Modelling Information Systems as Object Societies[*†]

Gunter Saake
Thorsten Hartmann

Abt. Datenbanken, Techn. Universität Braunschweig
POBox 3329, D–38023 Braunschweig
E-mail {saake|hartmann}@idb.cs.tu-bs.de

Abstract

Conceptual modelling of complex information systems requires the use of a formal design approach covering both static and dynamic aspects of the system and the modelled Universe of Discourse. Viewing an information system as a collection of communicating objects is close to the intuitive perception of such systems on a conceptual level. Objects have a local state, show a specific behaviour, communicate with other objects and may be itself composed from smaller objects. This article presents an abstract concept of such dynamic objects and discusses language features of a specification language for describing object systems. The presented language TROLL2 supports structuring mechanisms of semantic data models together with process specification constructs to cover object dynamics. Extensions of the presented framework are discussed covering the step from communicating objects to cooperating agents allowing more flexible system structures.

1 Introduction

Conceptual modelling of information systems requires the description of the application domain, the so-called *Universe of Discourse* (UoD), on a high abstraction level. This description should be independent from any implementation details and should be build on an exact formal description technique. Such a *conceptual model* should cover both the structural aspects, i.e. the information

[*]This work was partially supported by the CEC under ESPRIT-III BRA WG 6071 IS-CORE (Information Systems – COrrectness and REusability). The research work of Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3.

[†]In: Proceedings 3. Workshop Informationssysteme und Künstliche Intelligenz: Verwaltung und Verarbeitung komplexer Strukturen, IS/KI-94, Hamburg 28.2.94-2.3.94, LNCS 777, pages 157–180, Springer, 1994.

entities and their relationships, as well as the dynamics aspects, i.e. application specific functions and processes.

Many current description approaches agree in conceptually modelling the UoD as a *collection of interacting objects*. Objects encapsulate structure *and* behaviour, and the commonly used inter-object relations (like aggregation, communication, specialization with inheritance etc.) offer the basic abstraction mechanisms known from semantic data modelling [Saa93]. Moreover, the object paradigm may be used for describing not only the UoD but also the information system itself allowing to integrate existing subsystems into the conceptual model [SJH93].

Looking at an information system (and its environment) as a collection of interacting objects seems to be a very natural way for conceptualizing information structures and processes. This observation is confirmed by the current success of object-oriented analysis and design frameworks [Boo91, CY91, RBP+90]. Several conceptual modelling approaches follow this idea by offering a framework based on a formal concept of dynamic objects. Languages supporting this object paradigm are for example Albert [DDP93], CMSL [Wie90], Oblog [SSE87], Object Behaviour Diagrams [KS91], Troll [JSHS91] and TROLL *light* [CGH92]. These languages combine language features known from semantic data models [HK87] with formal techniques for describing object behaviour.

In this paper we will discuss some features common to these object-oriented conceptual modelling approaches. The formal foundations of Troll-like languages have stabilized during the last years and will be sketched only. The language Troll2 [HSJ+93], a revised version of the Troll language [JSHS91], will be presented using small examples. Shortcomings of the underlying object model will be discussed afterwards and a possible extended framework is sketched.

The rest of the paper is organized as follows. Section 2 gives a short overview of the underlying semantic structures and related logics. The two following sections present the basic language features of the Troll2 language: first we discuss the basic abstraction principles to relate objects used in information systems and their realization in Troll2, and afterwards we present the concrete logic-based specification of object structure and behaviour. Section 5 discusses extensions of the presented object concept to capture more flexible and adaptable object behaviour.

2 Basic Concepts and Formal Background

An object is constructed from a possible behaviour and observations over that behaviour associated with local states. In this section, we are concentrating on the model level rather than on the specification of such models in terms of a language. Some language features will be used for presentation purposes.

2.1 Objects as Observable Processes

Each object has a set of *events* EVT. An event can be regarded as an abstraction of a *possible atomic state transition* and is strictly *local to an object*. The declaration `debit(Amount:money)` declares a set of events, one for each possible value in the carrier set of the data sort `money`. A particular event is described by an *event term*, e.g., `debit(m)` where `m` is a variable instantiated by a value of sort `money`. Some events may be marked as being *active*, i.e. in a closed world of a specification they may occur without request.

Over a set of events EVT we define *snapshots* s as sets of simultaneous events $s \in 2^{EVT}$, i.e. each s is a set of simultaneous occurrences of local events.

Definition 2.1 (*Object Life Cycle*) Let EVT be a set of events. A sequence $\hat{s} = s_0 s_1 s_2 ...$ such that all s_i ($i \in \mathbb{N}$) are snapshots over EVT, is called an *object life cycle*. A finite prefix of length i $\sigma = s_0 s_1 ... s_i$ for $i \in \mathbb{N}$ is called the *i-th state*. The last snapshot s_i in a state σ is denoted by $last(\sigma)$.

Thus, the evolution of an object over time is described by a sequence of snapshots, i.e. the sequence of sets of local events. Many occurrences of the same event can occur in several snapshots in an object life cycle. We do not exclude the presence of empty snapshots in object life cycles. Empty snapshots represent states where no event can be observed. This way, we may "stretch" life cycles by inserting empty snapshots and we may restrict ourselves to infinite life cycles since finite ones can be expanded by appending empty snapshots.

Definition 2.2 (*Object Behaviour*) Let EVT be a set of local events. A set B of object life cycles over EVT is called *possible object behaviour*.

Events may not occur in arbitrary states within an object life cycle. In each state, we have a set of *enabled events*. An event a may occur in a snapshot s_i only if it is enabled in state σ where $last(\sigma) = s_i$. This is written as $\mathtt{enabled}_\sigma(a)$. In this paper, we will not enter into conditions telling us whether an event is enabled in a certain state or not – the reader is referred to [EDS93, Jun93].

A set of *attributes* as usual denotes the state-dependent observable properties of objects. We assume a fixed universe of data types and sorts S to be given. Each attribute $att \in ATT$ is *typed*, i.e. it has an associated sort $sort(att) \in S$. The interpretation of a sort is a suitable set of values and is denoted by $\mathcal{U}(s)$ for a sort $s \in S$.

The declaration `IncomeInYear(Year:nat):money` declares an attribute set, one attribute for each possible value in the carrier set of the data sort `nat`. Particular attributes are described by *attribute terms*, e.g. `IncomeInYear(y)` where `y` is a variable instantiated by a value of sort `nat`. With each state σ of an object life cycle we associate a *current observation* giving us the current *values* of the attributes.

Definition 2.3 (*Observations*) Let B be an object behaviour, $\mathcal{S}(B)$ be the set of states over B, and ATT a set of attributes. The set $O(ATT) = \{(att, d) \mid$

$att \in ATT, d \in \mathcal{U}(sort(att))\}$ denotes the possible observations over ATT. A mapping

$$obs : \mathcal{S}(B) \to 2^{O(ATT)}$$

is called an *observation structure over* B iff for each state σ of an object life cycle $\hat{s}$ the observation $obs(\sigma)$ fulfills the following condition:

$$[(att, d) \in obs(\sigma) \ \wedge \ (att, d') \in obs(\sigma)] \Rightarrow d = d'$$

That is, attributes cannot have more than one value in each state, but they are allowed to be undefined in a state.

An object now is composed from an object behaviour and suitable observations over the possible object life cycles.

Definition 2.4 (*Object Model*) An *object model* $\mathbf{ob} = (EVT, B, ATT, obs)$ consists of a set of events EVT, an object behaviour B over EVT, a set of attributes ATT, and an observation structure obs such that for each pair of subsequent states $\sigma, \sigma' \in \mathcal{S}(B)$ where $\sigma' = \sigma\, s$ the following condition holds:

$$(s = \emptyset) \ \Rightarrow \ [obs(\sigma') = obs(\sigma)]$$

s is the current snapshot in state σ'.

Only the occurrence of events may alter an observation, i.e. empty snapshots do not have any effect on observations over an object life cycle.

When we put objects together to form composite objects, we are interested in the behaviour of the composite object as a whole. Now, the events local to all components are local to the composite object, i.e. the components are *embedded* into the composite object. Each snapshot in the object life cycle of the composite object may include events of components. A composition is admissible if the restriction of each snapshot of each object life cycle of the composite object to the events local to a component yields an object life cycle equivalent to an object life cycle of the component.

Composition of objects thus means the following:

- Snapshots over components are merged, i.e. we are constructing unions of component snapshots (please recall that snapshots may be empty, i.e. we may have stretched life cycles);
- observations over components are merged according to the merging of life cycles.

The composition of snapshots and observations includes the *synchronization of events* since all events in a snapshot are occurring simultaneously. Synchronization along with parameter passing models *communication*.

In our model, communication is only possible *inside composite objects*. Certain composite objects, however, can be regarded as being virtual, i.e. a system of communicating objects may be regarded and modeled as a composite object.

2.2 Object Interaction

As mentioned in the previous section, communication is achieved by synchronization. We have, however, to introduce a notation that allows us to describe communication. Furthermore, communication almost always involves control flow, i.e. there are objects that initiate communications and there are objects that are requested to communicate by others.

The basic primitive in our approach to describe communication is *event calling*. Specificationwise, it is defined as a *state dependent relation* over events.

Definition 2.5 (*Calling Relation*) Let EVT be a set of events. The *calling relation* $\ggg_\sigma \subseteq EVT \times EVT$ relates events wrt. a state σ of an object life cycle $\hat{s}$.

The calling relation is *transitive*, i.e. the following condition always holds:

$$(e_1 \ggg_\sigma e_2) \wedge (e_2 \ggg_\sigma e_3) \;\Rightarrow\; (e_1 \ggg_\sigma e_3)$$

Intuitively, calling should be understood as *asymmetric, synchronous communication*. Calling is written as

```
{ condition } event_term1 >> event_term2 ;
```

This means that the calling relation only holds between events denoted by the event terms if the condition holds in the current state. This notation can also be interpreted as some kind of *ECA-rule* [DBM88]: an event denoted by `event_term1` causes other *Events* (or *Actions*) denoted by `event_term2` depending on certain *Conditions*.

Definition 2.6 (*Closure of Calling*) Let a be an event occurring in state σ. The *closure of a wrt. calling in state σ* is defined as

$$\mathtt{cl}_\sigma(a) = \{a' \mid a \ggg_\sigma a'\}$$

As mentioned earlier we distinguish between active and passive events, the former defining the events that can occur by own initiative. For calling closures we require that for all passive events $\mathtt{a}$ there exists an active event $\mathtt{a}'$ such that $\mathtt{a} \in \mathtt{cl}_\sigma(\mathtt{a}')$.

2.3 Temporal Logic View on Objects

The basic semantic concept of the presented object model is the characterization of object behaviour by a set of life cycles, i.e. *observable linear process runs*. Another view on life cycles is to see them as a *sequence of states*, where state transitions are labelled with event snapshots. These state transitions can even be encoded into the states using an **occurs** predicate [SSC92, Jun93] or — equivalently — an **after** predicate [JSHS91, Saa93].

State sequences are the interpretation structures for *linear temporal logic* [MP92]. Therefore, a logic-based specification of object behaviour can be done in temporal logic. The concrete syntax of the TROLL languages allows at some places direct formulation of temporal axioms (for example, as temporal integrity constraints); other language features can be translated to temporal logic (see [Jun93, Saa93] for concrete translations). Instead of introducing a complete temporal logic and the translation of TROLL language features into it we will present some typical specification patterns and show how they can be expressed in temporal logic.

- The temporal constraint "The `Counter` attribute always increases." can be expressed by the formula

$$\forall x(\textbf{always}((\texttt{Counter} = x) \Rightarrow (\textbf{next}(\texttt{Counter} \geq x))))$$

- The effect of the `Increase(n)` event to increase the `Counter` attribute by the value n can be expressed as

$$\forall x \forall n(\textbf{always}(((\texttt{Counter} = x) \wedge \textbf{occurs}(\texttt{Increase}(n))) \Rightarrow$$
$$(\textbf{next}(\texttt{Counter} = x + n))))$$

  The variable x stores the 'old' value of the `Counter` attribute to be used in the **next** state.

- We can restrict the permission of the `Decrease(n)` event to those states where the `Counter` attribute has a value larger than n.

$$\forall n(\textbf{always}(\textbf{occurs}(\texttt{Decrease}(n)) \Rightarrow (\texttt{Counter} \geq n)))$$

- The effect of a calling declaration a_1>>a_2 is simply expressed by

$$\textbf{always}(\textbf{occurs}(a_1) \Rightarrow \textbf{occurs}(a_2))$$

TROLL object specifications are equivalent to a set of temporal axioms describing the possible behaviour of an object. Observable processes are equivalent to state sequences serving as interpretation structures for this temporal logics, and therefore the usual satisfaction relation holds between object descriptions and object models. We will come back to this point in Section 5.

3 Abstraction Principles in TROLL2

In this section we will introduce the basic abstraction mechanisms of TROLL2 [HSJ+93] 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 between objects.

To talk about objects in some formal language 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 TROLL2 such symbols are *attributes*. To describe not only the observable part of objects, we have to specify the *state changes* an object can show during its life time. In TROLL2 *event symbols* denote such state changes. Last but not least objects can be composed of other objects. For this case we introduce *component symbols*. 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 TROLL2 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, i.e. attributes (see Section 4). Tuples of attribute values (an *identification*) are thus defined as *keys* to objects of a class. Apart from such attribute values, each object of a given class has a unique *identity*. An injective mapping from identifications to identities ensures that we can non-ambiguously refer to a *formal object* if we know a class and an identification of the corresponding real world object. As an example we may specify a 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` (not shown here), and the tuple `ByName` defining a (named) key for persons.

3.2 Object Roles and Specialization

The notion of classes is not sufficient if we want to model real world entities. Usually an abstraction in terms of similar structure and behaviour yields object descriptions that are untractable with respect to size and complexity. Object oriented programming languages introduce the notion of *inheritance* for structuring code and to support code reuse. TROLL2 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 [WJ91]).

As in programming languages, role classes in TROLL2 *inherit* the specification of their base classes. Moreover role class objects in TROLL2 *contain* their base objects leading to an inheritance mechanism based on *delegation* [Ste87]. Objects of role classes are born sometime during the lifetime of their base objects due to occurrence of *events*. Similarly they leave a role with occurrence of events. In contrast to syntactical inheritance also the objects themselves are *inherited*. The concept of roles resembles the *is-a* hierarchies known from semantic data models [HK87]. In TROLL2 the behaviour of objects is integrated in that a role object *is-a* base object for a specific time span. 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.

A role class *inherits* also the naming mechanism, that is, we can refer to `Employee`s via their `SocialSecNo` for example. 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`:

```
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 also *derived* for a role class as `Woman`.

3.3 Composite Objects

The use of classes and roles as abstractions still is not sufficient to map real world entities to objects. Nearly all objects we observe are *composed of parts*. Basically we can distinguish two kinds of part objects, *dependent* and *sharable* parts. The former are local to an object in that they cannot be parts in 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.

TROLL2 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, i.e. it has a template and a naming mechanism. The uniqueness of identifications 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.

Orthogonally to dependent and sharable components TROLL2 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 details how objects can be referenced the interested reader may refer to [HSJ+93]. 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*.

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 the communication between objects.

TROLL2 introduces the view concept similar to views known from relational databases, that is, we can specify selection, projection, and join views. Optionally views can contain *derived attributes* and also *derived events* describing information that can be calculated from existing information, and operations (events) that are defined in terms of existing events. Objects 'contained' in views can be referred to with keys defined in the view specification or directly via their identities (identities cannot be encapsulated). In case of join views such an identification mechanism via keys must be constructed from existing keys.

The view definition resembles the role specification in that the keyword role of is substituted by view of. A derived classification 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. The attributes, events, and components parts of a view specification introduced in the next section list the symbols visible for 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 TROLL2 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 introduced in Section 2.1 is extended to objects not related by inheritance or components using *relationships* leading to a system description where relationships between loosely connected objects is 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.

After having sketched the basic abstraction principles in TROLL2 we will now have a look at the template specification for TROLL2 objects.

4 Specifying Objects in TROLL 2

Specifications in the language TROLL2 are based on a number of sublanguages, that are the basic formalisms underlying the language TROLL2. These sublanguages are *data terms* for data values and expressions (involving signatures of constant symbols and operation symbols as well as terms over such 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.

Typed 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 TROLL2 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. TROLL2 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.

4.1 Features of Attributes

Attributes in TROLL2 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. A restricted attribute may have only values defined by a constraint formulated in first order logic. A constant attribute is an attribute that can never change its value after creation of an object. With an initialization we can define the initial value of attributes using data terms. Derived attributes are defined via a data term referencing other attributes.

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 ;
     initialised 0 .
   IncomeInYear(Year:nat):money
     restricted Year > 1870  and Year <= 2030 .
   HasBooks:set(tuple(Author:list(string),Title:string))
     initialised emptyset .
   NoOfBooks:nat derived card(HasBooks) .
   ...
```

The attribute PersonalID is classified as constant and must be set with 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.

4.2 Features of Events

Events in TROLL2 are specified in the same manner as attributes. 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 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.

The notion of *active* events comes in due to the requirement to abstract from causality. Usually an event must be triggered, i.e. caused by some other event. In modelling 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.

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.

Another 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.

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 !.

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 [HS93].

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 initialised Array(No) = undefined .
   Pointer:nat initialised 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.

4.3 Features of Components

Component specification are 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. Here we will only describe shared components.

In TROLL2 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 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 inhibit event occurrences in components.

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. The features set and list define multiobject components (object sets) and optionally ordered components (lists).

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. 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). Similarly 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 TROLL2 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+93] 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
    initialised false;  restricted ServiceQueue.Length < 10 .
events
  Arrival(P:|Person|)
    calling ServiceQueue.InsertLast(P) .
  Serviced(P:|Person|)
    calling ServiceQueue.RemoveFirst .
```

The `Manager` component is a single object component describing the person that is currently the manager of the bank. It can be changed using the events `Manager.Insert` and `Manager.Remove` where we have to supply identifiers for person instances. 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 initialised to the empty list by means of a condition qualifying for no objects and restricted to have at most the length 10.

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`.

4.4 Object Behaviour

The specification of life cycles of an object is the main part of the overall behaviour specification and describes the possible *evolution* of objects. *Constraints* formulated in future directed temporal logic are used to describe the evolution of attribute values. 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.

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.

Like for attributes, events and components we specify named processes that are further depicted with process features. The features are active, interleaving, start, and completeness. The process sequence is written down in a process language that mainly provides *sequencing* and *alternatives* described by a set of *guarded processes*. Recursion is introduced using process identifiers in the sequence.

As an example we provide a process defining the sequence of a user interacting with an automatic teller machine (ATM). We assume attributes `MyPIN` and `MyCard` 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) ;
     completeness strict .
```

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 further specified 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. The process starts in the state after the event `decisionToGetMoneyAtATM`, and the sequence *cannot* be exited without violating the specification (completeness strict). For details on process features see [HSJ$^+$93].

Behaviour involving several objects of a compound object is modelled via event calling as introduced in Section 2.1 and also sketched in Section 3.5. After having introduced the main features of TROLL2, we will now continue to sketch a possible future evolution of TROLL2.

5 Extending the Object Model

The semantic foundations and corresponding language proposals for object oriented conceptual modelling were investigated for several years now and have come to a state where the basic concepts seem to be stabilizing. In the following section, we will discuss some directions for future work extending this framework allowing more flexible object concepts. This work is in its early stages and the following subsections should be read as proposals for future research rather than as a presentation of stabilized results.

5.1 Limits of the Object Model

In section 2.1 we have characterized objects as linear processes observable by attributes. Each object state is completely characterized by a snapshot trace which determines the current attribute values. Therefore, the values of object attributes may change during object life time. The *current object behaviour* may depend on the attribute values, of course, but *the behaviour specification* remains fixed during the object life.

However, information system objects (like account objects) are persistent in the sense that they 'live' considerably longer than program runs or application sessions. Accounts (identified by an account number), for example, have a typical life span of several years. Bank rules, financial laws, computation of yearly interests etc may change several time during the life span of an account object — even without changing its attribute values.

To capture these effects we have to find a semantic model for objects where the behaviour specification of an object may be modified *during its existence*, which is not expressible in the framework underlying TROLL until now. In the following subsections, we will sketch a semantic structure enabling such behaviour evolution and give an outlook on applications of this extension.

5.2 Theories as Object States

A logical view on TROLL2 objects regards an object as an process connected with a temporal logic specification which is *satisfied* by the object behaviour. At the birth of an object, we have an initial set of temporal axioms restricting the future state evolution. Each event occurrence adds a new next time point to the object history, and at each of these transitions the set of temporal axioms is checked and the next set of axioms to be satisfied in the future is derived — a liveness requirement may be satisfied by this transition and therefore removed from the current temporal axioms. This complies to the known operational stepwise valuation of temporal formulae for a given state sequence [MP92, Saa91].

The idea to extend the object model to allow changing object behaviour is to keep the temporal valuation but to add the possibility to *modify the current set of temporal axioms depending on the transition.*

More formally, we keep the notion of object life cycle as a sequence of event snapshots as the formalization of the temporal object behaviour. For states, we use a *two-level* approach: the base level defines object states σ_i as before using a mapping to attribute-value-pairs extended by the **occurs** predicate. The second level characterizes states as sets of temporal logic axioms. Both together constitute a dynamic structure D, if for each tail sequence starting at time i the i-th state axiom set Ax_i is entailed by the base level. With other words, the state axioms of the second level restrict the future evolution of the base level using usual temporal logic entailment, i.e., we require

$$\langle \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots \rangle \models Ax_i$$

The axiom sets Ax_i are modified by events and may change from state to state. This formalization guarantees that temporal constraints are never overwritten by changing the state axioms. Therefore it offers a restricted way of modifying the temporal specification in contrast to completely override temporal obligations. A formalization of these ideas can be found in [SSSJ93].

An intuitive view on this extended object model is to think of an additional attribute `Axioms` with domain *set of temporal axioms.* An object specification has to fix the ways this attribute is modified during object life. The simpler object model described in Section 2 is a special case, where this attribute is initialized in the first state with the complete temporal logic specification. To handle temporal logic axioms as attribute *values* enables to transfer specification parts between objects using event parameters, too (see the alarm clock example below).

5.3 Applications of the Extended Model

In the following subsections we will sketch some application areas of the extended object model to show its expressive power.

5.3.1 Behaviour Evolution versus Object Migration

Typical information systems objects may have a longer life span than their functional aspects or restrictions on their behaviour. This effect partly results from having objects representing real world entities (like employees and accounts) in artificial systems realizing for example company policies — the way to compute current interests for an account or the maximal withdraw per month may change each year. With other words, existing objects have to be adapted to new behaviour patterns that cannot be anticipated in the original system specification.

A common solution for example in object-oriented database systems is to offer the possibility to change the class membership of existing objects. This concept of *class migration* [Su91] allows to handle this situation but does not fit very well to the conceptual role of classes — conceptually, the objects do not move to another class but the class description is adapted to new requirements.

The extended object model discussed above gives an intuitive semantic framework for adapting behaviour specifications to new requirements. Besides a stable immutable description of the invariant object properties we can add the possibility of an additional behaviour description modifiable during system runtime.

Another aspect of class migration is *signature modification*. In our framework, this should be handled by defining appropriate views or new role classes for objects rather than introducing object migration. In fact, many other needs for class migration only arise in data models which do not support a dynamic role concept.

5.3.2 Programmable Objects

As mentioned before, handling temporal axioms as attribute and attribute values allows to modify (or even 'program') objects via event calling. The expressive power of this approach is shown by the following example realizing an alarm clock which can be programmed to arbitrary behaviour. The syntax of the following examples should be understood as an ad hoc notation for presentation purposes, not as a language proposal.

The example is written in a syntactical style which is close to the original TROLL language features [JSHS91]. The reason is that these language features are closer to a temporal logic representation. Attribute modifications are written in the 'positional style' where for example the notation

```
    [ Tic ] Minutes = Minutes + 1 mod 60;
```

defines the new value of the `Minutes` attribute after the `Tic` event.

The following specification models a *programmable alarm clock* as an object. The clock has a basic specification part being invariant during the life of the clock. It can be programmed to arbitrary alarm behaviour by sending a temporal theory to it using an event parameter of type `SpecAxioms`.

```
object alarm_clock
template
   attributes Minutes, Hour : nat;
              Alarm: bool;
   events birth CreateClock;
          Tic;
          AlarmOn; AlarmOff;
          Pause; Reset;
          SetAlarmMode(AlarmProgram: TempAxioms);
specification
    BaseAxioms =
          begin axioms
              valuation
                [ CreateClock ] Minutes = 0, Hour = 0, Alarm = false;
                [ Tic ] Minutes = Minutes + 1 mod 60;
                { Minutes = 59 } ==>
                    [ Tic ] Hour = Hour + 1 mod 24;
                [ AlarmOn ] Alarm = true;
                [ AlarmOff ] Alarm = false;
          end axioms;
   behavior valuation
       [ CreateClock ]  Axioms = BaseAxioms;
       [ Reset ]  Axioms = BaseAxioms;
       [ SetAlarmMode(x) ]  Axioms = Axioms union x;
             /* changing the temporal theory by adding axioms */
end object alarm_clock;
```

The value of the attribute `Axioms` defines the current axiom set Ax_i at time i. This clock can be 'programmed' to an arbitrary alarm behaviour by sending a specification of the interplay of the `Tic` event with the other events. Additionally, we can even send a specification to the clock such that it triggers some object outside in case of alarm. To show the possibilities, we give two examples for the first case.

As a first example, we set the parameter `x` of the `SetAlarmMode` event to following behaviour specification:

```
begin axioms
   interaction
```

```
      { Minutes = 59 } ==> Tic >> AlarmOn;
      { Minutes = 0 } ==> Tic >> AlarmOff;
end axioms;
```

The resulting behaviour after the occurrence of `SetAlarmMode` is a one minute alarm each hour. The second example programs the clock to give alarm at noon for 10 minutes. The value of `x` is the following set of axioms. Additionally, we allow to interrupt the alarm using the `Pause` event.

```
begin axioms
   interaction
      { Minutes = 59 and Hour = 11 } ==> Tic >> AlarmOn;
      { Minutes = 9 and Hour = 12 } ==> Tic >> AlarmOff;
   valuation
      [ Pause ] Alarm = false;
end axioms;
```

These examples show modifications of the current axioms directly via the `SetAlarmMode` event. A more flexible approach is even to allow **behavior valuation** rules as part of the modifying axioms, which allows complete restructuring of behaviour specifications during runtime (and should therefore be used with care).

5.3.3 From Objects to Agents

A possible generalization of the object concept is the notion of *agent*. The concept of agent is used as an important concept in many fields of information technology, among them requirements engineering, software design, distributed databases, distributed artificial intelligence as well as organization technology. The key facets of agents can be characterized by the following properties:

- Agents may be active (in contrast to passive objects). Agents have *goals* which they try to achieve through cooperation under given constraints.

  Goals may be represented as temporal obligations to be satisfied by the agent.

- Agents have an internal *state* which includes an internal, imperfect representation of the world (including knowledge about the state of other agents).

  This concept of state goes far beyond the state concepts of objects where state information is coded in attribute values only. It requires disjunctive knowledge as well as handling of default knowledge [BLR93].

- Agents act, communicate and perceive, thus showing an external *behaviour* that obeys the given constraints [DDP93].

Following this characterization of agents, semantic models for agents should be similar to those discussed for objects: agents show behaviour resulting in life cycles like for objects, and they have a state determined by their history. But states are represented by theories rather than by attribute values, and state changes are therefore theory revisions.

We think that the presented extended object concept can serve as the first step towards an appropriate semantic model for agents. It offers temporal logic for describing goals, and (restricted) state changes are part of the model. However, we are aware of the fact that pure temporal logic is not adequate for handling all aspects of agents as mentioned above. Moreover, the two level formalization does not allow disjunctive information for attribute values. This restrictions avoids on the other hand some problems arising with arbitrary theory revision.

We are aware of the fact that the ideas presented in this section are in a preliminary state. The presentation aims at showing current research efforts rather than stabilized results.

6 Conclusions

In this paper we have introduced a framework for specifying information systems on a high level of abstraction. After describing the basics of the underlying object model, we sketched the language TROLL2 based on these formal ideas. We introduced the basic abstraction mechanisms of TROLL2 used for relating objects and continued with the specification of templates defining the structure and behaviour of objects. Then we discussed the limits of the object model described so far in terms of necessary adaptions of object behaviour during lifetime of objects. A possible integration of a theory based approach into the TROLL2 framework was sketched.

As mentioned already in the discussion of possible application areas, the approach can be extended to handle more application specific logics for example based on deontic logics [MW93]. An interesting extension is to combine the extended object model with non-monotonic logics for revising temporal logic goals. A related approach is reported in [BLR93].

A necessary extension to handle problems of schema evolution is to extend the state modification to allow changes of the object signature, too. To avoid problems with changes of the external object interface this may be restricted to internal 'hidden' attributes. Currently, schema evolution is handled by creating new views for existing object classes which allows to modify the external signature.

For the more practical areas of research in the direction of modelling of information systems we are currently working on an environment for TROLL2 specification integrating the modelling process, prototyping and documentation of specifications and on a graphical representation of TROLL2 object societies [WJH+93] that draws on concepts known from *OMT* [RBP+90].

Acknowledgements

We are grateful to Peter Hartel, Ralf Jungclaus, Jan Kusch, and Cristina Sernadas who developed the basic features of TROLL2 with us. The topics discussed in Section 5 are part of a current joint research effort with Amílcar and Cristina Sernadas. The concept of agents and the search for a common formal model for agents is topic of the ESPRIT BRA WG 8319 ModelAge (*A Common Formal **Mod**el of Cooperating Intelligent **Age**nts -a multidisciplinary approach-,* coordinator: P.-Y. Schobbens, Namur) which will start in early 1994. For many fruitful discussions on the TROLL2 language we are grateful to all our colleagues in Braunschweig, and to the members of IS-CORE, especially to Hans-Dieter Ehrich, Amílcar Sernadas, José Fiadeiro, Udo Lipeck and Roel Wieringa.

References

[BLR93] S. Brass, U. W. Lipeck, and P. Resende. Specification of Object Behaviour with Defaults. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 155–177, 1993.

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin / Cummings, Redwood City, 1991.

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

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[DBM88] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules are Objects Too: A Knowledge Model for an Object-Oriented Database System. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 129–143. Springer, Berlin, LNCS 341, 1988.

[DDP93] E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: An Agent Perspective. In O. Nierstrasz, editor, *ECOOP'93—Object-Oriented Programming (Proc. 7th European Conference)*, pages 458–481, Kaiserslautern, 1993. LNCS 707, Springer-Verlag, Berlin, 1993.

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

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

[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.

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

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

[HSJ+93] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL. "Informatik-Bericht", TU Braunschweig, 1993. *In preparation.*

[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.

[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.

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

[KS91] G. Kappel and M. Schrefl. Object / Behavior Diagrams. In *Proc. 7th Int. Conf. on Data Engineering*, pages 530–539, Kobe, Japan, 1991. IEEE Computer Society Press, Los Alamitos.

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

[MW93] J.-J. Ch. Meyer and R. J. Wieringa, editors. *Deontic Logic in Computer Science. Normative System Specification.* Wiley, Chichester, 1993.

[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. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6(1):47–74, 1991. North-Holland.

[Saa93] G. Saake. *Objektorientierte Spezifikation von Informationssystemen.* Teubner, Stuttgart/Leipzig, 1993. Habilitationsschrift.

[SJH93] G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. In M. Huhns, M.P. Papazoglou, and G. Schlageter, editors, *Int. Conf. on Intelligent & Cooperative Information Systems (ICI-CIS'93)*, pages 309–318. IEEE Computer Society Press, 1993.

[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.

[SSSJ93] Gunter Saake, Amílcar Sernadas, Cristina Sernadas, and Ralf Jungclaus. Evolving Object Specifications. Technical report, INESC Lisbon & TU Braunschweig, Draft Version, 1993.

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

[Su91] J. Su. Dynamic Constraints and Object Migration. In G. M. Lohmann, A. Sernadas, and R. Camps, editors, *Proc. Intern. Conf. on Very Large Databases VLDB'91, Barcelona*, pages 233–242, 1991.

[WJ91] R. Wieringa and W. de Jonge. The Identification of Objects and Roles – Object Identifiers Revisited. Technical Report IR–267, Vrije Universiteit, Amsterdam, 1991.

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

[WJH+93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. OMTROLL – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 267–283, 1993.