TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

INFORMATIK-BERICHTE

94–03

Revised Version of the Modelling Language TROLL

(TROLL Version 2.0)

Thorsten Hartmann
Gunter Saake
Ralf Jungclaus
Peter Hartel
Jan Kusch

Abt. Datenbanken
Techn. Universität Braunschweig
Postfach 3329
D–38023 Braunschweig, Germany

Braunschweig
April 1994

Revised Version of the Modelling Language TROLL[‡]

(TROLL Version 2.0)

Thorsten Hartmann[*]
Gunter Saake[§]
Ralf Jungclaus[†]
Peter Hartel[*]
Jan Kusch[*]

April 1994

[*]Abt. Datenbanken, Techn. Universität Braunschweig, Postfach 3329, D–38023 Braunschweig, Germany. E-mail: {hartmann,hartel,kusch}@idb.cs.tu-bs.de

[§]Institut für Technische Informationssysteme, Otto-von-Guericke-Universität Magdeburg, Postfach 4120, 39016 Magdeburg. E-mail: saake@iti.cs.TU-Magdeburg.DE

[†]Telekom, Technical IP-Systems TD42a, Postfach 2000, D–53105 Bonn, Germany. E-mail: jungclau@u9000mst.nez.telekom.de

[‡]This work was partially supported by CEC under ESPRIT-III Basic Research Action Working Group No. 6071 IS-CORE II (Information Systems – COrrectness and REusability). The work of Thorsten Hartmann, Ralf Jungclaus (until 12/93), and Jan Kusch is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3. The work of Peter Hartel is supported by OBLOG Software SA, Lisbon.

Abstract

Conceptual modelling, sometimes in conjunction with requirements acquisition, is widely accepted as the first formal step in information system specification and design. Since knowledge about the world to be modelled is often vague in early development phases, formalisms for conceptual modelling must support a wide variety of concepts for describing real entities. Furthermore a conceptual modelling language must be declarative so that later implementation is not restricted. A large amount of features often hinders learnability and applicability of such languages whereas restriction to few basic features may result in unreadable specifications. In this report we introduce the revised version of the conceptual modelling language Troll that provides a suitable set of declarative concepts that are orthogonal and tailored to description of real world contexts. To support the acceptance of the language, the notation is as near as possible to accepted formalisms like object oriented programming languages without leaving the way of high level specifications. The version of Troll presented here is suited especially for the conceptual modelling and later design phase for information systems specification. As main abstractions Troll supports classes, roles and derived roles (specializations), composite objects, views, and relationships.

Acknowledgements

For many fruitful discussions on the language we are grateful to all members of IS-CORE, especially to Hans-Dieter Ehrich, José Fiadeiro, Gerhard Koschorreck, Udo Lipeck, Amílcar Sernadas, and Cristina Sernadas. The colleagues in Braunschweig have been of great help, too. Special thanks to: Stefan Conrad, Grit Denker, Martin Gogolla, Rudolf Herzig, Perdita Löhr-Richter, Karl Neumann, and Nikolaos Vlachantonis. Thanks also to Scarlet Schwiderski for discussions on example specifications written in a preliminary language version.

Thanks are also due to the students working (or having worked) with us. Especially Cesar Ortiz provided valuable comments on various language features.

Contents

Chapter 1

Introduction

The language TROLL, Version 2.0 we describe in this report is a language designed for the conceptional modelling phase of information systems. It is the successor version of the language TROLL, Version 0.01 introduced in [JSHS91]. We will henceforth call the new version just TROLL, if necessary referring to the old version as TROLL1. Since a survey of related approaches and previous work can be found in [JSHS91, Jun93, Saa93] we will concentrate on language issues in this report.

1.1 Features of Troll

A basic feature of TROLL1 and TROLL is that the description of static and dynamic properties of conceptual entities is *integrated* in object descriptions. These object descriptions are the basic building blocks of system specifications. The concept of object employed by TROLL is different from the notion of object in object oriented programming languages in several aspects. In the next chapters we will try to make these differences clear.

The roots of TROLL can be found in earlier work mainly devoted to semantics of object oriented specification respectively a common formal model of objects [SSE87, ESS88, ESS89, SFSE88, SFSE89, ESS90]. These articles have been the starting point for several language dialects based upon this common model of objects. The different dialects are focused on several paradigms as for example the early version OBL-89 [CSS89] for specification of real world entities (but quite close to the semantic models), or the diagrammatic version OBLOG [SSG$^+$91] that is the root for a more pragmatic, high level programming language directed version.

TROLL1 [JSHS91] is a dialect mainly based on early textual OBLOG versions like OBL-89 and was formerly known as **Oblog$^+$** (see for example [SJ91, JSS91, SJ92]). The main goal of TROLL is the support for declarative specification of conceptual models mainly by means of temporal logic and a variety of concepts backing (hopefully) natural and intuitive specifications. The language TROLL *light* [CGH92] is a simplified version of TROLL without a class concept, inheritance, and temporal logic but with an operational process language for object life cycles. It is tailored towards verification [Con94] and direct execution purposes.

The revised version of TROLL introduced here is focused on more operational aspects and prototyping issues without leaving the way towards high level, declarative specification as introduced with the predecessor version. It is our opinion that a language for the purpose of conceptual modelling has to be as intuitive for the user as possible (a main goal for the graphical versions of OBLOG and the transition from OBL-89 to **Oblog**$^+$, too). The predecessor version lacks this requirement in several aspects. Furthermore the language was not as orthogonal as possible which leads to a long learning phase for newcomers.

We therefore firstly decided to introduce a more traditional notation as known from object oriented programming languages. The basic ideas however remain unchanged. Secondly we got rid of several restrictions of TROLL1 that made specifications unreadable and counter-intuitive. The main changes to the last version are summarized as follows:

- Specification formulae for a *concept* like e.g. attributes, events, or components were scattered over the specification text for one object in TROLL1. We now decided to *group* formulae that describe one *concept*. A concept like an attribute is now introduced with features: a name, parameters, initialization rules, local constraints, and possibly derivation rules. A concept like an event is described by a name, parameters, an enabling condition, a set of attribute changing rules, a set of calling rules, etc. This view is more natural in the sense that specification parts that belong to one concept are grouped together in a concise description of the concept.[1]

- The class and type concept of the former version was based on a *static identification concept* that led to problems when describing highly dynamic objects (often we do not find *constant* properties of objects that can easily be used for object naming purposes). The solution to this problem is to use the widely accepted concept of *object identity* (unchangeable and unprintable internal id's of objects) and combining this concept with a notion of *identification* (object properties (attribute tuples) that uniquely identify objects in real world contexts such as name and birthdate of a person). In the database community identification is usually called a *key*. To manage keys, *class container objects* are introduced as first class (specification) entities in an object society specification. Class container object specifications are not specified by the user of the language but are implicitly generated for a class specification. In a more operational view these implicit object specifications can serve as implementation aids as well.

[1]These are only changes in representation and do not change what is sometimes called *abstract syntax*. There are however some features newly introduced in TROLL2 that could not be expressed in the last version.

- TROLL1 introduced various concepts of *part-of* relationships [HJS92] as for example the *including* concept based on semantic issues of the underlying object (aggregation) model, *subtemplates* as a means to (syntactically) structure object specifications, and *components* as a dynamic counterpart of the including concept that is based on static aggregation of objects. These concepts were not *orthogonal* to each other. For example the naming of components was based on object identities whereas the naming of instances generated from subtemplates was based on parameters of subtemplate symbols. In TROLL as it is now there are only components that can be local or global respectively non-sharable or sharable. Furthermore components may be declaratively described by aggregation predicates (derived components).

- In TROLL1 we distinguished between *specialization* and *roles*, the former a static the latter a dynamic concept to structure the specification of objects (and classes) by *inheritance*. Both concepts can be described by the concept of roles. We decided to syntactically support *roles* as the only mechanism for TROLL, introducing a declarative specialization feature by means of *derived roles*.

  For derived components and derived roles we can also give an operational description with predicate based birth and death.

1.2 A Concept of Objects

Before we introduce the language TROLL let us have a short look at the underlying ideas. For a detailed description the reader may refer to [JSHS91, JSS91, HJS92, HS93]. The basic concept of *object-oriented design* is the concept of *objects as units of structure and behaviour* [SE91]. This way, 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, which are abstractions of methods.

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*). We need snapshots for the modelling of communication (see below). 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:

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

Up to now we only talked about single objects. Often we observe, that a set of entities are of the same kind, in other words we can *classify* objects. We distinguish between *classes* and *class types*. The former denote a *collection* of existing objects, the extensional classification, the latter describes the *possible* instances of an object description, the intensional description.

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 the sum of the observations of the parts where observations of the parts calculated in isolation are maintained. The latter requirement plays the role of *encapsulation* in our model: although the composite entity can be observed as a whole, changes to observations of the parts may be performed only locally.

Real world entities somehow communicate with each other. This is equally true for the system objects representing the real world entities in a computer. 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. Here sets of concurrently occurring events – the already mentioned snapshots – come into consideration again. For details on these issues see [HS93].

Description of objects and object classes may be structured with a notion of inheritance. We distinguish two sorts of inheritance relations, *syntactic inheritance* denotes inheritance of structure and behaviour *definition* and is defined on the type level. *Semantic inheritance* denotes inheritance of the objects themselves and is therefore defined on the instance level. The latter kind of inheritance is known from semantic data models, where it is used to model one object that appears in several *roles* or *aspects* in an application.

1.3 Structure of this Report

As the basic ideas are described in [JSHS91] we will mainly concentrate on language and syntactical issues in this report. Wherever necessary, we will provide references to recent papers.

The report is structured as follows: In the second chapter we will introduce the basic language features for TROLL *templates* as the basic items for object specification. In this chapter we will only talk about *object descriptions* by means of necessary sublanguages, declarations, attribute and event specifications, constraints for attribute observations, life cycle descriptions, and parameterized templates.

In the third chapter we will then go on to use templates to describe *objects* as instances of *classes* including single objects as the sole instance of a class. Closely related to classes are *composite objects* or object aggregations. We decided to introduce components in this chapter because components are later on used to describe the already mentioned class container objects. The end of Chapter 3 is devoted to *object interaction* in composite objects.

In Chapter 4 we will introduce the above mentioned *implicit class object specifications*, the operational specification of object creation and destruction, and some issues concerning the now dynamically changeable keys of objects. The second part of the chapter is used to depict the *role* concept of TROLL. Roles are the only way to describe inheritance relations between object specifications and between objects.

In Chapter 5 we will firstly introduce mechanisms to describe *views* on objects and object classes. These views (and objects as well) are then used by an explicit *relationship* construct describing *global communication* and *global constraints* between separately specified objects.

Chapter 6 describes accompanying work on tool support for specifying object systems with TROLL and finally draws some conclusions and points out further work on implementation of the tools.

Chapter 2

Language features for Troll-Templates

In this chapter we will introduce the *sublanguages* TROLL is based on. The sublanguages are then used to describe the various features of TROLL *templates*. Throughout the rest of this report we will introduce the syntactical patterns by means of grammar productions. A comprehensive grammar is given in Appendix A.

2.1 Sublanguages

Specifications in the language TROLL are based on a number of sublanguages, i.e. specification of concepts like attributes etc. are *sentences* of sublanguages. The basic sublanguages of TROLL are the following:

- *Data terms* for data values and expressions (involving signatures of constant symbols and operation symbols as well as terms over such signatures).

  The specification of *data types* is considered external to TROLL. We only import data type *signatures*.

- *First order logic* for a variety of assertions that can be formulated for objects.

- *Temporal logic dialects* for dynamic constraints on attribute evolution (future tense) and enabling conditions for event occurrences (past tense).

- A language for *process specification* to specify fragments of life cycles explicitly.

The sublanguages define the basic formalisms underlying the language TROLL and are described in detail in [JSHS91]. For the revised version of TROLL described here, we will concentrate mainly on language issues. The sublanguages remain the same as introduced in [JSHS91] except for the process language. However, some comments about temporal logic and process specification are in order here.

Temporal logic is a special logic that provides a system for describing and reasoning about how the truth values of assertions change over time. Thus, temporal logic is well suited to describe *behaviour* of objects, processes, computations etc. In particular, temporal logic is useful for describing and reasoning about the behaviour of non-terminating or continuously operating concurrent systems [Pnu86, Saa89, Eme90].

We use two different temporal logic dialects for constraints on object evolution. The first one, directed to the future, describes possible attribute evolutions. We may for example state properties about future values of attributes like *sometime in the future the value of attribute xyz must be greater than 10*. The second dialect, directed to the past, describes possible enabling conditions for event occurrences depending on previous observations (attribute values) and event occurrences like *sometime after event* e_1 *occurred event* e_2 *may occur*. Note that the last assertion has nothing to do with *activity*, i.e. e_2 is not forced to occur.

The logic sublanguages of TROLL are used to state *properties* of objects, determine the applicability of certain rules in certain states etc. The semantics of data terms and the logic sublanguages in terms of algebras and state sequences is described in [JSHS91]. We will not elaborate this issue here.

2.1.1 Data Terms

Data terms are used throughout a specification in many different places as for describing the change of attribute values, event parameters etc. Data terms are typed so that we have an additional criteria for their correct construction.

We take the practical assumption that only a *framework* for the denotation of data values via data terms is integrated into the language. That is, data terms may be constructed out of sub-data terms connected with infix or unary operators or as function applications denoted by a function name and a list of parameter (sub)data terms. The semantics of such constructions must be described in a suitable framework for data type specification. Nevertheless we introduce the basic numerical operators in the syntax. For enumeration data types that can be constructed with the data type constructor ***enum*** operation symbols without parameters may be used denoting constants of data types (see below).

Data terms denoting values of the data type `bool` serve a special purpose in that they may be used in *formulae* (and vice versa, see also the grammar production for formulae) as well as formulae may be used as data terms of type bool. The equality *predicate* `'='` as well as the comparison operators are predefined as already mentioned are some basic arithmetic operators.

We introduced the connection between the data sublanguage and the formula sublanguage for convenience.

Syntax

<*inf_op*> ::= **+** | **-** | ***** | **/** | **div** | **mod**

```
<compare_op>      ::=  = | <> | < | > | <= | >=

<unary_op>        ::=  ~ | -

<post_op>         ::=  [<data_term>] | .<tuple_sel_id>

<const_symbol>    ::=  <nat_const> | <float_const> | <bool_const>
                       | <char_const> | <string_const>

<data_term>       ::=  <data_term> <inf_op> <data_term>
                       | <data_term> <compare_op> <data_term>
                       | <unary_op> <data_term>
                       | <data_term><post_op>
                       | <op_id>[(<data_term_list>)]
                       | (<data_term>)
                       | if <formula> then <data_term> [else <data_term>] fi
                       | <var_id> | <parameter_id> | <const_symbol> | <att_term>
                       | undefined | <formula>
```

A data term is recursively constructed out of *(sub)data terms* and *operators*. The leaves of the so defined syntax trees consist of *attribute terms*, *variable id's*, *parameter id's*, or *constant symbols*. The special value *undefined* is considered as available in all data types. Furthermore we assume that all operations applied to the value *undefined* yield undefined and that comparisons with the value *undefined* yield *false*. The predicate *undef* (see again production for formula below) is used to test if for example an attribute is not defined i.e. *undefined*.

The *if then else* operator is assumed to have an implicit *else undefined* part if the *else* part is missing.

The only *postfix* operators used in TROLL are the *tuple selector* and the *list selector*. Data terms of type *tuple* have special postfix operators defined with their declaration.[1] For example a variable declaration like:

```
variables xpto:tuple(itemA:nat,itemB:bool)
```

defines the postfix operators (selection functions) `.itemA` and `.itemB`, i.e. we may write `xpto.itemA` to denote a value of type `nat`, the first position of the tuple etc.

The list selector `[..]` can be used for data types of type *list*:

```
variables xpto:list(string)
```

We can write `xpto[3]` to denote the third element of the list `xpto` provided the length of the list is ≥ 3. Note that tuple selectors and list selectors are written without separating blanks.

[1]For the introduction of *declarations* refer to Section 2.3.

A special treatment must be devoted to *attribute terms*. As we will see in the next sections, attributes may be locally defined in a template, be included by means of *components*, or be *inherited* from another template specification. So we possibly have to supply *prefixes* denoting the *paths* to such attribute symbols:

Syntax

```
<att_term>      ::=  [<selector>]<att_id> [(<data_term_list>)]
<selector>      ::=  <selector><select_id>. | <select_id>.
<select_id>     ::=  <class_id> | <ovar_id> | <cmp_term>
<cmp_term>      ::=  <cmp_id>[(<data_term_list>)] [<obj_ref>]
<obj_ref>       ::=  (<data_term>) | ()
```

For the various identifiers we may only use symbols that can be found following the inheritance chain or resp. components chain. This will be made clear in the sections on inheritance respectively composite objects.

2.1.2 Logic Sublanguage

As in the previous language version we basically introduce three logical sublanguages for specifying properties of *states* (*first order logic*), properties of state sequences of the *past* and *future* object life (*past directed temporal logic, PDTL* and *future directed temporal logic, FDTL*). In contrast to TROLL1 however, the keywords are clearly separated using *always* and *sometime* for the past directed version and *henceforth* and *eventually* for the future directed counterparts.

For past formulae we introduce special predicates *after* and *occurs* that have an event term as parameter. The predicate *after*`(evt)` is true in states reached by occurrence of an event denoted by event term `evt` whereas *occurs*`(evt)` is valid in a state where the event denoted by event term `evt` occurs. We included both – an *after* and an *occurs* predicate in TROLL although we will often only use the *after* predicate. *Occurs* is necessary for special applications as for example integrity monitoring for past temporal enabling conditions [SHS93]. In that context it is necessary to restrict events from occurring in snapshots relative to other events of the snapshot. In general, the *occurs* predicate can be used to restrict events that must not occur together in a snapshot.

Syntactically all sublanguages are introduced by means of a general production for `formula`[2]:

Syntax

```
<formula>        ::=  <formula> <bool_op> <formula>
                      | not <formula> | <bool_const>
                      | (<formula>)
                      | <quantifier> (<var_decl_list>:: <formula> )
                      | <data_term>
                      | undef(<data_term>)
                      —— predicates for past tense ——
                      | after(<evt_term>) | occurs(<evt_term>)
                      —— past tense temporal logic ——
                      | always <formula> | sometime <formula>
                      | previous <formula>
                      —— bounded past predicates ——
                      | always <formula> sincelast <formula>
                      | sometime <formula> sincelast <formula>
                      —— future tense temporal logic ——
                      | henceforth <formula> | eventually <formula>
                      | next <formula>
                      —— bounded future predicates ——
                      | henceforth <formula> until <formula>
                      | eventually <formula> before <formula>

<var_decl>       ::=  <var_id_list> : <domain>

<bool_op>        ::=  implies | and | or

<bool_const>     ::=  true | false

<quantifier>     ::=  forall | exists
```

We can decide syntactically to which sublanguage a specific formula belongs. The latter property is necessary, since we want to restrict the use of certain temporal dialects to clearly defined parts of a TROLL specification. For example we do not want to use FDTL as enabling conditions (see Section 2.4.2.3) for events.[3] A past formula thus may only contain past predicates and boolean connectives, a future formula only future predicates and boolean connectives, and a simple formula may only contain boolean connectives. The occurrence of ***after*** and ***occurs*** predicates is restricted to past formulae.

[2]Note that this definition is introduced for convenience. Formally we have to distinguish between the different logical sublanguages.

[3]Although semantically possible for a declarative specification language, for TROLL it is not desired to restrict state transitions depending on the future object behaviour. This way we would introduce implicit obligations for an objects life in enabling conditions.

In this report we will not go into details of the semantics of the temporal logic languages but refer the reader to [JSHS91] or a textbook like [MP92] Part II. Note that there are some differences in operator names to TROLL1. To give a flavour of the meaning of the temporal operators we will give some short descriptions. As usual temporal formulae are evaluated in *state sequences*:

$$\sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_{j-1}, \sigma_j, \sigma_{j+1}, \sigma_{j+2}, \ldots$$

(for details see [JSHS91]). In the following we assume p and q to denote formulae of the "right" sublanguage, i.e. if we describe past operators, p and q are past formulae etc.

Past Temporal Operators

always p holds at position j iff p holds at position j and all preceding positions.

sometime p holds at some position in the sequence from position 0 to position j.

previous p holds at position $j > 0$ if p holds at position $j - 1$. It also holds at position 0.[4]

always p **sincelast** q holds at position j iff q held in state i and p held continuously between states i and j (exclusively i, inclusively j). The state i is defined as the maximum state where q held in the past, or as $i = -1$ if q did not hold in the past.

sometime p **sincelast** q holds at position j iff q held in state i and p held in *some* state between states i and j (exclusively i, inclusively j). The state i is defined as the maximum state where q held in the past, or as $i = -1$ if q did not hold in the past.

Future Temporal Operators

henceforth p holds at position j iff p holds at position j and in *all* successive states ("from now on").

eventually p holds at position j iff p holds at *some* future state, or at position j.

next p holds at position j iff p holds at position $j + 1$.

henceforth p **until** q holds at position j iff p holds at position j and in *all* successive states until the *first* state i where q holds $(i > j)$.

eventually p **before** q holds at position j iff p holds at *some* future state or at position j before the first state i where q holds $(i > j)$.

[4] We do not introduce existprevious and existnext operators in TROLL as in TROLL1 [JSHS91].

For a formal definition see [JSHS91]. Note that we changed the names of the operators from *alwaysf* to ***henceforth***, and from *sometimef* to ***eventually*** and that the bounded operators for the future look like the bounded operators for the past.

2.1.3 Process Specification Language

The definition of processes is a little bit more subtle. A process definition is intuitively seen as a *sequencing schema* for event occurrences. That is, with a process definition we can describe (parts of) life cycles of objects.

As usual in process description languages we introduce "operators" for *sequencing*, *choice* and *guarded processes*, *loops*, *parallel*[5], and for *recursion* presented in a human readable syntax.

2.1.3.1 Process Event Terms and Process Terms

Basic ingredients for a process definition are *process event terms*. Process event terms consist of an event symbol, optionally of a preceding selector, and a list of *process parameters* denoted by data terms. The use of selectors is motivated the same way as for attributes:

Syntax

<*evt_term*>	::=	[<*selector*>]<*evt_id*> [**(**<*proc_param_list*>**)**]
<*process_term*>	::=	[<*selector*>]<*process_id*> [**(**<*proc_param_list*>**)**]
<*proc_param*>	::=	['**?**']<*var_id*> | <*data_term*> | ['**?**']<*param_id*>

Event terms are special for processes in that they may contain optionally *?* preceded parameters. These parameters are *set* during the course of the process executing. Operationally these parameters are bound somewhere during execution and henceforth refer to this value.

A similar role play process terms that are place holders for a process declared elsewhere. Like for process event terms we may prefix some parameters with a *?*. The meaning of such parameters is the property to be *set* during the process associated with the identifier. Note that for process event terms and process terms parameters with a *?* can only be *variables* or *parameters*, i.e. local to the process.

2.1.3.2 Process Specification

The process sublanguage defines sequences of events that are considered as *allowed* (parts of) object life cycles. Additional to the operators *sequencing*, *choice*, *parallel*, and *guarded processes* we provide a *for each* construct for *finite iterations*. *Recursion*

[5]The parallel operator here implies synchronization.

is introduced via usage of process terms that is described in Section 2.6.1. The following productions generate the syntactically possible process descriptions:

Syntax

```
<process>            ::=  <process> -> <process_unit> | <process_unit>
<process_unit>       ::=  <process_term> | <evt_term>
                          | <choice> | <parallel> | <foreach> | nil
<choice>             ::=  ( <choice_alternative> )
<choice_alternative> ::=  <guarded_process> | <choice_alternative>'|'<guarded_process>
<guarded_process>    ::=  [{ <formula> }] <process>
<parallel>           ::=  ( <parallel_events> )
<parallel_events>    ::=  <evt_term> | <parallel_events>'||'<evt_term>
<foreach>            ::=  foreach <var_id>:<data_term> do <process> od
```

TROLL does not only introduce a *pure* process language since we may refer to state dependent parts of objects via formulae and terms build over attributes.

Process sequencing schemata are constructed out of process units. A process unit is:

- A *process term* constructed from a process identifier (see Section 2.6.1) and a list of parameters referring to attributes, constants, variables etc. A variable can be preceded with a '?' denoting that this variable is not free but must be bound to a value in the course of the process associated with the process identifier.

- A *process event term* constructed from an event identifier and a list of parameters like for process terms. '?' preceded variable parameters are bound to values during "execution" of the process, i.e. they are *set* by other objects.

- *Nil* to denote the empty process.

- A set of *guarded (sub)processes* grouped with round brackets and separated with vertical bars '|' denoting a *choice*. A guarded process is a process (optionally) preceded with a (past) formula in curly brackets. Such an alternative or choice is a valid continuation of a process if its guard is valid in the state reached so far. To describe a process that uses neither of the alternatives we can supply the process unit *nil* to the set of alternatives.

- A *for each construct* as a finite iteration depending on the state of the object. A *foreach* is defined with a reference to a *set valued* data term. The semantics of *foreach* is a nondeterministic *sequence* of subprocesses for all values of the set (the data term denoting the set is evaluated in the state before the *foreach* unit starts). Usually the variable declared with *foreach* is used in the process itself. We do not introduce a *co-routine* facility into the process language. In other words: concurrency must be modelled at the object level or in a restricted way by means of the parallel operator.

- *The parallel operator* is used to describe events that occur at the same time in the course of a process execution. The semantics of a parallel construct is *synchronization* of events, i.e. all events mentioned as parallel have to occur in one snapshot. We will see later on that this object behaviour resembles *calling* of events. Here however such synchronization is only done for a particular process. In another process specified for an object the mentioned events may be unrelated again.

In Section 2.6 we will describe how process specifications can be further refined and how they are used to describe the long term behaviour of objects. For the time being, we will provide a simple example of a process describing the user interaction with an automatic teller machine:

A sample process specification (not real TROLL text)

```
variables Bal:money; A:money; Valid:bool;
arrival -> enterCard(MyCard) ->
enterPINCode(MyPIN) -> ( checkPinCode(?Valid) || recordPinCode ) ->
(  {Valid}
     readBalance(?Bal) -> enterAmount(?A) ->
     throwOutCard -> removeCard ->
     throwOutMoney -> getMoney(A)
  |
   {not Valid} throwOutCard -> removeCard
) -> leave
```

On arrival at the machine the user enters his/her card (`MyCard`), enters the PIN code (`MyPIN`), the PIN code is checked and recorded in parallel. Depending on this check two different alternatives can be executed: normal operation or an exit by the machine. `MyCard` and `MyPIN` are assumed to be attributes of the object specified (see below) whereas `Valid`, `A`, and `Bal` are *variables* set during 'execution' of the process. For illustration this example may be enough. We will return to process descriptions in Section 2.6.

2.2 Template Structure

In this section, we present the components of templates before we describe the specification of objects based on templates and object *identification*. In the course of discussion we will often talk about *objects* meaning objects that behave like specified in templates. This way templates are *generic descriptions* of properties of objects. They define the shape of objects in terms of observable attributes as well as their possible behaviour in terms of possible events and event sequences. Objects then are introduced by means of classes, collections of objects behaving like specified in their associated templates.

Templates in TROLL have the following structure:

```
template Name
   < Imports and Declarations >
   < Component, Attribute, and Event Specification >
   < Object Behaviour Specification >
end template Name
```

Although the structure is not fixed due to the syntax rules (see below) the sequence shown above should be used as a convention. A template name must be introduced if we specify a template for later reuse in one or more class descriptions. All template specification parts mentioned above are optional, but of course there are some constraints on the existence of some concepts depending on other ones.

Syntax

<*template_spec*>	::=	**template** <*template_id*> [**(**<*dt_param_id_list*>**)**] <*template_desc_items*> **end template** <*template_id*>
<*template_desc*>	::=	**local classes** <*class_spec_items*>
	|	**components** [<*var_spec*>] <*cmps_spec_items*>
	|	**attributes** [<*var_spec*>] <*atts_spec_items*>
	|	**events** [<*var_spec*>] <*evts_spec_items*>
	|	**constraints** [<*var_spec*>] <*constr_seq*>
	|	**process declaration** [<*var_spec*>] <*process_decl_items*>
	|	**processes** [<*var_spec*>] [<*process_use_items*>]
	|	**interaction** [<*var_spec*>] <*c_interaction_seq*>

In a template specification the description entities ***components***, ***attributes***, and ***events*** describe the *signature of a template*. As such, their names must be *unique* in the template at hand. The same is true for names of local classes that also define local signature elements. As we will see later on, (instances of) local classes can be referenced only via components. Since parameters are allowed for components, attributes, and events, uniqueness is defined for the identifier together with its parameter sorts.

Optionally we can use *parameterized templates* where we introduce a template parameter that is later on substituted by a data type when the template is used in a class description. We will elaborate issues concerning parameterized templates in Section 2.8.

In the following sections we will describe most of the beforehand mentioned template features in detail. Some parts will be sketched only and discussed more deeply later on.

2.3 Declarations

2.3.1 Data Types

Data types are not specified in the TROLL framework. Nevertheless data types are necessary to describe the data storage of objects, the transfer of data during communication and not at least changes of the observable states of objects. Data types are once incorporated into a specification by means of the society specification data type part and can then be used in templates as domains for attributes, etc. (see Section 5.3). As for a variety of programming languages we assume the following data types as predefined for the specification language TROLL, i.e. they do not have to be imported:

```
bool, nat, integer, real, char, string
```

For the boolean and arithmetic data types we also introduced the basic operators like ***and***, ***or***, ..., `+`, `-`, etc. into the TROLL-syntax (see the relevant grammar productions in Appendix A). Additionally we may *construct* data types applying the data type constructors:

set, *list*, *tuple*, and *enum*:

to one of the predefined data types or an incorporated (externally specified) data type. For operations defined for data types declared with these constructors see Appendix B.

Data type domains are introduced by means of the following grammar productions:

Syntax

```
<domain>        ::=  <data_type_id> | '|'<class_id>'|'
                     | tuple(<domain_item_list>)
                     | list(<domain>) | set(<domain>)
                     | enum(<enum_id_list>)

<domain_item>   ::=  <tuple_sel_id>:<domain>
```

For the tuple constructor we introduce tuple selector identifiers to refer to components of a tuple by means of symbolic names. The tuple selector identifiers can be postfixed to data terms using the traditional dot notation.

For an explanation of the possible operations on the data types constructed with type constructors see [JSHS91]. Special treatment is necessary for identifier data types. As we will see, identifier data types are used as *"handles"* to objects. Together with class names they are thus something like *references*. However, the only operation on identifier data types visible for TROLL *users* is the equality test. The most important feature in this context is that identifier types can be stored as attribute values etc.

2.3.2 Local Classes

As briefly mentioned in the previous section classes are abstraction mechanisms above templates. Classes are *collections* of *similar* objects. As such objects can be visible to the whole society, i.e. being components in several other objects, or be visible for only one object. The local class section makes it possible to define such local classes.

Syntax

local classes <*class_spec_items*>

Local classes basically have the same representation as global classes that are introduced in Section 3.1. For the time being, we refer the reader to Section 3.2.8.

2.3.3 Variable and Parameter Declarations

To describe various properties of objects we introduce several kinds of formulae all based on sorted first order predicate calculus. Variables used in these formulae must be declared. Variable declaration are expressed according to the following syntactical rules:

Syntax

<*var_spec*>	::=	**variables** <*var_decl_seq*>**;**
<*var_decl*>	::=	<*var_id_list*> **:** <*domain*>

The meaning of such variable declaration is a universal quantification for each formula in a given block, e.g. a constraint or interaction description etc. (see below).

Parameters on the other hand may be used to describe attributes, events and components in more detail. Syntactically they are declared similar to variables.

Syntax

<*param_decl*>	::=	[<*parameter_id*>**:**]<*domain*>

One reason to introduce the second form of variable declaration is the possibility to declare parameters *locally* to attributes, events, and components together with their data types and to refer to them locally. The use of symbolic names for parameters locally in attribute, event and component descriptions is thus supported to enhance the readability of a specification document. Another reason for such parameter declaration is to be closer to traditional notations (e.g. formal parameters of procedures etc.).

2.4 Attributes and Events

The attribute and event specification defines the observable properties of objects respectively the state changing operations. They define the local signature of an object. Attributes and events are specified with names and named parameters together with properties further refining the specification. Although parameters need not be named, names will enhance the readability of a specification text. In most cases names are necessary to *refer* to parameters (see below). In the next two sections we will introduce these properties in more detail.

2.4.1 Attributes

The attribute section of a template specification defines the *observable properties* of objects. Attributes in TROLL can be compared to instance variables of languages like e.g. Smalltalk. However there is a difference in that attributes in TROLL are *visible* at the object interface. Encapsulating attributes and only providing *access methods* is considered an *implementation related* concept that seems not to be useful for semantic data modelling (for an overview of the mechanisms supported by semantic data models see also [BMS84, UD86, HK87, PM88]).

Attributes in TROLL are specified with a name and type. Optionally attributes may have parameters, thus introducing *attribute sets*. Attribute parameters are specified with an optional name, which is considered as a formal parameter declaration similar to a variable declaration. Parameterized attributes define one attribute for each possible value of the parameter sorts (data types).

Syntax

(For: **attributes** [<*var_spec*>] <*atts_spec_items*>)

```
<atts_spec>      ::=  <att_id>[(<param_decl_list>)] :<data_type>
                        <att_desc_items>.

<param_decl>     ::=  [<parameter_id>:]<domain>
```

```
<att_desc>        ::= inherited from <class_id>
                      | hidden | constant
                      | restricted <formula>
                      | initialized <data_term> [default]
                      | derived <data_term>
```

The naming of parameters is needed to make it possible to refer to them in the description part of attributes. In the next paragraphs we will introduce the various options of an attribute specification in more detail. The *inherited from* clause is explained in the section on inheritance (Section 4.2.1). There we describe how features of attributes can be *refined* in role classes.

2.4.1.1 Attribute Constraints

Attributes in TROLL are typed. Additional to this simple form of attribute constraint we may specify *constraints on the possible values attributes can take*. For example an attribute `Age` of type `integer` in a `Person` object may take values in the range `0` to `150` (to be optimistic). For such constraints, attributes can be specified as being *restricted*. A formula after the keyword *restricted* must hold for every value that is to be stored with this attribute. For example:

```
...
  attributes
     Age:nat restricted Age >= 0 and Age <= 150.
...
```

The formula may be constructed out of the attribute name, names of attribute parameters, constants, and suitable operators. For this kind of constraint it is not allowed to specify dependencies among attributes. See the section on *constraints* below (Sec. 2.5). One minor deviation from this rule is introduced in an example below.

Another kind of constraints on attributes that must be considered local are constraints on possible *attribute parameter values*. Attribute parameters are used to describe sets of attributes of the same kind. As an example see attributes `IncomeInYear` of type money that record the earnings of an employee object for each year:

```
...
  attributes
     IncomeInYear(Year:nat):money
       restricted Year > 1870  and Year <= 2030 .
...
```

This time, not the *possible values* of the attribute defined are restricted *but* the possible *set of attributes* defined by the parameterized attribute specification. Since this constraint is also *local* to one attribute *specification*, it can be formulated with the *restricted* option for attributes.

The semantics of such a restriction is explained as follows. The formula has to be valid for a given attribute and its value. For variable substitutions where the formula is invalid, the value of the attribute denoted by `IncomeInYear(Year)` has to be *undefined*. The value *undefined* is a value of all data types in TROLL. The following formula describes this formally:

```
forall(Year:nat ::  (Year>1870 and Year<=2030)
                       or undef(IncomeInYear(Year)) )
```

where the predicate *undef* is defined as yielding ***true*** if the attribute has value *undefined* else *false*. We have to use the build in predicate *undef* because comparison with the value *undefined* always leads *false* as all data type operations that take at least one actual parameter value that is *undefined* yield *undefined*.

Above we mentioned the rule "attribute restrictions can only refer to the attribute itself, parameter values and constants." Formally, an attribute term with different parameter values denotes *different attributes*. We allow one deviation from this rule as depicted in the following example:

```
...
  attributes
     IncomeInYear(Year:nat):money;
       restricted IncomeInYear(Year+1) >= IncomeInYear(Year).
...
```

Here *different attributes* can be used in a *restricted* clause. The example shows that we may introduce restrictions that are considered not local to one attribute but local to an attribute *specification*.

2.4.1.2 Initializations and Defaults

Attributes as observable object properties may change their values upon occurrence of state changing operations (events, see below). The first state plays a special role in the life of an object. By default attributes have the value *undefined* upon birth of an object. In contrast to the default case we may specify attribute values for the first object state. This can be done in two different ways.

Firstly we can provide a data term constructed in similar ways like the formulae in attribute restrictions, namely constructed out of names of attribute parameters, constants, and data type functions. The initialization now is specified as a *data* *term* after the keyword *initialized*. Furthermore an initial value can be *strict* or may be *overridden*. The latter case is marked with the keyword *default*.[6]

As example see the following specification:

```
...
  attributes
    Age:nat initialized 0 default .
    HasBooks:set(|LibBook|) initialized emptyset .
...
```

The first initialization rule assigns `0` to the attribute `Age` upon birth of an object. *Default* is used to state that a special birth event (see below) may *override* this value. This does not hold for the second rule describing `HasBooks`. `HasBooks` is initialized to an empty set of *identifiers* like `|LibBook|` for `LibBook`s. Overriding of the initial value *emptyset* is not possible. Object identifiers will be introduced in the section about classes. For the time being, identifiers are simple data type values.

Secondly as already mentioned we may introduce *birth events* with initialization parameters (for examples see below). If this variant is used in conjunction with the first initialization, the initialization rule must be classified as *default*.

With the specification of *initialized...default* rules the specifier states that he/she wants to have the possibility of supplying another initial value for some objects. Omitting the *default* keyword states that it is forbidden to supply another value upon birth. This feature can thus be used as a hint for the reader of a specification and to document the semantics of a certain initialization rule.

In the next example, we introduce initialization of attributes with parameters. Now we have to supply initial values for all attributes defined this way:

```
...
  attributes
    IncomeInYear(Year:nat)
      restricted Year > 1870  and Year <= 2030 .
      initialized 0 .
    Xpto(N:nat)
      initialized N+1 .
    ....
...
```

The attribute (set) `IncomeInYear` is initialized to zero for all parameter values that do not violate the restriction specified whereas the value of the attributes `Xpto` depends on its parameter. Thus the semantics of this specification can be illustrated as follows (pseudo code notation):

[6]We used the word *default* further classifying initialization to stress that the initialization rules become invalid for object creation events that want to provide different initial values.

```
 “ upon birth of an object ”
variables Year,N:nat;
IncomeInYear(Year) :=
    if Year>1870 and Year<=2030
        then 0
        else undefined
    fi ;
Xpto(N) := N + 1 .
```

In principle we can also define initialization rules like the following:

```
attributes
  F(N:nat)  initialized if N=0 then 1 else N*F(N-1) fi .
```

Clearly this is an interesting issue for a specification language like Troll. For executing such a specification we have to provide some kind of *lazy evaluation*. As mentioned earlier we want to leave this point open in the syntactical representation.

In Section 2.4.2 we will introduce *birth* events that describe object creation and in Section 2.4.2.8 we will introduce initialization of attribute values based on *birth events* that is introduced for convenience.

2.4.1.3 Attribute Derivation

Attributes of an object may be either *stored* or *computed*. Latter attributes are named *derived attributes* in Troll. A `Person` template may specify the `Birthdate` or the `Age` property of a person and derive either of them in terms of the current date. The value of a derived attribute is thus determined by a data term over the observable properties of an object. This data term is written after the keyword *derived*.

For example see the following specification:

```
...
  attributes
    BirthYear:nat
      derived YearOf(Today) - Age .
...
```

The `BirthYear` attribute is *calculated* from the current date (assuming `Today` is another attribute of the object at hand)[7] and the `Age` attribute of the person. `YearOf` is an operation of the data type `date`, namely extracting the year part of a date value.

Attribute derivations must be specified carefully. As an example look at the following derivation rule:

[7]In a real example this would be an attribute of an object `Calendar` that has to be referenced somehow.

```
...
  attributes
    xpto(n:nat):nat
      derived
        if n>0 then n * xpto(n-1) else 1 fi .
...
```

Regarding only attribute *names* without parameters this definition contains *cycles*. Nevertheless it is a consistent specification of the factorial function described with parameterized attributes. To avoid such situations that may not always be obvious at first reading of such specification we may restrict the derivation data term to contain only stored attributes or perform tests on the circularity of definitions. The syntax is liberal so that analyzer tools can warn the specifier depending on the current strategies.[8]

2.4.1.4 Interfacing

As mentioned in the beginning of the attribute description section attributes are visible in Troll. In other words attribute values can be observed from other objects. In contrast to other languages, mainly programming languages, state information is not completely encapsulated. To obtain *privacy* for some specification details, attributes can be specified as being ***hidden***. Hidden attributes are used inside the template at hand like ordinary attributes. For another object hidden attributes cannot be accessed. For example we may specify some attributes as hidden (with stored values) and some others as *derived* attributes with values determined via the hidden attributes.

As an example for derived and hidden attributes we provide the following specification fragment describing persons:

```
template person
  attributes
    Likes:set(|Person|) hidden.
    Birthdate:date hidden .
    Age:nat derived YearOf(Today) - YearOf(Birthdate).
 ...
end template person
```

where the `Likes` and `Birthdate` attributes are not made public but can be used in local formulae.

[8]In general we avoided to restrict the syntactical rules too far so that tools supporting the specification process can be adjusted to the features that can be handled in a given phase of development of a specification document. For example in early phases of specification we may use arbitrary temporal formulae. In later phases we may want to *prototype* such specification but the prototyping tool can handle only a restricted set of temporal formulae. Then the tool has to insist that such formulae must be transformed to simpler ones. The language however must be open.

Since *hiding* or *interfacing* in this sense is useful for specifying private properties of objects, the keyword ***hidden*** can be used also in the *events* and *components* specifications (see below) that define the *operation* and *composition interface* of template specifications.

We will provide examples for the *encapsulation* property for events in the following sections. The feature ***hidden*** can be used for events and components similar as for attributes.

2.4.2 Events

An important part of an object behaviour description is the specification of events that can take place in an objects life. According to approaches of object oriented languages in general we identify three different aspects of events:

1. An event can be allowed in a given object state or be forbidden in a given state. In terms of functional specification we speak of *necessary preconditions*, *safety rules*, or *enabling conditions*.

2. An event can have *effects* on the local state of an object. In other words we observe a new state after an event occurs. In more traditional terms events change state variables (attributes).

3. An event can involve other events in different objects or in the object at hand. To achieve the desired functionality an event of an object triggers *communication* with other objects.

These aspects are among others grouped together into an *event description* for a given event and a given list of formal parameters. In former TROLL versions the description of the three different aspects mentioned above where introduced in different sections of the template. For TROLL we propose an integrated view where an event is described by the necessary conditions that must be fulfilled (***enabled***), the local change of state (***changing***), associations with other events (***calling***), and return values (***binding***) of events (see below). A *derivation* feature is not introduced for events.

The description of events has the following syntactical representation:

Syntax

(For: **events** [<*var_spec*>] <*evts_spec_items*>)

<*evts_spec*>	::=	<*evt_id*>[**(**<*df_param_list*>**)**] <*evt_desc_items*> **.**
<*df_param*>	::=	[**!**][<*param_decl*>]

```
<evt_desc>          ::=  inherited from <class_id>
                         | birth | death | active | hidden
                         | enabled <formula>
                         | changing <c_changing_seq>
                         | calling <c_calling_seq>
                         | binding <c_binding_seq>
```

This concrete syntax is a merge between the logic based view of specification languages (‘a specification is a set of formulae’) and the operational view of object oriented programming languages (‘methods define state changes’). The semantics of most of the sections introduced so far however is *not* changed compared to former TROLL-versions.

The head of an event specification is an event description build from an event name and a list of (typed) formal parameters with optional *binding responsibilities*. A ‘!’ preceding a parameter denotes parameters that are instantiated by the specified event itself, that is, they *must not be instantiated by the caller*. The instantiation or binding of parameters to values is in the *local* responsibility of the event respectively object specified.

In the next paragraphs we will introduce the various options of an event specification in more detail. Again, the ***inherited from*** feature for events is explained in Section 4.2.1.

2.4.2.1 Activity

TROLL specifications describe some Universe of Discourse (UoD) in its whole (according to some abstraction level). In such UoD some events may be observed as *passive*, i.e. events that only occur if they are triggered from some other events. Other events may occur spontaneously, i.e. there is no *causal dependency*[9] specified. In a UoD of a library for example we may specify Users with events like the following:

```
template User
  ...
  events
    borrowBook(|LibBook|) active .
  ...
end template User
```

Such events – although causally dependent on other events (the need for reading a book) – must be classified as active if we abstract away this dependency. This declaration is performed with the keyword ***active***.

[9]Causal dependency must not be confused with *causality*. Causality usually means *reactions* on stimuli later on in the objects life.

Active events are not events which are *driven by the running system itself* in other words certain events that are given *cpu resources* in the sense of method execution. For implementation and design purposes such an *initiative* concept is necessary because describing implementation, we have to model event sequences that are more than active events in the sense that there is additionally a *driving force* for their occurrence. A method execution or the execution of a sequence of statements in a program is an example for such a notion of initiative.

Active events are events that are not *forced to occur* by events specified in a TROLL document. They are thus the *interfaces* to the real world: they describe the boundary to real objects and abstract away a causal dependency to parts of the real world that are not specified.

2.4.2.2 Birth and Death

Two special classes of events are provided that mark the *creation* and *destruction* of objects (keywords *birth* and *death*). According to the life cycle of an object, *birth* events may only occur as the *first* event in an objects life whereas death events are the *ultimate* events in a life cycle. Person objects may be specified with events like the following:

```
events
  born birth .
  die death .
```

to denote the first event in a `Person` objects life (`born`) and the last event in a `Person` objects life (`die`).

Birth events are obligatory for an object specification. Death events on the other hand are optional. Objects without death events are supposed to *live forever*. For example a `Book` may be considered an ever lasting entity that once written and published will never be destroyed (in technical terms: we forbid to remove the book info from a database).

As mentioned in the attributes section, birth events with parameters can be used to *initialize attributes*, too. We will elaborate this topic in Section 2.4.2.8.

2.4.2.3 Enabling Conditions

One way to restrict event occurrences in an object is by means of *enabling conditions* mentioned after the keyword *enabled*. Enabling conditions state conditions that must be fulfilled in order to allow an event to occur, i.e. the other way round, they ensure that *something must not happen* [Lam83] if the condition is not valid in a given state. Enabling conditions in TROLL are formulae build over state variables (attributes) and parameters of events.

The meaning of an enabling condition can informally be described as follows. An event, instantiated with actual parameters, can only occur if the condition holds in the current state. Inside the condition the event parameters can be used to refer to the actual parameter values when the event occurs. In a bank context a `withdraw` event in an `Account` object may for example only occur if the current `Balance` attribute of the object is greater or equal to zero:

```
events
  withdraw(Amount:money)
    enabled Balance >= 0.0 .
```

To be more general, we may not only refer to the current state of an object in defining enabling conditions. To describe more complex life cycles, we can also use a past directed temporal logic to refer to the current *history* of an object. The special predicate ***after*** frequently used in this context is true in the state *after* the mentioned event took place. Together with the temporal quantifiers, conditions refer to object histories. We may for example specify that a `withdraw` event can only occur if a `deposit` event has occurred with an arbitrary parameter value sometime in the past:

```
events
  variables m:money;
  withdraw(Amount:money)
    enabled  sometime after(deposit(m)) .
```

The enabling condition remains valid after the first occurrence of a `deposit` event that occurs in an objects life.

Positively speaking, enabling conditions only state that an event *may* occur (is enabled), not that it occurs in a given state. The validity of a general past formula can be evaluated with knowledge of the whole history of an object. For a given formulae we can reduce the necessary information about the history. Therefore we have to collect information about attribute values and event occurrences in the past (according to the point in time where the formula is evaluated) that are used in the formula. First steps to investigate the derivation of this information are reported in [Sch92, SS93, SHS93].

Conditions in past temporal logic thus may be used to describe the possible occurrences of events depending on other events that occurred in the former life of an object. To describe more complex life cycles we may specify further conditions with the *process description feature* of templates. Often the conditions used as enabling conditions will be relatively simple involving only first order logic or unnested temporal formulae with an after predicate. Nevertheless the syntax allows arbitrary past temporal logic conditions.

2.4.2.4 State Change Specification

An important part of an event description are *state change rules*. Here we describe how the values of attributes changes with the occurrence of events. State change rules are defined by a restricted form of sentences of a positional logic [FS90]. Sentences in this logic refer to *positions* in the life of objects, namely event occurrences and describe formulae valid in the state *after* the event occurrence.

In case of TROLL these formulae are restricted to equalities with a single attribute term on the left hand side, i.e. assignments. We allow for conditional rules that are only valid if the given condition holds in the current state. For convenience we allow lists of assignments for a given condition.

Such assignment rules in event descriptions have two parts:

- An optional condition formulated in first order logic. The condition describes if the assignment has to be applied in the current state of the object. In contrast to the last TROLL version we only allow first order conditions because past temporal logic should be reserved to describe possible life cycles via enabling conditions.
- A change list consisting of a list of assignments that become observable *after* the event occurred. An effect is denoted by an attribute term (left hand sides of the assignment) and an arbitrary data term on the right hand side.

Syntax

```
<c_changing>        ::=   [{<formula>}] <changing_list>

<changing>          ::=   <att_term> := <data_term>
```

State change rules are expressed as ***changing*** rules as the following example taken from the specification of a book in a library shows:

```
...
  attributes
    OnLoan:bool initialized false.
    Due:date .
    Borrowers:list(|Person|) initialized emptylist .
    BadGuys:list(|Person|) initialized emptylist  .
  events
    borrow(From:|Person|,At:date,Days:nat)
      changing
        OnLoan := true;
        Due := AddToDays(At,Days);
        Borrowers := insertfirst(From,Borrowers).
    return(From:|Person|,At:date)
      changing
        OnLoan := false;
        { At > Due } BadGuys := insertfirst(From,BadGuys),
                          ... ;
        ...
```

The event `borrow` changes attributes `OnLoan`, `Due`, and `Borrowers`, whereas the `return` event changes `OnLoan` but the attribute `BadGuys` is only changed if the return date is to late. An alternative form of the last rule is the following:

```
return(From:|Person|,At:date)
  changing
    ...
    BadGuys := if At > Due
                  then insertfirst(From,BadGuys)
                  else BadGuys
               fi ;
    ...
```

The data type sublanguage contains an *if then else operation* for all predefined data types. In this example the former conditional changing rule is more natural since we need not specifying that nothing is changed if the condition is not valid. In fact there is a *frame assumption* in TROLL that attribute values can only be changed by occurrence of events if and only if there is a changing rule for a particular event/attribute pair [JSHS91, Saa93, Jun93].

The general form of the change specification allows *lists* of attribute changes for *one* condition and several such conditional change lists. The *if then else* form is semantically different in that a new value is assigned in every case (maybe the old value) whereas the general form allows for no assignments at all (if there is no condition valid). For the latter case the mutual exclusion of arbitrary conditions cannot be proven syntactically.

2.4.2.5 Compound State Changes

The causal relationship to other events is described using so called *calling rules*. A calling rule consists of two parts: a *condition* determining if the calling should be performed in the current state, and a list of *event terms* denoting the events participating in the calling for the given conditions. Since different conditions can be specified for different event term lists, it is possible to specify *several* conditional calling rules.

Syntax

<*c_calling*>	::=	[{<*formula*>}] <*evt_term_list*>
<*evt_term*>	::=	[<*selector*>]<*evt_id*> [(<*proc_param_list*>)]

The meaning of the sequence of calling rules can be depicted as follows. An occurrence of the event at hand causes the occurrence of the events denoted after the keyword *calling*. However, this is only the case if the corresponding conditions are fulfilled. If one of the caused events is not allowed (for example it violates its enabling condition or after its occurrence a constraint is not fulfilled any more) *all* participating events cannot occur. Conceptually, all events caused this way occur at the same time [HS93].

This means also, that there is *no* direct correspondence to *procedure calling* or *message transmission* in other object oriented languages. Calling thus describes no *control flow* in the original sense of this term. Calling describes only the *causal dependency* between different events.[10] The events listed are forced to occur if the originating event occurs, but not vice versa. All events called together, precisely the transitive closure of the first event triggered, describe the state change of the system.

As an example for a *calling* rule we provide the following specification fragment, a slightly modified `return` event for `LibBook`s as introduced above:

```
...
  attributes
    BadGuys:list(|Person|) initialized emptylist  .
    Due:date .
  events
    insertBadGuy(Person:|Person|)
      changing
        BadGuys := insertfirst(From,BadGuys) .
    return(From:|Person|,At:date)
      changing
        OnLoan := false
      calling
        { At > Due } insertBadGuy(From) .
    ...
```

Here the changing of the attribute `BadGuys` is performed using an auxiliary event `insertBadGuy` (just for the sake of the example). In a more realistic example such events will be introduced only if there are more effects to be described.

The concept of calling as it is used here is a way to structure a state change using different events. Later on we will see that event calling can be generalized for *synchronous communication* between parts of a *composite object* and between separately defined objects of an object society (***interaction*** and ***relationship***).

2.4.2.6 Parameter Binding

The occurrence of an event may be triggered by the environment of an object for example by means of event calling or interaction (see below). Responsibility to provide parameter values is not limited to the caller. Some parameters may be left

[10]Again, causal dependency should not be understood as *causality*. The term *causality* is sometimes used to describe *reactive* behaviour of systems, i.e. an event causes an object to *react* later on in its life cycle.

unknown on behalf of the caller, whereas the callee is responsible to provide values if the event occurs.

In traditional terms a *data flow against* the causal relationship between the caller and the callee is specified. Data flow on the other hand is often used to describe data exchange between *procedure calls* with input and return values. In TROLL, the relationship between the caller and the callee is more subtle. Conceptually both events occur *at the same time* (see above). Thus there is no data flow in the traditional sense but more or less a *unification* process takes place where exactly *one* value is substituted. Syntactically, the binding rules are formulated similar to attribute updates but with an equality sign instead of the assignment “operator”.

Syntax

```
<c_binding>        ::=  [{<formula>}] <binding_list>
<binding>          ::=  <parameter_id> = <data_term>
```

In the former TROLL version this process of returning a value to he caller was modelled with *in* and *out* parameters together with *enabling conditions* for events selecting suitable values for these parameters. This property induces the following problem: The enabling condition must select *exactly one* data value for a given parameter. Otherwise the effects of such a specification interferes with the intuitive semantics of the calling relationship namely the callee event will be disallowed (no value selected) or several events take place (something like a multicast). To avoid such problems, parameter binding is specified explicitly.

As an example we may *return* a value to the caller of a `borrow` event in a `LibBook` object modelled as a ***binding*** rule in the following manner:

```
attributes
  MaxBorrowDays:nat initialized 21 .
events
  borrow(From:|Person|,At:date,!ReturnAt:date)
    binding
      ReturnAt = AddDaysToDate(At,MaxBorrowDays) ;
    ...
```

Upon calling this event, the parameter `ReturnAt` is equal to the value calculated by adding the current date `At` and the maximal time allowed for this library book.

This language feature reflects the idea of a specification of an atomic state transitions that is supposed to return a value that is calculated with the *knowledge of another object*. How such a specification is implemented in later design steps is irrelevant at this stage. It should be noted that enabling conditions or calling conditions of other events calling for events with binding responsibilitiy must not refer to the ‘returned’ parameters. Although this feature would be possible from a semantics point of view, it is counterintuitive to refer to a value that is 'returned' in some sense by a calling.

2.4.2.7 Hidden Events

As for attributes, also events may be declared as being hidden for other objects. Remember the example of the `insertBadGuy` event of library books:

```
...
  events
    insertBadGuy(Person:|Person|)
      hidden
      changing
        BadGuys := insertfirst(Person,BadGuys) .
    return(From:|Person|,At:date)
      changing
        OnLoan := false;
      calling
        { At > Due } insertBadGuy(From) .
    ...
```

Clearly this event is only local to the library book specification. It must not be called from outside the object because all necessary information for its occurrence is available inside the object. Otherwise other objects would have the possibility to call this event, thus inserting persons into the bad borrowers list *without* returning a book late. The hidden feature is thus not used to hide implementation details but as documentation that a special event makes sense only locally.

Another example for the use of hidden events is relevant for later design steps. In TROLL we may start specifying a `NatStack`-object template providing only the signature and some initial constraints and enabling conditions first:

```
template NatStack
  attributes
     Top:nat initialized undefined .
     Empty:bool initialized true .
  events
     Create birth .
     Push(Elem:nat) .
     Pop enabled not Empty .
end template NatStack
```

Later on in the design process we decide to *implement* the stack on top of an internal "array and pointer structure" that should be hidden at the external interface of the `NatStack` object:

```
template NatStack
  attributes
     Top:nat
       derived
          if not Empty then Array(Pointer-1) fi .
     Empty:bool
       derived (Pointer = 0) .
     Array(No:nat):nat
       restricted No >= 0
       hidden
       initialized Array(No) = undefined .
     Pointer:nat
       hidden
       initialized 0 .
  events
     Create birth .
     Push(Elem:nat)
       changing
          Array(Pointer) := Elem;
          Pointer := Pointer + 1 .
     Pop
       enabled not Empty ;
       changing
          Pointer := Pointer - 1 .
end template Stack
```

Although this is a rather simple example of hiding information inside an object it shows the basic concepts. It is also a very simple form of *reification* that deals only with snapshots of events [HS93] as *implementation* of event behaviour not with event sequences as needed in more general settings [ES90, EDS93, FM93]. Later on we will see that the concept of hiding information about objects is orthogonally extended to part objects. Then we may reformulate the stack example in terms of "pointer and array *objects*" not visible outside the stack (see Section 3.2.8, Page 70).

2.4.2.8 Initialization with Birth Events

As discussed in the attribute specification section we may override attribute initializations if the initialization rule for a given attribute is classified as ***default***. The ***default*** classification must therefore be used if we want to introduce specific birth events parameterized so that they may deliver initial values.

For example a specification for a bank application may contain an account template specification that has birth events providing a different set of parameters (the account number, the id of its holder, an initial balance etc):

```
template account
  attributes
    No:nat .
    Holder:|BankCustomer| .
    Balance:money initialized 0.0 default .
  events
    open(No:nat, Holder:|BankCustomer|) birth .
    openWithMoney(No:nat, Holder:|BankCustomer|,Balance:money) birth .
end template account
```

Parameters of these events are named after the name of attributes. This is an abbreviation of changing rules for these attributes, that is, introduced for convenience. Without this abbreviation we would have to specify:

```
template account
  attributes
    No:nat .
    Holder:|BankCustomer| .
    Balance:money initialized 0.0 default .
  events
    open(InitNo:nat, InitHolder:|BankCustomer|)
      birth
      changing
        No := InitNo ;
        Holder := InitHolder .
    openWithMoney(InitNo:nat,
                  InitHolder:|BankCustomer|,
                  InitBalance:money)
      birth
      changing
        No := InitNo ;
        Holder := InitHolder ;
        Balance := InitBalance .
  ...
end template account
```

thus inventing new names for the parameters and specifying the state update rules for the initialization parameters. For attributes that are initialized with a special *initialization rule* (like `Balance`) the classification as default makes possible the overriding of the initial value.

We introduce this abbreviation for birth events only. For other events the changing rules have to be specified explicitly even for such simple events that only provide a new value for an attribute. For birth events as a special class of events such initializations are frequently found in a specification. They are for example used to initialize *key attributes* (see below).

2.5 Constraints

In a constraints specification we may impose restrictions on the observable states (attributes) and on the evolvement of attribute values over time. Constraints that restrict the possible observable states are called *static* constraints or *invariants*. Constraints that restrict attribute *evolution* are called *dynamic constraints*.

Static constraints are formulae of first order logic whereas dynamic constraints are formulae of Future Tense Temporal Logic as described in Section 2.1.2 [Ser80, Lip89, SL89, Saa91]. Constraints implicitly also restrict the admissible behavior of objects in that certain state transitions (i.e. event occurrences) are not permitted.

Syntactically, constraints are expressed as a sequence of FDTL formulae optionally preceded with the keyword ***initially***. Initial constraints have to be fulfilled relatively to the *initial state* whereas general constraints are valid for the whole life of an object, that is, they must be valid in *each state*.

Syntax

constraints [<*var_spec*>] <*constr_seq*>

<*constr*> ::= [**initially**] <*formula*> .

As an example for an initial constraint we may specify, that a new `Account` object sometime must contain at least 100 $ before it can be overdrawn (first rule), that an account in the red condition must be in non red condition sometime in the future (second rule), and that an account can only be overdrawn up to 1000000.0 $:

```
constraints
   initially (eventually Balance > 100 before Red) ;
   Red => eventually(not Red) ;
   not (Red and Balance > 1000000.0) ;
   ...
```

`Balance` can only be positive in this example. Debts are modeled with the attribute `Red`.

In contrast to *restrictions* formulated for attributes with ***restricted*** rules, constraints allow for arbitrary dependencies *between* attributes respectively their possible values. Moreover the attributes mentioned in constraints can be incorporated into the object by means of components or be inherited from a parent object. For a discussion on components and inheritance see Sections 3.2 and 4.2.1.

Restrictions can be formulated as constraints as well. We introduced restrictions because from a modelling and documentation viewpoint it is more natural to group such local properties of attributes in the attribute specification and define general constraints as a different concept.

2.6 Life Cycle 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. An event description merely describes the conditions and effects local to event occurrences. To describe entire life cycles we have to talk about *sequences of events* and *dependencies* among events that occur sequentially.

Modelling real world entities, we have to deal with a variety of different behaviours. Some objects may be highly deterministic and their behaviour should be described by an explicit *process*. For example a clock or a part of simple robots often follow a static sequence of events. This kind of objects can be described by a *pattern of behaviour*.

Other objects sometimes can be described by their *restrictions* on event occurrences *(enabling conditions in event descriptions)* in different states of their life namely if there possible life cycles underlie only minor restrictions. Since TROLL does not contain an extra construct to describe *obligations* or *liveness properties* respectively *commitments* as TROLL1, the process language introduced here provides mechanisms to describe such properties without being too operational.

2.6.1 Specification of Processes

The process part of an object description is used to define the long term behaviour of objects in terms of sequences of events. Such sequences can be specified with various options that define the dependencies on events not mentioned in a particular process description. Such dependency can for example describe which events may interleave the specified process, when a specific sequence is in effect (start mode), or to which extend a sequence must be executed.

An object life cycle description is separated into two parts the *declaration part* and the *usage part*. With the former, processes are declared that can be composed of other processes also declared there or being (syntactically) inherited from parent object specifications. In the usage part, processes that describe the behaviour are listed up and are further refined with respect to the above mentioned dependencies on other events.

Syntactically processes are defined with the *process declaration* feature of a template description:

Syntax

(For: **process declaration** [<*var_spec*>] <*process_decl_items*>)

<*process_decl*>	::=	<*process_id*>[**(**<*df_param_list*>**)**] **=** <*process*> **.**
<*df_param*>	::=	[**!**][<*param_decl*>]
<*param_decl*>	::=	[<*parameter_id*>**:**]<*domain*>

```
<process>              ::=  <process> -> <process_unit> | <process_unit>
<process_unit>         ::=  <process_term> | <evt_term>
                            | <choice> | <parallel> | <foreach> | nil
<choice>               ::=  ( <choice_alternative> )
<choice_alternative>   ::=  <guarded_process> | <choice_alternative>'|'<guarded_process>
<guarded_process>      ::=  [{ <formula> }] <process>
<foreach>              ::=  foreach <var_id>:<data_term> do <process> od
<evt_term>             ::=  [<selector>]<evt_id> [(<proc_param_list>)]
<process_term>         ::=  [<selector>]<process_id> [(<proc_param_list>)]
<proc_param>           ::=  ['?']<var_id> | <data_term> | ['?']<param_id>
```

The composition of particular process patterns uses the process combination operators *sequencing*, *choice*, *parallel*, and *for each* that were described in the sub-language section on processes. Now we can give names to processes and have the possibility to also describe *recursive* processes.

The *top level processes* for an object are defined with the process feature for templates. Syntactically we write down the process terms used to define the behaviour of objects after the keyword ***processes*** as a template description feature.

Syntax

```
For:  processes [<var_spec>] [<process_use_items>]
<process_use>          ::=  <process_term> <process_desc_items> .
<process_desc>         ::=  initiative | weak | start <formula>
                            interleaving <interleave_mode>
<interleave_mode>      ::=  none | free | excluding <event_term_list>
```

Object life cycles are correct if they respect *all* listed processes, that is, the possible life cycles are defined as the intersection of the possible life cycle sets defined by each particular process listed.

2.6.2 Process Features

Similar to attribute, event, and component specifications process declarations are specified with a set of features relating the process at hand to other events and further refining its properties. The possible options include:

- a *mode for encapsulation against other events* that in various degrees restricts other events to occur during the "execution" of the process specified,
- a *start mode* that declaratively describes when a specific process sequence becomes valid in an objects life,
- a *weak* or *strict* mode that describes to which extend a process has to be executed or can be interrupted by a death event (thus defining some kind of normative behaviour), and
- an *activity* mode describing *initiative* behaviour of an object.

In the sequel, we will describe these options in more detail.

2.6.2.1 Degrees of Encapsulation

With the interleaving mode specification we describe the various possibilities events may interleave the specified sequence. Interleaving can be classified as *none*, *free*, and *excluding*.

The 'normal' case: A natural way to regard processes is to assume all events mentioned in a process declaration have to obey the specified sequencing schema. This means that there may be events interleaving the process but they must not be mentioned in its declaration. The events explicitly mentioned in a *description* must exactly respect the sequencing conditions. For example we may specify that an object locally has to behave in a specific manner but there may be events of part objects that are independent and may interleave.

None: Sometimes an interleaving of other events cannot be allowed because of interferences with the task of the process specified. A process that is specified with the interleaving mode *none* behaves in some sense as a *transaction*. Note that this classification has the same effect as *locking* the object for the time of process execution. An access in terms of other events cannot take place. The object can be *observed* however, it is not 'read locked'. This feature has to be used with care: it describes an object that can only execute one task for a specific time span (until the process in effect ends).

Free: The most liberal behaviour is defined as a *free* process. In this case we are not interested in *any* restrictions concerning interleaving. For example we can specify that a product object must suffer from `increasePrice` events as long as there is sometimes a `decreasePrice` event in its life cycle. Here we allow an event like `increasePrice` to occur several times, that is to "interleave" the sequence (pseudo notation):

```
ProductLife = increasePrice -> decreasePrice -> ...
```

a normal object behaviour requires a `decreasePrice` event after an `increasePrice` with no multiple `increasePrice` events interleaving. Other events are free to do so however. Several `increasePrice` events do *not* force additional `decreasePrice` events in the ongoing process. The semantics of such a process is intuitively defined as follows: A life cycle respects the specification if the sequence specified is at least one subsequence of the life cycle.

Excluding: For this mode (extending the normal mode) we can identify some events besides the ones mentioned in the description that are *also* not allowed to interleave. These can be listed with the ***excluding*** option.

The default for a process description is to assume that events mentioned in a process specification have to follow the sequence specified (the 'normal' case). The ***free***, ***none***, and ***excluding*** features thus denote more liberal respectively more restrictive behaviour.

2.6.2.2 Start Mode

Sequencing specifications may depend on the *situation* an object has reached so far. A situation in this sense is a particular point in time after a given sequence of events has occurred (a state). A particular situation is the state after the birth of an object. Other situations are classified using predicates over state variables. To refer not only to observable properties of objects (attributes) but also to events that occurred so far, TROLL supports *past formulae* to describe situations to be the "start" of a particular process. After the keyword ***start*** we write down a PDTL formula that denotes a state of an object where the process definition becomes valid. Frequently we will provide formulae with an *after* predicate at this position. The initial situation after the birth event is the default start of a specified process.

2.6.2.3 Weak and Strict Mode

To describe some form of *normative behaviour* of objects, i.e. a sequencing schema that is usual but may be violated by the object at hand is necessary to describe for example representations of *human objects*. A user of an automated teller machine (ATM) can be depicted writing down the normal way of interacting with such machine: first he/she puts his/her card in, enters his/her PIN code selects the amount of money, etc. and finally draws out the money. If for some circumstances he/she forgets to get the money this behaviour is unusual but not *forbidden*, the process can be classified as being ***weak***.[11] The weak feature thus models the distinction between an obligation that *must* occur in an objects life and a *commitment* that *should* occur in an objects life (compare commitments and obligations in TROLL1 [JSHS91]).

[11] Although we are able to specify this behaviour without a *weak* mode using *guarded processes*, the former way of specifying is more abstract and avoids low level specifications. It must be noted – as we have sketched above – that the introduced options and the process operators are not completely orthogonal to another. We may express some of them in terms of others.

If no weak feature is given, the process is assumed to define a *strict* behaviour that has to be executed eventually.

2.6.2.4 Initiative Mode

A process can be classified as being ***initiative*** to describe an internal *force* of an object to execute a process. An ***initiative*** process describes a behaviour of an object that need not be triggered from other objects. Such behaviour can be compared to an imperative program given CPU resources. In more concrete terms we describe this behaviour as an object entering a special role (see Section 4.2) where the events mentioned in the initiative process become at least ***active*** events in the sense described in the section on events. The reason to introduce a new keyword is the difference between active events that model an abstraction from real world dependencies and initiative events that *occur* if they are enabled.

The default initiative mode is *no* initiative. Declaring a process with ***initiative*** means *all* its events become active as long as the specified process is executed.[12]

2.6.2.5 Recursion

A usual operator for process languages is the *recursion operator*. Up to now we only talked about the process items *event terms*, *choice*, *for each*, *parallel*, and *process terms*. Using the identifier of a process declaration, i.e. a process term in the declaration itself defines a recursive process. Thus we may specify the famous swiss[13] `Clock` process as follows:

```
...
process declaration
  ClockProcess = Tic -> Tac -> ClockProcess .
processes
  ClockProcess  initiative .
```

Such a clock has initiative, i.e. its events occur by own initiative without being called from elsewhere. We abstract from the fact that later in the design process such a clock is integrated into the concrete implementation by means of an interface to low level system resources managing time. Interleaving of other events is allowed so that we may use the clock as a part in other objects, it cannot die and must tic forever. Interleaving must not be ***free*** because ***free*** would make possible sequences as for example:

```
Tic -> Tac -> Tic -> Tic -> Tic -> Tac -> Tic -> ......
```

[12]Again, the notion of *process execution* is operational. Declaratively we can talk about a period of abstract time during a life cycle where the process specified is valid.

[13]The clock *must* be from Switzerland because it cannot cease to tick [CSS89].

with several Tic events in a row which is not desired for a clock object.

Another example showing the possible use of process definitions and process usage is the description of the factorial function in terms of an object. The recursive process FAC is formulated accumulating the temporary results using attributes of the object. An object description incorporating most of the beforehand mentioned concepts and features is thus the following:

```
object Factorial
  attributes
    ResOut:nat
      hidden derived if ResultOK then Result else undefined fi .
    Result:nat
      hidden initialized 0.
    ResultOK:bool
      hidden initialized false .
  events
    create birth .
    calcFac(n:nat)
      enabled not ResultOK .
    readFac(!n:nat)
      enabled ResultOK
      binding n = ResOut
      changing ResultOK := false .
    init hidden
      changing Result := 1 .
    ready hidden
      changing ResultOK := true .
    step(n:nat) hidden
      changing Result := Result * n .
  process declaration
    STEP(n:nat) = ( {n>1} step(n) -> STEP(n-1) | {n<=1} nil ).
    CALC(n:nat) = init -> STEP(n) -> ready .
  processes
    variables n:nat;
    CALC(n)
      interleaving none
      start after(calcFac(n))
      initiative .
end object Factorial
```

This example is meant to illustrate the use of a recursive process definition in a well known example. In the *process declaration* section we define two processes STEP and CALC where CALC is the top level process that uses the STEP (calculation) process. CALC inits the object and then ‘executes’ the STEP process with a natural number supplied as parameter. STEP then executes the event step and recursively itself with decreasing parameter until the parameter becomes lower or equal than 1.

Until now we used only a very simple process definition language with well known operators like sequencing, choice, and recursion. In the ***processes*** section of the template specification we are *embedding* this process into the object described here by specifying

- when it is valid (start after event `calcFac`),
- that it may not be *interrupted* by other events (no interleaving),
- that it has to be executed until it has fulfilled its specification,
- and that it is an initiative process that is performed without intervention from outside.

We are aware that this is a rather operational example but it shows features of TROLL that cannot be described easily in 'half page' (real world) examples. It should be mentioned here, that the specification of this process as ***interleaving none*** prevents further `calcFac` events during the execution of `CALC`. The enabling condition for `calcFac` is thus redundant. Note that – just as an example – only the event interface is visible for this object.

2.6.2.6 Parallel Events

In process languages we have usually an operator to combine processes to execute concurrently. In TROLL parallel processes usually are specified *on the level of objects*, i.e. objects are the units of concurrency. Nevertheless objects may be composed of part objects (see Section 3.2) and we may sometimes want to describe events of such part objects that are synchronized.

For the process description language of TROLL we introduce a restricted *parallel operator* "`||`" for events with the following semantics. The specification:

```
...
enterPINCode(MyPIN) ->
( checkPinCode(?Valid) || recordPinCode ) ->
...
```

taken from the ATM-user interaction from Section 2.1.3.2, Page 17 states that the events `checkPinCode` and `recordPinCode` are *synchronized*, they occur during one state change. This means they can only occur in one snapshot of the objects life if the process specified is executed. We see that this synchronization can also be modelled with mutually *calling* of both events themselves. With the specification above however we specify synchronization *only for this process occurrence* whereas calling is specified globally for an object. Firstly we have to encode *process states* into the object description to model the above mentioned problem completely with calling and secondly we scatter knowledge about the properties of these events over the specification. Both points are not desired for a readable TROLL text.

2.6.2.7 Scope of Parameters

As noted above we may introduce “?’ prefixed parameters in process descriptions. Operationally speaking, the value of such parameters is not known when the process is executed. Usually a value is provided by means of *communication*, i.e. the event with such a parameter is specified with an output parameter or it is called from another object with a suitable value for the parameter. As an example see the following specification:

```
process declaration
  variables p:aType;
  ProcX = E1(...) -> .... -> En(...,?p,...) -> ... -> Ek ;
```

For such a process we assume the following restrictions to be valid:

- There must not be a reference to the variable `p` *before* the event `En`. We may assume that the process is embedded into an environment with a set of variables bound to values, the value of `p` is *hidden* for all events before `En`. This condition guarantees that we may not refer to a value that is intuitively not known. Operationally the variable is bound to a value with `En` occurring.
- All event terms `Ei` with `i>n` may refer to the value bound to `p` by event `En`. The same is true for *conditions* used “after” `En`.
- After the process has been executed, e.g. in the state where *after*(`Ek`) holds, the value of `p` is lost. In other words we assume a *strict block concept* (see next item).
- The value referred to by `p` can only be transferred to other subprocesses by means of process parameters. Note that a process unit can also be a subprocess (by means of a process identifier and suitable parameters). Thus together with the last item these parameters are strictly *local* variables for the process at hand.
- The last item mentioned can also be applied to *recursive* processes. In programming language terms we may say that parameter transfer is only *value based*.
- If a process unit is a process term with `?`-prefixed parameters we have another form of value transfer, namely from a subprocess to the surrounding process. Like for events the same rules as stated above apply. Additionally, the subprocess must be specified with a binding responsibility for the parameter transfer, i.e. it must have a ‘*!*’-prefixed parameter declaration (similar as for events).

The last point mentioned is exemplified with the following artificial process pattern:

```
process declaration
  variables p,r:aType;
  ProcX = E1(...) -> .... -> Pn(?p) -> ... -> Ek(...) ;
  Pn(!r) = Ep1(...) -> ... -> Epi(...,?r,...)  -> ... -> Epj(...) ;
```

In this artificial example the process `ProcX` contains a process term `Pn` in its definition. This process term contains a `?`-prefixed variable `p` that delivers a value to be used similar as in the former example. For this purpose the process `Pn` has the *responsibility* to deliver this value. It therefore has to be specified in the same manner as an event with a *return* value. Here however the value is determined during the execution of the process. Again, only a *value transfer* is modelled – the variable is lost at the end of the process however.

2.7 Interaction

Besides *calling* specification in an event description we may also specify *interaction* in a similar way as in TROLL1. Since we have to introduce composite objects before we can introduce such interaction rules, we refer the reader to Section 3.3 for interaction specification.

2.8 Parameterized Templates

As we have seen in the stack example we sometimes have to introduce templates that can be described as primarily representing *data structures*. The behaviour of these data structures in terms of possible operations etc. is fixed whereas the contents in terms of data values may vary in data types. For example we can use a generic description of a queue to describe queues that can store values of different data types.

Symbols of data type signatures (operation symbols and to some extend *formulae* that are used throughout the specification of a template) are the building blocks of *data terms*. Thus a template specification has to be checked for type safetyness. If we parameterize templates with data types, the data terms and formulae used in such a template determine the possible parameter data types. To keep things simple we consider parameterized templates as a derived feature of TROLL that can be described by transformation to a simpler language version without parameterized templates.

Syntactically parameterized templates have the following form (see also Section 2.2, Page 20):

Syntax

```
<template_spec>      ::=   template <template_id> [(<dt_param_id_list>)]
                              <template_desc_items>
                           end template <template_id>
```

For example if we specify a template for stacks storing arbitrary data values:

```
template Stack(EntryType)
  attributes
     Top:EntryType
       initialized undefined .
     Empty:bool
       initialized true .
     ...
  events
     Push(Elem:EntryType) .
     Pop enabled not Empty .
     ...
end template Stack
```

and use this template in another object specification (The concept of object classes is introduced in the next section):

```
object class InputStack
  template Stack(nat)
end object class InputStack
```

then we can *derive object specifications* for all uses of `Stack` object specifications where the formal data type parameter is substituted by the real parameter. In this example we have to construct the following (derived) specification:

```
template Stack_nat
  attributes
     Top:nat initialized undefined .
     Empty:bool initialized true .
     ...
  events
     Push(Elem:nat) .
     Pop enabled not Empty .
     ...
end template Stack_nat
```

that can *now* be checked formally on type correctness. Of course we have to rewrite the specification of `InputStack` according to this derived specification, namely by substituting the template definition by:

```
object class InputStack
  template Stack_nat
end object class InputStack
```

This way we have at our disposal a very simple language feature for parameterization that can be described on a lower language level. The parameterization feature can be handled easily because it is described as a *syntactic reuse* of a template specification without having in mind the *objects* that are generated from such specification later on using classes. For future developments of the language TROLL further parameterization is planned but is not incorporated into this version. For a discussion on parameterization see also [Saa93].

Chapter 3

Class and Components Specification

3.1 Specification of Object Classes

To come from prototype descriptions of objects (templates) to the objects themselves, we must introduce a *naming mechanism*. A template itself only defines the *shape* of an object, its structure and possible behaviour.

For the specification of objects and object classes we use a template specification, introduce a set of possible identifiers (a carrier set of a special data type, an object identifier data type) and end up with a set of (possible) *object instances* henceforth called *objects* defined by a *template*. Furthermore we introduce the notion of a *class* as a *container for similar objects* that behave as defined in their corresponding template.

In TROLL there is a distinction between the *description of a class* (its *class type*) and the *class itself* (the *extension*, *class container object* or shortly *class*). A class is thus regarded as an actual set of objects with a structure defined by its class type. A *class container object*, being an object in its own right *contains* the objects of the class as *components* (see Section 3.2 for components and Chapter 4 for a possible specification of classes as containers).

3.1.1 Object Identities and Object Identification

Each object has associated an unchangeable unique identifier. So it is possible to distinguish between different objects. In TROLL this identifier is conceptually a value of an *object identifier data type*. The identifier data type is denoted by the class name surrounded by vertical bars. We only assume that object identifier data types have an unbounded set of values, i.e. there must be *enough* identifiers[1].

Apart from object *identities* we talk about *object identification* as a means to

[1] For a discussion on concepts like object identifiers, keys, and surrogates see also [Wd91].

describe *externally meaningful names*. In other words object identities denote *unprintable* values associated to objects whereas object identifiers denote names that have *semantics in the context of the world modelled.* As an example see the following specification:

```
object class Person
  identification ByName:(Name, BirthDate)
  attributes
     Name:string .
     BirthDate:date .
     ...
end object class Person
```

Person objects have identities, in this case values of the data type `|Person|`. We say that for a particular person object the value `x` $\in$ `|Person|` is the *identity* of this object. The value `x` is *abstract* in the sense that it has *no meaning in the world modelled*. For specification purposes, each object of a class has an *implicit attribute* `OID`. In the case of the example above:

```
  attributes
     OID:|Person| .
```

This attribute can be used as an ordinary attribute and can for example be transferred to other objects during communication etc (see again [Wd91] where oid's are considered visible). In other languages this reference to the object itself often is called `self`. Smalltalk [GR83] uses `self` to send messages locally. Note that `self` in Smalltalk is a *reference* which is different to an *identity* in TROLL.

On the other hand the person object associated with identity `x` has attributes like `Name` and `BirthDate` that together constitute an *identifier*, the *identification* in the UoD at hand. Object identifiers are the interface to the real world whereas object identities are used throughout the specification to refer to objects – thus the latter are considered artificial and internal to the specification document (unprintable).

Identities can be stored as attribute values and can be used to refer to objects. However, identities are *not references* to objects as in other object oriented (programming) languages. In TROLL identities must be used in conjunction with the class name to refer to objects. As we will see later on, an identity value may be used to refer to different *aspects* [ES91] of an object if it is used together with *role classes*. With the specification of an object class (for example `Person`) semantically a function

$$\texttt{Person :}\quad \texttt{|Person|} \longrightarrow \texttt{PERSON-OBJECTS}$$

is implicitly defined taking an identifier and yielding the corresponding object, that is `PERSON-OBJECTS` denotes the set of all possible person objects. Consequently

`Person(x)` denotes the object associated to `x` itself. We use this pseudo formalization here to describe the mapping from identities to objects in an intuitive way.

For *identification* of objects there exist several ways. We have objects that have a clearly defined *externally meaningful name* (like persons), objects that are the only object of their corresponding class (single objects like a system clock), *abstract* entities that have no identification but identity (a formula in the modelling of a software repository) or *derived* objects that *inherit* an identification from some base class specification (an employee that *is-a* person).

In more detail we identify the following possibilities for object identification:

1. *An identification can be defined as a list of observable properties.*

   Such a list, denoted by attribute symbols, is specified as a *conceptually meaningful* name for objects (see the person class above). In the case of the example, tuples of (`string`$\times$`date`) serve as identification if we declare the *key* `ByName` as shown above. Key tuples must be introduced with a name.

   In contrast to the example there may possibly exist several keys. For persons there may be another attribute `SocialSecNo:nat` that serves the same purpose as the tuple `(Name:string,BirthDate:date)`.

2. *Objects without identification.*

   Sometimes it is not quite natural to find an observable property that can be used to *externally* identify objects. As an example suppose we describe a class `Formula` with objects used in class `Template` (for example in a Troll-repository specification). What is an external name for a formula? Another example is the specification of an `Engine` object. What is the identification of a `Bolt` or a `Screw` used as parts of this `Engine`?[2]

3. *The keyword* class *is omitted from the specification.*

   This case may be used to describe single objects, i.e. not real classes with several objects but a single object of the corresponding template. Semantically, such *objects* can be treated like objects of a class if we assume that we specified an object class that has an identity space with only one value. This viewpoint is necessary for a uniform treatment of single objects and objects of real classes. The syntactic notation stresses that there is only a single object specified by omitting *class*.

4. *There is no identification given but a class is defined as being a* role *class.*

   A role class inherits the properties of its base class(es), i.e. also the identification of the base class. In case of multiple inheritance this implies that there must be one class at the root of the particular part of the inheritance hierarchy.

[2]Several other properties of real world objects can be considered in this example too: the construction of composite objects, only locally visible classes etc. Often the question of identification spaces is closely related to the question of when to use classes with a local identification space.

Role specification is introduced in Section 4.2.

5. *There is no identification given but a class is defined as being a* view *class.*

   A view class is a means to describe restricted access to the set of objects of the class (selection, see Section 5.1.2) or to the set of properties of objects (projection, see Section 5.1.1).

   As for role classes the identification is derived from the base class(es). View specification is introduced in Section 5.1.

Syntactically, a class is introduced with a name, a class description and a template description similar as already described for templates. We can either introduce the template specification directly or specify a template separately and provide its name for the class specification. In case of a parameterized template we also have to supply the parameter data type(s). Then we can only use the latter case.

Syntax

```
<class_spec>      ::=  object [class] <class_id>
                         <class_desc_items>
                         [<template_desc_items>]
                       end object [class] <class_id>

<class_desc>      ::=  identification <key_spec_list>
                       | role of <class_item_list> [derived <formula>]
                       | view of <class_item> [derived <formula>]
                       | template <template_id> [( <data_type_id_list> )]

<key_spec>        ::=  <key_id> : ( <att_id_list> )
```

If a template is introduced together with a class specification, we may refer to this template with the class name, i.e. if we want to reuse the *template specification* for a different class. As sketched above the class name is used throughout a specification to refer to objects of a class (in conjunction with identities and identifications).

In the next sections we will introduce examples for the above mentioned possibilities for object identification (list no. 1 to no. 3). After introducing object components in Section 3.2 we will be able to have a closer look on classes as first class entities containing objects.

As already mentioned, classes are specified with templates. When specifying templates, usually one will have classes or at least single objects in mind. Thus the distinction between templates and classes is not always obvious. A template is a *static description* of object structure and behaviour. Classes however are dynamic in the sense that there is a *time varying set of objects* in them (although their description is also static; for possible future evolution of this model see [SH94]).

We distinguish classes and templates to stress such a difference. A template in this sense is an entity to be stored in a *library* of reusable components and can be used when specifying classes. A *class specification* can also be stored in such a library but has a closer correspondence to the real world by virtue of its identification mechanism. A *class container* however is an (abstract) dynamic entity and contains a varying set of objects. As a first class object the class container has a specification that is derived from the user specification of a class. We will provide an example later on.

3.1.2 Referencing Objects using Key Attributes

Attributes used in key descriptions have to obey special constraints. These constraints must be valid in the *actual set of objects of a class type at a certain point in time*, i.e. the actual class population. Objects have associated a unique identity. This way it is possible to distinguish objects that have the same attribute values. Attribute tuples that are mentioned in identification clauses however must be *unique in the set of currently existing objects.* Recall the specification of the class `Person`:

```
object class Person
  identification
     ByName:(Name, BirthDate),
     BySSN:(SocialSecurityNo)
  attributes
     Name:string .
     BirthDate:date .
     SocialSecurityNo:nat .
     ...
end object class Person
```

Apart from referencing objects of class `Person` via their object identities, that is, writing

$$\texttt{Person(x)} \text{ for an } \texttt{x} \in |\texttt{Person}|$$

we may also supply a tuple consisting of a string and a date and get back the object with the appropriate values for its `Name` and `BirthDate` attributes. The above mentioned constraints for key attributes guarantee, that *at most one* object qualifies for such a *query*. Now we are able to motivate the existence of *named* keys, here `ByName`. `ByName` denotes the (partial) function

$$\texttt{ByName: string} \times \texttt{date} \longrightarrow |\texttt{Person}|$$

taking a tuple consisting of a string and a date and yielding an *identity* of class `Person` that is in turn used to refer to an object.[3] `ByName` is local to the class `Person`. The syntactic expression:

```
Person(ByName(Thorsten,17-04-64))   or   Person(BySSN(4170464893))
```

thus refers to the object with the appropriate attribute values. As an abbreviation for objects that only define *one* key group we can – as syntactic sugar – even omit the map-name for references as above. Assuming we have only specified the key `ByName` for `Person` we can write:

```
Person(Thorsten,17-04-64)
```

provided we have a tool that expands these expressions to the original one. Additionally we have to detect ambiguities that arises with such kind of abbreviations. For Troll as it is now it is safe to use only the longer forms that additionally help the reader of a specification. In fact the abbreviation is not defined in the concrete syntax (see appendix).

The conditions necessary to guarantee the key constraint (only one object for a given identification tuple), and the existence of this particular object in the current class population etc. are managed in explicit class container objects (see Section 4.1). The function `ByName` changes during the life of a class container object. Changing key attributes and creation and deletion of objects have to be reflected in changing the function mapping. We will elaborate this in Section 4.1.2 and 4.1.3.

3.1.3 Referencing Objects using Identities

As already mentioned it is sometimes difficult or impossible to supply externally meaningful names for objects. This is often the case for *abstract* entities like e.g. a `Formula` in a specification document or a `Bolt` in a car manufacturing environment. Both entities are identifiable only *in the context of another object*.

A bolt object may be identifiable via the special role it has in the engine object e.g. connecting the body of the engine and the starter. Outside this context it has an *identity*, we may distinguish it from other bolts of the same shape but no *identification* other than *"a bolt"*. From the viewpoint of the engine the bolt is a *component*.

A similar observation is made for abstract entities like formulae in e.g. a repository specification. Although we may find an identification for a formula for example build out of its recursive structure, this is not quite natural or obvious. Changing the formula will induce changing this identification which is not desirable in this

[3]Compare also the notion of `KeyMaps` in [Jun93]. Here we define alternative key maps and therefore need names for these functions.

case. It seems to be more natural to identify such formula by its role in the overall design – as for the bolt – now to be a component of another object. For example a formula can be identified as being an enabling condition for an event specified with class `Event`.

Object classes without an *identification* are thus used to specify *anonymous* objects. As for ordinary classes there may be *potentially* infinitely many objects of such classes. Objects may also be shared between different objects (see below) by means of components if knowledge of their *identity* is available, e.g. for the above example the only possible reference to a bolt is:

`Bolt(b)` where `b∈|Bolts|`

this bolt may be used in a car as a `BodyToStarterBolt` and be "transferred" to another car to be a `BodyToSomethingElseBolt` by a mechanic. Thus the only mechanism to refer to such objects as described in this section are identities. Since identities are (nearly) ordinary data values in TROLL, they can be transferred between objects, too. The only difference is that identities are not printable in the sense that they have meaning to people. They are as abstract as the objects they identify.

3.1.4 Identification of Single Objects

A special case of a class is a class with only one object as class population: *single objects*. Single objects are represented the same way as classes are. The only difference is the missing keyword *class*. Here the condition that identity sets have to be sufficiently large is dropped. The framework for identification and identities is the same as for classes however. So classes and single objects (as classes with exactly one object) can be handled with equal rights.

Suppose we want to specify a world containing an object `Clock`. We do not want to specify the variety of clocks available but only one object counting time units.

```
object Clock
  attributes
     Seconds:nat initialized 0.
     Minutes:nat initialized 0.
  events
     Create birth .
     Count ....
     ...
end object Clock ;
```

Referring to `Clock` attributes can be done the same way as for classes, namely using the function taking object identities and yielding the object. To refer to the `Seconds` attribute of the `Clock` as an abbreviation for `Clock(x).Seconds` where `x` is the only element of `|Clock|` we write `Clock().Seconds`. The identification thus is the class name itself, the identity is redundant and confusing at the *language level*.[4] It is needed only to formally integrate classes and single objects.

3.2 Specification 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.

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*[5] components that are handled similar to set valued components plus additional access to objects at the head or tail of the list as well as indexed access.

3.2.1 Description of Components

The component specification has a similar notation as the attribute and event specification. In fact a component specification enriches the signature of an object in that we may use the attribute and event symbols (and as we will see also the component symbols) of the component objects in the enclosing object.

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 describe a stronger relationship than object references. Nevertheless, through component symbols we may *refer* to the component objects itself.

But also the component objects being part of a composition can be influenced by the embedding object. For example we can specify conditions – i.e. constraints – that directly inhibit attribute changes in components. Furthermore not only a communication relationship *to* the components, but also a relationship *from* the components to the embedding object is implied by the use of components. The latter includes communication that is however specified in the *embedding* object. We may not specify such communication in the part objects since the part objects have no knowledge of their use in other objects on the specification level: there is no

[4]The brackets `()` are necessary to distinguish between referencing the *object* itself (with the brackets), and referring to the class object itself. The difference will be more clear when we discuss class containers and referencing components.

[5]Do not take the word 'valued' too seriously in this context. We are talking about *objects* here that are different from *values* in Troll.

reference from the components to the embedding object on this level if not explicitly specified.

The syntactic representation of component specification is similar to attributes: we may specify *properties* or special *features* of components in a similar way. The difference is that we have to handle objects and object populations and no data values:

Syntax

```
(For:  components [<var_spec>] <cmps_spec_items> )
<cmps_spec>      ::=  <cmp_id>[(<param_decl_list>)]:<class_item>
                        <cmp_desc_items> .
<cmp_desc>       ::=  set | list
                      | inherited from <class_id>
                      | hidden
                      | restricted <formula>
                      | initialized <formula> [default]
                      | derived <formula>
<param_decl>     ::=  [<parameter_id>:]<domain>
<class_item>     ::=  <class_id> [<ovar_id>]
```

3.2.2 Single vs. Set Valued Components

Component specifications describe *aggregations* of objects. Similar to *object valued references* in conceptual data models (see for example [EGH$^+$92]), they provide access to parts of the object society. As such, component specifications can be seen as *locally visible populations of classes*. On the global level, classes can be restricted to contain only one object thus modelling single objects. Likewise components may also be depicted as being *"single valued"* or *"set valued"*.

As an example for component specification we model a `Bank` class containing a single `Manager` component and a set of `Account` objects, that are specified in the following way:

```
object class Account
  identification AcctID:(No);
  template accountTemplate
end object class Account
```

We have not specified the template for accounts here to keep the example small. Accounts should have usual attributes like `Balance` etc.

```
object class Bank
  identification BankID:(Name,No)
  attributes
    Name:string .
    No:nat restricted No>10000000 and No<99999999 .
    ...
  components
    Manager:Person .
    Acct:Account set .
end object class Bank
```

In the specification of `Bank` we may refer to the components with the well known *"dot notation"*. For example the age of the manager of the bank can be referred to writing:

`Manager().Age`

provided during the life of the `Bank` object the component has been *set* to a specific `Person` object. As we will see in the next section, this can be done using *implicitly generated events* like

`Manager.Insert(P:|Person|)`

If it was not set in this way the *component object* can be considered as not being existent in the sense of an object that is not yet born. Now we can also motivate the difference in notation between:

`Manager().Age` and `Manager.Insert(...)`

The first form refers to an *object*. Consequently we can use only signature elements from `Person`. The second form refers to the *component collection* and we may use implicitly generated signature for a single valued component.

The second component of the beforehand introduced example (`Acct`) describes a *set* of objects as part objects (set valued component). To refer to objects of this component it is not sufficient to supply the component name. We also have to supply an *identifier* for `Account` objects resembling the notation for access to objects of global classes in general. Than we may write:

`Acct(acc).Balance` for `acc` $\in$ `|Account|`

Alternatively we can also use the identification for `Account` objects, for example if we supply a customer-bank interface to transfer money from one account to another. In this case we typically have to describe some kind of a translation between *real account* *numbers* and *abstract identities* of accounts.[6] As we have seen in the discussion of classes, this translation can be performed using the *key maps* implicitly defined with classes. To refer to the `Balance` attribute of account number 123456 we have to write:

```
Acct(AcctID(123456)).Balance
```

implicitly using the fact that the function `AcctID` is a translation of natural numbers to account identities of the class `Account`. The function `AcctID` is visible due to the class `Account` being visible in the `Bank` object by means of the component specification (`Acct`). In the section on interaction (Section 3.3) of components we will introduce some more examples for object referencing.

3.2.3 List valued Components

Additionally to *set valued* components we may also specify *"list valued"* components. Compared to set valued components, list valued components are ordered with respect to insert and delete operations and may contain duplicates. Additionally there exist more implicitly generated attributes and events and also derived components for list components.

In the following example we model a queue of persons waiting for service in a bank.

```
object class Bank
  ...
  components
    ServiceQueue:Person list .
    ....
  events
    Arrival(P:|Person|)
      calling ServiceQueue.InsertLast(P) .
    Service(P:|Person|)
      enabled ServiceQueue.IDFirst = P
      calling ServiceQueue.RemoveFirst .
    ...
end object class Bank
```

The events `Arrival` and `Service` – parameterized with the identifier of a person – trigger the occurrence of insert and remove operations in the service queue of the bank. Another *implicitly generated component* `ServiceQueue.First` denotes the currently serviced person (not shown in the example above). A complete list of generated attributes, events, and components is given in the following section.

[6] In later specification steps we should use a *view class* `AccountToCustomers` with events like `LocalTransfer(Source:nat,Dest:nat,Amount:money)` where natural numbers as real world identification for accounts are used.

3.2.4 Implicit Signature for Components

To summarize and extend the examples of the last section, Figure 3.1 lists the

	Single Components	Sets Components	Lists Components
attributes	Empty:bool Card:nat IDSet:*set*(`|CN|`) In(`|CN|`):bool Exists(`|CN|
	```
	):bool | Empty:bool
	Card:nat
	IDSet:*set*(
	```
	|CN|`) In(`|CN|`):bool Exists(`|CN|
	```
	):bool | Empty:bool
	Card:nat
	IDSet:*set*(
	```
	|CN|`) In(`|CN|`):bool Exists(`|CN|`):bool IDList:*list*(`|CN|`) IDFirst:`|CN|` IDLast:`|CN|
	```
	Length:nat
	Pos(
	```
	|CN|`):*set*(nat)
events	Insert(`|CN|`) Remove(`|CN|`)	Insert(`|CN|`) Remove(`|CN|`)	Insert(`|CN|`) Remove(`|CN|`) InsertFirst(`|CN|`) InsertLast(`|CN|
			```
			)
			RemoveFirst
			RemoveLast
			Change(nat,
			```
			|CN|`)
components			First:`CN` Last:`CN` Element(nat):`CN` Elements:`CN` *set*

Figure 3.1: Generated Symbols for Component Manipulation and Observation

signature generated for components. A component specification of the form:

`Comp:CN` *list*

thus makes available e.g. an attribute `Comp.Empty` denoting the status of the component (***true*** meaning that there is no object assigned), an event `Comp.RemoveFirst` for removing an object from the first list position, and a component `Comp.Last` denoting the object at the last list position etc. Semantically a component specification is regarded as a local collection of visible objects just as a class collection is globally visible if we have access to a class collection.

The generated signature is developed having uniformity in mind. This means, that the generated symbols are the same for single valued, set valued, and list valued components, taking into account that single objects are treated globally as classes with at most one object contained. The same uniformity for list valued components is introduced for convenience plus some list specific signature. So for example there is an attribute for single objects `Card:nat` that can have values `0` or `1` only.

3.2.4.1 Implicitly Generated Attributes

The following attributes are generated for a component specification:

`Empty:bool` Is ***true*** if there is no component defined, i.e. the component set or lists are empty.

`Card:nat` Denotes the count of elements in sets and lists, in the latter without duplicates. For single components `Card` is 0 if `Empty=`***true*** else 1.

`IDSet:`**set**`(|CN|)` Denotes the identifier sets for set and list valued components as well as for single valued components where this set can contain at most one element.

`In(|CN|):bool` Takes an identifier and yields ***true*** if the associated object is assigned, respectively is in the set or in the list.

`IDList:`**list**`(|CN|)`, `IDFirst:|CN|`, `IDLast:|CN|` Only for lists. Denotes the list of identifiers, the identifier of the first resp. last object in the list.

`Length:nat` Only for lists. Denotes the length of a list. In other words we can derive this attribute also as: `Comp.Length = length(Comp.IDList)` using the data type operation `length` that is defined for list.

`Pos(|CN|):`**nat**`(nat)` Only for lists. Denotes the position(s) of an element in a list. Is an ***emptyset*** of naturals if argument is not member of the list.

3.2.4.2 Implicitly Generated Events

The following events are generated for a component specification:

`Insert(|CN|), Remove(|CN|)` Inserts respectively removes elements from single, set, or list valued components. For lists the event `Remove` works as for sets, i.e. all component objects for a given identifier are removed from a list. `Insert` for lists work for the head and is similar to `InsertFirst` (see below).

`InsertFirst(|CN|), RemoveFirst` Insert and remove events that are used only for lists and work at the head of a list (the first position).

`InsertLast(|CN|), RemoveLast` Insert and remove events that are used only for lists and work at the tail of a list.

`Change(nat,|CN|)` Changes the element at the position specified with the first parameter.

If we refer to list positions for example with event `Change`, than an explicit enabling condition must be taken into account that allows such events only for index values between `1` and `CN.Length` inclusively.

3.2.4.3 Implicitly Generated Components

The following components are generated for a component specification:

`First`, `Last`, `Element(nat)`, `Elements` Denotes the first respectively the last object of a list. `Element(i)` denotes the object at the i-th list position, and `Elements` is the transformation to a set valued component.

There are no derived components for single and set valued components since access is performed similar to access to object classes, namely by providing the component name and an identification or an identity. As for classes the *key map* functions can be used for components, too.

3.2.5 Composition Constraints

Constraints that may restrict objects from being components in another object can be formulated in a *restricted* clause. As an example for a composition constraint the relationship between a bank and accounts respectively a clerk is specified in the following way:

```
object class Bank
  ...
  components
    Acct:Account A set
      restricted A.No>100000 and A.No<999999 .
    Clerk:Person P set
      restricted P.Age>=65 .
end object class Bank
```

The object variable introduced here is a short form to denote an arbitrary object of class `Account`, i.e. only objects of class `Account` respecting the constraint can be components of `Bank`. The *restricted* clause describes conditions that must be fulfilled by `Account` respectively `Person` objects incorporated into the composition `Bank`. Of course these condition can also be formulated as a general *constraint*. To stress the belonging of this restriction to the component, we make this rule available similar to attribute restrictions that state conditions for the *range of possible values*. Here it is the "range" of possible objects with respect to their attribute values and the restricting formula.

There is an important difference to data values in that the interpretation of data values is fixed whereas attribute values change. For objects incorporated into a composition a ***restricted*** clause of this form (and alternatively equivalent constraints) define *additional constraints* that must be fulfilled in the life of `Account` objects that currently are members of the composition. As already mentioned, this feature (among others) distinguishes components from object valued attributes. Users of the language must be aware of this "*export*" of conditions to components. It models the fact, that an object – becoming part of an enclosing entity – may not behave as free as before.

3.2.6 Derived Composition

As composition constraints are analog to attribute constraints we define derived components analog to attribute derivations. In contrast to attributes where we have to specify a *data term* that describes the derived *value* we provide a *formula* as a *selection condition* for a single object or a set of objects. Such condition is thus a simple form of a *query*.

As an example we can use the constraint of the previous section that restricted the possible set of accounts in a bank to describe a *derived* set of components:

```
Acct:Account A set
  derived A.No>100000 and A.No<999999 .
```

As for attributes such derivation rule is also a constraint in the sense that there cannot be accounts in the composition that do not satisfy the formula given. Here however, changes of attributes involved in the formula have to be reflected in automatically provided inserts and removes for the component (set). Thus a derivation is not a constraint *for the component object* in the sense described in the last section. The connection between part and whole is weaker than for composition constraints.

As we cannot specify changing rules for derived attributes we cannot specify insert and delete operations for derived components nor we can specify restrictions for such components. Insert and delete events are hidden from the specifier and may only be used as an operational description how derived components may be semantically described on top of real components.

A different but equivalent description for this semantics is to include all possible accounts by default and to *check* the derivation formula in case of an access to such a component which shows the close relationship between restrictions and derivations of components. The analog strategies for derived attributes are *calculation on access* and *storing on change* the latter as an additional changing rule if properties used in the derivation conditions change.

Some remarks are in order here. The classification *set vs. single* valued used in the example is optional, it cannot be proven syntactically anyway. Often the condition will be constructed over key attributes of the component objects. Since attributes including keys are not constant, the composition may change if these attributes change. Keys are however not subject to frequent change.

3.2.7 Composition Initialization

Again in analogy to attributes, Troll provides a construct that allows for initialization of components. The initial component is selected similar to a derived component using a formula (a simple query). The formula is evaluated in the state where the birth event of the enclosing object occurs. To give an operational characterization, the population of the class where the component objects are drawn from is 'searched' for all objects qualifying, and for these objects implicit insert event occurrences are triggered (called from the birth event). Again we provide the example bank and accounts, now with the selection of account objects as an initialization rule:

```
Acct:Account A set
  initialized A.No>100000 and A.No<999999 .
```

Note that the `Account` class *cannot be a local class* (local classes are introduced in the next section). Instances of local classes do not exist upon creation time of the enclosing object although their creation can be triggered with the birth event. We face the problem, that the formula has to be evaluated in the context of a class population that becomes visible for the first time *after* the birth event took place.

As for attribute initialization, Troll provides a ***default*** classification of initialization rules for components. In contrast to attributes however, we do not introduce a short notation for initialization with birth event parameters. Components are *objects* and cannot be *transferred* as data values during communication. Nevertheless we can specify birth events with parameters from *identifier data types* together with application specific calling of component insertion events if an initialization rule exists for a component that is classified as ***default***.

3.2.8 Local Classes and Components

As already mentioned in the *local class* section (cf. 2.3.2) it is possible to describe classes that are only local to objects. The key difference is the definition of a *local identification space* for local classes, that means a *local key constraint*. A globally specified class defines a *globally visible identification space* in terms of a *key definition*. The key constraint implicitly derived with such a key definition specifies that there is *at most one* object for a given key tuple.

Suppose now we want to specify companies:

```
object class Company
  identification CompanyID:(Name)
  attributes
    Name:string .
    ...
  components
    Deps:Department D set
      restricted D.InCompany = OID.
    ...
end object class Company
```

relying on the specification of departments:

```
object class Department
  identification DepID:(Name,InCompany)
  attributes
    Name:string.
    InCompany:|Company| .
    ...
  components
    Emps:Employees set .
  ...
end object class Department
```

Thus departments are *dependent* on their *surrounding* companies. In the identification of `Department` this dependency is *explicitly encoded*, the company includes a constraint stating that only the departments having a suitable `InCompany` attribute value (an object id for companies) may be components. Such “backward pointers” are also needed to distinguish between `Departments` of different companies. A global identification space for `Departments` only containing their name is not sufficient here.

From a modelling point of view – regarding departments as entities in their own right – it is more natural to separate their *properties* like `Name` and their *usage* in terms of objects incorporating them. The concept of *local classes* supports this view in that a class is locally specified in the following way:

```
object class Company
  identification CompanyID:(Name)
  local classes
    object class Department
      identification DepID:(Name)
      attributes
        Name:string.
        ...
    end object class Department
  attributes
    Name:string .
    ...
  components
    Deps:Department set .
  ...
end object class Company
```

In this specification the identification space of departments is *localized* to objects of class `Company`. This means, that there may be different objects of class department with the same name but belonging to different companies. Moreover this concept reveals the strong dependency between a class and its local class in that objects of the latter cannot exist without their surrounding objects.

As we have seen in the introduction of this section, the concept of local classes introduced so far can be modelled with global classes as well if we provide a suitable identification space for objects. We introduce local classes in TROLL to support a more natural way of modelling abstracting from the concrete identification space that is needed. A specification like the latter one can then be used to *derive* global classes with an identification space similar to the one used in the former version that are transparent for the specifier and only used for implementation or prototyping issues.

An example for the use of local classes is the specification of stack objects introduced in Section 2.4.2.7, now however not implemented via attributes pointer and array but via *components* pointer and array.

```
object class Stack
  identification ByName:(Name)
  local classes
    object class Entries identification ByIndex:(No)
      template EntryTemplate
    end object class Entries
    object PointerObject
      template PointerTemplate
    end object PointerObject
  components
    Array:Entries set hidden .
    Pointer:PointerObject hidden .
  attributes
    Name:string .
    Top:nat
      derived if not Empty then Array(ByIndex(Pointer.Current-1)).Val fi .
    Empty:bool derived Pointer.Current = 0 .
    EntryExists(No:nat):bool derived Array.Exists(ByIndex(No))
  events
```

```
    variables NewPointer:|PointerObject|; NewEntry:|Entries|;
    Create
      calling
        PointerObject.Create(NewPointer),
        Pointer().Insert(NewPointer) .
    Push(Elem:nat)
      calling
        { EntryExists(Pointer.Current) }
           Array(ByIndex(Pointer.Current)).Set(Elem) ;
        { not EntryExists(Pointer.Current) }
           Entries.Create(NewEntry,Pointer.Current,Elem) ,
           Array.Insert(NewEntry) ;
        Pointer.Increment .
    Pop
      enabled not Empty
      calling Pointer.Decrement .
end object class Stack
```

The class `Entries` uses the following template defining attributes `No` and `Val` describing the position in an "array" respectively the "value" stored at this position. Entries can be created with initialization values and the value stored can be changed later on.

```
template EntryTemplate
  attributes
    No:nat .
    Val:nat .
  events
    Create(No:nat,Val:nat) birth .
    Set(N:nat)  changing Val := N .
end template EntryTemplate
```

The following template just defines a single natural number (attribute `Current`), that can be incremented and decremented after the creation of an object that uses this template:

```
template PointerTemplate
  attributes
    Current:nat initialized 0 .
  events
    Create birth .
    Increment changing Current := Current + 1 .
    Decrement
      enabled Current > 0
      changing Current := Current - 1 .
end template PointerTemplate
```

The example seems to be very complicated but it is in fact not rather natural to define stacks on such a low level with components. Here it is only used as a well known example that uses various concepts. Note that there are a lot of different possibilities to define for example the storage for values. We may also specify the `Array` component as a *list component*, insert elements at the end of the list if necessary, and can avoid the necessity of an *identification* in the `Entries` class. The reader may experiment with this example to explore further possible modellings of this *design* problem.

3.3 Interaction between Components

The signature of an object is composed of the *local signature* and the signature of all part objects (by prefixing with component names and id's). For events of part objects there are no event descriptions since event descriptions concern the *local behaviour* of objects. Communication between the embedding objects and the embedded objects can naturally be described with the events of the embedding object. They are the *cause* or the *origin* of the communication:

```
object class Bank
  ...
  components
    Acct:Account set .
  events
    LocalTransfer(Source:nat,Dest:nat,Amount:money)
      calling
        Acct(AcctID(Source)).withdraw(Amount),
        Acct(AcctID(Dest)).deposit(Amount) .
end object class Bank
```

Nevertheless event occurrences in the components of a composite object can trigger event occurrences of the object incorporating the parts as well as components may communicating among themselves. Thus we need not only *calling rules* defined locally in event descriptions but also *interaction rules* for events between part objects and the embedding object. A template feature called *interaction* serves this purpose. In Chapter 4 we will see a similar concept used to describe communication *between* global objects that are not related by *part-of* relationship.

Syntactically, interaction rules are build from event terms. Event terms can be specified with a selector denoting the path to the relevant component objects. Optionally calling rules can be specified with an application condition formulated as a first order formula. Again we forbid past temporal logic formulae that should be used only in enabling specifications:

Syntax

(For: **interaction** [<*var_spec*>] <*c_interaction_seq*>)

```
<c_interaction>      ::=  [{<formula>}]<interaction_rule_list>

<interaction_rule>   ::=  <evt_term> >> <evt_term_list> .

<evt_term>           ::=  [<selector>]<evt_id> [(<proc_param_list>)]
```

As an example we specify events of account objects that trigger events in the bank incorporating the account:

```
object class Bank
  attributes
    Transactions:nat initialized 0;
  components
    Acct:Account set .
  events
    CountTransaction
     changing Transactions := Transactions + 1 .
  interaction
    variables x:|Account|, m:money ;
    Acct(x).withdraw(m) >> CountTransaction ;
    Acct(x).deposit(m) >> CountTransaction ;
end object class Bank
```

The advantage not to specify the `CountTransaction` event as being called from the `LocalTransfer` event (see above) is a more concise specification of effects. Now all `withdraw` and `deposit` events, regardless how they are triggered, are counted. We may specify different kinds of money transfer inside the bank without having to specify calling to the `CountTransaction` events. The underlying execution model for event executions [HS93] guarantees that only *one* `CountTransaction` event is executed for bank transactions like `LocalTransfer` in this case.

Chapter 4

Class Objects and Roles

In this chapter we introduce implicit class objects as containers for instances and roles as different aspects of objects. The specification of implicit class objects given here is just *one possible* solution for an operational definition of object creation, class population management, etc. Furthermore it is only sketched and has to be worked out for more realistic prototyping issues. It should be seen as part of the *design* of an implementation of a prototyping system. Additionally the object specifications introduced here should be regarded as some more examples.

4.1 Class Objects vs. Object Classes

In the next sections we will introduce various aspects of class objects as containers for objects of a given class. Class objects are thus the basic ingredients to describe features like key constraints, object creation and deletion etc. in an *operational* way. Object classes on the other hand are a *logical concept on an abstract level* that are used to group objects with similar structure and behaviour. As for other features of Troll like e.g. parameterized templates, we consider class objects as being derived from a specification and not directly visible for the specifier. They are used to describe the semantics of Troll features like keys and object creation *operationally*.

In this section we will introduce implicit class objects (Section 4.1.1), an operational description of object creation and destruction (Section 4.1.2), and sketch necessary operations that must be performed in case of key attribute changes (Section 4.1.3). The description of such an *implementation* respectively *prototyping related* feature is nevertheless formulated on a level that abstracts from implementation.

Since this version of Troll is directed towards design and prototyping, the approach with implicit class objects makes sense here. It is in no way a deviation from the property of a specification language to be declarative. We introduce implicit features that are hidden in logical frameworks but have to be implemented in some way. As an example we mention key constraints that can be stated easily on the conceptional level (see for example the `KeyMaps` in [Jun93]) but we have to check this constraint *somewhere*. The features introduced by means of an example here seem to be a suitable way.

4.1.1 Implicit Class Objects

Class objects as containers for objects are introduced as objects in their own right. Such collections of objects of a given class type are used to manage the *extension* of a class type, e.g. the set of *all currently existing objects*, the identification space, etc. For TROLL we do *not* introduce *user defined class properties*. Such properties of collections as for example the average income of a class of employees must therefore be explicitly specified as *composite objects* and derived attributes.

Class container objects are only means to describe the necessary mechanisms to deal with key constraints, object creation and deletion. We will introduce such class container objects by means of an example class, the familiar class `Person`. Persons are specified in the following way:

```
object class Person
  identification
    ByName:(Name, BirthDate),
    BySSN:(SocSecNo)
  attributes
     OID:|Person| . /* implicit attribute, not user specified */
     Name:string .
     BirthDate:date .
     SocSecNo:nat .
     Age:nat initialized 0 .
     ...
  events
     born(Name:string, BirthDate:date, SocSecNo:nat) birth .
     die death .
     ...
end object class Person
```

having two key groups `ByName` and `BySSN`. Note that the birth event of this object class specification initializes all attributes that are part of the keys defined. This property is necessary, since otherwise we would have objects that have *undefined* key values which is known to be impossible if we want to use keys as access points to objects. In this example we exemplified the implicitly generated attribute `OID` that was already mentioned in the last chapter.

As already mentioned, such a specification induces an identification data type `|Person|` with values used as *handles* to person objects. For a given `x` $\in$ `|Person|` we can refer to the object with the expression `Person(x)`, provided that we got this identifier `x` sometime in the past and that there is an object associated to it.

Alternatively we can refer to objects of class `Person` with their *real world id's* for example `Person(ByName("John",6-12-66))`. Here it is also implied that there is a person with this name. However we need not know the identity of the associated object in terms of the identifier data type value.

Another situation arises if we want to trigger the event `born` of a person. We only know its real world name, there is no identity up to now and moreover there is no object existing as a representation of the real world entity. If there exists such an object, we firstly cannot trigger the event `born` since this violates the *key constraint*: "There exist no two objects with equal *identification*". Secondly the `born` event would be forbidden because an object can only execute one *creation event*. Thus we face (at least) three problems that can be solved with implicit class objects:

- Managing the population of currently existing objects of a class (the class type extension). For example to refer to the set of objects currently existing.
- Managing the mapping between identification (real world names) and identities (identity values associated to object instances) including the management of the key constraint.
- Providing means to *create* and *destroy* objects.

In the section on composite objects we have introduced components that play the role of *local* populations of objects visible in the enclosing object. For component symbols we introduced derived signature (attributes, events, and components). For implicit class objects we also derive an implicit signature for the component *set of objects of a given class*, attributes that manage the *mapping between keys and identities*, and events that model the *creation and deletion operations on the class level*. Constraints define the possible combinations of parameters and component object properties:

```
object Class_Person
  components
    Objs:Person set  .
  attributes
    NextID:|Person|
      initialized initialValue .
    ByName(Name:string,BirthDate:date):|Person| .
    BySSN(SocSecNo:nat):|Person|  .
    Empty:bool derived Objs.Empty  .
    Card:nat  derived Objs.Card  .
    IDSet:set(|Person|) derived Objs.IDSet  .
    In(OID:|Person|):bool derived Objs.In(OID)  .
    Exists(OID:|Person|):bool derived In(OID)  .
  constraints
    variables x,y:|Person|; N:string; B:date; S:nat;
    (Objs(x).Name=Objs(y).Name and
     Objs(x).BirthDate=Objs(y).BirthDate) implies  (x=y) ;
    (Objs(x).SocSecNo = Objs(y).SocSecNo) implies  (x=y) ;
    (BySSN(S) = x implies Objs(x).SocSecNo = S) ;
    /* ...etc ...*/
end object Class_Person
```

The component Objs is a set valued component that maintains the current set of objects that are alive. It is left open up to now *how* objects are created and destroyed and *how* identities for new objects are determined.

The attributes ByName and BySSN are used to describe how tuples of data values (identifications) are mapped to identity values. The specification as *attribute generators* automatically guarantees that there is at most *one* value for a given data tuple. However we need some additional constraints to guarantee that values of object attributes are identical to the parameters of the key maps etc.

4.1.2 Object Creation and Destruction

In the last section we have introduced the structure of implicit class objects in terms of attributes, components and constraints by means of an example specification of an implicit class object for class Person. We will now introduce also part of the event interface of such a class object:

```
object Class_Person     /* ...continued... */
  events
    PersonClassCreate        /* --- creates the class object --- */
      birth .
    born(!OID:|Person|,Name:string,BirthDate:date,SocSecNo:nat)
      enabled
        undef(ByName(Name,BirthDate)) and
        undef(BySSN(SocSecNo))
      changing
        ByName(Name,BirthDate) := OID ;
        BySSN(SocSecNo) := OID ;
        NextID := calculateNextID(NextID)
      binding OID = NextID
      calling Objs.Insert(OID),Objs(OID).born(BirthDate:date,SocSecNo:nat).
    die(OID:|Person|)
      enabled Exists(OID)
      calling Objs.Remove(OID) .
    ...
end object Class_Person
```

Such a class object is created with the event `PersonClassCreate` initializing the attribute `NextID` to a value determined by the data type operation `initialValue`. We will not elaborate this operation but consider identifier data types to have the operations `initialValue` and `calculateNextID` available. These operations are considered *invisible* for the *normal* user of TROLL but are necessary for the introduction of the basic features of class objects introduced here.

For class container objects we generate special events for all birth events of the original specification. These events are enlarged by one additional parameter. The parameter is named `OID` and has type `|Person|` in this example. `OID` is a parameter bound on calling this derived event.[1] The class container object is responsible to generate a new, up to now unused identity using the operation `calculateNextID` of `|Person|`. The key map attributes are changed to the appropriate value of `OID`. Needless to say that these events are only enabled if there is no object with the same key attribute values.

We left open *how* objects of this class are created *physically* since this is a matter of implementation (allocation of storage etc). Conceptually, the generated event then calls for the birth event of the object instance and inserts the new instance into the component `Objs`.

Up to now we have introduced the minimal (and partial) specification of class container objects necessary to understand the basic concepts of classes as sets of objects with similar structure and behaviour including the mechanism for naming objects on the abstract level of the world modelled (identification) together with the mechanism to *"internally"* identify objects (identity). The mechanism to create objects is thereby defined to rely on these class container objects. Whereas we trigger events in objects by means of calling them directly, for example writing:

`Person(x).birthday` for a suitable value of `x`

we have to explicitly refer to the class container object if we want to *create* objects. We can assume that a notation like above is an abbreviation for:

```
Class_Person.Objs(x).birthday
```

A similar abbreviation is also supported for birth events. For birth events we use the class name alone to denote the implicit class container object. For example:

```
Person.born(OID,"Jack",23-07-1961,3645230761234)
```

triggering the creation of an object with name Jack and suitable parameters. Again this is an abbreviation of:

[1]Note that in the the binding expression `OID=NextId` the data term `NextID` is evaluated *before* the state change. In other words: the attribute `NextID` always holds a *new, unused* id.

```
        Class_Person.born(OID,"Jack",23-07-1961,3645230761234)
```

omitting the `Class_` prefix for the class container name. Since this object does not exist when the event should occur, we have to trigger the event of the class container object. This in turn creates the new object, and delivers its identity back to the caller via the parameter `OID` of the implicitly generated object creation event.

The explicit reference to the creation events that deliver the identity back to the caller[2] sometimes is useful if we want to insert the newly created object into a component. In this case we need the new identity. It would require two state changes to refer to it via the new key map that can be observed earliest *after* the birth event. See for example the following specification of a company containing houses that may be build by the initiative of the company:

```
object class Company
  identification ByName:(CompanyName)
  components
    Employees:Person set .
    Houses:Buildings set .
    Manager:Person .
    ...
  attributes
    CompanyName:string .
    ...
  events
    variables OID:|Buildings| ;
    buildHouse(Position:Coordinates)
      calling
        Buildings.create(OID,Position) ,
        Houses.Insert(OID) .
  ...
```

The class container `Class_Buildings` is referenced (using the short form `Buildings`) to perform the creation of an object with the suitable birth event `create(...)` for `Buildings`. With the same state change, i.e. in the same snapshot, the new identifier is used to insert this object into the component set `Houses`. The somewhat complicated construction with two events called is only necessary because the object to be incorporated has to be created at the same instant of abstract time. Often we will need only the `Insert` event to incorporate an already existing object.

As we mentioned at the beginning of this chapter this is only one way to operationalize object classes. Here we may not create *several objects at the same time* for example.

[2]Be aware that this operational, procedure call like terminology is not quite correct since callee events and called events conceptually occur at the same time.

4.1.3 Key Attribute Change

To conclude the section on implicit class containers we will sketch another mechanism relevant for the management of key attributes and key maps. In the former TROLL version key values of objects had to be *constant*. This property led to unnatural specifications, namely difficulties to define suitable identifying attributes.[3]

With this version of TROLL the key attribute values may be changed if this is not explicitly forbidden. The key maps defined in the last section must be updated to reflect such a change. Moreover class wide conflicts concerning the key constraints must be detected and are used to forbid such changes. It is natural to derive events for class container objects that are to be called from all object events that change key attributes. For the mentioned person class we may have specified an event `changeName(New:string)`:

```
  events
    changeName(New:string)
      changing Name := New .
```

In the class container objects the following events and interaction rules have to be generated to control the implications of such key attribute changes:

```
object Class_Person
  ..
  events
    changeName(OID:|Person|,New:string)
      hidden
      enabled undef(ByName(New,Objs(x).BirthDate))
      changing
        ByName(New,Objs(x).BirthDate) := OID ;
        ByName(Name,Objs(x).BirthDate) := undefined   .
  interaction
    variables N:string; x:|Person| ;
    Objs(x).changeName(N) >> changeName(x,N) ;
    ...
```

Occurrences of events that change key attributes are thus reflected in the class container objects by means of interactions between the objects of a class and the class container object. Calling of checking events manages the necessary changes for the key maps. In case of conflicts in terms of the various constraints specified in the class container objects, such events may be forbidden due to the underlying execution model for event snapshots and the key attribute semantics.

[3]In general it is difficult to identify object properties that are constant.

4.2 Objects and Object-Roles

Constructing an object oriented model of some UoD using TROLL, we begin with general observations about some basic properties of objects. Object specifications can become rather large if we introduce more and more properties in *one* class specification only. A way to structure such specification is the use of *inheritance*. In TROLL, a notion of inheritance is introduced that goes one step further as inheritance in (most) programming languages. TROLL introduces *roles* as inheritance of object specifications (code reuse) and inheritance of *the objects itself* (see also [JSHS91]).

4.2.1 Structuring Object Specification

For example in a model of the real world containing persons we begin with specifying their general properties like age, living place etc. Than we go on to more specific properties like the ones of employees having an income, an employer, and so on. Employees however *are* persons, particularly they are an *aspect* of persons played during a specific time span in the life of an object [ES91]. A role thus defines an *is-a-relationship* factoring out specific properties of objects that belong together [Per90, Wie90, Wd91].

From the standpoint of conceptual modelling, we identify two possible kinds of such is-a-relationships: *derived* and *dynamic* relationships. The first kind (derived) is sometimes called *specialization* where the belonging of an object to some class is determined with its birth. For example a person can be specialized to belong to class male or class female. A later change should be possible as this example shows but it is not very frequently found for such objects. The second kind (dynamic) is dynamic in the sense that the belonging to a special class *changes* during the lifetime of an object due to occurrence of *events*. The previously mentioned employee is a possible candidate for such a dynamic is-a relationship.

The dynamic kind is more general in that it can simulate the derived is-a relationship. Objects from a derived object class are created and destroyed *together* with their 'base' objects. A role however starts at some point in time in the life of the root object and ends at some later point in time before (or latest with) the death of the root object. We will describe these two kinds of is-a relationships starting with the more general dynamic concept.

4.2.2 Dynamic Roles

Specifying a role, we extend the possible observation and operation interface of an object. An object starts playing a role with a special event, the *birth* event of the role. A role is introduced as a new class introduced as a *role of* `<class_list>`. A role class *inherits* the properties of the classes in the class list. As long as there are no name conflicts, the root of a property can be determined following all inheritance chains. If there are name clashes, it is necessary to qualify the name of the property with the appropriate class name.

A role not only inherits the signature of the base class(es) but also the *objects themselves* in terms of values of attributes, occurring events and so on. For example in an object of class employee we can observe an attribute `Age` of the *associated* person, and events like `Birthday` in case they occur.

Another problem that arises with the specification of roles is the problem of *identification*. On the one hand it is natural that we require the role objects to have the same *identification* as the original objects, i.e. they inherit not only the properties in terms of attributes but also the *mapping that relates identification with identities*. This feature is motivated from the viewpoint of modelling real world entities. For example a `Person` usually does not change its name if it starts playing the role of an `Employee`. In other words, we consider the role to represent a certain *aspect* of the original object. On the other hand, a role object is considered to be an *object in its own right* that *incorporates* the properties of the base object.

For a specification like:

```
object class Student
      role of  Person ...
```

we can refer to the age of the person with `Age` or explicitly writing the origin of the property `Person.Age`.[4] For convenience the second form need to be used only if the origin of the `Age` attribute is ambiguous. The introduction of more observable properties and behaviour thus is *handled* in separate objects described by separate classes embedded in a class hierarchy. The underlying semantics of such kind of inheritance is *delegation* [Ste87]. Requests for properties not defined locally are answered by the original objects. Moreover this view nicely reflects the idea of an *is-a-relationship* being a special kind of a *part-of-relationship* in the sense that the original object is a part of the derived object. Semantically, role objects exists from the birth of their parent objects but their properties are not observable nor are any events enabled except for role birth events (see for example [EDS93, Jun93]).

Syntactically role classes are introduced with the class feature *role of*.

Syntax

<*class_desc*>	::=	**identification** <*key_spec_list*>
		| **role of** <*class_item_list*> [**derived** <*formula*>]
		| **view of** <*class_item*> [**derived** <*formula*>]
		| **template** <*template_id*> [**(** <*data_type_id_list*> **)**]
<*class_item*>	::=	<*class_id*> [<*ovar_id*>]

[4]It is an abbreviation of `Person(OID).Age` denoting the parent object of the role object.

For the time being let us forget about the optional parts in this production as well as the other class features like identification, views, and templates. We only note that the following combinations of features are allowed for a class description:

- Templates may be combined with roles, views, and an identification. This means, that all abstractions can have their own templates.
- An identification can be combined with a role. This means, that role objects can define their own keys (see below).
- Views can only be combined with a template. Moreover this template is strongly dependent on the *base class* (see below).

These rules not only apply for the *reuse* of a template specified elsewhere, but also for a template directly specified for the class at hand.

Roles are characterized by their base classes as defined in the *class item list*. Class items with the introduction of object variables are necessary for options like e.g. derivation conditions.

As an example for roles consider again a `Person` class that has two role classes: `Employee` and `Student`.

```
object class Student
  role of  Person
  ...
  events
    becomeStudent birth .
  ...
object class Student

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

As mentioned earlier, in TROLL we are working with two closely related forms of inheritance, the *class* and *class type* inheritance. The difference between these two forms of inheritance can be depicted as follows. With class inheritance subclasses conceptually are related to *subsets* of the original class populations for a given point in time whereas with class type inheritance subtypes describe *additional* properties of objects. The latter can be compared to the inheritance hierarchy in object oriented programming languages. We speak of *semantic inheritance* in the former and *syntactic inheritance* in the latter case. Semantic inheritance also means that dependent objects like employees as roles of persons *include* their base objects together with their identification and identity as sketched above.

The inheritance of the naming mechanism implies that given an identity or identification of a base object, we may refer to the specialized objects using the identity respectively the identification of the base class and the role class name. A role class thus has not necessarily a private identification, i.e. the class header only mentions the root classes which in turn have a single predecessor class somewhere above in the hierarchy. An additional identification (a new key) can be defined however. An employee may have an attribute `PersonalID` valid in the context of the employer that can be used to define a new identification for `Employee`s

```
object class Employee
  role of   Person
  identification ByPID:(PersonalID)
  attributes
    PersonalID:nat .
  ...
  events
    becomeEmployee(PersonalID:nat) birth .
```

Note that the event that creates this role (`becomeEmployee`) must initialize the attributes used in the new key. We may now refer to `Employee`s with the expression:

```
Employee(ByPID(3573865))
```

As for other classes we have an implicit class object, say `Class_Employee` that manages the population in the role class and that can also manage the translation of external names of objects to identities of objects.

In the inheritance hierarchy we have to find a most general class for a given role, i.e. the base class. In this base class also the basic identification mechanism is defined. To continue the last example we can specify a class `WorkingStudent` as being a role of persons and employees (multiple inheritance). This example is only valid if `Student` and `Employee` are roles of *one* class somewhere 'above' in the inheritance hierarchy:

```
object class WorkingStudent
  role of Student, Employee ;
  ...
```

A specification of this kind has several properties (also summarizing some of the already mentioned cases):

1. We can refer to the base class objects qualifying inherited properties with the class name of the base classes. Note that the name `Person` defined as an abbreviated reference to the base class of employees and students in these classes is also inherited by `WorkingStudent`.

2. Referring to objects of the role class can be done by providing the role class name together with an *identifier or identification* of the base class. The translation of these identifiers to identities of objects is done in the role classes class objects.

3. Name conflicts must explicitly be resolved by the specifier. If there are locally specified properties that have the same name as inherited properties, the inherited properties can only be referred to qualified with the class name where the property is defined. There is *not* a notion of *overriding* in the sense that the semantics of attributes, events, or components is completely changed. We may only *enlarge* the specification. This means that additional restrictions for observations and life cycles are possible. We will discuss this topic in Section 4.2.4.

4. Name conflicts resulting from properties inherited from different classes must also be resolved by the specifier, i.e. such names can only be used in the role classes if they are qualified with the appropriate name.

For a conceptual modelling language it is reasonable *not* to use built in features to resolve name conflicts. This way the specifier is forced to identify the root of a property. Generally we want to avoid *hidden*, built in semantics such as class precedence lists for inherited properties as in CLOS for example [Moo89].

4.2.3 Derived Roles

The second form of inheritance is described by *static* or *derived roles*. The belonging of an object to a static role class is determined by the value of properties of the base class. From a conceptual as well as from an operational viewpoint, objects of the static role classes are implicitly created with the creation of an object of the base class depending on the constant properties of the base class object.

Syntactically, static role classes are distinguished from role classes by the presence of a *derived* clause in the role definition specifying the *specialization condition*. As mentioned above, the general definition of (dynamic) role classes allows a *simulation* of static role classes. The part of a life cycle that an object spends as a role is just extended to the whole life cycle of the parent object. With the birth of a more general object, the *derived* clause determines the creation of objects of the static role classes.

The formerly sketched mechanism can be formulated as conditional calling clauses in the role class specifications that trigger the creation of the specialized objects. Similarly the destruction is performed if the base object is destroyed. See for example the following specification fragment:

```
object class Person
   identification ByName:(Name,Birthdate)
```

```
  attributes
    Name:string .
    Birthdate:date .
    Sex: enum(male,female) constant .
  ...
  events
    birth born(Sex:enum(male,female)) .
    ...
```

and the specialized class:

```
object class FemalePerson
  role of Person derived Person.Sex = female
  attributes
    NoOfChildren: nat initialized 0 .
  events
    born(Sex:enum(male,female))
      inherited from Person
      birth .
    ...
```

with an additional attribute `NoOfChildren` that is initialized to zero upon birth of the role object. For this specialized class an implicit calling clause must be incorporated into the class container specification of `Person`:

```
  interaction
    variables x:|Person|; s:enum(male,female); id:|FemalePerson|;
    { s = female }
       Objs(x).born(s) >> Class_FemalePerson.born(x,s)
```

Conceptually, if an object of class `Person` is created, the calling specified with this rule automatically creates an object of the specialized class `FemalePerson` that is a role object of `Person`. Events that change attributes mentioned in the derivation condition must also trigger birth and death events of possibly different role classes if necessary. In this case there cannot be such events since the only property used in the derivation condition is a *constant* property. Some comments on this construction are in order here.

- The general mechanism is easily described on the conceptual resp. logical level. Since birth events are involved, the appropriate events to create objects must be triggered in the class objects.
- The role concept must be defined on top of the language kernel by means of appropriate components[5] and calling clauses. The above mentioned example describes the basic idea.

[5]Recall that inheritance is modelled as delegation [Ste87], so we may provide components that can be forwarded requests to inherited properties.

- Events of the parent objects that change properties *used* in specialization conditions may be restricted if the triggered birth and death events in case of a *role class change* are not allowed.

The specialization or role hierarchy defines *aspects* of objects in the sense of [ES91] thus we may look at different facets of *one conceptual object*. For implementing or animating this concept we have to deal with *different physical objects* that provide an interface *as if* they were one object.

4.2.4 Refined Base Object Properties

Since TROLL clusters the specification of properties around the specification of the signature symbols (event, attribute, and component symbols), we must also be able to describe various additional "constraints" for *inherited* events, attributes and components. For this purpose, the description of attributes, events, and components provides a feature *inherited from* not mentioned up to now. *Inherited from* is used to tag *properties* that are inherited and that are to be further refined in a role. Such property, e.g. an attribute, may be listed in the ***attributes*** part of the role specification but tagged as *inherited from* `<a base class>`. For example we can specify:

```
object class Employee role of Person ;
  ...
  attributes
    Age:nat
      inherited from Person
      restricted Age <= 65 .
```

further restricting the range of values for attribute `Age`. Such *refinement*[6] can only be done in a *conservative* style. Conservative means, that we are only allowed to *further restrict* the possible life cycles and possible observations of the base class objects.

Overriding however often means *change of behaviour*. We will not deal with overriding as it can be found in most of the popular object oriented programming languages. Since role objects in TROLL *contain* their base objects that are encapsulated in the sense that only local events can change their states, a notion of a more restricted behaviour is possible (additional restriction rules imply less possible life cycles and observations). A notion of overriding however implies that the behaviour can be *changed completely*, that is, we would have to *add* life cycles which contradict the specification of the base objects. In the sequel, we will therefore only describe possible conservative refinements.

[6]The word *refinement* used here has nothing to to with refinement of specifications or processes in the sense of transformations towards implementing specifications.

4.2.4.1 Refinement of Attributes

In this section we will list the possible attribute features that can be refined. Attribute specifications can be further refined in a role object if we respect the following conditions. Note that we only talk about *inherited properties*. The rules do not apply for newly defined properties of a role specification that can be introduced as usual.

- *Hidden*: An attribute *cannot* be newly hidden in a role object.
- *Constant*: To be constant is just a special constraint on the attribute evolution, namely a temporal constraint valid from the first state of the object (*initially*) stating that the initial value of the attribute is not allowed to change.
- *Restricted*: Is also just a special constraint. Note that additional constraints can *prevent* the entry event for a role if the condition does not hold after the birth event (which will be rejected in this case).
- *Initialized*: Makes no sense for dynamic roles. We cannot 'initialize' an attribute that has already a value. Even in the case of a static role that starts its life together with the base object, an initialization must be defined *locally in the base object specification* because an initialization rule is a shorthand for a special changing rule of a birth event.
- *Derived*: Derivation is a kind of constraint allowing only *one* value for a given attribute. On the other hand, attributes of base class objects are changed during the life of the object by means of events and changing rules. Thus refining such a normal attribute to a derived attribute is possible if the derivation rule directly corresponds to the original changing rules which normally makes no sense. We therefore forbid the refinement to a derived attribute.

4.2.4.2 Refinement of Events

Like attributes, events may be refined in different ways. Most of the event features can be specified more particularly in subclasses.

- *Birth*: An event that is inherited from a parent object can serve as a *local* birth event of the role object. This property is *local* to the role specification. It is not reflected directly in the parent object. A closer look on this property reveals, that being the birth event of a role is reflected indirectly in the base class object in the sense that this object is incorporated into the role object which has an effect on the base class object too: there may be further restrictions on life cycles of the base object in the sense described with composite objects.
- *Death*: An inherited event can be the death event of a role object. Like for birth events this means not that the event is a death event of the parent object.

- *Active*: An inherited event can become active in a role object. For example we can describe initiative processes where the role object shows a particular initiative during its lifetime.
- *Hidden*: Same as for attributes.
- *Enabled*: An event can be further restricted in a role object. This implies, that the set of admissible life cycles of the parent object becomes smaller when the parent object plays a role. It should be noted here that additional enabling conditions can lead to *inconsistent specifications* if we further restrict events that are specified to be *necessary* for a life of some parent object by means of explicit process specifications.
- *Changing*: We may specify changing rules for *local* attributes, but the inherited attributes can only be changed by explicitly calling events of the parent object (locality principle for updates, encapsulation of object behaviour).
- *Calling*: Inherited events may call for additional events. This is also a further restriction on possible life cycles because called events may not be enabled and therefore the original events can be forbidden also. In this sense a calling rule is an *enabling condition* for an event because it is only enabled if all events called are also enabled. Additional calling thus is a strengthened enabling condition.
- *Binding*: Additional binding is not allowed since parameters marked *!* have to be set *locally*. A binding for not *!* marked parameters implies a signature change which is not desirable (also some kind of overriding).

4.2.4.3 Refinement of Components

For components that are structurally similar to attributes, the same conditions as for attributes apply. This means that inherited components may not be classified as *hidden*. A restriction however is possible and this may prevent a role entry as is the case for attribute restrictions. As for attributes an initialization or derivation is not possible.

4.2.4.4 Refinement of other Features

We may refine the template specification of the base class templates in that we introduce additional attributes, events, components, constraints, interactions, processes etc. All these rules further restrict the possible attribute values, event sequences etc. in a conservative manner respectively define new properties.

Chapter 5

System Specification

The mechanisms described so far are not sufficient for the description of systems of interacting objects. Classes provide means to structure a society in terms of grouping similar objects, components are a means to describe closely related objects that may communicate, and roles are used to factor out common behaviour in super and special behaviour in subclasses. In system specification however, we have to deal with *relationships* between objects and with *views* or *interfaces* to objects.

For specifying whole systems of objects we must be able to describe relations between separate objects that do not correspond to some composite real world entity. For example the interconnection between customers and banks should not be described on the conceptual level by means of a composite object containing instances of `Customer` and `Bank` nor by *references* specified in `Customer` or `Bank`.

TROLL therefore introduces the concept of *relationships* between objects. Relationships serve two purposes. Firstly and most important they are used to describe *communication relations* between separately specified objects. The concept of calling events is extended to such communication relations. Secondly, relationships may specify *integrity constraints between objects*. Relationships are a structuring concept orthogonal to classes and have thus no attributes, events, and a life cycle definition of their own (cf. LCM [FW93] where relationships are introduced as a special kind of class).

Object and class specifications introduce an event, attribute, and component signature (their basic interface) used to refer to properties of objects. Modelling real world entities we often have to specify different degrees of *privacy* for objects. As we have already seen, events, attributes and components may be declared as *hidden* for observers of objects. The hiding concept alone is not sufficient however to describe *specialized interfaces* of objects and classes. To make such interfaces explicit, TROLL introduces *views* for objects and object classes.

Defining these interfaces we can identify different kinds of such views. Sometimes we want to provide only a subset of properties of objects to other objects. Then we talk about *projection views*. Another possible interface makes visible a certain *subset of a class extension*. Then we speak of *selection views*. Both concepts are similar to relational views and have an obvious semantics in this sense. Other possible views are views related to *several* objects respectively classes: *join* views or to *different* classes: *union* views or *generalizations*. We will not deal with the latter in this report. They are not currently part of TROLL.

5.1 View Classes

For reasons of orthogonality in language design, views are introduced as a special kind of class resembling the definition of role classes. Whereas role classes define a *particular behaviour for a certain time interval* in the life of objects and are fully fledged class specifications, view classes only have a restricted template. This template declares the properties of the base class objects to be used from other classes respectively the subset of objects that are *visible* for other objects.

5.1.1 Projections

Projection views restrict the possible properties of objects that can be observed (attributes, components) or triggered (events). Syntactically a projection view is introduced as a *class* being a *view* for another list. For the time being let us assume that we specify a view for a single class only.[1]

Associated to such a view class is a template. This template however is restricted in that it may not use all features normally available. For a view class we have to explicitly specify the attribute, event and component *signature* that should be visible in the view.

For example a book copy specification in a library can look like follows:

```
object class BookCopy
  identification ByNo:(DocumentNo)
  attributes
    DocumentNo:tuple(Branch:string,No:nat) .
    Title:string .
    Authors:list(string) .
    OnLoan:bool initialized false .
    LastCheckOut:date .
    MaxCheckOutTime:nat initialized 21 .
    Borrowers:list(|Person|) initialized emptylist .
    BadGuys:list(|Person|) initialized emptylist  .
    ...
  events
    buy(DocumentNo:tuple(Branch:string,No:nat),
        Title:string,Authors:list(string))  birth .
```

[1] The language is open however to define views for *several classes*, i.e. join and union views. We will not deal with the latter in this report.

```
    throwAway death .
    borrow(From:|Person|,At:date,Days:nat)
      changing
        OnLoan := true;
        LastCheckOut := At ;
        Borrowers := InsertFirst(From,Borrowers).
    return(From:|Person|,At:date)
      changing
        OnLoan := false;
        { At > AddDaysToDate(LastCheckOut,MaxCheckOutTime) }
                              BadGuys := InsertFirst(From,BadGuys) ;
        ... .
end object class BookCopy
```

For a user of a library we may only provide *information* about the book: its title, authors, and its status as well as the operations to borrow and return it. Then we provide a view in the following way:

```
object class BookCopyToUser
  view of BookCopy
  identification ByNo:(DocumentNo)
  attributes
    DocumentNo:tuple(Branch:string,No:nat) .
    Title:string .
    Authors:list(string) .
    OnLoan:bool .
  events
    borrow(From:|Person|,At:date,Days:nat) .
    return(From:|Person|,At:date) .
end object class BookCopyToUser
```

Since we only define a view on other objects we may not specify further *features* of attributes, events and components already in the base objects. We may only write down the visible signature. An identification can be selected from the base class. In this case there is no choice between different identifications and we used `ByNo` as visible for 'clients' of this view class. Note that the attributes used to construct the identification (key attributes) are also visible. Here the attribute `DocumentNo` is visible to clients of this view. By default we can access objects via a view using their identities – in other words identities cannot be encapsulated.

For attributes and events however we may additionally introduce new symbols that are closely connected to attributes and events of the base class. For example we want to provide information about the current borrower of a book and the date where the book must be returned. Then we may specify new derived attributes for example:

```
  attributes
    ...
    CurrentBorrower:|Person|
      derived if OnLoan then First(Borrowers) fi .
    ...
```

For an interface to a (real world) user we have to provide more information than only the identifier of the current borrower since this identifier is *unprintable* but this is not in the scope of this example. Here we only want to introduce some kind of *encapsulation*.

Events may be introduced newly using calling.[2] The event specified must have a set of called events that define its behaviour in terms of events of the original objects.

5.1.2 Selections

Additional to the projection feature for views we may not only describe a visible subset of attributes, events, and components for a given base class but we can also restrict the *population* that is visible. In this case we have to provide a suitable *selection condition* that is used similar to the selection condition for (static) role classes. If we want to make visible only the documents with a branch identification 'B' in their document number we have to write:

```
object class BookCopyFromBranchB
  view of BookCopy derived DocumentNo.Branch = 'B'
  identification ByNo:(DocumentNo)
  attributes
    DocumentNo:tuple(Branch:string,No:nat) .
    Title:string .
    Authors:list(string) .
    OnLoan:bool .
  events
    borrow(From:|Person|,At:date,Days:nat) .
    return(From:|Person|,At:date) .
end object class BookCopyFromBranchB
```

providing only objects in this view that respect the derivation condition mentioned.

For projection and selection views the identification mechanism is *obtained* from their base classes. This means, that we may use object identities as well as object

[2]Event derivation is thus introduced by means of calling as the simplest form of derivation. We do not allow an event derivation consisting of a *process* of events taken from the base class, called *reification* in [ES90, EDS93, FM93]. In general if we specify an event just by its name and calling to other events (no further features) this can be seen as a very simple derivation facility, mostly a change of names and translation of parameters.

identification via key attributes at the view level. We must however indicate which key groups are visible.

5.2 Relationships

Now that we have described views as specialized interfaces to objects and object classes we have to describe how separately specified objects can be put together to make up a society specification of interacting objects.

Whereas the concept of composite objects defines communication and integrity constraints between objects of a composition, yet we have no means to describe communication and constraints between *separate objects* so far. The use of composite objects for the general description of communication should not be used on the society specification level since communication relationships are *buried* in the object specifications of the interacting objects this way [JHS93]. TROLL – as its predecessor TROLL1 – propose to use an *explicit relationship construct* that is orthogonal to classes, roles, and views.

In TROLL there are two possible relationships: *global constraints* and *global interactions*. The former concept is used in early stages of design when we do not want to decide where constraints have to be formulated, the latter concept is used in later stages of design when we *put together* objects from different classes.

Global constraints are a means to relate objects depending on their observable properties (attribute values) whereas global interactions relate objects with respect to *communication*. Syntactically, relationships i.e. global constraints and interactions are defined as follows:

Syntax

```
<rel_spec>        ::=  relationship <rel_id> between <class_item_list>
                         [ constraints [<var_spec>] <constr_seq> ]
                         [ interaction [<var_spec>] <c_interaction_seq> ]
                       end relationship <rel_id>
```

We may introduce an object variable for each class participating so that we can refer to properties of the objects associated through the global constraint respectively global interaction.

5.2.1 Global Constraints

The only *inter object constraints* that we have introduced so far were *key constraints*. They are special in that they relate *different* objects from *one* class. To do this we introduced *class container objects* to have a context containing objects for which the constraint has to be formulated.

Just as an example for the use of global constraints we may want to specify the key constraint for person objects (see Page 58):

```
relationship PersonKeyConstraintSSN between Person P1, Person P2
  constraints
    (P1.SocSecNo = P2.SocSecNo) implies (P1.OID = P2.OID)
end relationship PersonKeyConstraintSSN
```

Here we have a the special case that we want to refer to related objects of *one* class which further motivates the use of object variables to distinguish between objects mentioned in this specification. The condition expresses, that an equal value for the `SocSecNo` attribute of objects of class Person implies that the objects are the same. This is a rather special global constraint and is only introduced here because of the well known example.

5.2.2 Global Interactions

Similar to interaction between parts of a composite object we may also specify interaction relationships between objects from different classes. An interaction is a constraint for possible life cycles of the communicating objects: they must synchronize. For example we may specify the interaction between a bank and a customer by means of a relationship as follows:

```
relationship BankCustomer between Bank B, Customer Cus
  interaction
    variables Amount:money; Acc:|Account|;
      Cus.sendMoneyTo(Amount,Acc,B.OID) >>
                    B.getMoneyFor(Amount,Acc,Cus.OID)
end relationship BankCustomer
```

The `Customer` specification defines an event `sendMoneyTo` with parameters denoting the amount of money (`Amount`), the destination account number (`Acc`), and the bank identification (`BID`) whereas the `Bank` specification defines an event `getMoneyFrom` with the amount, the destination, and the source of the money. The expressions `B.OID` and `Cus.OID` define the relevant *objects* that communicate, i.e. if a `sendMoneyTo` event occurs in a customer object with the specific `Bank` id value, in the corresponding bank object the `getMoneyFrom` event occurs. As for interactions in composite objects interaction rules may be *conditional*, i.e. communication only takes place if the associated condition holds.

5.3 Society Specification

The specification of a society groups the various specification items together and introduces a name for it. An object society is specified as follows:

```
object society XPTO
   including a list of data types to be imported from elsewhere
   followed by data type declaration,
               template,
               class,
               role,
               view, and
               relationship specifications
```

To be more particular we introduce *object society specifications* with implicitly generated *specifications* for class container objects. The class container *objects* possibly generated from such specification manage the *real instances* in the sense of a running (prototyped) system: the object society itself. On the language level however there is a clear distinction between the *specification* in terms of templates, class types, roles, views, and relationships and the instances that may be *generated* from the specification of classes. In this report we introduced some ideas how theses issues can be combined using implicit class containers. There is a clear separation between these issues however.

Syntactically an object society specification is represented as follows:

Syntax

<*society_spec*>	::=	**object society** <*society_id*> <*society_desc_items*> **end object society** <*society_id*>
<*society_desc*>	::=	**including** <*domain_list*> | **data types** <*dt_decl_list*> | <*template_spec*> | <*class_spec*> | <*rel_spec*>
<*dt_decl*>	::=	<*dt_id*> = <*domain*>

The data type declaration feature is used to introduce short names for *constructed data types*. This is especially useful if we use enumerations. For example we may introduce an enumeration like the following:

```
object society XPTO
  data types
    SexType = enum(Female, Male)
    ...
```

and can henceforth use the data type `SexType` as if it is included as e.g. data type `nat`.

Chapter 6

Conclusions and Outlook

In this report we have presented the revised version of the language TROLL. Major enhancement is done to make the language more usable than the previous version. The result is a more practical syntax, i.e. properties are specified beneath their related concepts. Language features are made orthogonal which offers a more intuitive way of specification.

TROLL as an object-oriented specification language allows for a declarative description of information systems. The basic concepts of TROLL can be characterized as follows:

- Basic building blocks of information systems are objects.
- Objects are observable processes that may interact.
- Objects encapsulate an internal state.
- Objects have a unique identity and can be classified.
- An object interface is described through a set of attributes and events.
- Object evolution is specified by the possible sequence of event occurrences.
- Specifications can be structured by using the concepts of classes, object composition (aggregation), inheritance and relationship that together make up the society specification.

Different formalisms like first order logic, temporal logic, and process specifications similar to CSP [Hoa85] are integrated into TROLL.

The specification elements of the TROLL language are divided into *concepts* and *features*. Concepts are the basic language elements. They either describe the society structure (e.g. classes, relationships, data types), class structure (e.g. attributes, components, events, processes) or the interconnection structure (e.g. interactions, global constraints).

Features specify concepts more detailed (e.g. enabling condition for events, value constraints for attributes, ...). The distinction between concepts and features is not important for the understanding of the language but provides a taxonomy for the supporting tools.

6.1 Tool Support for Troll

The need for tool support in information system modelling is widely accepted [Wij85]. This support should be facilitated by an integrated software engineering environment (SEE) rather than by loosely coupled tools [SB93]. A SEE enables the use of certain methods and languages. Many informal as well as formal modelling and specification languages have been proposed in the past [RBP+91, SM90]. The advantages of the latter over the former are well identified. Although the area of SEEs is still dominated by the informal approaches. Most tools supporting formal specification languages are developed in research institutions. They are rather prototypes than robust tool sets which can be used in serious project development or by non-experts.

We believe that formal specification and modelling languages open a variety of opportunities for tool support which can hardly be achieved by informal approaches.

- Formal languages allow for language sensitive editors, which can check at specification time syntactic correctness as well as completeness and consistency.
- The conceptual system model should be understandable for the domain specialist who has to validate the model against the problem domain. Specification animation either in an interpretative way or through a generated prototype is well suited for validation.
- Tool assisted verification of specification properties is another important aspect of formal specifications. It enables the user to prove the absence or the existence of certain facts in a specification.

Besides these aspects of tool support which are directly related to the formality underlying the model, we have to support many other activities related to the engineering of large systems, e.g. project management, multi-user support, versioning, traceabilitiy, etc.

Last but not least tool support will make formal methods accessible to a much larger community than it is the case nowadays. It will allow us to show the usability and the opportunities of such approaches.

Many informal modelling languages gain a lot of attraction due to their intuitive and convenient syntax. OMT [RBP+91] is currently among the most popular approaches and we adapted partially the OMT notation to represent Troll concepts [WJH+93]. We use the object model notation to specify the structural relationships between objects. For the process specification a variation of state diagrams, the dynamic model of OMT, is proposed. The specification of expressions over data values stays textual. The usage of the functional model did not seem appealing to us. Additionally to the OMT models we introduced new graphical features to represent object communication in an adequate way

Our current effort is on building such an integrated tool set based on Troll, which is called the Tbench. The first phase in developing the SEE Tbench emphasizes on the aspects of tool supported creation and manipulation of specification documents.

6.2 Specification Support

The software process consists of peoples (or roles of people), various products (or documents), activities performed by people or performed automatically and long term processes. These components could be supported by means of notation, procedure and several mechanisms. Especially in the early phases of the software process tool support for maintaining documents of conceptual modelling is mandatory.

Troll models information systems as object societies. Each object society specification consists of classes, aspects of classes (roles) interfaces for classes (views) and relations between them (relationships). Each class consists of structural and behavioural aspects which could be partitioned into *concepts* and *features*. The tool support for specification reflects the structure of object societies. It is separated in two levels (society level and object level) and provides one society at a time.

A Troll specification describes a society entirely but an adapted 'popular' graphical notation on top of a formal object-oriented specification language helps to make Troll more usable in practice. Graphical notations are advantageous in giving overviews. Detailed information is specified more concise textual.

For the above mentioned reasons, the Tbench support is split up in two tools. The change between this tools within the specification process is not strictly prescribed. Depending on the currently necessary information during the specification process we identify two levels:

Level 1: Graphical specification of global information of a society.

The graphical notation used for the depiction of overviews is omTroll (OMT-notation adapted to Troll [WJH$^+$93]). Different graphical elements represent specification of *templates*, *classes*, *relationships* and connections of *role*, *view*, *relationship* and *component* of Troll.

On this level of specification, classes, aspects of classes and connections between them could be created or deleted. Public offered services of the Tbench can be applied with the displayed elements as parameters, e.g. export and editing of classes are performed this way.

The tool *TGSM* (Tbench Graphical Specification Manager), which is frontend of the Tbench, provides the needs of the first level of specification.

Level 2: Textual refinement of one object specification document.

One object specification selected from the TGSM is depicted textually

The structure of editing is generic (simplifies language variants) and reflects the structure of one class which is partitioned into *concepts* and their *features*. Only features of the selected concept are offered for modification. This is provided via feature-dependent dialogues which also perform online syntax check. It has to be possible to maintain erroneous or incomplete information to be able to specify only parts of classes at a time. Automated refinement helps to keep up afterwards at position of next error or next incomplete. Object structure driven schedules guide to perform 'next' in sense of the actual concept (e.g. return parameter specification of events involve parameter binding specification). Additionally session related annotations and feature related comments are supported.

The tool *TED* (Tbench Editor) provides the needs of the second level of specification.

Our approach has several advantages against general purpose editors:

- only selected information is represented (confusing details are hidden),
- more formal information is depicted if desired,
- only syntactically correct information is maintained,
- specification text is automatically parsed for posterior access,
- identifiers, suitable concepts, and features are visualized and suggested for specification.

6.3 Prototyping Support for Validation

The prototyping process, which will be supported by the first prototype of the Tbench, pictures the connection from graphical modelling of conceptual aspects of the universe of discourse to independently executable applications. It is partitioned into the following phases. Each of them requiring the effect of its previous phase to be performed:

Configuration. The prototyping process starts and is accompanied by configuring the environment, ie. the society to be edited, location of the necessary services and specification documents, etc.

Specification. As sketched in the previous section, the specification support is a cycle of: global society arrangement as creation and deletion of parts of the society (classes, relationships, etc.) and its connections (components, roles, views, etc.) and syntax controlled refinement of textual representations.

Consistency check. A set of consistency checkers controls different aspects of consistency of one society.

Transformation. A set of transformators map high level TROLL concepts into operational TROLL-kernel.

Generation. Different generation tools map operational TROLL specifications to each different available programming languages (C++, Prolog, etc.)

Execution. Generated prototypes can be executed to validate specifications against user needs [HJS93].

Each phase is supported by a set of tools. The schedule within the mentioned phases is interactive and not strictly prescribed. Parts of it can be specified by the process concepts of TROLL.

Centralized communication within the TBENCH enables logging of all kind of information. Versioning support submits specification document evolution control. This accumulated information about habits of developers helps effecting case studies and improving the method of specification.

6.4 Outlook

Our main emphasis for the future is to instantiate a straightening way from graphical specification of TROLL object societies to executable applications (prototypical information systems). It is appointed to support the phase of conceptual modelling in the area of education. Project management aspects are secondary.

To be as flexible as possible the environment is developed with respect to the following claims:

Integrated. The integration frame of the environment is developed by ourselves to fit precisely our needs: data integration on syntactical level, control integration on semantical level and presentation integration on lexical level [Sof92]. The aspects of integration are briefly described by the following topics:

Open and Extensible. The environment consists of a set of tools and a repository. The repository maintains specification and administration data which are both represented as TROLL specifications. The data access is performed via enhanced data structures of the tools. Explicit database binding for each tool could be dropped.

Tool services are grouped into completed applications managed by a centralized supervisor tool, the TBENCH Integration Manager which implements transparent distribution. Each tool is exchangeable because of component independence. Needed functionality could by ‘plugged in’ as a new service. Available services are dynamically offered and used. Unified interfaces with public descriptions (specified in TROLL) of all services of the TBENCH enable the possibility to utilize *reuse on application level*.

Portable. Depending on our appointment of use, the environment is build of and uses only common spread tools and languages (Tk/Tcl, RPC, SQL, C/C++, Lex and Yacc).

Intuitive. Unitary clear arranged graphical user interfaces (window based, mouse driven) with a widget-oriented helpsystem support the use of the environment.

A prototyping environment based on an object-oriented specification language supports validation, verification, consistency control of the modeled universe of discourse. We concentrated on the phase of specification strategy which is twofold: Graphical support for specification of global information with textual syntax controlled refinement. The integrated environment framework enables the extensibility and exchangeability in the direction of needs for the future (e.g. other graphical notations on top).

Bibliography

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

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

[Con94] Conrad, S.: On Certification of Specifications for TROLL *light* Objects. In: Orejas, F. (ed.): *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (WADT/Compass'92.* Springer, LNCS, 1994.

[CSS89] Costa, J.-F.; Sernadas, A.; Sernadas, C.: OBL–89 User's Manual (Version 2.3). Internal report, INESC, Lisbon, 1989.

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

[EGH$^+$92] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, Vol. 9, No. 2, 1992, pp. 157–204.

[Eme90] Emerson, E. A.: Temporal and Modal Logic. In: Leeuwen, J. van (ed.): *Formal Models and Semantics.* Elsevier Science Publishers B.V., 1990, pp. 995–1072.

[ES90] Ehrich, H.-D.; Sernadas, A.: Algebraic Implementation of Objects over Objects. In: deBakker, J. W.; deRoever, W.-P.; Rozenberg, G. (eds.): *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness".* LNCS 430, Springer, Berlin, 1990, pp. 239–266.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Information Systems –*

Correctness and Reusability. TU Braunschweig, Informatik Bericht 91-03, 1991, pp. 1–24.

[ESS88] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Abstract Object Types for Databases. In: Dittrich, K. R. (ed.): *Advances in Object-Oriented Database Systems*, Bad Münster am Stein, 1988. LNCS 334, Springer, Berlin, 1988, pp. 144–149.

[ESS89] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Objects, Object Types, and Object Identification. In: Ehrig, H.; Herrlich, H.; Kreowski, H.-J.; Preuß, G. (eds.): *Categorical Methods in Computer Science.* LNCS 393, Springer, Berlin, 1989, pp. 142–156.

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

[FM93] Fiadeiro, J.L.; Maibaum, T.: Actions are Objects: Refinement in the temporal Logic of Objects. In: Lipeck, U.W.; Koschorreck, G. (eds.): *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, 1993, pp. 213–226.

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

[FW93] Feenstra, R.; Wieringa, R.: LCM 3.0: A Language for Describing Conceptual Models — Syntax Definition. Report ir-344, Faculteit der Wiskunde en Informatica, Vrije Universiteit, Amsterdam, 1993.

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

[HJS92] Hartmann, T.; Jungclaus, R.; Saake, G.: Aggregation in a Behavior Oriented Object Model. In: Lehrmann Madsen, O. (ed.): *Proc. European Conference on Object-Oriented Programming (ECOOP'92).* Springer, LNCS 615, Berlin, 1992, pp. 57–77.

[HJS93] Hartmann, T.; Jungclaus, R.; Saake, G.: Animation Support for a Conceptual Modelling Language. In: Mařík, V.; Lažanský, J.; Wagner, R.R. (eds.): *Proc. 4th Int. Conf. on Database and Expert Systems Applications (DEXA), Prague.* LNCS 720, Springer, Berlin, 1993, pp. 56–67.

[HK87] Hull, R.; King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol. 19, No. 3, 1987, pp. 201–260.

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

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

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

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

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

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

[Lam83] Lamport, L.: Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, 1983, pp. 190–222.

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

[Moo89] Moon, D.A.: The Common Lisp Object-Oriented Programming Standard. In: Kim, W.; Lochovsky, F.H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press, Frontier series, 1989, pp. 49–78.

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

[Per90] Pernici, B.: Objects with Roles. In: *Proc. ACM/IEEE Int. Conf. on Office Information Systems*, Boston, 1990. Special Issue of SIGOIS Bulletin, Vol. 11, No. 2&3, ACM Press, New York, 1990, pp. 205–215.

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

[Pnu86] Pnueli, A.: Application of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Current Trends in Concurrency.* LNCS 224, Springer-Verlag, Berlin, 1986.

[RBP$^+$91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-oriented modeling and design.* Prentice-Hall, 1991.

[Saa89] Saake, G.: On First Order Temporal Logics with Changing Domains for Information System Specification. Informatik-Bericht 89–01, Technische Universität Braunschweig, 1989.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991, pp. 47–74. North-Holland.

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

[SB93] Schefström, D.; Broek, G. van den: *Tool Integration.* Wiley Professional Computing, 1993.

[Sch92] Schulze, J.: Vereinfachung von dynamischen Objektspezifikationen. Diplomarbeit, TU Braunschweig, 1992.

[SE91] Sernadas, A.; Ehrich, H.-D.: What Is an Object, After All? In: Meersman, R.; Kent, W.; Khosla, S. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, Amsterdam, 1991. North-Holland, pp. 39–70.

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

[SFSE88] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: Abstract Object Types: A Temporal Perspective. In: Banieqbal, B.; Barringer, H.; Pnueli, A. (eds.): *Proc. Colloq. on Temporal Logic in Specification.* LNCS 398, Springer, Berlin, 1988, pp. 324–350.

[SFSE89] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis*, Namur (B), 1989. North-Holland, Amsterdam, 1989, pp. 225–246.

[SH94] Saake, G.; Hartmann, T.: Modelling Information Systems as Object Societies. In: von Luck, K.; Marburger, H. (eds.): *Management and Processing of Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence, Hamburg.* Springer, Berlin, LNCS 777, 1994, pp. 157–180.

[SHS93] Schwiderski, S.; Hartmann, T.; Saake, G.: Monitoring Temporal Preconditions in a Behaviour Oriented Object Model. Informatik-Bericht 93-07, TU Braunschweig, 1993.

[SJ91] Saake, G.; Jungclaus, R.: Konzeptioneller Entwurf von Objektgesellschaften. In: Appelrath, H.-J. (ed.): *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft BTW'91*. Informatik-Fachberichte IFB 270, Springer, Berlin, 1991, pp. 327–343.

[SJ92] Saake, G.; Jungclaus, R.: Views and Formal Implementation in a Three-Level Schema Architecture for Dynamic Objects. In: Gray, P.M.D.; Lucas, R.J. (eds.): *Advanced Database Systems : Proc. 10th British National Conference on Databases (BNCOD 10), July 6-8, 1992, Aberdeen (Scotland)*. Springer, LNCS 618, Berlin, 1992, pp. 78–95.

[SL89] Saake, G.; Lipeck, U.W.: Using Finite-Linear Temporal Logic for Specifying Database Dynamics. In: Börger, E.; Kleine Büning, H.; Richter, M. M. (eds.): *Proc. CSL'88 2nd Workshop Computer Science Logic*. Springer, Berlin, 1989, pp. 288–300.

[SM90] Shlaer, S.; Mellor, S.J.: *Object Lifecycles: Modeling the World in States*. Prentice-Hall, 1990.

[Sof92] Software, IEEE: *Integrated Case*. IEEE Computer Society, March 1992.

[SS93] Schwiderski, S.; Saake, G.: Monitoring Temporal Permissions using Partially Evaluated Transition Graphs. In: Lipeck, U.; Thalheim, B. (eds.): *Proc. 4th International Workshop: Modelling Database Dynamics, Volkse 1992*. Workshops in Computing, Springer, Berlin, 1993, pp. 196–217.

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Stoecker, P.M.; Kent, W. (eds.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*. VLDB Endowment Press, Saratoga (CA), 1987, pp. 107–116.

[SSG^{+}91] Sernadas, A.; Sernadas, C.; Gouveia, P.; Resende, P.; Gouveia, J.: OBLOG – Object-Oriented Logic: An Informal Introduction. Internal report, INESC, Lisbon, 1991.

[Ste87] Stein, L.A.: Delegation is Inheritance. *SIGPLAN Notices, Special Issue OOPSLA87*, Vol. 22, No. 12, 1987, pp. 138–146.

[UD86] Urban, S. D.; Delcambre, L.: An Analysis of the Structural, Dynamic, and Temporal Aspects of Semantic Data Models. In: *Proc. Int. Conf. on Data Engineering*, Los Angeles, 1986. ACM, New York, 1986, pp. 382–387.

[Wd91] Wieringa, R.; de Jonge, W.: The Identification of Objects and Roles – Object Identifiers Revisited. Technical Report IR–267, Vrije Universiteit, Amsterdam, 1991.

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

[Wij85] Wijers, G. M.: *Modelling Support in Information Systems Development.* Thesis Publisher Amsterdam, 1985.

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

Appendix A

The Troll-Syntax

The following symbols are meta-symbols for the grammar:

- **terminal symbols** and <*non-terminal symbol*>
- optional [**terminals**] and [<*non-terminal symbols*>]
- alternatives are separated by |

To clearly distinguish terminal symbols from non terminals we use a **boldface sans serif** font whereas the examples in the text use *sans serif* font.

The following abbreviations are used to simplify production rules for repeating occurrences of nonterminals. Let x be a non-terminal-symbol:

<*x_list*> ::= <*x_list*> **,** <*x*> | <*x*>

Commalists are used to separate closely related specification items as for example changing specifications or called events for *one* associated application condition and for example parameter lists.

<*x_seq*> ::= <*x_seq*> **;** <*x*> | <*x*>

Sequences are used to separate specification formulae that further describe for example features of events above the level of the commalists.

<*x_items*> ::= <*x_items*> <*x*> | <*x*>

We need no separators between parts of a specification that start with keywords.

A.1 Sublanguages

A.1.1 Formula Sublanguage

```
<bool_op>          ::=  implies | and | or

<bool_const>       ::=  true | false

<quantifier>       ::=  forall | exists

<formula>          ::=  <formula> <bool_op> <formula>
                        | not <formula> | <bool_const>
                        | (<formula>)
                        | <quantifier> (<var_decl_list>:: <formula> )
                        | <data_term>
                        | undef(<data_term>)
                        —— predicates for past tense ——
                        | after(<evt_term>) | occurs(<evt_term>)
                        —— past tense temporal logic ——
                        | always <formula> | sometime <formula>
                        | previous <formula>
                        —— bounded past predicates ——
                        | always <formula> sincelast <formula>
                        | sometime <formula> sincelast <formula>
                        —— future tense temporal logic ——
                        | henceforth <formula> | eventually <formula>
                        | next <formula>
                        —— bounded future predicates ——
                        | henceforth <formula> until <formula>
                        | eventually <formula> before <formula>
```

A.1.2 Data Sublanguage

```
<inf_op>           ::=  + | - | * | / | div | mod

<compare_op>       ::=  = | <> | < | > | <= | >=

<unary_op>         ::=  ~ | -

<post_op>          ::=  [<data_term>] | .<tuple_sel_id>

<const_symbol>     ::=  <nat_const> | <float_const> | <bool_const>
                        | <char_const> | <string_const>
```

```
<data_term>           ::=  <data_term> <inf_op> <data_term>
                           | <data_term> <compare_op> <data_term>
                           | <unary_op> <data_term>
                           | <data_term><post_op>
                           | <op_id>[(<data_term_list>)]
                           | (<data_term>)
                           | if <formula> then <data_term> [else <data_term>] fi
                           | <var_id> | <parameter_id> | <const_symbol> | <att_term>
                           | undefined | <formula>
```

A.1.3 Process Sublanguage

```
<process>             ::=  <process> -> <process_unit> | <process_unit>

<process_unit>        ::=  <process_term> | <evt_term>
                           | <choice> | <parallel> | <foreach> | nil

<choice>              ::=  ( <choice_alternative> )

<choice_alternative>  ::=  <guarded_process> | <choice_alternative>'|'<guarded_process>

<guarded_process>     ::=  [{ <formula> }] <process>

<parallel>            ::=  ( <parallel_events> )

<parallel_events>     ::=  <evt_term> | <parallel_events>'||'<evt_term>

<foreach>             ::=  foreach <var_id>:<data_term> do <process> od
```

A.2 Terms

```
<att_term>            ::=  [<selector>]<att_id> [(<data_term_list>)]

<evt_term>            ::=  [<selector>]<evt_id> [(<proc_param_list>)]

<process_term>        ::=  [<selector>]<process_id> [(<proc_param_list>)]

<proc_param>          ::=  ['?']<var_id> | <data_term> | ['?']<param_id>

<selector>            ::=  <selector><select_id>. | <select_id>.

<select_id>           ::=  <class_id> | <ovar_id> | <cmp_term>

<cmp_term>            ::=  <cmp_id>[(<data_term_list>)] [<obj_ref>]

<obj_ref>             ::=  (<data_term>) | ()
```

A.3 Data Types and Declarations

```
<domain>          ::=  <data_type_id> | '|'<class_id>'|'
                       | tuple(<domain_item_list>)
                       | list(<domain>) | set(<domain>)
                       | enum(<enum_id_list>)

<domain_item>     ::=  <tuple_sel_id>:<domain>

<var_spec>        ::=  variables <var_decl_seq>;

<var_decl>        ::=  <var_id_list> : <domain>

<param_decl>      ::=  [<parameter_id>:]<domain>

<dt_decl>         ::=  <dt_id> = <domain>
```

A.4 Template Structure

```
<template_spec>   ::=  template <template_id> [(<dt_param_id_list>)]
                          <template_desc_items>
                       end template <template_id>

<template_desc>   ::=  local classes <class_spec_items>
                       |  components [<var_spec>] <cmps_spec_items>
                       |  attributes [<var_spec>] <atts_spec_items>
                       |  events [<var_spec>] <evts_spec_items>
                       |  constraints [<var_spec>] <constr_seq>
                       |  process declaration [<var_spec>] <process_decl_items>
                       |  processes [<var_spec>] [<process_use_items>]
                       |  interaction [<var_spec>] <c_interaction_seq>
```

A.5 Template Signature

A.5.1 Components

```
<cmps_spec>       ::=  <cmp_id>[(<param_decl_list>)]:<class_item>
                          <cmp_desc_items> .

<cmp_desc>        ::=  set | list
                       | inherited from <class_id>
                       | hidden
                       | restricted <formula>
                       | initialized <formula> [default]
                       | derived <formula>
```

A.5.2 Attributes

<atts_spec>	::=	*<att_id>*[(*<param_decl_list>*)] :*<data_type>* *<att_desc_items>*.
<att_desc>	::=	**inherited from** *<class_id>*
		| **hidden** | **constant**
		| **restricted** *<formula>*
		| **initialized** *<data_term>* [**default**]
		| **derived** *<data_term>*

A.5.3 Events

<evts_spec>	::=	*<evt_id>*[(*<df_param_list>*)] *<evt_desc_items>* .
<df_param>	::=	[**!**][*<param_decl>*]
<evt_desc>	::=	**inherited from** *<class_id>*
		| **birth** | **death** | **active** | **hidden**
		| **enabled** *<formula>*
		| **changing** *<c_changing_seq>*
		| **calling** *<c_calling_seq>*
		| **binding** *<c_binding_seq>*
<c_changing>	::=	[{*<formula>*}] *<changing_list>*
<changing>	::=	*<att_term>* **:=** *<data_term>*
<c_binding>	::=	[{*<formula>*}] *<binding_list>*
<binding>	::=	*<parameter_id>* **=** *<data_term>*
<c_calling>	::=	[{*<formula>*}] *<evt_term_list>*

A.6 Behaviour

A.6.1 Constraints

<constr>	::=	[**initially**] *<formula>* .

A.6.2 Processes and Life Cycles

<process_decl>	::=	*<process_id>*[(*<df_param_list>*)] **=** *<process>* .
<process_use>	::=	*<process_term>* *<process_desc_items>* .
<process_desc>	::=	**initiative** | **weak** | **start** *<formula>*
		interleaving *<interleave_mode>*

```
<interleave_mode>    ::=  none | free | excluding <event_term_list>
```

A.6.3 Interaction

```
<c_interaction>      ::=  [{<formula>}]<interaction_rule_list>

<interaction_rule>   ::=  <evt_term> >> <evt_term_list> .
```

A.7 Abstractions

A.7.1 Classes

```
<class_spec>         ::=  object [class] <class_id>
                            <class_desc_items>
                            [<template_desc_items>]
                          end object [class] <class_id>

<class_desc>         ::=  identification <key_spec_list>
                          | role of <class_item_list> [derived <formula>]
                          | view of <class_item> [derived <formula>]
                          | template <template_id> [( <data_type_id_list> )]

<key_spec>           ::=  <key_id> : ( <att_id_list> )

<class_item>         ::=  <class_id> [<ovar_id>]
```

A.7.2 Society Specification

```
<rel_spec>           ::=  relationship <rel_id> between <class_item_list>
                            [ constraints [<var_spec>] <constr_seq> ]
                            [ interaction [<var_spec>] <c_interaction_seq> ]
                          end relationship <rel_id>

<society_spec>       ::=  object society <society_id>
                            <society_desc_items>
                          end object society <society_id>

<society_desc>       ::=  including <domain_list>
                          | data types <dt_decl_list>
                          | <template_spec> | <class_spec> | <rel_spec>

<dt_decl>            ::=  <dt_id> = <domain>
```

Appendix B

Operations for Constructed Data Types

The following *generic* or parameterised sorts are assumed to be predefined (where *elem* is an arbitrary predefined sort, *sel* a selector, and *sym* a symbol):

$set(elem)$	(Sets)
$list(elem)$	(Lists)
$enum(sym_1, \ldots, sym_n)$	(Enumeration)
$tuple(sel_1 : elem_1, \ldots, sel_n : elem_n)$	(Tuples or records)

- For sets, we have the following operations:

$emptyset : \rightarrow \texttt{set}$	(set constructor)
$in\ \_\ \_ : elem \times \texttt{set} \rightarrow \texttt{bool}$	(element of?)
$empty\ \_\ \_ : \texttt{set} \rightarrow \texttt{bool}$	(emptypset?)
$insert\ \_\ \_ : elem \times \texttt{set} \rightarrow \texttt{set}$	(insert element into set)
$remove\ \_\ \_ : elem \times \texttt{set} \rightarrow \texttt{set}$	(remove element from set)
$card\ \_ : \texttt{set} \rightarrow \texttt{nat}$	(cardinality)

- For tuples, we have only selectors and constructors:

$tuple\ \_ \ldots \_ : elem_1 \times \ldots \times elem_n \rightarrow \texttt{tuple}$	(tuple constructor)
$\_\ .sel_1 : \texttt{tuple} \rightarrow elem_1$	(selector)
$\vdots$	$\vdots$
$\_\ .sel_n : \texttt{tuple} \rightarrow elem_n$	(selector)

- For enumeration, we have only the symbols as constructors:

$sym_1 : \rightarrow \texttt{enum}$	(symbol 1)
$\vdots$	$\vdots$
$sym_n : \rightarrow \texttt{enum}$	(symbol n)

- For lists, we have the following operations:

emptylist : → list	(list constructor)
in _ _: *elem*×list → bool	(element of?)
empty _ _: list → bool	(list empty?)
pos _ _: *elem*×list → nat	(position of *elem*)
_ [_] : list×nat → *elem*	(selection)
insert _ _: *elem*×list → list	(insert element into first list position)
remove _ _: *elem*×list → list	(remove elements(!) from list)
insertlast _ _: *elem*×list → list	(insert element into list, last position)
removelast _ _: *elem*×list → list	(remove element from last position)
insertfirst _ _: *elem*×list → list	(like insert)
removefirst _ _: *elem*×list → list	(remove element at first position)
delete _ _: nat×list → list	(delete i-th element)
length _: list → nat	(length)
card _: list → nat	(count without duplicates)

Index