describing *optional features* like enabling conditions, changing rules, calling rules etc. This strategy serves three purposes. Firstly we have a certain degree of *modularization in small scale* inside an object specification. Secondly the notation is syntactically near to object oriented programming languages and more familiar to beginners. Thirdly the beginner can concentrate on the basic concepts and learn about the meaning of features successively.

The idea of a concept refined by features is then carried over to objects and object classes. Single objects are specified with a name and a template consisting of structure and behaviour specification. Optionally (although this is the usual case) with the features *class* and *identification* we specify collections of similar structured objects. Additional features for objects are different kinds of *views* and *roles* that can be orthogonally combined with *object relationships*. Relationships are mainly used to define *communication* between objects.

Parallel to the language development we worked on tool support for TROLL. Up to now we have implemented a repository for specification documents, a syntax directed editor for TROLL-text and a graphical tool based on OMT notation [RBP+90, JWH+94]. Planned are prototyping tools (translation to C++ [Str86]), tools for checking specifications for inconsistencies, and translation tools that transform high level constructs to an operational language kernel.

Acknowledgements

We are grateful to all our colleagues in ISCORE stimulating our work with a lot of discussions on language and semantics issues. Especially we thank Hans-Dieter Ehrich and Amílcar Sernadas for their early work on semantics of objects. Thanks also to Ralf Jungclaus and Cristina Sernadas who worked with us on the development of the previous language version. For many fruitful discussions on the TROLL language we are grateful to all our colleagues and students in Braunschweig.

References

[BM86] M. L. Brodie and J. Mylopoulos, editors. *On Knowledge Management Systems*. Springer-Verlag, Berlin, 1986.

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

[Bor85] A. Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, 2(1):63–73, 1985.

[BP92] W. Bischofberger and G. Pomberger. *Prototyping-Oriented Software Development*. Springer, Berlin, 1992.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[Che76] P.P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Con94] S. Conrad. A Basic Calculus for Verifying Properties of Synchronously Interacting Objects. Informatik-Bericht 94–05, Technische Universität Braunschweig, 1994.

[EDG88] 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.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, pages 453–467. Springer, Berlin, LNCS 668, 1993.

[EGH+92] 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.

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

[EGS90] 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.

[Ehr86] 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.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[ES90] 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.

[ES91] 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.

[ESS92] H.-D. Ehrich, G. Saake, and A. Sernadas. Concepts of Object-Orientation. In *Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany)*, pages 1–19. Springer IFB 303, 1992.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

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

[HG88] U. Hohenstein and M. Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In C. Batini, editor, *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, pages 129–148. North-Holland, Amsterdam, 1988.

[HJS92] 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.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

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

[HS93] T. Hartmann and G. Saake. Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 1993.

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[Hum90] W.S. Humphrey. *Managing the Software Process.* Addison-Wesley, 1990.

[JHS93] R. Jungclaus, T. Hartmann, and G. Saake. Relationships between Dynamic Objects. In H. Kangassalo, H. Jaakkola, K. Hori, and T. Kitahashi, editors, *Information Modelling and Knowledge Bases IV: Concepts, Methods and Systems (Proc. 2nd European-Japanese Seminar, Hotel Ellivuori (SF))*, pages 425–438. IOS Press, Amsterdam, 1993.

[JHSS91] R. Jungclaus, T. Hartmann, G. Saake, and C. Sernadas. Introduction to TROLL – A Language for Object-Oriented Specification of Information Systems. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability*, pages 97–128. TU Braunschweig, Informatik Bericht 91-03, 1991.

[JSHS91] 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.

[JSS91] 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.

[Jun93] R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-Based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[JWH+94] R. Jungclaus, R. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining TROLL with the Object Modeling Technique. In *Proc. IFIP'94 Workshop F1, GI-Fachgespräch, Hamburg.* Springer, Informatik aktuell, 1994. *To appear.*

[MB89] J. Mylopoulos and M. Brodie, editors. *Readings in Artificial Intelligence & Databases.* Morgan Kaufmann Publ. San Mateo, 1989.

[MBW80] 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.

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

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification.* Springer-Verlag, New York, 1992.

[MR91] J.A. McDermid and P. Rook. Software Development Process Models. In J.A. McDermid, editor, *Software Engineer's Handbook*, Chapter 15. Butterworth-Heinemann Ltd., 1991.

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

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

[Saa91] 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.

[SE91] 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.

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

[SFSE89] 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.

[SSC92] A. Sernadas, C. Sernadas, and J. F. Costa. Object Specification Logic. Research report, INESC/DMIST, Lisbon (P), 1992. *To appear in Journal of Logic and Computation.*

[SSE87] 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.

[ST89] J. W. Schmidt and C. Thanos, editors. *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.

[Ste87] L.A. Stein. Delegation is Inheritance. *SIGPLAN Notices, Special Issue OOPSLA87*, 22(12):138–146, 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, 1986.

[Wd92] R. Wieringa and W. de Jonge. The Identification of Objects and Roles – Object identifiers revisited. Technical Report IR–267 (version 2), Vrije Universiteit, Amsterdam, 1992.