TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

INFORMATIK-BERICHTE

91–04

Object-Oriented Specification of Information Systems: The TROLL Language

(Version 0.01)

Ralf Jungclaus

Gunter Saake

Thorsten Hartmann

Cristina Sernadas

Abt. Datenbanken
Techn. Universität Braunschweig
Postfach 3329
W–3300 Braunschweig, Germany

Braunschweig

Dezember 1991

Object-Oriented Specification of Information Systems: The TROLL Language [§]

(Version 0.01) [†]

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

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

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

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

[†]This is the first completely defined version of the language – any comments, suggestions and error messages are welcome and greatly appreciated. They should be sent to the first author (email: jungclau@idb.cs.tu-bs.de).

Abstract

In this report we present the language TROLL. It is a language particularly suited to be used in the early stages of information system design where the problem domain or Universe of Discourse must be described. In TROLL, the description of the static and dynamic aspects of entities is integrated in object descriptions. TROLL is based on sublanguages for data terms, for first-order and temporal assertions and for processes. These sublanguages are the tools to describe static properties, behavior and evolution over time of objects.

Object descriptions can be related in many ways. The abstraction mechanisms provided by TROLL are classification, specialization, generalization, roles, and aggregation. Abstraction mechanisms together with the basic structuring in objects help in organizing the system design.

In order to support the composition of system descriptions from component description TROLL provides language features to state interactions and dependencies between components when put into the global context of the system. Furthermore, interfaces that can be defined on top of object descriptions help in modularizing the system design and make cooperative and distributed design possible

All language features of TROLL are explained in detail. We define a formal syntax of TROLL and give some ideas towards the semantics of TROLL language constructs.

Acknowledgements

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

The colleagues in Braunschweig have been of great help, too. Special thanks are due to: Martin Gogolla (who has asked critical questions and gave valuable remarks), Christine Müller, Perdita Löhr-Richter, Stefan Conrad, and Grit Denker (for discussions on TROLL).

Thanks are due to the students working (or having worked) with us, too: Scarlet Schwiderski, Axel Stein, Rüdiger Stöcker, and Michael Thulke.

Outside IS-CORE and Braunschweig, we received helpful comments from Andreas Heuer, Gerti Kappel, Ralf Kutsche, and Philippe Du Bois.

Last but not least we appreciate the comments of numerous people after presentations of the language.

Contents

1 Introduction

The language TROLL we describe in this report is a language especially designed for being used in the early stages of information system design. A basic feature of TROLL is that the description of static and dynamic features is integrated in object descriptions. These object descriptions are the basic building blocks of system specifications.

TROLL has been developed based on the language OBLOG presented in [SSE87, CSS89]. A diagrammatic version of this language is described in [SSG+91]. Previous versions of TROLL have been described in [JSS91, JSH91].

1.1 Information Systems

Information systems represent the relevant aspects of some portion of the real world (called the *Universe of Discourse* (UoD) in the sequel) [Ken78]. The UoD sometimes is called the *problem domain* elsewhere [RBP+90]. In the UoD we find the aspects that are to be computerized as well as the environment of these aspects. Aspects in general are *abstractions* of real-world phenomena. (Please note that by real-world phenomena we also mean historical or planned or abstract phenomena which are not really "real" [Ken78]). An information system becomes part of the UoD since it is designed to represent parts of the UoD. It is capable of storing, processing and producing information about the UoD.

Information must be understood as covering both static and dynamic aspects. Traditionally, with the notion of information in information systems we used to denote values that represent *properties* of certain real-world situations or entities. But the notion of information must be applied also to *behavior*, i.e. to the ways properties change.

The stored information changes over time according to interactions with the environment. Information systems ideally run forever once having been started – thus, they do not describe an input-output function like many programs do but are systems with a *state* that changes over time due to interactions with the environment. Such systems are called *reactive systems* [Pnu77, Pnu86].

Recent trends towards distributed information systems (which may even be spanning continents) and federated information systems (without central control) make the need for modularization and communication between bits of information significant.

1.2 Information Systems Design

Information systems are becoming more and more complex. The domains represented by them tend to become larger and larger and more and more complex (think of Computer Integrated Manufacturing, Banking and Stock Exchange Systems, or Decision Support Systems).

Today, one can often observe that already in the early stages of information systems design implementation issues are addressed. It is well known, however, that design flaws in the analysis phase that are revealed during implementation are very costly to fix. Moreover, addressing of implementation issues too early restricts choices that could be made in later design stages – the information system may not be of the quality expected by the user.

Unfortunately, even formal approaches do not prevent system developers from making mistakes in the design that are due to misunderstandings between them and the users. Thus, it is essential to focus solely on *conceptual* issues in the early phases of design. That is, developers should think in terms of the application rather than in terms of the implementation.

Since system developers in general are not specialists in the application domain, the process of collecting knowledge about the concepts in the UoD should be interactive, i.e. the developer must work with the user. The result of this process is a representation of the relevant static and dynamic aspects in the UoD. In this phase, it is only relevant to know *which* aspects are relevant and *what* are their properties. Thus a modeling approach should support a *declarative* description.

A model of the UoD should be *formal* in the sense that mathematical structures are used to represent real-world aspects. Only then it is possible to formally assess a model [CHJ86, Win90]. Thus, we need *formal specifications* in a formal language that denote such models as early as possible in the design process. A desirable feature of specification languages is a logical representation with a *proof theory* for reasoning about specifications based on a suitable logical framework. A proof theory allows to *derive knowledge* from specifications in order to support assessment of solutions and to formally verify implementations against specifications.

On the other hand, specifications as descriptions of formal models should also be understandable by application domain specialists. Thus, languages to be used for UoD modeling should provide a wide variety of constructs that can be used to describe real-world aspects as natural as possible.

1.3 Object-Oriented Conceptual Modeling

There is a long research tradition in the fields of *semantic data modeling* and *knowledge representation* on describing real-world aspects. The process of modeling the UoD-concepts is called *conceptual modeling*.

Semantic data modeling came up when several researchers noticed that "pure" data models were not suitable to be used in the design of database schemas (see e.g. [Ken78]).

The data structures in these "pure" models are relatively close to the ones used for the physical representation in computers. When designing a database schema, database designers should think of data as descriptions of concepts in the problem domain – a higher level of abstraction is needed for that task. One of the first proposals was the Entity-Relationship Model proposed in [Che76] based on the primitive ontological notions of entity and relationship. The first semantic models, however, did not provide for a variety of of abstraction mechanisms [SS77]. An overview of the mechanisms provided by modern semantic data models can be found in [BMS84, UD86, HK87, PM88].

Knowledge representation traditionally focused on representing "knowledge" about an application domain (like facts, rules, strategies etc.). In [McC58] the use of *logical formulas* to represent knowledge about situations and actions was taken for granted already. Other approaches in this area were semantic networks [RM75]. Knowledge is represented in semantic networks as a graph formed of data items connected by edges. An important contribution of semantic networks is the natural modeling of *is-a-*, *is-part-of-*, and *is-instance-of*-relationships. The major drawback of semantic networks is that they do not distinguish between schema and data. Another prominent approach in knowledge representation is the *frame-based* approach [Min75]. A frame represents a class of entities (like am abstract entity type in semantic data models). A frame consists of of properties and slots. Slots represent properties of which the value changes over time. In frame models we typically find concepts of inheritance along *is-a* relationships between frames and overriding.

A major drawback of the approaches described above is the lack of constructs to represent behavioral or dynamic aspects. Only very few approaches address the problem explicitly, among them TAXIS [MBW80] and RML [GBM86]. In these proposals we are able to specify *scripts* that represent activities on entities.

Many approaches in other research fields have addressed the modeling of system dynamics, especially approaches towards the modeling of *concurrency*. The central concept in many of these approaches is the concept of a *process*. The idea is that a system is composed from sequentially working components that operate concurrently and are synchronized by synchronous one-way message-passing. The most popular approaches in this field are CCS [Mil80] and CSP [Hoa85]. In general, the field of *process algebras* is a vivid research area [Mil90]. In Petri nets, the central issue is concurrency: actions and states are distributed and local to the components of the system and are thus independent. A Petri net is a bipartite graph of states and actions. A third approach towards describing the behavior of reactive systems is *temporal logic* [Kro87, Eme90]. Here, a logical approach to specifying the change of validity of propositions about the state of a reactive system is presented.

However, these approaches do not take into account the static or structural aspects. Thus, very often one model is chosen to represent the structural aspects of the UoD and another model is chosen to represent the behavioral issues. Usually, both models do not fit together very well. Structures manipulated in the behavioral model cannot be found in the static model (because the behavioral model is not a integrated) and the behavior or evolvement of structures or entities cannot be found in the behavioral model (because it is not integrated).

Object-oriented approaches like the one described in [RBP$^+$90, Boo90] aim at overcoming these problems. Object-oriented concepts have their roots in object-orient programming [Weg90]. Here, programs are regarded to be collections of interacting objects that encapsulate an internal state and offer a functional interface consisting of so-called *methods* to the outside. Recently, object-oriented concepts have also taken the step into the design of software systems [Mey88] and even into domain analysis [CY89, Boo90, RBP$^+$90]. The approaches to object-oriented analysis, however, do not fully integrate the description of static and dynamic aspects – in the approaches a mixture of (mostly informal) entity-based concepts to represent structural aspects and data-flow diagrams to represent the dynamics are used which have to be combined in some way to result in a complete description of structure and behavior.

Object-oriented conceptual modeling in our view should be integrating the advantages of work on algebraic specification of data types [EM85, EGL89] and databases [Ehr86, EDG86], process specification [Hoa85, Mil89], the specification of reactive systems [Pnu86, Eme90], conceptual modeling [Che76, MBW80, BMS84, Bor85, HG88, EGH$^+$90] and knowledge representation [BM86, ST89, MB89]. In a nutshell, object-oriented conceptual modeling approaches organize system specifications as a collection of discrete object descriptions that incorporate both the description of static local aspects and dynamic local aspects of entities (or objects). Object descriptions are thus encapsulated units of structure and behavior description. An object instance has an internal state that can be observed and changed exclusively through an object *interface*. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e. *methods*), object-oriented databases put emphasis on the observable structure of objects (through *attributes*). We propose to support both views in an equal manner, i.e. the structural properties of objects may be observed through attributes and the behavior of objects may be manipulated through *events* which are abstractions of state changing operations.

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

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

Objects can be composed from other objects (*aggregation*). Aggregation of objects imposes a *part-of* relation on a collection of object-descriptions. This kind of inheritance is known from semantic data models where it is used to model objects that appear in several roles in an application. It is also useful to distinguish *disjoint* from *non-disjoint* complex objects. In non-disjoint complex objects, components may be shared with other complex objects, whereas in non-disjoint complex objects only belong to one complex object. This implies that the existence of components of disjoint complex objects is strongly dependent on the existence of the aggregation. For non-disjoint complex objects, components may exist independently from aggregations. Non-disjoint complex objects can be defined based on static properties (*static aggregation*) or operations to alter the composition (*dynamic aggregation*).

Object descriptions may also be embedded in a *specialization* hierarchy. Usually, specialization implies reuse of specification code and allows to treat instances both as instances of the base class and the specialization class. A related concept is *generalization* that allows to treat conceptually different instances uniformly as instances of the generalization class.

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

Last, but not least system descriptions should be organized in several levels of modularity. That is, we may want to compose a system not only from objects but from subsystems. Therefore, we need to have a means to *relate* objects from several subsystems. Moreover, it is sometimes not very natural to bury relationships in objects instead of making them explicit in a system specification. Addititonally, the hiding of relationships in object descriptions hinders reuse. Thus, a framework for object-oriented conceptual modeling should support *relationships* between objects [RBP$^+$90]. Relationships between dynamic objects are, however, not as simple as relationships in e.g. the Entity-Relationship model – here, we must be able to specify *interactions* between objects and *dependencies* between objects.

1.4 A Concept of Object

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

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

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

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

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

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

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

A formalization of these ideas can be found in [JSS91, SJ91, JSH91]. For recent developments towards a more sophisticated semantic domain for object systems see [ES91].

1.5 Running Example

Throughout the rest of this report, we use fragments of the following example taken from commercial applications. Consider a bank. A bank maintains a number of accounts for customers. It also owns a number of automatic teller machines (ATMs) that are operated remotely. Accounts have the usual properties such as a current balance, a credit limit and a holder. The balance is always the absolute balance – each account has a special indicator for being in the red or not. Accounts may not be overdrawn (i.e. below credit limit). Accounts accept update requests that are completed succesfully if possible. Associated with a checking account is a number of cash cards that can be used to withdraw money at an ATM. An ATM accepts a cash card, communicates with the user and the bank to carry out the transaction and dispenses cash if the transaction was successful and the ATM is not empty. The bank coordinates the card verification requests and the bank transactions issued concurrently from the ATMs.

Occasionally, other well known concepts like books in a library context are used to explain certain language features. This report explains all features of TROLL and it is sometimes a bit strange to press everything into one example.

1.6 Organization of this Report

In Chapter 2, we will introduce the sublanguages used by TROLL. These sublanguages are based on well known languages used in algebraic specifications, predicate logic, temporal logic, and process specification. We will in detail explain the the language constructs (in terms of a formal syntax) and the semantics of them over suitable mathematical structures. The sublanguages are very important since they are the tools to specify certain properties (either static or dynamic) of system components.

In the next chapters we will introduce the higher-level language features of TROLL. We will try to explain the basic ideas behind them and give some motivation why they are part of the language. For every construct, we will give the formal syntax and some ideas of the semantics.

In Chapter 3, we will explain how templates are specified. All sections of a simple template specification are explained in detail using examples.

In Chapter 4, the issue of classes and class types is explained. Classification is one of the abstraction mechanisms in TROLL.

Chapter 5 then is about composing objects from other objects. In the TROLL-approach, such objects are called complex objects. TROLL supports different kinds of object aggregation explained in this chapter.

Chapter 6 explains abstractions that aim at modeling different aspects of entities. An entity can be looked at from different angles but it is still the same entity. The abstractions we explain in this chapter are specialization, generalization, and roles.

In Chapter 7 we introduce language features that support the assembly of components of systems to specify systems. On the one hand, TROLL provides a concept of relationship to specify the interactions and dependencies between system components, on the other hand TROLL contains language features to specify additional restricted interfaces to object specifications to support information hiding.

In Chapter 8 we asses the current version of the language and provide an outlook on topics to be addressed concerning TROLL in the future.

In the bibliography we not only collect the references from the text but also give additional references to work which we feel is related.

2 Sublanguages

Specifications in the language TROLL are based on a number of sublanguages, i.e. certain sections of a TROLL-specification are sentences of a sublanguage. The sublanguages of TROLL are

- *data terms* for data values and expressions (involving signatures of constant symbols and operation symbols as well as terms over such signatures);
- *predicate logic* for (static) constraints;
- *temporal logic* for dynamic constraints and preconditions and
- a language for *process specification* to specify fragments of life cycles.

In the following subsections, we explain these sublanguages in more detail. Please note that the concepts are well known – we only made some slight syntactic modifications from time to time to increase readability and usability.

2.1 Data Terms

Underlying a TROLL-specification we assume a universe of *data types* to be given. Data types are the resources for *values* that describe the state of objects and can be exchanged between objects. Furthermore, objects are identified by values. Please note that we distinguish objects and values – values are immutable, they have no current state and they identify themselves.

The specification of data types has been a vivid field of research for many years. In the school of *abstract data types*, the elements of data types are specified through their operations. In general, we have to distinguish between the actual values and operations and the symbols and terms that denote these values and operations. A data type consists of a set of values and a collection of operations on them. A set of data types (or models) interprets a specification. Data types as known from programming languages usually provide symbols for a great number of values. Expressions then compute values from other values. In the abstract data type field, values are usually denoted by *terms* involving operations on other terms.

In TROLL, we do not define a fixed sublanguage for data types and expressions but assume a framework for the algebraic specification of data types to be given. The reason for this is the fact that it should be possible for users to define their own data types. Thus, we just state which components of a data type description we need and which kind of terms over these components can be used in TROLL specifications.

2.1.1 Signatures

The *signature* of a specification consists of the *symbols* used in that specification. In the following, we will refer to the set of values of a particular data type as a *sort*. A signature declares the symbols that are used to define data types. For convenience, we distinguish the function symbols that yield a result of sort `bool` from the other functions and refer to them as *predicates*.

Definition 2.1.1 (*signature*)
A *signature* $\Sigma = (S, \Omega, \Pi)$ consists of

1. a set $S = \{s, r, \ldots\}$ of *sorts*,
2. a set $\Omega = \{f\colon s_1 \times \cdots \times s_n \to s_0, g\colon r_1 \times \cdots \times r_m \to r_0, \ldots\}$ of *function symbols*, and
3. a set $\Pi = \{p\colon s_1 \times \cdots \times s_n, q\colon r_1 \times \cdots \times r_n, \ldots\}$ of *predicate symbols*. For each sort $s \in S$, a binary equality predicate $=_s\colon s \times s$ is included in Π.

Constants are expressed by null-ary functions, e.g. `zero`: $\to$ `integer`. ■

The symbols of a signature are interpreted by elements of an *algebra*. Please note that the priorities of the operator symbols affect only the denotation of terms, they do not affect interpretations of symbols.

We furthermore assume that each operation symbol and each predicate symbol is defined either as an infix operator/predicate or a prefix operator/predicate. This can be done in an OBJ-like fashion [GKK$^+$87].

Definition 2.1.2 (*interpretation*)
An interpretation of a signature is an algebra $A(\Sigma) = (A(S), A(\Omega), A(\Pi))$, where

1. $A(S)$ is a family of carrier sets for each $s \in S$.
2. $A(\Omega)$ is a family of functions $A(f)\colon A(s_1) \times \cdots \times A(s_n) \to A(s_0)$ for each function symbol $f\colon s_1 \times \cdots \times s_n \to s_0 \ \in \Omega$, and
3. $A(\Pi)$ is a family of relations $A(p) \subseteq A(s_1) \times \cdots \times A(s_n)$ for each predicate symbol $p\colon s_1 \times \cdots \times s_n \ \in \Pi$

■

The set of algebras for a signature are the possible interpretations of the symbols of signatures by values and functions over values.

Over signatures, we may define *variables* that take values from a particular sort.

Definition 2.1.3 (*variables*)
For a given signature $\Sigma = (S, \Omega, \Pi)$ we have a collection of s-sorted *variables* $X = \{x: s_1, y: s_2, \ldots\}$ with $s_i \in S$. The variables of a sort s are denoted by X_s. ■

We allow also for *generic* or *parameterized* data types. Basically, these data types have as formal parameters data sorts in their specification and are *instantiated* with a particular sort when they are used. An example for a generic type is the type **set**(`<a_sort>`) which describe sets of elements of sort `a_sort`. The operations that work on generic types do not manipulate the elements but the instances of the generic type. For a deep explanation of generic types please refer to [EW86, VHL89].

For TROLL, we assume a number of data types to be predefined. The sorts are

- `bool` for the boolean values;
- `nat` for the natural numbers, i.e. positive integers and zero;
- `integer` for the integer numbers;
- `real` for rational numbers;
- `char` for characters and
- `string` for sequences of characters.

The operations defined over these sorts are listed and described in Appendix B.

We also assume the following generic types to be predefined:

- **set**(`a_type`) for sets;
- **list**(`a_type`) for lists, and
- **tuple**(`list_of_types`) for tuples or records of elements.

The operations defined on these generic types are also listed and described in Appendix B.

2.1.2 Terms

From the symbols defined by signatures and variable declarations, we may construct *terms*. Terms denote a value of a particular sort and thus are of the same sort than the value they denote. Terms are constructed from variables, function symbols (please note that constants are nullary function symbols) and predicate symbols.

Let us now formally define terms, boolean terms and term evaluation before we present the syntax by means of a grammar.

Definition 2.1.4 (*terms*)
Let a signature Σ and a set of variables X be given. Each term t in the set of *terms* $T_\Sigma(X)$ has a corresponding sort s and is notated as $t: s$.

1. Each variable $x\colon s$ is a term of sort s.
2. If $t_1\colon s_1, \ldots, t_n\colon s_n$ are terms of sort $s_1, \ldots, s_n$, and $f\colon s_1, \ldots, s_n \to s_0$ is a function symbol in Σ, then $f(t_1\colon s_1, \ldots, t_n\colon s_n)\colon s_0$ is a term of sort s_0.
3. Nothing else is a term in $T_\Sigma(X)$.

■

In the TROLL syntax for terms, we can also have infix-operator symbols. Parentheses can be used wherever wanted.

Boolean terms are defined by the predicate symbols from Π.

Definition 2.1.5 (*boolean terms*)
The boolean terms $BT_\Sigma(X)$ are defined as

1. Variables of sort `bool` are boolean terms in $BT_\Sigma(X)$.
2. If $t_1\colon s_1, \ldots, t_n\colon s_n$ are terms of sort $s_1, \ldots, s_n$, and $p\colon s_1, \ldots, s_n$ is a predicate symbol in Π, then $p(t_1\colon s_1, \ldots, t_n\colon s_n)$ is a boolean term in $BT_\Sigma(X)$.
3. Nothing else is a boolean term in $BT_\Sigma(X)$.

■

Please note that in our approach, we try to integrate logic and data types in that we refer to operators that yield a value of sort `bool` as predicates. Usually, in the syntax of first order logic, predicates applied to data terms are referred to as atomic formulae (see below).

To define the evaluation of terms w.r.t. a given interpretation $A(\Sigma)$, we have to define the *substitution* of a variable $x\colon s$ by a value $\vartheta(x\colon s) \in A(s)$.

Definition 2.1.6 (*substitution*)
A *substitution* ϑ of variables X w.r.t. an interpretation $A(\Sigma)$ is a family of mappings

$$\vartheta\colon X_s \to A(s)$$

substituting each variable $x\colon s \in S$ by a value from the corresponding sort carrier $A(s)$. The set of all substitutions for a given $A(\Sigma)$ and X is denoted as $\Theta(A(\Sigma), X)$. ■

The denotation of a term is evaluated recursively:

Definition 2.1.7 (*term evaluation*)
For a given interpretation $A(\Sigma)$ and a given substitution $\vartheta \in \Theta(A(\Sigma), X)$, the *valuation* function of a term is defined by

$$(A(\Sigma), \vartheta)[\![x]\!] = \vartheta(x) \qquad \text{for } x \in X$$

$$(A(\Sigma), \vartheta)[\![f(t_1, \ldots, t_n)]\!] = A(f)((A(\Sigma), \vartheta)[\![t_1]\!], \ldots, (A(\Sigma), \vartheta)[\![t_n]\!])$$

This recursive definition resembles the well-known term evaluation strategy. ■

Finally, let us give the syntax for terms in the TROLL-language. Note that terms denoting data values are referred to as *data terms* in the sequel Here and in the following syntax descriptions, we use a BNF-like notation. Note that we may refer to the values of *attributes* of objects. For further details please have a look at chapter 3. Attribute selectors are treated as variables semantically.

<*data_term*> ::= <*data term*> <*inf_op*> <*primary*>
| <*primary*>

<*primary*> ::= '('<*data_term*>')'
| <*op_id*>'('<*data_term_list*>')'
| <*variable_id*>
| <*constant_symbol*>
| <*attribute_sel*>

2.2 First Order Logic Formulae

Based on an underlying universe of data types, we now define first order logic formulae. We consider *many-sorted* logic, i.e. the terms and variables are sorted as described in the previous section.

Logical formulae are build over signatures as defined in the previous section. Formulae are interpreted over an interpretation structure for the corresponding signature.

The basic building blocks for formulae are boolean terms as defined in the previous section. Let us now define the syntax of first order predicate logic $\mathcal{L}_\Sigma(X)$:

Definition 2.2.1 (*syntax of predicate logic*)
The *formulae* in $\mathcal{L}_\Sigma(X)$ are defined as

1. The symbols **true** and **false** are formulae in $\mathcal{L}_\Sigma(X)$.

2. Each boolean term in $BT_\Sigma(X)$ is a formula of $\mathcal{L}_\Sigma(X)$.

3. Let φ and ψ be formulae of $\mathcal{L}_\Sigma(X)$. Then $(\varphi \wedge \psi)$,$(\varphi \vee \psi)$,$(\neg\varphi)$,$(\varphi \Rightarrow \psi)$, $(\exists x\colon s)\varphi$ and $(\forall x\colon s)\varphi$ are formulae in $\mathcal{L}_\Sigma(X)$.

4. Nothing else is a formulae in $\mathcal{L}_\Sigma(X)$.

■

Let us now give the formal definition of the semantics of first order formulae over an interpretation structure for a given signature:

Definition 2.2.2 (*semantics of $\mathcal{L}_\Sigma(X)$*)
The semantics of a formula φ w.r.t. an interpretation $A(\Sigma)$ and a substitution ϑ is given by the following rules. We notate the validity of φ by $(A(\Sigma),\vartheta) \models \varphi$ for a given $A(\Sigma)$ and ϑ.

$$(A(\Sigma),\vartheta) \models \mathbf{true}$$

$$(A(\Sigma),\vartheta) \not\models \mathbf{false}$$

$$(A(\Sigma),\vartheta) \models p(t_1,\ldots,t_n) \quad \text{iff} \quad ((A(\Sigma),\vartheta)[\![t_1]\!],\ldots,(A(\Sigma),\vartheta)[\![t_n]\!]) \,\epsilon\, A(p)$$

$$(A(\Sigma),\vartheta) \models \neg\varphi \quad \text{iff} \quad \text{not } (A(\Sigma),\vartheta) \models \varphi$$

$$(A(\Sigma),\vartheta) \models \varphi \wedge \psi \quad \text{iff} \quad (A(\Sigma),\vartheta) \models \varphi \text{ and } (A(\Sigma),\vartheta) \models \psi$$

$$(A(\Sigma),\vartheta) \models \varphi \vee \psi \quad \text{iff} \quad (A(\Sigma),\vartheta) \models \neg(\neg\varphi \wedge \neg\psi)$$

$$(A(\Sigma),\vartheta) \models \varphi \Rightarrow \psi \quad \text{iff} \quad (A(\Sigma),\vartheta) \models \neg\varphi \vee \psi$$

$$(A(\Sigma),\vartheta) \models \forall(x{:}\,s)\varphi \quad \text{iff} \quad (A(\Sigma),\vartheta') \models \varphi$$

for all $\vartheta' \,\epsilon\, \Theta(A(\Sigma),X)$ where ϑ' differs from ϑ only in the value of x.

$$(A(\Sigma),\vartheta) \models \exists(x{:}\,s)\varphi \quad \text{iff} \quad (A(\Sigma),\vartheta) \models \neg\forall(x{:}\,s)\neg\varphi$$

■

The TROLL-sublanguage syntax for first-order formulae is the following:

<*formula*> ::= <*formula*> '**<==>**' <*rel_form*>
| <*rel_form*>

<*rel_form*> ::= <*rel_form*> '**==>**' <*s_form*>
| <*s_form*>

<*s_form*> ::= <*s_form*> **or** <*m_form*>
| <*m_form*>

<*m_form*> ::= <*m_form*> **and** <*a_form*>
| <*a_form*>

<*a_form*> ::= '('<*formula*>')'
| **not** <*formula*>
| **forall** '('<*decl_list*>')' '(' <*formula*> ')'
| **exists** '('<*decl_list*>')' '(' <*formula*>')'
| <*data_term*>
| **true** | **false**

2.3 Temporal Logic Formulae

Temporal logic is a special logic that provides a formal system for *qualitatively* describing and reasoning about how the truth values of assertions change over time. Thus, temporal logic is well suited to describe *behavior*. In particular, temporal logic is useful for describing and reasoning about the behavior of nonterminating or continuously operating concurrent systems [Pnu86, Eme90].

In TROLL, temporal logic formulae are used to specify the behavior of objects. In this section, we want to introduce the particular dialect of temporal logic we chose and want to give an intuitional introduction to the underlying semantical structures.

Underlying a system specification is a notion of *time*. By notion of time we do not mean a notion of *explicit* time but a notion about the *order* of arbitrary points in time that are not necessarily related to any concrete time. We only consider *linear time*, i.e. the structure of time is a totally ordered set of points in time $(S, <)$. This structure is isomorphic to the natural numbers with their usual ordering. Thus, time is discrete, there exists an initial point in time and there are infinitely many future states.

We distinguish between assertions that are to be valid in the future and referring to assertions that were valid in the past. Thus, we end up in having two related temporal frameworks of the past and the future.

2.3.1 State Sequences

In this section, we want to introduce state sequences and sequences of state transitions. This is what distinguishes a reactive system from a transformational one: A transformational system describes the mapping of an initial state to a final state – this view is not sufficient for reactive systems. We have to deal with states and state transitions.

Definition 2.3.1 (*state*)
A *state* σ is defined as an algebra $A(\Sigma) = (A(S), A(\Omega), A(\Pi))$ over a signature Σ. ■

Definition 2.3.2 (*state sequence*)
A *state sequence* is an indexed sequence of states σ_i over a given signature Σ. Finite state sequences are notated as $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ and infinite state sequences are notated as $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle$. The carrier sets of state algebras do not change. The length of a state sequence is notated as $\mid \hat{\sigma} \mid$. ■

Let us now define state transitions. State transitions are sets of atomic events that describe transitions between states. An atomic event is the least unit that can change the state of a system component [JSS91, HJ91].

Definition 2.3.3 (*state transitions*)
Let $\sigma_1\, \sigma_2 \;\epsilon\; S$ be states and E be a set of atomic events. A subset $\kappa \subseteq E$ of simultaneously occurring events in E is a *state transition* between σ_1 and σ_2 if the effects of κ transform σ_1 into σ_2. A state transition is notated as $\sigma_1 \xrightarrow{\kappa} \sigma_2$. ■

On the sort of events, a predicate **after** is defined. It is valid in exactly those states that are reached by a transition with an atomic event $e \epsilon E$. This is formalized as follows:

Definition 2.3.4 (*after-predicate*)
Let e be an event, $\hat{\sigma}$ a state sequence and σ_i the current state. Then

$$(\hat{\sigma}, i) \models \textbf{after}(e) \quad \text{iff} \quad \sigma_{i-1} \xrightarrow{\kappa} \sigma_i \ \wedge\ e \,\epsilon\, \kappa$$

■

Since the domains do not change over time, a *substitution* of variables in the sense of Definition 2.1.6 remains constant for all states in a sequences.

2.3.2 Future Tense Temporal Logic

Our aim in introducing Future Tense Temporal Logic (FTTL) is to state properties on the future object evolution. For stating properties on the *current* state of an object, we use formulae of $\mathcal{L}_\Sigma(X)$ being evaluated w.r.t. the current values of attributes. For stating propositions on the future evolution of object properties, we need temporal quantifiers like 'always in the future', 'sometime in the future' or 'something will become true before something else will happen' etc. A special operator is the nexttime operator referring to the next state.

The syntax of such a logic is defined as follows.

Definition 2.3.5 (*syntax of future tense temporal logic*)
The formulae in the future tense temporal logic $\mathcal{F}L_\Sigma(X)$ are defined as follows:

1. Each first order logic formula $\varphi \,\epsilon\, \mathcal{L}_\Sigma(X)$ is a formula of $\mathcal{F}L_\Sigma(X)$.

2. If φ and ψ are formulae of $\mathcal{F}L_\Sigma(X)$ and x a variable of sort $s \,\epsilon\, S$, then also $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$, $\neg\varphi$, $(\exists x\colon s)\varphi$ and $(\forall x\colon s)\varphi$ are formulae of $\mathcal{F}L_\Sigma(X)$.

3. If φ and ψ are formulae of $\mathcal{F}L_\Sigma(X)$, then also (**next** φ), (**existsnext** φ), (**alwaysf** φ), (**sometimef** φ), (φ **until** ψ) and (φ **before** ψ) are formulae of $\mathcal{F}L_\Sigma(X)$.

■

The semantics of formulae in $\mathcal{F}L_\Sigma(X)$ is defined relative to a *reference moment*, the current state σ_i. Please note that the semantics of the connectives $\vee$, $\wedge$, $\Rightarrow$,and $\Leftrightarrow$ are as for first order logic.

Definition 2.3.6 (*semantics of future tense temporal logic*)
The semantics of $\mathcal{F}L_\Sigma(X)$ formulae is given by the following definitions of validity in a state sequence $\hat{\sigma}$ at a reference moment i under a given global substitution ϑ (this validity is denoted by $(\hat{\sigma}, i, \vartheta) \models \varphi$ for a formula φ). The evaluation of first order logic formulae under a given substitution is defined as usual.

$(\hat{\sigma}, i, \vartheta) \models \varphi'$	iff	$(\sigma_i, \vartheta) \models \varphi'$ for first-order logic formulae $\varphi' \in \mathcal{L}_\Sigma(X)$.	(1)
$(\hat{\sigma}, i, \vartheta) \models \neg\varphi$	iff	not $(\hat{\sigma}, i, \vartheta) \models \varphi$.	(2)
$(\hat{\sigma}, i, \vartheta) \models \varphi \wedge \psi$	iff	$(\hat{\sigma}, i, \vartheta) \models \varphi$ and $(\hat{\sigma}, i, \vartheta) \models \psi$.	(3)
$(\hat{\sigma}, i, \vartheta) \models (\forall x)\varphi$	iff	for all substitution sequences ϑ' differing from ϑ only in the value of x, the following holds: $(\hat{\sigma}, i, \vartheta') \models \varphi$.	(4)
$(\hat{\sigma}, i, \vartheta) \models$ **existsnext** φ	iff	$\lvert\hat{\sigma}\rvert > i$ and $(\hat{\sigma}, i+1, \vartheta) \models \varphi$.	(5)
$(\hat{\sigma}, i, \vartheta) \models$ **next** φ	iff	$\lvert\hat{\sigma}\rvert > i$ implies $(\hat{\sigma}, i+1, \vartheta) \models \varphi$.	(6)
$(\hat{\sigma}, i, \vartheta) \models$ **alwaysf** φ	iff	$(\hat{\sigma}, j, \vartheta) \models \varphi$ for all $i \leq j < \lvert\hat{\sigma}\rvert$.	(7)
$(\hat{\sigma}, i, \vartheta) \models$ **sometimef** φ	iff	there exists j, $i \leq j < \lvert\hat{\sigma}\rvert$, such that $(\hat{\sigma}, j, \vartheta) \models \varphi$.	(8)
$(\hat{\sigma}, i, \vartheta) \models \varphi$**until** ψ	iff	$(\hat{\sigma}, j, \vartheta) \models \varphi$ for all $i \leq j < l$ where l is defined as $l = \min\{\{k \mid (\hat{\sigma}_k, i, \vartheta_k) \models \psi\} \cup \{\lvert\hat{\sigma}\rvert\}\}$.	(9)
$(\hat{\sigma}, i, \vartheta) \models \varphi$ **before** ψ	iff	there exists j, $i \leq j < l$, such that $(\hat{\sigma}, j, \vartheta) \models \varphi$ where l is defined as in (9).	(10)

■

We will not give a detailed description of properties of $\mathcal{F}L_\Sigma(X)$. The interested reader can find more about this logic in [SL89].

Let us now give the formal syntax definition of FTTL-formulae in TROLL. We decided not to introduce the **existsnext** temporal quantor since **existsnext** φ = $\neg$**next**$\neg$ φ always holds.

<*fut_formula*> ::= <*fut_formula*> '**<==>**' <*fut_rel_form*>
| <*fut_rel_form*>

<*fut_rel_form*> ::= <*fut_rel_form*> '**==>**' <*fut_s_form*>
| <*fut_s_form*>

<*fut_s_form*> ::= <*fut_s_form*> **or** <*fut_m_form*>
| <*fut_m_form*>

<*fut_m_form*> ::= <*fut_m_form*> **and** <*fut_a_form*>
| <*fut_m_form*> **until** <*fut_a_form*>
| <*fut_m_form*> **before** <*fut_a_form*>
| <*fut_a_form*>

<*fut_a_form*> ::= **not** <*fut_formula*>
| **forall** ‘(’<*decl_list*>‘)’ ‘(’<*fut_formula*>‘)’
| **exists** ‘(’<*decl_list*>‘)’ ‘(’<*fut_formula*>‘)’
| **sometimef** <*fut_formula*>
| **alwaysf** <*fut_formula*>
| **next** <*fut_formula*>
| ‘(’<*fut_formula*>‘)’
| <*formula*>
| **true** | **false**

2.3.3 Past Tense Temporal Logic

With Past Tense Temporal Logic (PTTL), we may refer to properties that were valid in the history of an object. Such properties may not be valid anymore or they may have been valid long ago. When referring to the past, we are only interested in a finite prefix of a state sequence starting with the initial state and ending with the current state. The definition of the syntax and the semantics of PTTL formulae is defined as follows.

Definition 2.3.7 (*syntax of past tense temporal logic*)
The formulae in the past tense temporal logic $\mathcal{P}L_\Sigma(X)$ are defined as follows:

1. Each first order logic formula $\varphi \in \mathcal{L}_\Sigma(X)$ is a formula of $\mathcal{P}L_\Sigma(X)$.

2. If φ and ψ are formulae of $\mathcal{P}L_\Sigma(X)$ and x a variable of sort $s \in S$, then also $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$, $\neg\varphi$, $(\exists x\colon s)\varphi$ and $(\forall x\colon s)\varphi$ are formulae of $\mathcal{P}L_\Sigma(X)$.

3. If φ and ψ are formulae of $\mathcal{P}L_\Sigma(X)$, then also (**previous** φ), (**existsprevious** φ), (**always** φ), (**sometime** φ), (**always** φ **since last** ψ) and (**sometime** φ **since last** ψ) are formulae of $\mathcal{P}L_\Sigma(X)$.

■

The semantics of formulae in $\mathcal{P}L_\Sigma(X)$ is defined relative to a *reference moment*, the current state σ_i:

Definition 2.3.8 (*semantics of past tense temporal logic*)
The semantics of $\mathcal{P}L_\Sigma(X)$ formulae is given by the following definitions of validity in a state sequence $\hat{\sigma}$ at a reference moment i under a given global substitution ϑ (this validity is denoted by $(\hat{\sigma}, i, \vartheta) \models \varphi$ for a formula φ). The evaluation of first order logic formulae under a given substitution is defined as usual.

$(\hat{\sigma}, i, \vartheta) \models \varphi'$	iff	$(\sigma_i, \vartheta) \models \varphi'$ for first-order logic formulae $\varphi' \ \epsilon\ \mathcal{L}_\Sigma(X)$.	(1)
$(\hat{\sigma}, i, \vartheta) \models \neg\varphi$	iff	not $(\hat{\sigma}, i, \vartheta) \models \varphi$.	(2)
$(\hat{\sigma}, i, \vartheta) \models \varphi \wedge \psi$	iff	$(\hat{\sigma}, i, \vartheta) \models \varphi$ and $(\hat{\sigma}, i, \vartheta) \models \psi$.	(3)
$(\hat{\sigma}, i, \vartheta) \models (\forall x)\varphi$	iff	for all substitutions ϑ' ϑ differing from ϑ only in the value of x, the following holds: $(\hat{\sigma}, i, \vartheta') \models \varphi$.	(4)
$(\hat{\sigma}, i, \vartheta) \models$ **existsprevious** φ	iff	$i > 0$ and $(\hat{\sigma}, i-1, \vartheta) \models \varphi$.	(5')
$(\hat{\sigma}, i, \vartheta) \models$ **previous** φ	iff	$i = 0$ or $(\hat{\sigma}, i-1, \vartheta) \models \varphi$.	(6')
$(\hat{\sigma}, i, \vartheta) \models$ **always** φ	iff	$(\hat{\sigma}, j, \vartheta) \models \varphi$ for all $0 \leq j < i$.	(7')
$(\hat{\sigma}, i, \vartheta) \models$ **sometime** φ	iff	there exists j, $0 \leq j < i$, such that $(\hat{\sigma}, j, \vartheta) \models \varphi$.	(8')
$(\hat{\sigma}, i, \vartheta) \models$ **always** φ **since last** ψ	iff	$(\hat{\sigma}, j, \vartheta) \models \varphi$ for all $l < j \leq i$ where l is defined as $l = \max\{\{k \mid 0 \leq k \leq i \wedge (\hat{\sigma}_k, i, \vartheta_k) \models \psi\} \cup \{-1\}\}$.	(9')
$(\hat{\sigma}, i, \vartheta) \models$ **sometime** φ **since last** ψ	iff	there exists j, $l < j \leq i$, such that $(\hat{\sigma}, j, \vartheta) \models \varphi$ where l is defined as in (9').	(10')

■

Finally, we list the formal syntax description of PTTL-formulae in TROLL. Please note that the **after**-predicate defined in Definition 2.3.4 is used very frequently in PTTL formulae and therefore is introduced in the formal syntax of PTTL-formulae in TROLL as a special predicate. Furthermore, we decided not to introduce the **existsprevious** temporal quantor since $existsprevious \varphi = \neg previous \neg \varphi$ always holds.

<*past_formula*> ::= <*past_formula*> ‘<==>’ <*past_rel_form*>
| <*past_rel_form*>

<*past_rel_form*> ::= <*past_rel_form*> ‘==>’ <*past_s_form*>
| <*past_s_form*>

<*past_s_form*> ::= <*past_s_form*> **or** <*past_m_form*>
| <*past_m_form*>

<*past_m_form*> ::= <*past_m_form*> **and** <*past_a_form*>
| **always** <*past_m_form*> **since last** <*past_a_form*>
| **sometime** <*past_m_form*> **since last** <*past_a_form*>
| <*past_a_form*>

<*past_a_form*> ::= **not** <*past_formula*>
| **after**‘(’<*event_term*>‘)’
| **forall** ‘(’<*decl_list*>‘)’ ‘(’<*past_formula*>‘)’
| **exists** ‘(’<*decl_list*>‘)’ ‘(’<*past_formula*>‘)’
| **sometime** <*past_formula*>
| **always** <*past_formula*>
| **previous** <*past_formula*>
| ‘(’<*past_formula*>‘)’
| <*formula*>
| **true** | **false**

2.4 Process Specification

The basic state transitions in objects are called *events* in our approach. The intuition behind an event is an action an object performs or suffers. This idea is very similar to the motivation of e.g. CSP [Hoa85] except that CSP does not take into account any state information about the objects of which the behavior is being described. A state sequence of an object corresponds to a sequence of event *occurrences*, the transitions between the states.

In a single, non-composite object, we only have to deal with event sequences since we do not allow for concurrency inside such objects. In composite objects, we may have concurrency but this may only be described by communication.

Events denote *classes* of state transitions: each occurrence of an event is an instance of this class and is distinguished from other instances in the *time* of their occurrence.

To describe an *order* with respect to time on event occurrences we use a *process language* that resembles the CSP-formalism [Hoa85] in many respects. Basically, we can describe precedence relationships between event occurrences and choices between possible alternatives. In the following, we refer to such descriptions as *process descriptions*.

The basic building blocks of process descriptions are *event terms*. Event terms describe events. Thus, the order of event terms in a process description describes the order of corresponding event occurrences.

Before we give a formal syntax of process descriptions in the TROLL language, let us briefly give a formal treatment of process descriptions.

2.4.1 Process Descriptions

Process descriptions are based on an *event signature*.

Definition 2.4.1 (*event signature*)
An *event signature* $\mathcal{E} = (\Delta, E)$ consists of

1. a sort Δ of events, and

2. a set $E = \{e_1 \colon s_1 \times \cdots \times s_n \rightarrow \Delta; \ldots\}$ of *event generators*.

■

An event signature thus introduces the symbols for events.

Let us now define the language of processes $\mathcal{P}(V)$ over a data signature Σ and a set of variables V over Σ as defined in Definition 2.1.3.

Definition 2.4.2 (*process language*)
Let $\mathcal{E}$ be an event signature. Let Σ be a data signature and X be a set of variables over Σ. The sentences of the *process language* $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ are defined as follows:

1. **nil** is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$.
2. If $x_1 : s_1, \ldots, x_k : s_k \;\epsilon\; X$ and $e \;\epsilon\; E$ is an event generator in $\mathcal{E}$, then $e(x_1, \ldots, x_k)$ is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*event term*).
3. If P is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ and $f \;\epsilon\; \mathcal{F}L_{\Sigma}(X)$ is a PTTL formula, then $\{f\}P$ is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*guarded process*).
4. If P, Q are processes in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$, then $P \longrightarrow\; Q$ is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*sequence*).
5. IF P, Q are processes in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$, then $P \mid Q$ is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*choice*).
6. If p is a process identifier, then p is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*identification*).
7. If M is a set and P is a process definition in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$, then **foreach** $m \;\epsilon\; M$: $P(m)$ is a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ (*iteration*).

■

Note that until now we are not able to bind processes to names and thus are not able to define recursive processes yet. We need the following definition for process definitions to do that.

Definition 2.4.3 (*process definition*)
Let P be a process in $\mathcal{P}_{\mathcal{E},\Sigma}(X)$ and $V = \{v_1 : s_1, \ldots, v_k : s_k\}$ a set of variables over Σ such that $V \subseteq X$. Then $p(v_1, \ldots v_k) = P$ is a *process definition* with free variables $v_1, \ldots, v_k$.

An *event occurrence* is defined as the application of an event to a state σ_i in a state sequence $\hat{\sigma}$. Pleasenote that the same event may occur several times in a process.

Definition 2.4.4 (*event occurrence*)
Let $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ be a state sequence and e be an element of the sort of events Δ. Then an *occurrence* of e in $\hat{\sigma}$ is defined as a mapping $e(\sigma_i) = \sigma_{i+1}$. Note that in σ_i we may need to refer to the history, i.e. the sequence of event occurrences up to that moment. ■

The semantics of process definitions is described over state sequences as defined in section 2.3.2. Before we define the semantics, we give some auxiliary definitions. Firstly, we define the *event trace* of a state sequence. The event trace of a state sequence contains the events of which the occurrences were the state transitions in that particular state sequence.

Definition 2.4.5 (*event trace*)
The *event trace* $Trace(\hat{\sigma})$ of a state sequence $\hat{\sigma}$ of length n is defined as the sequence of all event occurrences in $\hat{\sigma}$:

$$Trace(\hat{\sigma}) = \langle e_0, \ldots, e_{n-1} \rangle$$

such that $\forall i, 0 \leq i \leq n-1: e_i(\sigma_i) = \sigma_{i+1}$. With $Trace^i(\hat{\sigma})$ we denote the tail of the trace $Trace(\hat{\sigma})$ that starts in e_i. The trace of a state sequence is finite. ∎

We need to have a notion of substitution for process identifiers. We do not want to give a formal definition here. A substitution of a process identifier is the process it identifies. The substitution may involve unfolding as described in [Hoa85] for recursive process definitions.

A brief note about the variables in a process is in order here. In a process description, we may want to describe events which are fixed in tre moment of their occurrence, i.e. we have event generators of which some variables may be bound to values when an event occurs. Such a phenomenon occurs when we have events that describe a communication with data flowing from the environment to the process. Thus, we may have variables in a process that are not bound by a process definition or by a guard.

Now we are able to define the semantics of a process definition over a state sequence $\hat{\sigma}$ relative to a reference moment:

Definition 2.4.6 (*semantics of process definitions*)
The semantics of a process definition $P \in \mathcal{P}_{\mathcal{E},\Sigma(X)}$ is given by the following definitions of validity in a state sequence $\hat{\sigma}$ at a reference moment i under a given global substitution ϑ (this validity is denoted by $(\hat{\sigma}, i, \vartheta) \models P$ for a process definition P.

$$\begin{array}{lll}
(\hat{\sigma}, i, \vartheta) \models \mathbf{nil} & \text{iff} & \mathbf{true} \\
(\hat{\sigma}, i, \vartheta) \models e(\vec{v}, \vec{x}) & \text{iff} & \exists \vec{x}\colon (\hat{\sigma}, i+1, \vartheta) \models \mathbf{after}(e(\vartheta(\vec{v}), \vec{x})) \\
(\hat{\sigma}, i, \vartheta) \models \{f\}P & \text{iff} & \sigma_i \models_{\mathcal{P}L_{\Sigma(X)}} f \ \wedge \ (\hat{\sigma}, i, \vartheta) \models P \\
(\hat{\sigma}, i, \vartheta) \models e(\vec{v}, \vec{x}) \rightarrow \ P & \text{iff} & (\hat{\sigma}, i, \vartheta) \models e(\vec{v}, \vec{x}) \ \wedge \ (\hat{\sigma}, i+1, \vartheta) \models P \\
(\hat{\sigma}, i, \vartheta) \models P \mid Q & \text{iff} & (\hat{\sigma}, i, \vartheta) \models P \ \vee \ (\hat{\sigma}, i, \vartheta) \models Q \\
(\hat{\sigma}, i, \vartheta) \models \mathbf{foreach}\ m \in M : P(m) & \text{iff} & (M = \emptyset) \ \vee \ \dot{\exists} m \in M\colon \ (\hat{\sigma}, i, \vartheta) \models P(m) \rightarrow \\
 & & \mathbf{foreach}\ m' \in \{M \setminus \{m\}\}\colon P(m') \\
(\hat{\sigma}, i, \vartheta) \models < P > & \text{iff} & (\hat{\sigma}, i, \vartheta) \models P \ \wedge \ Trace(\hat{\sigma}) = Trace(\hat{\sigma}) \downarrow \Delta
\end{array}$$

∎

Finally, let us list the productions of the TROLL-grammar that generate process descriptions:

```
<process_def>        ::= process <process_id> '=' <process>
                         [with <process_def_seq>] end process
                         | process
                           <process_id> ['('<data_term_list>')'] '='<process>
                           [with <process_def_seq>] end process

<process>            ::= <process> '->' <process_term>
                         | <process_term>

<process_term>       ::= <process_id>
                         | <event_term>
                         | <choice>
                         | <loop>
                         | <guarded_process>
                         | '<'<process>'>'

<event_term>         ::= <event_id>
                         | <event_id>'('<data_term_list>')'

<guarded_process>    ::= '{'<past_formula>'}' <process>

<loop>               ::= foreach <set_term> do <process> od

<choice>             ::= case <guarded_process_seq>
                           [else <process>]
                         esac
                         | case on <set_term>
                             <guarded_process_seq>
                             [else <process>]
                           esac

<set_term>           ::= <decl_list>
                         | <data_term>
```

2.5 Process Interaction

The behavior of a simple object is described as a process. When we want to describe the behavior of objects that are connected we have to describe how processes interact or communicate.

In our approach, we use the mechanism of *event calling*. Event calling denotes the simultaneous occurrence of events that are called by an event. Suppose an event e_1 calls another event e_2, e_1 `>>` e_2. Then whenever e_1 occurs, e_2 must occur simultaneously, but e_2 can occur without e_1 being obliged to occur. In other words, if e_1 `>>` e_2 then e_1 subsumes e_2. Using this, we may construct a process describing the joint behavior of the communicating subprocesses.

Definition 2.5.1 (*event calling relation*)
Let $\hat{\sigma}$ be a state sequence and e_1 and e_2 be events. Then

$$e_1 \texttt{>>} e_2 \text{ iff } \forall \kappa\colon\ e_1 \epsilon \kappa \Rightarrow e_2 \epsilon \kappa$$

■

Please note that by default event calling means broadcasting – a specified event can call several other events.

In TROLL, the calling relation is notated as `>>`. Please recall that event terms denote events – thus, in TROLL calling is specified between event terms:

$$\mathit{event\_term} \texttt{ >> } \mathit{event\_term}$$

We also allow for *conditional calling*. Here, the calling only is valid if the guarding condition holds. A conditional calling is specified as follows:

$$\{\ \mathit{condition}\ \}\ \texttt{==>}\ \mathit{event\_term} \texttt{ >> } \mathit{event\_term}$$

A special form of interaction is *event sharing*. It is explained as being a calling relation in both directions.

Definition 2.5.2 (*event sharing*)
Let $\hat{\sigma}$ be a state sequence and e_1 and e_2 be events. Then

$$e_1 \texttt{==} e_2 \text{ iff } e_1 \texttt{>>} e_2\ \wedge\ e_2 \texttt{>>} e_1$$

■

The following productions generate interaction specifications in TROLL:

<interaction_def>	::=	'{'*<past_formula>*'}' '`==>`' *<simple_interaction>*
		| *<simple_interaction>*
<simple_interaction>	::=	*<event_term>* '`==`' *<event_term>*
		| *<event_term>* '`>>`' *<event_term>*
		| *<event_term>* '`>>`' *<process_term>*

3 Templates and Objects

The basic building blocks of object-oriented conceptual specifications of systems are (as discussed in Chapter 1) *object descriptions*. Object descriptions are not the objects in a system, they are *generic* descriptions of their properties. An object description is called a *template* in TROLL. A description along with a unique identifier defines an *instance description*. An instant description and the current state make up an object instance. Please recall that the instances we want to specify with TROLL are dynamic in the sense that their state changes over time.

In this chapter, we want to present the components of templates and describe the specification of instances based on templates. Furthermore, we explain our conventions concerning the specification texts and naming.

3.1 Structure of Template Specifications

In this section, we want to give an overview of the sections of a template. The choice of these sections is motivated by the following requirements on a description of objects:

- Each object has an *interface*. This interface consists of *attribute symbols* and *event symbols*. These symbols can be used to observe attribute values and to invoke events from outside the object.
- Attributes and parameters of events are data values. State transitions involve the manipulation of attribute values, i.e. operations on data values.
- Not all values of the corresponding type may be admissible for an attribute. Thus, observations may be restricted. Furthermore, the evolution of attribute values over time may be restricted in the sense of [Lip90, Saa90]. Therefore, in object descriptions we must be able to state constraints on the observations and on the evolution of observations over time.
- The state transitions have to be specified. In our approach, we are interested in the effects of event occurrences on attribute values.
- Finally, the dynamic behavior of objects over time is relevant in our approach. Sequences of events specify the evolution of objects over time. Thus, we have to be able to state precedence relationships for events, state-dependent restrictions on

possible future events, and short-term as well as long-term goals (i.e. events that must happen in the future).

Please note that in this chapter we are only concerned with the specification of single, isolated, non-composite objects. All issues about semantic links or abstractions between descriptions, specification of communications, description of composite objects and compositional description of systems composed from components are postponed.

Simple templates have the following sections:

template [*template name*]
  data types *import of data type signatures*
  attributes *attribute name and type declarations*
  events *event name and parameter declaration*
  derivation *rules for the derivation of attributes and events*
  constraints *static and dynamic constraints on attribute values*
  valuation *effects of events on attributes*
  behavior
    permissions *enabling conditions for event occurrences*
    obligations *completeness requirements for life cycles*
    commitments *state-dependent short-term goals*
    patterns *transactions and scripts*
 [**end template** *template name*]

Please note that templates may be given a name. Thus, we can reuse template specifications just by giving the name. Furthermore, the sections of a template are optional – if sections are not used in the template specification, they can be omitted.

In TROLL, this structure of a template is represented by the following productions in the TROLL-grammar.

<template>	::=	**template** *<template_id>* *<template_sig>* *<template_body>* [**end template** *<template_id>*';']
<template_sig>	::=	[*<data_type_section>*] [*<attributes>*] *<events>*
<template_body>	::=	[*<derivations>*] [*<constraints>*] [*<valuation>*] [*<behavior>*]

<*behavior*> ::= **behavior**
[<*permissions*>]
[<*obligations*>]
[<*commitments*>]
[<*patterns*>]

In the following subsections, we explain in detail the sections of a template specification. In general, keywords of TROLL are in **boldface**, names chosen by the user are in `teletype` font.

3.2 Signature

The **data types**-section, the **attribute**-section and the **events**-section make up the *signature* of a simple template definition. The symbols declared or imported in the signature are the elements of the alphabet that can be used in the template.

In the **data types**-section, we import the signatures of data type specifications. The data type signatures contain the constant symbols and the operation symbols along with their parameter types for a data type. The signatures of all data types used in a template must be imported. The motivation is that it should be possible to formally check a template specification and for doing that we must know which constant symbols and operations can be used in a specification. Furthermore, all attributes are strongly typed in TROLL. To do static type checking at compile time, we must be able to find out the type of data terms. This is possible if we know the signature.

Data types are imported using their unique name. As an example, suppose we want to import the (predefined) data types `nat` (for the natural numbers) and `string` (for strings of characters).

data types `nat, string;`

The **data types**-section of a template is generated by the following production rule:

<*data_type_section*> ::= **data types**
<*data_type_id_list*>';'

3.2.1 Attribute Symbols

The next section of the signature of a template is the **attributes**-section. Attributes (as mentioned earlier) are observable properties of objects. Each attribute takes values from a codomain. As an example, consider attributes of a template that describes *accounts* maintained by a bank. Please note that we use a data type `money` that has the expected properties of data values describing amounts of money.

```
attributes
  Balance:money;
  Red:bool;
  CreditLimit:money;
```

We may declare also *constant* attributes. Such attributes are not allowed to change their value over time. Instead of having to restrict the behavior of object instances such that all state transitions changing that value have to be prohibited, we allow for marking those attributes with the keyword **constant**. Consider for example the holder of an account – in our miniworld at hand, we may want to require that the holder of an account never changes:

```
constant Holder:|BankCustomer|;
```

In this tiny example the codomain of the attribute `Holder` is a unusual data type: it is the type of identifiers for instances of the class type `BankCustomer` notated as `|BankCustomer|`. Thus, the elements of this type are values that may identify object instances of type `BankCustomer`. The issue of identifiers for class types will be explained in detail in Chapter 4.

The codomains of attributes can also be domains of *complex structured values* like sets or lists. Please recall that these are only data values. Consider for example a customer of a bank. Each customer may be the holder of a number of accounts. Thus, the specification of a template for bank customers may include the following attribute specification:

```
Accounts:set(|Account|);
```

Please note that again we are dealing with identifiers here.

Furthermore, attributes can be declared to be *derived*. The values of derived attributes are computed from the values of other attributes. In our example, we want accounts to be able to provide information about the maximum amount that can be withdrawn from an account without overdrawing it:

```
derived MaxWithdrawal:money
```

Another type of symbols that can be declared in the **attributes**-section is an *attribute generator* . An attribute generator is a *parameterized* attribute name. This way, we may specify arrays of attributes or even infinitely many attributes. Consider for example that we want to have an attribute for the balance of an account at the end of each year. We may model that by an indexed attribute symbol in the following way:

```
Bal(nat):money;
```

The **attributes**-section of a template is generated using the following productions of the TROLL-grammar:

<attributes> ::= **attributes**
<att_def_seq>';'

<att_def> ::= *<attribute_def>*
| **constant** *<attribute_def>*
| **derived** *<attribute_def>*

<attribute_def> ::= *<attribute_id>* ':' *<data_type_id>*
| *<attribute_id>*'('*<data_type_id_list>*')'':'*<data_type_id>*

3.2.2 Event Symbols

In the **events**-section of a template, the event generators of a template specification are introduced. Please recall that event generators generate events. Event generators may have parameters. These event parameters may be used to define the effects of an event occurrence on the current state of an object or they may be used to describe the exchange of data during communications.

Event generators denote events which are the basic state transitions that objects can go through (see Section 2.4). Event generators may have parameters or not. Consider the **events**-section of the `account`-template:

```
events
  birth open(in Holder:|BankCustomer|,in Type:checking,saving);
  death close;
  new_credit_limit(in Amount:money);
  accept_update(in Type:UpdateType, in Amount:money);
  withdrawal(in Amount:money);
  deposit(in Amount:money);
  update_failed;
```

For a template, we require that at least one birth event generator be specified. Birth event generators are marked with the keyword **birth**. Please note that the values of constant attributes that are not defined by constraints (see Section 3.4) must be provided as parameters of a birth-event generator. For the `account`-template, the birth event generator may be specified as follows:

```
events
  birth open(in Holder:|BankCustomer|,in Type:checking,saving);
```

The parameters of a birth event usually are used to set the constant attributes, here the attributes `Holder` and `Type`. The optional keywords **in** and **out** refer to the data flow when events are invoked. This is only useful in communications. For details please see Section 2.5. Briefly, **in**-parameters are set by the environment, i.e. the data flows into the object. The values of **out**-parameters are set by the object, i.e. the value is delivered by the object.

Event generators that denote events that put the life of an object to an end (*death events*) are marked with the keyword **death**. Please note that it is not required to declare death-events – if no death-events are specified objects with this template cannot be destroyed or destroy themselves. For the `account`-template, there is a death-event:

death `close;`

Events may be derived, i.e. events may be defined in terms of other event generators. In this case, event generators are marked with the keyword **derived**. Derived events are discussed in the next section.

Finally, we may have *active events.* Active events represent initiatives, i.e. they can be invoked by the object on its own initiative. For a detailed discussion about activity of objects see Section 3.6.5. Briefly, active events are the only events that can occur without being called by other events. Thus, for a society to be able to act at all, at least one active event must be declared. Active events are declared in the **events**-section by marking an active event with the keyword **active** .

The following productions of the TROLL-grammar generate the**events**-section of a template:

<*events*>	::=	**events**
		<*birth_event_seq*>';'
		[<*event_seq*>]';'
<*birth_event*>	::=	[**active**][**derived**]**birth** <*event_def*>
<*event*>	::=	<*event_def*>
		| **death** <*event_def*>
		| **derived** <*event_def*>
		| **derived death** <*event_def*>
<*event_def*>	::=	<*event_id*>
		| <*event_id*>'('<*evt_par_list*>')'
<*evt_par*>	::=	[**in**] <*data_type_id*>
		| [**out**] <*data_type_id*>

3.3 Derivation of Attributes and Events

In TROLL, it is possible to specify how the values of certain attributes (*derived attributes*) depend on the values of other attributes and how certain event generators (*derived events*) denote other events or whole sequences of events. In this section we are going to explain these issues in more detail.

The **derivation**-section of a template is generated by the following syntactical rule:

<derivations> ::= **derivation**
[*<variable_declaration>*]
[*<process_def_seq>*';']
[*<derivation_rule_seq>*';']
[*<event_der_seq>*';']

3.3.1 Derived Attributes

The rules that specify the derivation of the values of derived attributes are specified in the **derivation**-section. Basically, derivation rules are expressions of the following form:

<attribute_term> '=' *<data_term>*

That is, every data term yielding a result of the appropriate type (of the attribute) may be used to compute the value of a derived attribute. Just for the sake of an example, consider a derived attribute `Black` which is set to true if the account is not in the red:

```
Black = not Red;
```

Additionally, it is possible to state *conditional* derivation rules. Conditional derivation rules are only applicable if the condition is valid in the current state. They have the following general form:

'{' *<condition>* '}' '==>' *<attribute_term>* '=' *<data_term>*

As an example consider the following conditional rules that describe the computation of the attribute `MaxWithdrawal` of an `account`-template. Since the value of the attribute `Balance` is always positive even if the account is in the red, two rules are required that state the computation in the red case and in the **not** red case:

```
{ Red } ==> MaxWithdrawal = CreditLimit - Balance;
{ not Red } ==> MaxWithdrawal = Balance + CreditLimit;
```

As usual, we list the relevant productions of the TROLL-grammar:

<derivation_rule> ::= '{'*<formula>*'}' '==>' *<simple_derivation>*
| *<simple_derivation>*

<simple_derivation> ::= *<attribute_term>* '=' *<data_term>*

3.3.2 Derived Events

Derived events are only renamings of ordinary events and whole sequences of events. The motivation is to restrict certain event generators or to make the invocation of events with many parameters easier. For ordinary templates, derived events in object templates are of minor importance. Derived events are used in object interfaces, though. Derived events are specified with rules of the following general form:

$$<\!event\_term\!> \textbf{ is } <\!event\_term\!>$$

Event terms are instantiated event generators that denote an event. What we do here is to specify that we may have different event terms that denote the same event. In our account template we may have something like the following:

```
events
  derived clear_account;
...
derivation
  clear_account is withdraw(Balance);
  ...
```

Derived events may also denote whole transactions. Therefore, event derivations may also be of the following form:

$$<\!event\_term\!> \textbf{ is } \text{'}\mathbf{<}\text{'}<\!process\_id\!>\text{'}\mathbf{>}\text{'}$$

The processes referred to by process identifiers are declared in the **derivation**-section according to the syntax of process definitions described in Section 2.4.

The formal syntax of event derivations is described by the following syntactical rules:

$$\begin{array}{lll} <\!event\_der\!> & ::= & <\!event\_term\!> \textbf{ is } <\!event\_term\!> \\ & & \mid\ <\!event\_term\!> \textbf{ is } \text{'}\mathbf{<}\text{'}<\!process\_id\!>\text{'}\mathbf{>}\text{'} \end{array}$$

3.4 Constraints

In the **constraints**-section, we may impose restrictions on the observable states 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 the possible evolutions of attribute values over time are called *dynamic* constraints. Static constraints are formulae of first order logic as described in Section 2.2. Dynamic constraints are formulae of Future Tense Temporal Logic as described in Section 2.3.2 [Ser80, Lip89, SL89]. Please note that constraints implicitly also restrict the admissible behavior of objects in that certain state transitions (i.e. event occurrences) are not permitted. Please note that syntactically, we do not distinguish static and dynamic constraints.

For accounts, we may e.g. state the following:

```
constraints
  initially Red = false;
  initially CreditLimit = 0;
  initially Balance = 0;
  initially ((Balance > 100) before Red);
  Red ==> (Balance <= CreditLimit);
  Red ==> sometimef(not Red);
```

Constraints with the keyword **initially** state conditions to be fulfilled relative to the initial state. Consider the initial constraint

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

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

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

states that if an account is in 'red condition', sometime in the future it has to leave this condition.

The syntax of the **constraints**-section of a template in TROLL is defined by the following productions:

<constraints> ::= **constraints**
  [*<variable_declaration>*]
  [*<constr_seq>*]';'

<constr> ::= *<fut_formula>*
  | **initially** *<fut_formula>*

3.5 Transitions

The values of attributes may change with the occurrence of events. Thus, to describe the change of objects over time, we have to describe how the occurrence of events affect the values of attributes. *Valuation formulae* stated in the **valuation**-section of a template are based on a positional logic [FS90]. The sentences in the valuation section refer to positions in a sequence of state transitions – hence the name positional. In general, valuation rules have the following form:

[*event term*] *attribute* = *data term*

Such a rule must be read like ([*event term*] *attribute*) = *data term*. It says that immediately after the event denoted by *event term* the attribute has the value denoted by *data term* in the previous state. In more traditional terms, this describes an assignment by an event occurrence. Please note that the term on the right side of the equality sign is evaluated in the state *before* the event occurred.

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

For our `account`-template, we, may have the following valuation rule:

```
valuation
  variables m:money;
  [new_credit_limit(m)]CreditLimit = m;
```

It says that for a substitution of the variable `m` with a suitable value the attribute `CreditLimit` has this value of `m` after the event denoted by `new_credit_limit(m)` occurred.

Valuation rules may also be *conditional*, i.e. they are only applicable if the guarding condition holds in the current state. They have the following general form:

{ *guarding condition* } `==>` [*event term*] *attribute* `=` *data term*

Please note that the condition is evaluated *before* the event occurrence.

In the **valuation**-section of our account example we may have rules like the following:

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

Please note again that the data term on the right side of the equality sign and the guarding condition are evaluated in the previous state.

In the following rule, we state that the occurrence of an event has an effect on more than one attribute. In that case, we may use a comma to combine several equations:

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

Valuation rules are generated by the following productions in the TROLL-grammar:

<valuation> ::= **valuation**
  [*<variable_declaration>*]
  <val_rule_seq>';'

<val_rule> ::= '{'*<past_formula>*'}' '`==>`' *<simple_valrule>*
  | *<simple_valrule>*

<simple_valrule> ::= '['*<event_term>*']' *<effect_list>*

<effect> ::= *<attribute_term>* '`=`' *<data_term>*

3.6 Behavior Specification

A major part of a template specification is made up of the behavior specification. By behavior specification we mean here the specification of the *permitted event sequences*. That is, we have to specify precedence relationships for events, completeness requirements for life cycles, and activity. Each event trace must satisfy the behavior specification.

Permitted object life cycles may be specified using several different formalisms. The reason for this is that the overwhelming variety of real-world patterns of behavior may not always be specified conveniently using just one formalism. In TROLL, we offer four formalisms:

- *Permissions* state preconditions that must be fulfilled in order to allow certain events to occur. Events are not allowed to occur in the current state if the precondition is not fulfilled. Preconditions may, however, not only refer to the current state – they may refer also to the trace of events that occurred in the life of an object so far. Thus, preconditions for permissions are formulae of *Past Tense Temporal Logic* as described in Section 2.3.3. Permissions are described in detail in Section 3.6.1.

- *Obligations* state completeness conditions for life cycles of objects. A life cycle of an object can only end if the obligations are fulfilled. In other words, obligations are preconditions for the death of an instance. Obligations are explained in detail in Section 3.6.2.

- *Commitments* state goals that an instance must try to fulfill as soon as possible. Commitments are usually guarded by a formula of past tense temporal logic. Whenever a commitment can be applied, it should be fulfilled. The only reason for it not being fulfilled may be that the activity may involve communications which are not possible in the current state. The semantics of commitments are not completely explored yet. In the execution model, the semantics can be explained by precedence on events. The declarative semantics, however, are not explained up to now. Commitments are considered to be very important in the early stages of system design and are thus part of TROLL. The specification of commitments is explained in Section 3.6.3.

- *Patterns* describe patterns of behavior an object goes through. Basically, pattern specifications are process descriptions as introduced in Section 2.4. A brief explanation of the usage of process descriptions in the **patterns**-section of a template can be found in Section 3.6.4.

3.6.1 Permissions

Permissions ensure that something bad does not occur in the life of an instance. They are stated in the **permissions**-section of a template. The general form of permissions is

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

For convenience, it is possible to have a sequence of event terms after the condition if the same condition applies for all the event terms listed. Permissions may refer to the current observable state (*simple permissions*) or to the history of events that occurred in the life of an instance so far (*temporal permissions*).

In simple permissions, the condition is a formula of first-order logic over the signature of a template. As an example for a simple permission look at the following rule that requires an account to be empty before it can be closed:

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

Simple permissions may also be what is called a *weak* precondition: they may refer to the parameters of the event which is guarded by a permission condition. Here e.g. we have a precondition that prohibits withdrawals of more than the maximum withdrawal being the current value of the attribute `MaxWithdrawal`:

```
variables m:money;
  { m <= MaxWithdrawal } withdrawal(m);
```

Please note that by default all variables are quantified *universally.*

Temporal permissions have a condition that refers to the history of the instance. Thus, the condition is a formula of Past Tense Temporal Logic as defined in Section 2.3.3.

Very often, the condition refers to event occurrences in the past. The **after**-predicate explained in Section 2.3.3 is used to state that an event just occurred. An example for a simple permission is the rule that the credit limit can only be updated if there has been at least one deposit before:

```
variables m,m1:money;
  { exists(m)(sometime(after(deposit(m)))) } new_credit_limit(m1);
```

Note that we have to bind the variable `m` with an existential quantifier since we only require the precondition to be true for at least one value for `m`.

The rules may become quite complex. When we use bounded temporal operators, we are only interested in the tail of a prefix of a life cycle. The following rule for example states that after a transaction has been completed with the occurrence of one of the events `update_failed` or `deposit(m1)` or `withdrawal(m1)`, at most one event of the class `accept_update` may occur (i.e. we do not allow to handle interleaved updates in an account):

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

Note that there are a number of implicit permissions, e.g. for birth-events: they are not allowed to occur after a birth event has occured in the life of an instance.

The TROLL-syntax of the **permissions**-section is defined by the following productions of the TROLL-grammar:

<permissions> ::= **permissions**
[*<variable_declaration>*]
<per_rule_seq>';'

<per_rule> ::= '{'*<past_formula>*'}' *<event_term_list>*

3.6.2 Obligations

In the **obligations**-section, we state completeness requirements for life cycles. These requirements must be fulfilled before the object is allowed to die. Obligations can be regarded as preconditions for the death of instances.

The simplest form of an obligation is an event term. This means that the events denoted by the event term must occur in the life cycle of an object. A simple example of this is the requirement that each account must be closed sometime, i.e. an instance must die:

```
obligations
  close;
```

Please note that parameters of events are universally quantified implicitly if not otherwise stated. This is not always suitable. An obligation may have to be fulfilled for at least one element of a set. An example for this may be that at least once a deposit must be made:

```
exists(m:money)(deposit(m));
```

Obligations may also be whole processes that must occur in a life cycle of an object. Thus, also a process description may be stated as an obligation. For accounts, this may be that at least one deposit transaction must be performed successfully:

```
exists(m:money)(accept_update(deposit,m) -> deposit(m));
```

We also admit combinations of simple obligations. Obligations may be combined disjunctive and conjunctive. For a disjunction, at least on of the alternatives must be fulfilled. As an example consider the obligation that at least one update (a withdrawal or a deposit) must occur in the life of an account:

```
exists(m:money)(deposit(m) or withdrawal(m));
```

For a conjunction, all the alternatives must be fulfilled. Thus, another possibility to state that at least one deposit transaction must be completed successfully is the following:

```
exists(m:money)(accept_update(deposit,m) and deposit(m));
```

Usually, obligations depend on the history of the object. The following requirement states that once an event `accept_update(t,m)` occurs, this update must be completed eventually by an occurrence of one of the events `update_failed` or `deposit(m)` or `withdrawal(m)`:

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

A complete life cycle is only admissible if the obligations are fulfilled before the death event. Intuitively speaking, whenever the condition is true, the obligation is appended to a list of open goals to be fulfilled. When an obligation is fulfilled, it is removed from that list.

As usual, we list the productions of the TROLL-grammar to generate the **obligations**-section:

<obligations> ::= **obligations**
  [*<variable_decaration>*]
  [*<process_decl_seq>*';']
  <obl_expr_seq>';'

<obl_expr> ::= '{'*<formula>*'}' '==>' *<obl_s_expr>*
  | *<obl_s_expr>*

<obl_s_expr> ::= *<obl_s_expr>* **or** *<obl_term>*
  | *<obl_term>*

<obl_term> ::= *<obl_term>* **and** *<obligation>*
  | *<obligation>*

<obligation> ::= '('*<obl_expr>*')'
  | *<process>*
  | **exists** '('*<decl_seq>*')' '('*<obl_expr>*')'

3.6.3 Commitments

Commitments are similar to obligations in that they also describe goals to be fulfilled by instances. The intuitive semantics of *commitments* is, however, a bit more subtle. Commitments describe internal activity of objects. If a commitment is applicable, the instance should actively try to fulfill the commitment. In contrast to obligations however, the instance does not violate its specification if it is unable to fulfill it due to communications. Currently, we do not have a satisfactory semantics for commitments. We therefore only give an example that hopefully makes the differences to obligations more clear.

In our account example, a request to update the account should be processed actively as soon as possible. As an example the following formula states that after a request for an update of type deposit was accepted the corresponding `deposit`-event instance should occur:

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

An object should try to fulfill commitments as soon as possible. In contrast to obligations (which just have to be fulfilled eventually), commitments describe short-term goals. Furthermore, if the environment of an object does not permit the object to fulfill its commitments, the specification is not violated.

The syntax of the **commitments**-section in TROLL is similar to the syntax of the **obligations**-section:

<commitments> ::= **commitments**
[*<variable_declaration>*]
[*<process_decl_seq>*';']
<obl_expr_seq>';'

3.6.4 Patterns

Patterns describe how events are ordered in the life cycle of an object. Usually, patterns are used to describe processes, i.e. sequences of events. In TROLL, patterns are only valid for events of process definitions: events that are not covered by a process definition (i.e. not an element of the set $Events(P)$ for a process definition P (see Section 2.4)) may interleave the process defined by a pattern specification. Process definitions have been explained in detail in Section 2.4.

In principle it is possible to specify arbitrary behaviors with permissions and obligations. The motivation to introduce the specification of patterns as yet another means to specify behavior is convenience. When we have to specify a very determined behavior (i.e. if the object has to gho through certain restricted patterns of behavior), then we must specify many permissions and obligations – the specification of patterns is just more concise and more natural.

The **patterns**-section of a template consists of variable and process declarations along with process definitions.

As a first short example consider the specification of a simple clock that just counts the ticks:

```
template OversimplifiedClock
```

```
  data types nat;
  attributes
    time:nat;
  events
    birth start(in time:nat);
    tick;
  valuation
    variables t:nat;
    [start(t)]time=t;
    [tick]time=time+1;
  behavior
    patterns
      process CLOCK = tick -> CLOCK end process;
      CLOCK;
end template OversimplifiedClock;
```

For the semantics, it is possible to describe the intended behavior of a behavior pattern (or process) by permissions and obligations.

Let us now turn to a more realistic example. To illustrate the use of the process sublanguage consider the description of automatic teller machines (ATMs):

```
template ATM
  data types nat,money,bool,|CashCard|;
  attributes
    CashOnHand:money;
    derived Dispensed:bool;
  events
    birth set_up;
    death remove;
    ready; cancel;
    refill(in Amount:money);
    read_card(in C:|CashCard|);
    check_card_w_bank(in Acct:nat:, in PIN:nat);
    card_accepted; bad_PIN_msg; bad_account_msg;
    issue_TA(in Acct:nat,in Amount:money);
    TA_failed_msg; eject_card;
    dispense_cash(in Amount:money);
  constraints
    initially CashOnHand = 10000;
  derivation
    Dispensed = (CashOnHand < 100);
  valuation
    variables m:money;
    [refill(m)]CashOnHand = CashOnHand + m;
    [dispense_cash(m)]CashOnHand = CashOnHand - m;
  behavior
    permissions
```

```
    variables n:nat; m:money; C:|CashCard|;
    { not Dispensed } read_card(C);
    { m <= CashOnHand } issue_TA(n,m);
  patterns
    variables n,p:nat; m:money; C:|CashCard|;
    process ATM_USAGE = read_card(C) -> case
                                          cancel;
                                          check_card_w_bank(n,p)
                                               -> GO_ON
                                        esac -> eject_card -> ready
    end process
    where process GO_ON = case
                            bad_account_msg;
                            bad_PIN_msg;
                            card_accepted -> case
                                                cancel;
                                                DO_IT;
                                             esac;
                          esac
    end process
    where process DO_IT = issue_TA(n,m) -> case
                                             dispense_cash(m);
                                             TA_failed_msg
                                          esac
    end process;
  ATM_USAGE;
end template ATM;
```

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

The behavior of an ATM during a service-session is quite complex. An ATM only accepts a cash card if the cash on hand exceeds 100. After the card has been read, the session may either be canceled or it may go on with the corresponding `check_card_w_bank`-event. Please note that through the variables we are able to use the values read by the `read_card`-event, i.e. variables are not universally quantified but chosen by the environment, i.e. they are existentially quantified inplicitly (refer to Section 2.4). The keyword **case** usually indicates an *external choice*, i.e. the decision for one of the alternatives is left to the environment. The `GO_ON`-pattern starts with another external choice which depends on whether the inserted card is valid or not (a decision which is not made by the ATM itself). If the card is accepted, the process may be canceled again or may go on with the launch of a transaction that can either be terminated with the dispense of cash or a failure message. That decision is again not made by the ATM but the environment. In any case, the service session with an ATM ends by the eject of the cashcard. Please note that an external choice requires communication with the environment that decides on how to proceed – this is, however, not addressed in this chapter.

Patterns can be transformed into equivalent behavior descriptions with permissions and obligations. This can, however, become a bit lengthy. To illustrate this, let us give a fragment of the above specification with just permissions and obligations:

```
behavior
  permissions
    variables n,p:nat; m:money; C,C1:|CashCard|;
    { exists(C)(previous (after(read_card(C)))
      since last(after(cancel) or
                 after(dispense_cash(m)) or
                 after(TA_failed_msg)))
    } cancel, check_card_w_bank(n,p);
    { exists(m,C1)(always
                 (not(after(read_card(C1)))
                 since last(after(cancel) or
                            after(dispense_cash(m)) or
                            after(TA_failed_msg))))
    } read_card(C);
    { exists(n,p)(previous (after(check_card_w_bank(n,p)))
      since last(after(cancel) or
                 after(dispense_cash(m)) or
                 after(TA_failed_msg)))
    } bad_account_msg,bad_PIN_msg,card_accepted;
    { exists(m)(previous
      (after(cancel) or
                 after(bad_account_msg) or
                 after(bad_PIN_msg) or
                 after(TA_failed_msg) or
                 after(dispense_cash(m)))
                 since last(after(cancel) or
                            after(dispense_cash(m)) or
                            after(TA_failed_msg)))
      } eject;
      ...
    obligations
      variables C:|CashCard|;
      { exists(C)(after(read_card(C))) } ==> ready;
      ...
```

The syntax of the **patterns**-section is described by the following productions:

<patterns> ::= **patterns**
    [*<variable_declaration>*]
    [*<process_def_seq>*‘;’]
    <process_seq>‘;’

Please refer to Section 2.4 for a detailed explanation of process descriptions.

3.6.5 Activity

Let us now briefly comment on the specification of object activity. From our point of view we identified the following forms of object activity:

- An object *triggers* another object or a collection of other objects.

This is the classical case of objects being regarded as either active or passive entities. An active object may initiate something whereas a passive object suffers something. Note that the classification of objects using this criterium is valid only in the current instant of time since an object that triggers something can be requested to do something by another object immediately after having been active in this sense. The object that triggers has the authority to cause activity. A trigger can be regarded as a *directive*.

- An object performs a number of actions as a *reaction* on being triggered.

This is the traditional computer systems view of activity. Programs need to be triggered (ie started) to do a number of tasks that have been specified in a certain order (ie programmed) before.

- An object can do something *on its own will* without having to be asked to do it.

While modelling the real world it is often the case that system components (eg users or models of entities whose behavior is unpredictable) may do something in a nondeterministic way. A formalism has to offer a means to describe such spontaneous activity. Spontaneous activity often arises in abstracting from causality of actions. Take for example a user of the library – if we say that (s)he spontaneously may decide to borrow a book, we abstract from reasons that cause this behavior.

- In a certain state, an object may *commit itself* to do something on its own will.

This is again a form of activity that can be surveyed in the real world. Objects may not be capable to do something spontaneously anytime, but if certain conditions are fulfilled, they may take the initiative and do something. As an example, consider a user of a library who commits himself to return a book once he borrowed it.

From these considerations, we now want to lay down the types of activity which should be supported by an approach to formal specification of object systems in an early phase of system development. We classify activity of objects in three categories:

- Reactive activity
- Spontaneous activity
- Activity commitment

Reactive activity means that an object is capable of performing a number of tasks once having been triggered to do so. Thus, we are able to model *causality* in actions and the *ordering* of actions in a system. An example for this may be the process that is triggered by a user requesting to borrow a book from the library: A copy of the book has to be fetched from the depot, the return date has to be put down for that particular copy, and the copy has to be handed to the user.

Spontaneous activity means that an object is capable of making a local event happen anytime, provided that the occurrence of the event is allowed. As an example for spontaneous activity consider a user of a library: (S)he may decide on borrowing a book anytime unless (s)he has borrowed the maximum number of books already.

Activity commitment means that an object commits itself to perform an event on its own will whenever certain preconditions are fulfilled, but the occurrence may be prevented by conditions of the environment. In contrast to obligations, life cycles not including events specified to be commitments are correct. Thus, activity commitments intuitively state goals that are *desirable* to be achieved *in the current situation.*

We start with explaining how spontaneous activity is specified. To recall, we want to model the situation that some object as a representation of some entity should be capable of behaving in a nondeterministic way. Consider the users of a library. When we model a user, we want to abstract from the reasons that cause her/him to issue a request for a book and assert that he decide spontaneously. Users are thought to be roles of persons, which are assumed to have the usual properties. In the specification, we only have to mark the events that can occur spontaneously with the keyword **active**.

Reactive activity is concerned with the modeling of actions that have been caused by events that have occurred either in the object itself or in another object. This causality is perfectly specified with the event calling mechnism described in Section 2.5.

Finally, let us consider activity commitments. Being in a certain state, an object may commit itself to doing something. This is useful to model that in a system an *actor* promises to *perform* certain actions as soon as possible. Well, we all know that this may be prevented by certain circumstances. So what distinguishes commitments from obligations is that they do not have to be fulfilled before an object dies.

3.7 Object Specification

In this section we want to describe how single object instances are specified. Basically, a specification of a single object is a template identified by an *object identifier.* Please note that the specification of single objects is not a class specification. A single object specification denotes one and only one object instance. The instance must satisfy the template.

As an example consider a system which models exactly one bank. Then, we would specify a single bank object identified by a name like "BankOfTroll". We do not give a complete specification here, however.

```
object BankOfTroll
  template
```

```
    ...
end object BankOfTroll;
```

For object specifications, we may also reuse templates. Suppose we have specified a template `bank` for a bank in general before which is a suitable description of the `BankOfTroll`. Then, the specification would have the following form:

```
object BankOfTroll
  template bank
end object BankOfTroll;
```

The production in the TROLL-grammar to generate an object specification is the following:

<*object_spec*>	::=	**object** <*ob_id*>
		<*template_spec*>
		end object <*ob_id*>

<*template_spec*>	::=	<*template*>
		| **template** <*template_id*>

3.8 Temporal Semantics of Template Specifications

In this section, we want to give some ideas towards the semantics of template specifications in terms of temporal logic as described in Sections 2.3.2 and 2.3.3. In fact, we use a general temporal logic which includes the past tense and future tense temporal quantifiers.

The possible object evolution can be described by state sequences as described in Section 2.3.2. Here, we want to give some hints towards a translation of template specifications in sets of sentences over temporal logic which we want to call a *temporal object specification* in the sequel.

The signature of a temporal object specification is composed from the data type signatures imported in the template specification, the signature of the local events and functions for the attributes (attribute generators). Let us now briefly sketch the translation of the sections of a template specification.

In the derivation section, we have stated equations between attributes and data terms. These equations are first order formulas. They have to be valid in every state the object goes through – thus, they are translated into invariants. Let φ be a derivation equation. It is translated into

$$\mathbf{alwaysf}(\varphi)$$

In the **constraints**-section, we have stated formulas of future tense temporal logic. In addition, we are able to state initial constraints. Initial constraints do not have to be translated. Other constraints are invariant, i.e. they are valid for the whole lifetime of an object. Let φ be a constraint formula. It is translated into

$$\textbf{alwaysf}(\varphi)$$

Valuation rules describe the effects of event occurrences on attribute values depending on the old state. Let e an event term, a an attribute term and t a data term. Then a valuation rule

```
[e]a = t
```

is transformed into a temporal formula of the following form:

$$\forall x\colon \textbf{type}(\mathtt{a})\colon (\textbf{alwaysf}((\mathtt{t} = x) \;\Rightarrow\; \textbf{next}(\textbf{after}(\mathtt{e}) \Rightarrow (\mathtt{a} = x))))$$

where the variable x is used to save the value of the term `t` evaluated in the old state. All free variables in valuation rules have to be bounded explicitly by a universal quantifier. Please note that we use a function **type** to compute the type of an attribute.

Recall that we employ a frame rule: each attribute which is not affected by an event occurrence does not change its value in such a transition. This has to be stated for each transition in a formula like

$$\forall x\colon \textbf{type}(\mathtt{a})\colon \textbf{alwaysf}((\mathtt{a} = x) \;\Rightarrow\; \textbf{next}(\textbf{after}(\mathtt{e}) \Rightarrow (\mathtt{a} = x)))$$

For the **behavior**-section, the translation is more complicated. First of all, we have to ensure that the initial state is reached by a birth-event transition and that no following transition includes a birth-event. Let $s_1, \ldots, s_n$ be data sorts and $b(s_1, \ldots, s_n)$ be a birth event generator. Then the temporal object specification must include the following formulas:

$$\exists x_1\colon s_1, \ldots, x_n\colon s_n\colon \textbf{after}(b(x_1, \ldots, x_n))$$

$$\forall x_1\colon s_1, \ldots, x_n\colon s_n\colon \textbf{next}(\textbf{alwaysf}(\neg\textbf{after}(b(x_1, \ldots, x_n))))$$

For death-events, we must ensure that transitions including them will lead to an empty state, i.e. only the constant **false** is valid. Let $s_1, \ldots, s_n$ be data sorts and $d(s_1, \ldots, s_n)$ be a death event generator.

$$\forall x_1\colon s_1, \ldots, x_n\colon s_n\colon \textbf{alwaysf}(\textbf{after}(d(x_1, \ldots, x_n))) \Rightarrow \textbf{next}(\textbf{false})$$

Permissions have the general form

$\{\ \varphi\ \}$ `e`

where φ is a formula of past tense temporal logic and `e` is an event term. Permissions are translated into

$$\textbf{alwaysf}(\neg\varphi \Rightarrow \textbf{next}(\neg\textbf{after}(\mathtt{e})))$$

In a second step, all free variables must be universally quantified. Please note that in the temporal object specification permissions are formulas in our general temporal logic.

For obligations, we have several forms. An event term `e` as an obligation is translated into

$$\textbf{sometimef}(\textbf{after}(\mathtt{e}))$$

For an obligation of the form

exists`(x:atype) ... e(...,x,...)`

with an event generator `e` the following temporal formula is created:

$$\exists \mathtt{x:atype}: \ldots \textbf{sometimef}(\textbf{after}(\mathtt{e(...,x,...)}))$$

Conjunctions and disjunctions of obligations are translated to conjunctions and disjunctions of the translations of the components, respectively.

We do not want to go into detail on the transformation of process definitions into temporal logic formulas. In general it is known that explicit process formalisms like CSP or Petri nets can be translated into temporal logic with the possible necessity to use an extra variable to encode the current state.

4 Class Types and Classes

The first abstraction mechanism we introduce for TROLL is *classification*. We distinguish between classes and class types. We distinguish between a template (which is the schema for an instance), a class type (which is the schema for a class) and a class (which is a collection of instances with the same template).

Basically, a class type defines the potential instances of a class of that type. Thus, it can be regarded as being the *schema* for a class. An instance is specified by a template identified by an identifier. Thus, a class type is defined by a template along with a set of possible identifiers (the *identification space*). An instance identifier is a value of a special data type, the type of identifiers of a class type.

A class can be seen as a container for instances of the corresponding class type. The class type defines the possible extensions of a class.

In the following section, we explain class types and their specification. In general, class types may not be specified separately but their specification is part of a class specification. Once a class type is defined as the type of a class, however, we may define more classes having the same type.

4.1 Class Types

Basically, a class type consists of a template and a data type for the possible identifiers of instances. This is the crucial point in class type specifications.

4.1.1 Identification Space Definition

Objects are identified by *values* in our framework. An object identifier remains unchanged for the whole lifetime of an instance. Moreover, it is unique in the system.

For each type an associated identification space is defined. An identifier along with the class name then identifies an instance of a class uniquely.

For the definition of identification spaces, we use an approach that combines the advantages of key mechanisms known from the database field (see e.g. [Dat86]) and surrogates (see e.g. [Ken78, Cod79]). The approach is based on abstract data types. The idea is to characterize a type of invisible identifiers (i.e. surrogates) by operations that construct these values from key values. The basic advantages are:

- the identifier is not directly visible;

- the construction of identifiers is based on key data types;
- key values can be obtained from an identifier and
- identifiers are *typed*.

For identification spaces, we use the following notational convention: for a class type `ctype` the identification space is notated as `|ctype|`. The name of the identification space can be used in template specifications like an ordinary data type name.

In the following let us briefly show how identification spaces can be specified using a specification language for abstract data types. The specification for the identification space for a class type `BOOK` would look like this:

```
data type |BOOK|
  sorts |BOOK|, string;
  opns
    BOOK:string × string -> |BOOK|;
    FirstAuthor:|BOOK| -> string;
    Title:|BOOK| -> string;
  axioms
    variables a,t:string;
    FirstAuthor(BOOK(a,t)) == a;
    Title(BOOK(a,t)) == t;
end data type |BOOK|;
```

We specify three operations on the data type `|BOOK|`, namely the operation `BOOK` to construct a `|BOOK|`-value from two key values (each of them of type `string`) and to map `|BOOK|`-values to the corresponding constructor values of type `string`. In the sequel, we will call these constructor values *key values*, their types *key types*, and the cartesian product of the key types *key domain*. The result of this is that

- the data type `|BOOK|` is isomorphic to the type **tuple**(`string`, `string`);
- we do not say anything about the representation of `|BOOK|`-values and
- a carrier set of `|BOOK|` is the set of possible identifiers.

Since we use key values to construct identifiers, the language features to specify identification spaces in TROLL are similar to the definition of key attributes in data definition languages. Consider for example the class type `BOOK` that describes possible book classes. We assume that books can be identified by the first author's name and the title of the book. Then, the identification space `BOOK` (as explained above) would be specified as follows:

```
identification
  data types string;
  FirstAuthor:string;
  Title:string;
```

4.1.2 Class Type Specification

The specification of the identification space makes up the first part of a class type specification. Please recall that it is, however, not possible to specify a class type in isolation.

A class type specification in TROLL is made up of a type header (for the moment this is an identification space definition like the one above) an a template specification as defined in Chapter 3. Thus, the corresponding productions of the TROLL-syntax are the following:

<type_spec>	::=	*<type_header>* *<template>*
<type_header>	::=	**identification** *<data_type_section>* *<id_def_seq>*';'
<id_def>	::=	*<attribute_id>* ':' *<data_type_id>*

4.2 Class Specification

For a class type as explained in the last section, we may specify one or more classes. Please note that the class type in general is specified along with the specification of the corresponding class. The class type then has the name of that class.

What about the identification space when we define more that one class for a class type? Basically, the identification space then is an isomorphic copy of the identification space of the class type being used – the constructor operation is only renamed in order to provide unique identifiers for instances of the class. Suppose we want to define another class with the type `BOOK` called `LibraryBOOK`. Then the constructor function would be

```
LibraryBook:string × string -> |LibraryBOOK|;
```

As an example for a class definition let us give the specification of an object class `Account` that describes accounts. Please note that implicitly we define also the class type `Account` with the identification space `|Account|`.

```
object class Account
  identification
    data types nat;
    No:  nat;
  template
    data types |BankCustomer|,money,bool,UpdateType;
    attributes
      constant Holder:|BankCustomer|;
      constant Type:{checking, saving};
```

```
    Balance:money;
    Red:bool;
    CreditLimit:money;
    derived MaxWithdrawal:  money;
  events
    birth open(in Holder:|BankCustomer|,in Type:checking,saving);
    death close;
    new_credit_limit(in Amount:money);
    accept_update(in Type:UpdateType, in Amount:money);
    withdrawal(in Amount:money);
    deposit(in Amount:money);
    update_failed;
  constraints
    initially Red = false;
    initially CreditLimit = 0;
    initially Balance = 0;
    initially ((Balance > 100) before Red);
    Red ==> (Balance <= CreditLimit);
    Red ==> sometimef(not Red);
  derivation
    { Red } ==> MaxWithdrawal = CreditLimit - Balance;
    { not Red } ==> MaxWithdrawal = Balance + CreditLimit;
  valuation
    variables m:money;
    [new_credit_limit(m)]CreditLimit = m;
    { not Red and (m <= Balance)} ==>
      [withdrawal(m)]Balance = Balance - m;
    { not Red } ==> [deposit(m)]Balance = Balance + m;
    { not Red and (m > Balance)} ==>
      [withdrawal(m)](Balance = m - Balance) and (Red = true);
    { Red } ==> [withdrawal(m)]Balance = Balance + m;
    { Red and (m >= Balance)} ==>
      [deposit(m)](Balance = m - Balance) and (Red = false);
    { Red and (m < Balance)} ==> [deposit(m)]Balance = Balance - m;
  behavior
    permissions
      variables t,t1:UpdateType; m,m1,m2:money;
      { Balance = 0 } close;
      { not sometime(after(accept_update(t1,m1)))
        since last(after(update_failed) or
                   after(deposit(m2)) or
                   after(withdrawal(m2)))} accept_update(t,m);
      { sometime(after(accept_update(t,m)))
        since last after(accept_update(t1,m1)) and
        t = withdraw and (m > MaxWithdrawal)} update_failed;
      { sometime(after(accept_update(t,m)))
```

```
            since last after(accept_update(t1,m1)) and
            t = deposit } deposit(m);
          { sometime(after(accept_update(t,m)))
            since last after(accept_update(t1,m1)) and
            t = withdraw and (m <= MaxWithdrawal)} withdrawal(m);
        commitments
          variables m:money;
          { after(accept_update(deposit,m)) } ==> deposit(m);
          { after(accept_update(withdraw,m)) and
            (m <= MaxWithdrawal) } ==> withdrawal(m);
  end object class Account;
```

Suppose now that we want to specify another class `OtherAccount` with the type `Account`. The specification would then be the following:

```
  object class OtherAccount
    type Account;
  end object class OtherAccount;
```

In the TROLL-syntax, a class specification is generated by the following productions:

<*class_spec*>	::=	**object class** <*class_id*>
		<*typedef*>
		end object class <*class_id*>
<*typedef*>	::=	<*type_spec*>
		| **type** <*class_id*>

4.3 Semantics of Classes

A class type defines the potential instances that can be members of the associated class. A class is something like a container that can contain only instances of the associated class type. In other words, class types are *stateless* whereas for a class one can always refer to the current *state of a class*, i.e. the current set of instances being contained along with their respective current states. That is, *a class can be regarded as an object!* It is, however, not a simple object but a complex one that has all the current instances as subobjects. Let us now briefly sketch the semantics of a class.

A class type denotes a tuple $ct = (I, ob)$, where I is a sort of identifiers (i.e. a set of data values) and *ob* is a model for a template (called object model in [JSS91, HJ91]). A model for a class then has to take the dynamics into account. Thus, we define a model for a *state* of a class:

Definition 4.3.1 (*state of a class*)
The *state of a class oc* is defined as

$$oc = (ct, \zeta)$$

where ct is the corresponding class type and

$$\zeta\colon\ ct.I \longrightarrow \mathbf{Prefix}(ob.P.\Lambda)$$

is a function that assigns to each identifier an admissible prefix of a life cycle of an instance of ob. Please note that also the empty life cycle belongs to $ob.P.\Lambda$ describing instances that are not created yet, i.e. that are not in the class oc. ■

We are now able to regard *sequences of states* for object classes. The semantics of a class is then defined over state sequences as follows:

Definition 4.3.2 (*life cycles of a class*)
Let oc be a class and ct the corresponding class type. Let σ^*_{oc} be the set of all possible state sequences for oc. The *life cycles of a class* are sequences $\hat{\sigma}_{oc} \in \sigma^*_{oc}$ such that the projection of $\hat{\sigma}_{oc}$ on an instance $i.ob$ in oc yields a valid life cycle of $ct.ob$. ■

The semantics of a class is a complex object that is obtained through the parallel composition of the contained instances. Thus, a class model is a model of a complex object.

We are in principle able to extend a class type with a *class signature* which includes attributes and event generators for the class (as an object) and a specification of the behavior and evolution of a class. This is, however, not reflected in the current version of TROLL.

5 Complex Objects

In the last chapter we sketched the concept of object classes as complex objects. We will now describe constructors for complex objects in more detail.

Complex objects are a means to describe aggregation of subobjects to structured objects. This construction is known from semantic data models. Complex objects are especially useful for the description of non standard applications.

Basically, we can identify two kinds of complex objects [BB84]:

- *Non-disjoint* complex objects may share components. Thus, components are autonomous objects. In TROLL, we further distinguish two kinds of non-disjoint complex objects: *dynamic* complex objects and *static* complex objects. The former describing objects which change their composition at runtime, the latter describing objects with a composition determined at specification time. The introduction of two different constructions for dynamic complex objects is motivated by the possibility of checking the specification of static aggregation at compile time.

- *Disjoint* complex objects do not share any components. This implies that components cannot exist outside the complex object, they are strongly dependent and the components are *local* to the complex object. The composition is always static. The components are referable only inside the complex object.

For disjoint and non-disjoint complex objects the components are encapsulated in the sense that their state may only be altered by events local to the components. Their attribute values, however, are visible. The coordination and synchronization between the complex object and its components or between the components must be done by communication.

The semantic foundation for the specification of complex objects is object inclusion, i.e. safe object import. Object inclusion is defined as a structure and behaviour preserving mapping between object models [ESS90, EGS90]. Thus the construction of complex objects is static. We can also define object inclusion as *semantic inheritance*, i.e. inheritance of objects itself to be distinguished from *syntactic inheritance* which can be described as code reuse. The idea of semantic inheritance goes somewhat further than the notion of inheritance implemented in most object-oriented programming languages.

In the next three sections we explain the basic constructs of TROLL to describe complex objects. The first section deals with explicit object inclusion, the simplest notation used to explain the other possible constructs. The second section introduces dynamic complex objects, a high level language construct able to describe complex objects with *changing structure*. In the last section the special case of disjoint complex objects is explained.

5.1 Explicit Object Inclusion

The aggregation of static complex objects is described using predicates over identifiers and constants. In contrast to dynamic aggregation where the structure of the complex object may vary over time, the structure of a static complex object never changes. The composition therefore consists of all objects which satisfy the constraints given by the aggregation predicate (see the following example). Possibly there are components belonging to the aggregation that are not yet born. In the underlying semantics, these objects have an empty trace.

The concept of inclusion or embedding denotes that an object is included into another object without violating any of its properties. Using an informal definition of object inclusion, this means that :

- Attribute values of the embedded object may only be altered by events local to it although they may be read inside the enclosing object.
- Projecting a life cycle of a complex object to the events local to a component always yields a life cycle of the component. Note that there may be life cycles of a component that can not be obtained by projection, i.e. those ones that are not allowed *in the context* of the complex object.
- Interactions do not cause any violation of the life cycle specification of both objects (see below).

Object embedding is directly reflected in the TROLL-language. Therefore TROLL supports a primitive language construct to define explicitly embedded objects. This construct may be applied to single object specifications as well as to object class specifications. We have the following syntactic patterns for the inclusion facility for single objects :

```
including
  instance object identification
    in ClassName
    [ as newName ];
  ...
```

This construct can be used to describe the inclusion of an object instance of some class `ClassName`, that is, ***object identification*** is a data term identifying an *object instance* in class `ClassName`. As a special case, the object identification may be given by an object name of a single object, thus ommitting the **in** part of the including clause (see example below). The second part of the instance including clause – the **as**-clause – is optional and can be used to rename the included component.

To include an instance of a given class `Class` where we have an identifier of this class say `I`ϵ`|Class|` we can write: **instance** `I` **in** `Class`. The object identifier must be a constant property or be derived from constant properties of the object instance. Renaming is especially useful for including an instance of a class: we may avoid the full object denotation, i.e. class name and identifier (`Class(I)`) when referring to the part object.

For inclusion of subsets of class populations, we have another including clause, dealing with the selection of a proper subset of class instances :

including
  `ObjectVariable` **in** $\texttt{Class}_X$
    where *`aggregation predicate`*
    as $\texttt{Class}_Y$;
  ...

`ObjectVariable` ranges over objects of class $\texttt{Class}_X$. The ***`aggregation predicate`*** is a bool valued term over constant properties of the current and the (possibly) included object. A new name for the included population may be introduced with the optional **as**-clause. For the included subpopulation $\texttt{Class}_Y$, the following two conditions hold : $\texttt{Class}_Y \subseteq \texttt{Class}_X$ and $\textbf{current}(|\texttt{Class}_Y|) \subseteq \textbf{current}(|\texttt{Class}_X|)$. With $\textbf{current}(|\texttt{C}|)$ we denote the set of object identifiers associated with instances with a non empty trace, i.e. the objects where a birth event and no death event occurred so far.

The subobjects included in the actual templates can be referenced by *indexing*. Indexing is accomplished by prefixing the attribute and event symbols with the (possibly renamed) object or class name using the dot notation : `PartName.symbol` . In the latter case, an identifier data type valued term must be supplied to select the actual instance : `PartName(`***`identifier term`***`).symbol`. If there are no name clashes, the former case can be reduced to the attribute or event name of the inherited object. Note that with subpopulation inclusion, indexing can not be avoided since there is no other way to select the proper instance to be referred to.

The last point mentioned leads to another syntactical construct, renaming of attribute, event and component symbols :

including
  `single instance or subpopulation including`
    renaming
      `Signature_Symbol` **by** `New_Signature_Symbol`;
      ...
  ...

Renaming can automatically be translated to full *path expressions* denoting the actual including path for the signature component. This can be done at compile time. For included subpopulation of classes, renaming can be seen as syntactic sugar to enhance the readability of specifications. The user of the language should use the **renaming** construct carefully.

As an example let us describe a bank object including a clock object (with properties like `Time`, `Date` etc.) and a set of account objects with identification (natural numbers) from 100000 to 999999. This example also shows the inclusion of a class (sub)population into a single object :

```
object class Bank
  template
    data types set(|Acct|),nat,UpdateType,money;
    including
      instance Clock renaming Date by Today;
      A in Account
        where A.No>=100000 and A.No<=999999;
        as Acct;
    attributes
      TheAccounts:set(|Acct|);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      process_TA(in Acc:nat,in Type:UpdateType,in Amount:money, Time:date);
      TA_failed(in n:nat);
    valuation
      variables n:nat;
      [open_account(n)]TheAccounts = insert(Acct(n),TheAccounts);
      [close_account(n)]TheAccounts = remove(Acct(n),TheAccounts);
    behavior
      permissions
        variables n,p,p1:nat; m,m1:money; t,t1:UpdateType; d,d1:date;
        { not in(Acct(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
        { not sometime after(process_TA(n,t,m,d)) since last
          (after(TA_failed(n)) or after(TA_OK(n))) and
          d1 = Today } process_TA(n,t1,m1,d1);
        ...
end object Bank
```

The including section consists of two including clauses. The first clause denotes the inclusion of a single object `Clock`. This object may be used to timestamp bank transactions. Inside the bank template the current date is referred to with the attribute symbol `Today`.

The second including clause denotes the inclusion of a proper subset of possible `Account` objects. The object variable `A` can be used to refer to constant properties of `Account` objects. The component selector `A.No` used in the selection condition is a *constant* property of account objects and may therefore be used in the selection condition. A reference to variable object properties e.g. attribute values is neither allowed nor possible in this context (see below).

The attribute `Today` is used in the last permission to timestamp bank transactions. This permission enables only those events `process_TA(n,`t_1`,`m_1`,`d_1`)` instantiated with the current value of `Today` (among the other parameters).

5.1.1 Communication Inside Complex Objects

The components of a complex object may somehow interact with each other. The bank example above only states that the set of accounts and the single object clock is included in the bank. Up to now the only interaction specified is the access to the attribute `Today` of the `Clock` instance :

```
{ not sometime after(process_TA(n,t,m,d)) since last
  (after(TA_failed(n)) or after(TA_OK(n))) and
  d1 = Today } process_TA(n,t1,m1,d1);
```

This access is only a read access to some attribute of the component object. Recall that we may only alter attribute values with the occurrence of *local* events, that is, for example the `Balance` attribute of some account may only be changed by events local to this `Account` object.

The mechanism to synchronize life cycles of objects is *calling* (cf. Section 2.5). The example must be extended to show possible interaction between the bank object and the accounts. Since the aggregated object is constructed from a set of accounts, we must use an operation `Acct:nat`→`|Acct|` to generate identifiers for included objects. The class name `Acct` may be used inside the bank object as if it were defined as a global class name. Note that **current**(`|Acct|`) ⊆ **current**(`|Account|`), i.e. for all X ϵ `|Account|` the aggregation predicate holds iff X ϵ **current**(`|Acct|`).

Identifiers are only *references* to objects, thus a second operation taking an object identifier and yielding the object itself is needed. Since there is no ambiguity – the accounts are *subobjects* of the bank – we could leave out the second operation, assuming that `Acct(n)` delivers an object instance. Thus the current balance of the account number 123456 may be referred to with the expression: `Acct(123456).balance`.

In the following specification fragment we will complete the specification of the bank with an additional section, the **interaction** section following the **behavior** section :

```
interactions
  variables t:UpdateType; m:money; n:nat; d:date;
  open_account(n) >> Acct(n).open;
  close_account(n) >> Acct(n).close;
  process_TA(n,t,m,d) >> Acct(n).accept_TA(t,m,d);
  { sometime after(process_TA(n,t,m,d) } ==>
              Acct(n).withdrawal(m) >> TA_OK(n);
  { sometime after(process_TA(n,t,m,d) } ==>
              Acct(n).deposit(m) >> TA_OK(n);
```

```
{ sometime after(process_TA(n,t,m,d) } ==>
              Acct(n).update_failed >> TA_failed(n)
```

As an example of communication between the complex object and one of its parts see the following calling expression :

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

In this case for example the `Bank` event `open_account(n)` calls the `Acct` event `open` in component object `Acct(n)`. Thus the creation and destruction of `Acct` instances is triggered by the `Bank` object. Nevertheless, the syntactic structure of the bank object is static. Static aggregation describes the *potential* object composition at specification time. Possibly there are component objects that have an empty trace (cf. Section 4.3.1).

Sometimes we want to restrict calling according to the current state or the history of the object. Conditional calling described in the interaction section of the above `Bank`-template (intuitively) only occurs if the condition evaluates to true in the current state :

```
{ sometime after(process_TA(n,t,m,d) } ==>
              Acct(n).withdrawal(m) >> TA_OK(n);
```

that is, an occurrence of event `Acct(n).withdrawal(m)` calls for the event `TA_OK(n)` only in some state after the event `process_TA(n,t,m,d)` occurred.

We will now give a few words on the semantics. The signature of the `Bank` object is obtained by disjoint union of the local signature of the `Bank` (without the accounts) and the local signature of `Account`. Since we included a set of objects here we have to deal with indexed symbols: To refer to an included object instance of a class we have to supply the (possibly renamed) class name and a data type value used to identify a class instance. In the case of an `account` instance we may write `Account(123456)` to denote the instance with the key value 123456.

But there is a little bit more to say about semantic inheritance. We must also deal with the object model inherited, that is, the effects of this construction on the process and observations of the participating objects.

For all possible life cycle and the associated observations of attribute values of the complex object we state the following properties :

- For any life cycle of the complex object we obtain a valid life cycle of a component object if we restrict this life cycle to the events local to the component. This means, that a valid life cycle of the complex object *contains* the life cycle of a component as a "subprocess". Thus, the aggregation does not change the behavior of the parts.

  For example take one possible life cycle λ of the `Bank` object. This life cycle may contain event occurrences from all accounts. Consider now, that we only want to observe the account associated with the identifier `Account(123456)`. Therefore we remove all events in λ that are not local to this account and we obtain a valid life cycle of `Account(123456)`.

- A somewhat similar condition must hold for the observation of the parts. An observation for a given life cycle denotes the binding of attributes to associated data sort values.

  Suppose we have an observation of the complex object derived from one of its possible life cycles λ. Then we take into account only those attributes, local to some subobject say again the object associated with the identifier `Account(123456)`. We require this (subobject)-observation to be the same as the observation of the part object derived from the life cycle of the complex object restricted to the events of the part. That is, an observation of the part does not change in the context of the composite object.

A more detailed formal treatment of embedding relationship and calling can be found in [EGS90, SJ91, HJ91].

To guarantee the stated properties, we must not have valuation rules for inherited attributes but the set of possible life cycles of the embedded objects may be further constrained if it is part of a complex object (cf. page 54). This is expressed in the condition that there *exists* a life cycle of a part object if we constrain our view to events of the part but there *may be* other life cycles in the part that are not allowed in the context of the composition. This view stems from the observation, that an object, communicating with its environment, may have a restricted but not a new behavior.

Communication is possible using calling or sharing expressions. This way calling and sharing expressions imply such further restrictions on the behavior of the component objects: some of their life cycles may not be possible in the context of the enclosing complex object and the specified communication structure.

Object embedding is used as the basic mechanism to describe the semantics of other high level language constructs throughout the rest of this paper. In our framework it is the only way to relate different objects on the semantic level.

5.2 Dynamic Complex Objects

The use of dynamic complex objects allows for high level description of object composition. Components may be specified as *single components* as well as *sets* or *lists* of components. They have a life of their own and may be shared by other objects as well.

With the specification of a dynamic complex object denoted with the keyword **components** we have implicitly defined events to update the composition. Additionally we have implicit attributes to observe the composite object. For example for a set of component objects we have events to insert and delete objects and an attribute to observe set-membership. For lists of objects we have events to append and to remove objects and for single objects there are events to add and remove them as well as an attribute to test if the component is assigned. A complete list of implicitly generated events and attributes for complex object constructions in the current language version is supplied at the end of this section.

This view of complex objects is operational instead of declarative like the concept of static aggregation. In another terminology we can say that static aggregation is *value based* whereas dynamic aggregation is *event driven*. Before we introduce how dynamic aggregation fits in our semantic framework of object embedding, we will give some specification fragments and examples as syntactic patterns :

```
template
  components
    SingleComponentName₁(dt₁,...,dtₙ₁):ComponentObject;
    SingleComponentName₂(dt₁,...,dtₙ₂):ComponentClass;
    SetComponentName(dt₁,...,dtₙ₃):SET(Component_Class_S);
    ListComponentName(dt₁,...,dtₙ₄):LIST(Component_Class_L);
    ...
  ...
```

The four different component specifications in the above example show the definition of a single instance as component (either a single object or an instance selected from a class), a set of class instance components and a list of class instance components. Note that the component symbols may be indexed.

We now give an example of another bank specification, including the accounts as a set component and additionally a single instance of class `Person` denoting the manager of the bank. Some parts of the bank specification are omitted since we want to deal with the important parts of a component specification :

```
object Bank
  template
    data types nat,|ATM|,UpdateType,money,|CashCard|;
    components
      Manager:Person;
      Accounts:SET(Account);
    events
      birth establish;
      death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      card_request(in AcctNo:nat);
      card_return(in C:|CashCard|);
      ...
    behavior
      permissions
        variables n,p,p1:nat; m,m1:money; t,t1:UpdateType; atm:|ATM|;
        { not Accounts.IN(Account(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
        ...
      interaction
        variables t:UpdateType; m:money; n:nat; C:|CashCard|;
```

```
        open_account(n) >> Accounts.INSERT(Account(n));
        open_account(n) >> Account(Account(n)).open;
        close_account(n) >> Accounts.REMOVE(Account(n));
        close_account(n) >> Account(Account(n)).close;
        { Accounts.IN(Account(n)) and not in(C,Account(n).Cards) } ==>
          card_request(n) >> Accounts(Account(n)).assign_card(C);
        card_return(C) >>
          Accounts(Account(CashCard(C).ForAccount)).cancel_card(C);
        ...
end object Bank;
```

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

For the behavior of the complex object the communication inside the complex object must be specified. Communication can take place between the component objects and between the complex object and the component objects. See for example the clauses :

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

which state, that every time an account identified by the natural number `n` is opened, it becomes a member of the set of components and the event `open` is called in the corresponding object `Account(n)`. Note that the event `open_account` in the `Bank` object may only occur if the expression **not** `Accounts.IN(|Account|)` evaluates to true. A calling in the opposite direction – from the component to the complex object – is also supported in this framework. Even communication between components can be accomplished by referring to components at both sides of interaction equations.

5.2.1 Implicitly Generated Signature Components

As mentioned above, with the definition of components, implicitly generated events and attributes are added to the signature of the base object. These special events are used to update the composition, that is to alter the structure of the complex object. Similar the attributes are used to observe this structure.

The simplest components are *single objects*, introduced by supplying the class name `ClassN` (short for `ClassName`) as the component type. We refer to components with `CompN` (short for `CompName`). Assume the following specification :

```
components
  CompN:ClassN;
```

Then, the following attributes are generated for single component instances of some Class `ClassN` (the same applies for single object components) :

`CompN.DEFINED:bool` is **true** if the component is defined, **false** otherwise.

`CompN.ID:|ClassN|` denotes the identifier data type value associated with the included instance of `ClassN`.

Two events are generated to initialize and reset the component :

`CompN.ASSIGN(|ClassN|)` used to assign the instance. After the occurrence of the event `CompN.ASSIGN(C)` for a `C`ϵ`|ClassN|`, the value of the attribute `CompN.ID:|ClassN|` is `C` and `CompN.DEFINED:bool` is bound to **true**.

`CompN.REMOVE(|ClassN|)` used to remove the instance. This is the dual operation to `ASSIGN` with obvious semantics.

The event `CompN.ASSIGN(|ClassN|)` can always be used to alter the included object instance. The event `CompN.REMOVE(|ClassN|)` restores the state of the composition to the state where no component is assigned. One may wonder why this operation is necessary, but see for example the complex object `FootballTeam` with the single object component `Trainer` – here the remove operation is (obviously) necessary. The name `CompN` can be used to refer to the component as subobject.

The second possible component type are *set valued* components defined by instantiating the set component constructor **SET**. Assume the following specification :

components
`CompN:`**SET**`(ClassN);`

The following attributes are generated for set valued components

`CompN.EMPTY:bool` is **true** if the component set is empty, **false** otherwise.

`CompN.CARD:nat` denotes the cardinality of the component set.

`CompN.ID_SET:`**set**`(|ClassN|)` describes a set containing all object identifiers of the objects currently in the component set. Note that we used the parameterized set constructor for data types to denote the codomain of this attribute.

`CompN.IN(|ClassN|):bool` is **true** if the parameter is element of `CompN.ID_SET`, **false** otherwise.

For manipulating the current set of objects included in the complex object, the following events are generated for set valued components :

`CompN.INSERT(|ClassN|)` the object associated with the parameter identifier is included in the component set. Note that it is possible that this object does not exist at the moment this event occurs. A birth event may be induced by the complex object.

`CompN.REMOVE(|ClassN|)` the dual event to `INSERT`.

Access to the objects of the component set is possible by supplying the name `CompN` and an identifier for the included class. The syntax for this access is the same as introduced for explicit embedding of subobjects. Conceptually there is no difference to the explicit case as we will see in the next section.

The last component type constructor mentioned in this report is the list constructor. Lists of components are defined by instantiating the list component constructor **LIST**. Assume that we specified :

components
`CompN:`**LIST**`(ClassN);`

The following attributes are generated for list valued components :

`CompN.EMPTY:bool` is **true** if the component list is empty, **false** otherwise.

`CompN.LENGTH:nat` denotes the length of the current component list.

`CompN.ID(nat):|ClassN|` the object identifier associated with the the i-th list component. This attribute is only defined for i <= `CompN.LENGTH`.

`CompN.ID_LIST:`**list**`(|ClassN|)` describes a list containing all object identifiers of the objects of the component list (including duplicates).

`CompN.ID_SET:`**set**`(|ClassN|)` describes a set containing all object identifiers of the objects currently in the component list. Note that duplicates are eliminated.

For manipulating the current list of objects included in the complex object, the following events are generated for list valued components :

`CompN.APPEND(|ClassN|)` the object associated with the parameter identifier is appended to the component list.

`CompN.SET(nat,|ClassN|)` the i-th element of the component list is set to the object associated with the object identifier denoted by the second parameter. This event may only occur for i <= `CompN.LENGTH` (an implicit permission).

`CompN.REMOVE_FIRST` the object on top of the list is removed from the component list.

`CompN.REMOVE(|ClassN|)` the object associated with the parameter object identifier is removed from the component list. Here we can assume an implicitly generated permission: **in**(C,`CompN.ID_SET`) for an actual parameter `C` ϵ `|ClassN|`. Since we may have the same objects at several positions in the list, all of them are deleted with this operation.

Access to the list of objects is possible with the following implicitly generated components :

`CompN.FIRST:ClassN` assuming that the first element of a list is a distinguished element, we supplied the operation `FIRST` yielding an instance of `ClassN`.

`CompN.ELEMENT(nat):ClassN` i-th element (class instance of `ClassN`) of the list. Only valid for i <= `CompN.LENGTH`. This operation is necessary since a list of objects may contain duplicates.

The list of component constructors supplied here is by no means exhaustive but merely a useful set for the introduced component types. Other possible sets of constructors may include bags and queues. This language version is restricted to the above mentioned constructors for single objects, sets and lists of included objects.

5.2.2 Semantics of Dynamic Object Aggregation

Dynamic complex object may not be described with safe object inclusion directly, since this concept only allows for static object composition. Nevertheless, we may introduce a two level translation to describe the semantics of dynamic complex objects [JHSS91].

On the *semantic level* this is done in the following way: for each possible composition the corresponding semantic object is obtained by including the objects that are part of *this* composition. For example the current manager (person)-object of the bank and all account objects currently in the set of components (see example below).

Whenever the composition changes, that is a component is inserted, deleted etc. a new semantic object is obtained in the same way, the old one may be forgotton. The observable properties of the unchanged components remain the same in the new object. Note that also the identity of the *conceptual object* remains the same. This two level construction is used *only* to describe the semantics of dynamic complex objects.

Consider for example the `Bank` object containing a set of `Accounts`. Just to explain the idea, we may construct a *semantic object class* `ComplexBank` (according to the object `Bank` introduced above) identified by instances of `|Person|` and **set**`(|Account|)` :

```
object class ComplexBank_Semantics
  identification
    Manager_ID:|Person|;
    Accounts_ID:set(|Account|);
```

```
  template
    including
      instance Manager_ID in Person as Manager;
      A in Account where in(A.Id,Accounts_ID) as Accounts;
    ...
end object class ComplexBank_Semantics
```

The first including clause denotes the inclusion of the instance of `Person` associated with the value of the key attribute `Manager_ID`. The second clause includes the set of object instances from class `Account` for all object identifiers in the set `Accounts_ID`. The expression `A.Id` denotes the identification of some object instance `A`ϵ`Account`. The operation **in** is an operation defined for the parameterized data type **set**.

For each semantic object of this class its (syntactical) structure is defined by embedding the `Person` object and all `Accounts` that are elements of its (external) identifier. Changing the composition thus yields a new semantic object instance. We require this new semantic instance to have the same observable state as the old instance. The old semantic instance can be forgotton.

The rest of the specification, especially the translation of the generated attributes and events is straightforward and therefore omitted.

5.3 Disjoint Complex Objects

Disjoint complex objects may not share components with other objects. The parts of disjoint object structures are strongly dependent on the composition. They cannot exist independently from the aggregation. Access to the subobjects is only possible from the enclosing object.

In TROLL, disjoint complex objects are described using *subtemplates*. Subtemplates are a means for structuring the specification, i.e. we can specify nested structures like chapters, sections and subsections etc. in some book object. Conceptually, subtemplates describe local subobjects not usable outside the complex object. A subtemplate is a complete template description. We may specify a subtemplate in the complex object or reuse a previously specified template.

The subobjects are specified together with attributes and events of the complex object and they have a local name. The name can optionally be supplied with one ore more parameters – using the same syntax as for attributes – describing several subobjects with the same structure. See for example the following template specification to be used as a subobject description :

```
template subTemplateName
  attributes
    Att1:attType1;
    Att2:attType2;
    ...;
```

```
  events
    birth bEvt;
    death dEvt;
    event_1;
    ...;
  ...;
end template subTemplateName
```

The next template description uses the above template as subtemplates. Note that we show only the template specification since subtemplates can be specified for single objects and object classes :

```
  template
    subtemplates
      SubTemp(dt_1,...,dt_n):subTemplateName;
      ...
    ...;
```

Attribute and event symbols of the subobject can be uniquely identified by prefixing them with the declared subtemplate name `SubTemp` and possible parameters according to the data types $\mathtt{dt}_1,\ldots,\mathtt{dt}_n$. For example we can access the attribute $\mathtt{Att}_1$ (of the subtemplate) writing :

$\mathtt{SubTemp(p_1,\ldots,p_n).Att_1}$

where $\mathtt{p}_1,\ldots,\mathtt{p}_n$ denote values of the data types $\mathtt{dt}_1,\ldots,\mathtt{dt}_n$. We may go one step further and assume, that we have internally usable *template identifier spaces* as in the global case. Then we have the data type `|SubTemp|` defined using the tuple constructor (see also Section 4.1.1) :

$\mathtt{|SubTemp|} = \mathbf{tuple}(\mathtt{dt}_1,\ldots,\mathtt{dt}_l)$

We can refer to the component supplying an (internal) identifier of the data type, e.g. `|SubTemp|` to the operation `SubTemp` yielding the subobject itself. The special case of non-parameterized subtemplate names is implicitly included in this definition, assuming, that the identification space reduces to the simple name – an analogy to the reference to single objects that are uniquely identified using the object name.

Some comments to the proposed constructor for disjoint complex objects are in order. For non-disjoint complex objects the occurrence of birth and death events may be induced by several objects possibly sharing the same components. In the case of disjoint complex objects, the birth of a subobject may only be induced by the parent or enclosing object. The parent object can be titled as the root of the disjoint object structure.

The signature of a disjoint complex object is fixed, i.e. there is no way to change it as in the case of dynamic complex objects. However not all of the subobjects must exist at a given point in time. Only the (sub)objects that have a non empty life cycle are existing.

Let us now turn to our example bank world. In this context we specify a class of banks each containing a set of divisions. The first step is to model the `division` template :

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

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

```
object class Bank
  data types string;
  identification
    name:string;
  template
    data types string,|Person|,...;
    subtemplates
      Divisions(string):division;
    ...
    events
      newDivisionManager(|Division|,|Person|);
      ...
    interaction
      variables D:string, P:|Person|;
      newDivisionManager(D,P) >> Divisions(D).setManager(P);
      ...
end object class Bank;
```

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

```
newDivisionManager(D,P) >> Division(D).setManager(P);
```

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

Note that different `Bank` instances can have the same names for their divisions. The names of the subobjects are local to the enclosing object as are the objects themselves. They may not be shared with other bank instances. The (sub)object identifier data type is only visible inside the bank.

5.3.1 Semantics of Subtemplate Specifications

One way to explain the semantics is static aggregation. Objects specified using subtemplates may be transformed to simply structured objects. Therefore we may generate an object class description employing the identification space of the original object together with the internal template identification. The template of this new object class is obtained from the subtemplate specification. We do not want to go into details of this construction but give a simple example. The `Bank` object may be transformed to the object classes `Division` :

```
object class Division
  data types |Bank|,string;
  identification
    Bank:|Bank|;
    subObjectID:string;
  template division
end object class Division;
```

and a slightly modified object class `Bank` :

```
object class Bank_Semantics
  data types string;
  identification
    name:string;
  template
    including
      D in Division where D.id.Bank = SELF.id;
  ...
end object class Bank_Semantics;
```

The last two specifications are not visible to the designer. Each object instance of the class `Division` may be shared only by exactly one `Bank` class object. This property must be guaranteed with the including condition in the modified `Bank` object. The condition is again an example for using predicates to describe the aggregation of static complex objects, i.e. using the basic mechanism of explicit object inclusion to describe the semantics of high level language constructs.

5.4 Syntax of Complex Object Specification

The following production rules generate the syntax for complex objects :

```
<including_section>   ::= including
                             <including_clause_seq>';'

<including_clause>    ::= <ovar_id> in <class_id>
                             [where <formula>] [as <class_id>]
                             [<renaming>]
                          | instance <data_term> in <class_id> [as <ob_id>]
                             [<renaming>]
                          | instance <ob_id> [as <ob_id>]
                             [<renaming>]

<renaming>            ::= renaming
                             <renaming_clause_seq>';'

<renaming_clause>     ::= <attribute_id> by <attribute_id>
                          | <event_id> by <event_id>
                          | <comp_id> by <comp_id>

<components_section>  ::= components
                             <comp_def_seq>';'

<comp_def>            ::= <comp_id> ':' <comp_domain>
                          | <comp_id>'('<domain_list>')' ':' <comp_domain>

<comp_domain>         ::= <ob_id>
                          | <class_id>
                          | LIST'('<class_id>')'
                          | SET'('<class_id>')'

<sub_temp_section>    ::= subtemplates
                             <sub_temp_def_seq>';'

<sub_temp_def>        ::= <subtemplate_id>':' <template_id>
                          | <subtemplate_id>(<domain_list>) ':'<template_id>
```

6 Aspects

Up to now we only talked about object templates, individual objects object identities and object classes. But an object is more than a template with an attached identity. We can look at an object from different perspectives, able to see special facets or concentrating on special properties. This way, we can speak about *aspects* of objects [ES91, SJE91].

Different aspects may be specified separatly in TROLL. We may start with some general information about a real world entity and construct a basic object description for this entity. For example a person may be described by its name, birthday etc.

As a next step we concentrate on special aspects of this entity and construct specialized object descriptions. These aspects may be either static properties: a person can be looked at as male or female, or dynamic properties: a person may show specialized behavior for some time in its life e.g. (s)he may become a customer, car-driver etc. The former case is modeled as a *specialization* of person, the latter as a *role* of person.

On the other hand we may have started the modeling process in the same manner as stated above but sometime realize that we want to look at some entities from a more general point of view, i.e. we state that these entities have some *common* behavior and structure. This direction is known as *generalization* and can be characterized as a restricted, more general aspect in our framework.

Modeling real world entities using specialization, roles and generalization imposes a structure on the specification. Structuring mechanisms in object oriented languages are usually described using *inheritance hierarchies.* TROLL supports two different levels of inheritance :

- *syntactic inheritance*: denotes reuse of specification code. This means, that attribute and event symbols of the parent object may be used in its ancestors. For example the birthdate attribute is known by a role or specialization of person.

- *semantic inheritance*: denotes reuse of the objects itself, meaning that the behavior, i.e. the dynamic part of a parent object is known by its ancestors. For example role and specialized objects can refer to attribute *values* and the *current life* of their parent objects.

The next three sections introduce the main abstraction mechanism supported in TROLL. Roles and specialization as temporary and static aspects of objects, generalization as hiding special properties.

6.1 Roles

The concept of *role* describes a dynamic (temporary) specialization of objects, i.e. a special view or aspect [ES91] of objects [Wie90]. As an example consider the roles customer or employee of persons. For instance, a given person may be looked at as an employee or a customer for a given period of time. Even more, the person may play different roles at the same time.

Roles may be introduced for single objects and for class types. Furthermore we may have several parent classes, i.e. multiple inheritance. Thus, we have the following syntactical patterns for the specification of role objects and role class types :

object [**class**] `ABC`
  role of X_0 , ... , X_n;
  template
    ...
end object [**class**] `ABC`

For the role object specification we require the following conditions to hold :

- The role object optionally introduces a set of new attribute symbols. Since the inherited attribute symbols can be identified by qualifying their names with the name of the base object/class, we have no name conflicts. If there are no name clashes, the qualification can obviously be omitted. Overriding of attribute and event symbols is not possible.

- The role object has birth and optional death events. Birth and death events induce the creation and destruction of a role. Thus an object may play a role several times. We require the birth and death events either to be inherited from the parent of the role or be locally defined. The two mentioned possibilities reflect that the creation may be under local control (the first case) or due to communication with the environment (the second case). As usual we require the birth event to be the first event of a life cycle and a death event the last event of the life cycle of a role object. The death event is optional, that is, an object may play a role for the rest of its life.

- Valuation rules may be specified as usual. They are restricted to be applied only to *locally* defined attributes. Otherwise the update of an inherited attribute violates the locality of attribute updates. On the other hand, valuation rules may contain inherited events. The role object may suffer from events occurring in the base object and change state according to the state changes in the parent object. As an example consider the `birthday`-event of a `person` object which may have effects on the role `employee` of `person` (for example a higher salary!).

- Update of attribute values of the parent object is still possible. For this purpose, inherited events may be used by calling. Note that the locality of attribute updates may not be violated in this way since it is definitely the only way to change the state of the parent object. Otherwise, the encapsulation – unity of process and observation in our sense – of the parent object is not guaranteed.

- Attribute values from inherited objects may be read, they are visible. All data terms defined in the role object may therefore contain inherited attribute symbols.

- Constraints may be specified for local and inherited attributes. An object playing a role can thus have a restricted behaviour. Some attribute updates may be allowed, others, violating the newly specified constraints, are forbidden. For example, an employee must not be older than 65 years.

- For permissions we can state the same as for constraints. The possible occurrences of events inherited from the parent object may be further constrained during the time between the birth and the death of the role object. Note, that the possibilities for defining new constraints and permissions for inherited attributes and events respectively, induce a necessary translation process during implementation. We will not deal with this topic in this paper.

- For explicit process specification the whole event alphabet of the role object, including inherited events, can be used. Such processes may be seen as special constraints on life cycles, even for the parent object. It should be noted, that contradictionary constraints may be specified but lead to 'senseless' specifications.

After this general requirements, let us consider the following example. Assume, that we specified a template `person` having the usual attributes like `Name`, `Birthdate`, `Address`, `Age` etc. In a bank world, the person may play the role of a customer. Therefore we include an event inducing the start playing the role `BankCustomer` :

```
object class Person
  identification
    Name:string; ...;
  template
    ...
    events
      ...
      bc_bank_customer
      ...
end object class Person
```

The role object `BankCustomer` is described by the following specification :

```
object class BankCustomer
  role of Person;
  template
    data types nat,set(nat);
    attributes
```

```
    Accts:set(nat);
  events
    birth Person.bc_bank_customer;
    death cancel;
    open_account(out Acct:nat);
    close_account(in Acct:nat);
  constraints
    Age >= 14;
  valuation
    variables n:nat;
    [bc_bank_customer]Accts = emptyset();
    [open_account(n)]Accts = insert(n,Accts);
    { in(n,Accts) } ==> [close_account(n)]Accts = remove(n,Accts);
  behavior
    permissions
      variables n:nat;
      { sometime(after(open_account(n))) } close_account(n);
end object class BankCustomer;
```

The identification space of the role object class is the same as for the parent class: `|BankCustomer|` = `|Person|` but the actual population of the role class is a time varying set of objects depending on the occurrence of birth and death events of the role class.

The role introduces new attribute and event symbols extending the signature of the original person object. An (inherited) event `Person.bc_bank_customer` induces the object `Person` playing the role `BankCustomer` and creates the role object. Clearly, the newly introduced events for opening and closing accounts, make sense only for bank customers. The sole attribute `Accts` register the currently available accounts for this person. According to the above mentioned requirements, this attribute can be updated after occurrences of local *and* inherited events, i.e. we may have additional valuation rules for inherited events.

In the **constraints**-section we refer to an inherited property. Bank customers must be older than 13 years (just for the sake of the example). Referring to the age of the bank customer (as a person) may be written also as `Person.Age`, explicitly qualifying the identifier with the source object name. In the absence of ambiguities we allow the shorter notation. Note that the mentioned constraint may also prevent the birth event of the role to occur.

One may wonder why we specified the birth event of the role to be inherited from the parent object. Here we want to show that the start of playing a role may be induced by the parent object, that is, the birth event is not a 'new' event but comes from the parent and is enriched in its semantics: it creates the role object. On the other hand, the death of the role is induced by the role itself and thus newly introduced in the template of `BankCustomer`, that is, `Person` *as* `BankCustomer` induces the cancel event.

The semantics of role specification is given by including the parent object into the role object. Including denotes enrichment of the object signature by the parent object signature (syntactic inheritance) and including the actual object properties of the parent objects (semantic inheritance) like introduced in the beginning of this section. See for example the semantic role object class for the above person role bank customer :

```
object class BankCustomer_Semantics
  identification
    Name:string; ...;
  template
    including
      instance Person(Name) in Person as Person;
    ...
end object class BankCustomer_Semantics
```

Following this line, aspects of objects make up complex objects. The life cycles of the parent and role objects can be related by calling (see Chapter 2.5).

Recall that multiple inheritance can also be formulated in this context. On the level of the role class, we have an intersection of the class populations of the parent classes. Thus we require, that all parent classes of a role inherit from some shared object class located somewhere in the inheritance chain.

6.2 Specialization

Another case of an object aspect is *specialization*. A specialization is a special case of an object role, namely a role played for the whole life of an object. Therefore, we do not have birth and death events for a specialization. The general requirements introduced for role object specification also apply to the specification of specialized objects.

The concept leads to a structured specification, factoring out the common properties in the description of the base objects, describing the special properties in the specialized objects. Therefore the specialization construct may be frequently used in a system specification. In conceptual modeling this concept is known under the term *is-a* or *kind-of* relationship. In this context, roles may be looked upon as event driven whereas specialization is value based.

With the birth of an object we must be able to decide to which class the object belongs. This is done using a specialization condition, a predicate over the constant attributes and identification components. Restricting the specialization predicate to constant properties of the object implies that no change of object classes is possible for a specialized object.

As in the case of object roles, an object may belong to different specialized classes of objects at the same time, it may be a specialization of several objects/classes and specialization may be specified for single objects and class types. We have the following syntactical patterns for the description of class specialization :

```
object class Name
  specializing from X0 , ... , Xn;
    where specialization predicate
  template
```

```
    ...
end object class Name
```

For specialization, as a special case of a role, the same conditions as stated above apply. The only exception is the absence of birth and death events for specialized objects. The **specializing from**-construct may also be applied to single objects. Naturally there is no specialization predicate :

```
object Name
  specializing from X0 , ... , Xn;
  template
    ...
end object Name
```

As an example, consider two specializations of our `Account`: A `SavingsAccount` and a `CheckingAccount`. Both have special properties in addition to their common ones which are described in the base template `Account`. A special aspect of savings accounts is that the bank (usually) pays interest for it. Furthermore, the balance of a savings account is always non-negative, i.e. the credit limit is 0. Every once in a while, interest is paid (computed by some interesting function) and is added to the balance. Please note that we may not update directly the attribute `Balance` of the `Account` – we have to *call* explicitly the event `deposit` of the `Account` to update the `Balance`. The specialization class `SavingsAccount` is specified as follows :

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

```
                    (days_between(d,last_interest_paid)/360) *
                    (Balance * interest_rate / 100)
                  );
end object class SavingsAccount;
```

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

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

With the creation of an Account (once and for all the life of the object) the belonging to one of the classes CheckingAccount or SavingsAccount is determined. The (constant) attribute Type can only have one of the values checking or saving, in this case the object can belong to only one of the classes.

Note that we do not impose further restriction on the life cycles directly – cashcards may be assigned anytime, and for example the credit limit may be updated anytime due to the occurrence of the event new_credit_limit inherited from the base object. On the other hand we have an implicit restriction in the case of the SavingsAccount, a restriction of the possible attribute values and therefore restricted update events for some attributes. For example the event new_credit_limit for SavingsAccounts may only have the integer parameter 0.

Since specialization is only a special case of a role, namely a role played for the whole life of the object, the semantics of specialized class specification can be directly transferred from the general case of object roles.

6.3 Generalization

Generalization is used to construct a uniform view on (several) different base classes. Basically, generalization is similar to the second is-a concept specialization. The difference is that it goes in the other direction: Using specialization, one defines a number of subclasses for a base class whereas by generalization, one defines a common superclass for a number of subclasses. In contrast to specialization with several input classes generalization is based on disjoint union of input class populations.

Consider for example the very popular example of legal persons being either 'real' persons or companies. The issue of specialization vs. generalization has been discussed in the semantic data modeling community for some time, see for example [SS77, UD86, HK87, PM88].

The syntactic pattern for generalized classes can be described with the following fragment:

```
generalization class ABC
  of X_0, ... , X_n;
    template
      data types ...;
      components
            common components ;
      attributes
            common attributes ;
      events
            common events ;
      renaming
        case
          SELF in X_0 :
            ... ;
          ...
          SELF in X_n :
            ... ;
end generalization class ABC;
```

For generalized classes we require the following:

- The identification space of the generalized class is derived from the disjoint union of the input class id spaces $|X_0|$, ..., $|X_n|$.
- The identifier SELF denotes the *current* instance. That is, we can decide to which object class the current instance belongs. This information is used in the renaming section.
- We only allow for renaming component, attribute and event symbols.
- Every component/attribute/event of the generalization class must have a source component/attribute/event in all input classes. All components/attributes/events have counterparts in the input classes.

As an example concerning the bank world introduced above, let us suppose a bank wants to treat persons and companies in the same way. For this purpose, the specification of `BankCustomers` and `Company` (which is not defined here but can be understood from the context) may be generalized to the specification of `Customers` :

```
generalization class Customer
  of BankCustomer, Company;
  template
    data types nat,set(nat),string;
    attributes
      Accts:set(nat);
      Address:string;
    events
      open_acct(out Acct:nat);
      close_acct(in Acct:nat);
    renaming
      case
        SELF in BankCustomer :
          Accts by Accts;
          Address by Address;
          open_account by open_acct;
          close_account by close_acct;
        SELF in Company :
          Accts by Accts;
          Location by Address;
          new_account by open_accnt;
          delete_account by close_acct;
end generalization class Customer;
```

The specification of a generalization starts with the listing of the *input* classes. In the template section, the signature of the generalization template is defined in the usual way. In the renaming section component, attribute and event symbols of the corresponding input classes are renamed according to the signature of the generalized object. In this section the identifier `SELF` denotes the object itself, i.e. either an instance of `Company` or an instance of `BankCustomer`.

Note that the semantics of this simple kind of generalization can be depicted as a specialized (common) *interface* for several base classes which is reflected in the keyword selection **generalization class** (resembling **interface class** introduced in the next chapter). More sophisticated language versions — to come up in the future — may include derivation relationships between attributes of the generalization class and the base classes as well as arbitrary process calling for the newly introduced events.

Describing the semantics of generalization in terms of the language constructs introduced so far is somewhat subtle. In fact, the parameterized data type constructors **set**, **list** and **tuple** are not sufficient to describe the name space of a generalization class.

In the case of specialization, the *current* sets of identifiers for roles and specializations are subsets of the original name spaces. For generalization, we work in the opposite direction. Thus, the name space of a generalized class is the *union* of the *input* name spaces.

Compared to the case of specialization with several input classes, where we have only identifiers of *one* identifier type, we must deal with identifiers of several types in the case of generalization. The parameterized data type constructor **union** is not included in this language version but used only to describe the semantics of generalization. Suppose we specified the generalized class `Customer` described above. Then we can define the semantics of this specification with the following class and interface descriptions :

```
object class Customer_Semantics
  identification
    Id:union(|BankCustomer|,|Company|)
  template
    including
      B in BankCustomer where B.id = Id as BC;
      C in Company where C.id = Id as COM;
    attributes
      derived Accts:set(nat);
      derived Address:string;
    events
      derived open_acct(out Acct:nat);
      derived close_acct(in Acct:nat);
    derivation
      variables n:nat;
      { in(Id,|BankCustomer|) } ==> Accts = BC(Id).Accts;
      { in(Id,|Company|) } ==> Accts = COM(Id).Accts;
      ...
      { in(Id,|BankCustomer|) } ==> open_acct(n) is BC(Id).open_account(n);
      { in(Id,|Company|) } ==> open_acct(n) is COM(Id).new_account(n);
      ...
end object class Customer_Semantics
```

Note that each instance of this object class includes exactly *one* instance of either `BankCustomer` or `Company`. To restrict the visibility of attribute and event symbols we must additionally define an interface (see Section 7.2) for this class :

```
interface class Customer_Semantics_Interface
  encapsulating
    Customer_Semantics;
  attributes
    Accts:set(nat);
    Address:string;
  events
    open_acct(out Acct:nat);
    close_acct(in Acct:nat);
end interface class Customer_Semantics_Interface
```

This definition hides the fact that we have something like an object class with varying template, i.e. a heterogeneous object class. The use of the **union** operator to construct a suitable name space and the definition of *common* events helps to define the semantics of generalized classes. But note that the **union** operator is *not* part of the TROLL-syntax.

6.4 Syntax of Aspect Specification

The following productions generate the specification of role classes specialized classes and generalized classes :

<object_spec> ::= **object** *<ob_id>*
  [*<dependencies>*';']
  <template>
end object *<ob_id>*

<dependencies> ::= **specializing from** *<ob_id_list>*
| **role of** *<ob_id_list>*

<class_spec> ::= **object class** *<class_id>*
  <type_header>
  <template>
end object class *<class_id>*

<type_header> ::= *<data_type_section>*
| **specializing from** *<class_id_list>*
  [**where** *<formula>*]';'
| **role of** *<class_id_list>*';'

<id_def> ::= *<attribute_id>* ':' *<domain>*

<gen_class_spec> ::= **generalization class** *<class_id>*
  of *<class_id_list>*';'
    <gen_template>
end generalization class *<class_id>*

<gen_template> ::= **template**
  [*<data_type_section>*]
  [*<components_section>*]
  [*<attributes>*]
  [*<events>*]
  renaming
    case
      <selection_seq>

<*selection*> ::= ‘`SELF`’ **in** <*class_id*>‘`:`’
<*renaming_clause_seq*>

<*renaming_clause*> ::= <*attribute_id*> **by** <*attribute_id*>
| <*event_id*> **by** <*event_id*>
| <*comp_id*> **by** <*comp_id*>

7 Specification of Systems

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

Object descriptions are the units of design corresponding to real world entities. In the case of complex real world structures, these design units may be aggregated to complex objects like described in chapter 5. The parts of a complex object may communicate, interchange data values and show restricted behavior.

For specifying whole systems of objects we must also be able to describe relations between separate objects that do not correspond to some complex real world entity. For example the interconnection between customers and automatic teller machines should not be described on the conceptual level by means of a complex object containing instances of `Customer` and `ATM`.

For this reason TROLL supports the specification of communication and integrity relations between objects via the *relationship construct*. Relationships on the one hand are used to depict the interconnection between separately defined objects, defining the communication patterns between these objects. On the other hand relationships can be used to describe integrity constraints between instances of (possibly different) classes.

Objects in TROLL like introduced up to now consist of a behavior and an observation component. The former is specified as the possible evolution of objects in terms of allowed events in some given object state, the latter corresponds to attribute values for a given state and their change according to the occurrences of events. Both components may be seen from outside the object, that is *all* attribute values may be read and event occurrences may be observed or triggered.

Specifying a system of objects, we sometimes want to hide details of objects. It may be necessary to explicitly encapsulate some attributes and events since they must not be known outside the object. This is particularly true for implementation issues, where the user must not know about complicated data structures or event traces used to describe internal activities of the system. In TROLL such restricted views may be defined as *projection interfaces*.

When specifying object (class) types, we describe possible extensions in terms of object instances with a structure and behavior according to the type specification. These objects are classified into object classes with the corresponding type. Object classes may be further structured when we select subpopulations of class instances. In TROLL this is achieved in defining *selection interfaces*. This way, only some instances may be referred to by a special class of users.

Sometimes the specification of projection and selection interfaces is not sufficient to describe customized interfaces for special purposes. For example when we want to describe interfaces for several object classes at once, i.e., for modeling sets of loosely coupled objects. Here, we may use *join interfaces*, explicitly specifying the instances that belong together in terms of some conditions on their properties.

At last, the specification of single objects, object class types, interfaces and relationships between objects make up a society of interacting objects, the specification of a system as a whole. In the next two sections we describe especially the structuring mechanisms for object societies, namely relationships and interfaces. The last section of this chapter concludes with the specification of object societies.

7.1 Relationships

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

In TROLL, two types of relationships are supported : (*global*) *interactions* and (*global*) *constraints*.

7.1.1 Global Interactions

Global interactions describe communication between objects. We may use the syntax for interactions depicted for complex objects. Global interactions along with the specification of the connected objects describe patterns of communication between objects. As usual, communication is described using event calling and event sharing.

Global interactions and constraints – as we will see in the next section – are specified with a special language construct, the **relationship** construct. First, the relationship is given a proper name. Second, the participating objects are specified giving their object names or class names. In the third part of a relationship specification, the possible *calling* and *sharing* equations are given :

```
relationship RelName between Name_1 OVar_1, ... , Name_n OVar_n ;
  data types ... ;
  interaction
    variables ... ;
    EvtTerm_l <op> EvtTerm_r;
    ... ;
end relationship RelName;
```

As can be seen in the above example, the participating objects can optionally be described using object variables, ranging over the class population. This kind of *renaming* is especially useful for describing interactions between instances of the same class. For example a communication between two instances of `ClassA` may be described with the following construction :

```
relationship RelName between ClassA X1, ClassA X2;
  interaction
    X1.EvtTerml <op> X2.EvtTermr;
    ... ;
end relationship RelName;
```

We will give an example for this kind of constraints between members of the same class in the next section.

As usual, the specification of global interactions contain the declaration of used data types. Under the keyword **interaction** the actual possible interactions between instances of the specified classes are described. For data interchange and description of source and target objects a **variables** section can be included.

Each interaction equation consists of three parts – a left and right hand side and an operation. The operation can be either *calling* denoted by `>>`, or *sharing* denoted by `==`. The left and right hand side are event terms like described in section 2.4. The event terms may contain the newly introduced object/class names. This renamed object/class names can be used like ordinary object/class names besides that they are only valid inside the relationship construct.

To give an example for a relationship describing interactions, we supply the following relationship between banks and ATMs :

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

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

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

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

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

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

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

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

7.1.2 Global Constraints

When we model systems by putting together objects, we sometimes have to state constraints that are to be fulfilled by related but independently specified objects. Such *global constraints* set up another kind of relationship between objects.

To specify global constraints, we need to know the participating objects. It is also possible, that we want to describe constraints between different objects of the same class. The two points mentioned lead to the decision to use the same syntax for the specification of global constraints as of global interactions. The close relationship between constraints and interactions, in fact, global interactions are a special kind of global constraints, leads to two blocks, an **interaction** and a **constraints** block inside a **relationship** construct :

```
relationship RelName between Name_1 OVar_1, ... Name_n OVar_n ;
  data types ... ;
  interaction
    variables ... ;
    ...
  constraints
    variables ... ;
    ...
end relationship RelName;
```

Only one of the blocks need to be present in a given relationship description. Consider the following example. When modeling our banking world, there may be a regulation that one particular bank customer may only be holder of at most one checking account. In TROLL, this would be specified as follows :

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

This constraint is an example for a relationship since it cannot be specified to be local to *one* instance of the class `CheckingAccount`. Here we also have an example using two different class names for the same class.

Please note, that the specification of global constraints may infer with contraints specified local to the participating objects, i.e. we may produce inconsistent specifications. The task to check the specification against such contradictionary constraints must be done with some verification tool. We will not deal with these aspects here.

7.1.3 Semantics of Relationships

One possible choice to define the semantics of relationships is again the use of embedding. Describing the semantics of aspects, we included the *parent* object into the specialized object. Since relationships are not objects on the specification level, we first have to construct an auxiliary object to be included into the participating objects. The other way round, we may include the objects participating in the relationship into the newly constructed object.

The former case can be seen as a direct modeling of interactions between independent objects, thus define the new object as a communication channel. This approach can also be adopted to describe the implementation of objects on different machines [HJ91]. Therefore we must introduce for example new events inside the communication channel. For this report we adopt the latter case since it is somewhat easier and sufficient to sketch the semantics of relationships.

Consider the example `RemoteTransaction` mentioned in the last section. The first step is to introduce an auxiliary object class type :

```
object class RemoteTransaction_Semantics
  data types |Bank|, |ATM|
  identification
    Bank_ID:|Bank|,
    ATM_ID:|ATM|
  template
    including
      instance Bank_ID in Bank as Bank;
      instance ATM_ID in ATM as ATM
    ...
end object class RemoteTransaction_Semantics
```

Instances of this class are identified by a tuple consisting of object identifiers of the participating classes. Each instance includes explicitly an instance of `Bank` and an instance of `ATM`. All possible pairs of communicating objects can be described.

Including the appropriate communicating instances yields an object where both components are referable. That is, the sharing and calling equations can be described as communication inside a complex object which is constructed via static aggregation. Furthermore we may describe the constraints as further restricting the behavior of the related objects.

7.2 Interfaces

An object or object (class) interface is first of all a mechanism to describe access control to objects. Interfaces in TROLL support the explicit encapsulation of object properties. Access control is achieved by projecting the attribute and event symbols to external visible symbols.

Interfaces may be defined for single objects as well as for object classes. For object classes, we may additionally define selection interfaces, thus restricting the visible population of the class to some proper subset. Selection interfaces resemble the well known views from relational databases [SJ91]. Other notably interface type are join interfaces and interfaces with derived properties.

7.2.1 Projection Interfaces

Up to now all properties specified in terms of attribute and event symbols are visible at the interface of the object. Attribute values of an object are accessible from the outside world although only read access is granted. Event occurrences inside an object may be observed as well as triggered from other objects.

Nevertheless, when specifying a system of objects, we sometimes want to hide or encapsulate some details. Especially in the process of implementing objects, an event `Bank_TA` may be further specified (and implemented) in terms of other events occurring in sequence or parallel. We do not want to see these additional events on the level described so far.

In this paper we will not present a formalism to describe this kind of object refinement, but we need object interfaces for example if we want to describe ATMs for a specialized class of users (in this case customers) :

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

The only observation for customers is the status of the ATM in terms of the bool-valued attribute `dispensed`. Information about the amount of money available inside the machine should (for obvious reason) not be public.

An interface to dynamic objects must define also the possible operations visible at this level of system description. Here a customer should only be able to talk to the ATM at 'customer level', i.e. he must be able to insert cards, issue transactions, cancel the transaction and not at least withdraw money. Customers must not be able to refill a machine or even remove it. Also they should not see details of the internal operations, for example the event `check_card_w_bank` is not relevant at 'customer level'.

The semantics of this simple kind of interface is just a signature restriction to the explicitly noted event and attribute symbols.

7.2.2 Selection Interfaces

Another kind of access restriction in contrast to the above mentioned projection interface is the selection interface. In contrast to projection interfaces that may be applied to single objects or object classes, selection interfaces may only be used for object classes. Here we can restrict the access to subsets of object classes.

In this language version we support only selections based on constant properties of objects (see [SJ91, SJE91] for selection based on arbitrary object properties). For example we may define an interface for another class of customers. Since ATMs only have the constant key attribute `IdentNo`, we can specify an (artificial) interface for a proper subset of ATMs with an identification number between 100 and 199 :

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

Here the actual visible population is limited using a predicate over the key attributes. Naturally, the selection interface can be combined with the projection interface.

The semantics of a selection interface definition is given by an object class specialization determining the proper subset of the encapsulated class. The condition in the **where**-clause can directly be used as selection predicate (see Section 6.2) :

```
object class ATMToCustomer2_Semantics
  specializing from ATM;
    where ATM.id.IdentNo >= 100 and ATM.id.IdentNo <= 199;
  ...
end object class ATMToCustomer2_Semantics
```

Access to instances of this specialized class may be further restricted by a projection interface. The predicate used for the definition of the specialized class can be seen as a filter allowing only those instances to pass, that satisfy the selection condition evaluated locally to the object instances.

7.2.3 Join Interfaces

Like mentioned above, interfaces are a well known construction in relational data base systems in that context named *views*. Views in relational systems may be defined for joins between different relations. In our approach, joins are described by aggregations. Therefore we support another kind of interface, the join interface which is based on selection interfaces. See for example an interface for accounts holders and the related accounts :

```
interface class AccountHolder
  encapsulating Account A; BankCustomer B;
  selection
    where A.id.Holder = B.id;
  ...
end interface class AccountHolder
```

For join views we must introduce variables ranging over the joined class populations. In this example the variables are optional, since we can identify the different populations of accounts and holders with their original names. Like for relationships we can introduce join views for instances of *one* class. Therefore we definitely need the renamed class variables. Naturally, the join interface can be combined with the projection interface.

The selection predicate identifies the instances of the joined classes. In this case corresponding instances of `Account` and `BankCustomer` are selected where the `Holder` (key) attribute of the account object is equal to the customer identification.

The semantics of join interfaces is described by aggregating the objects according to the selection condition :

```
object class JoinAccountHolder_Semantics
  identification
    BankCustomer_ID:|BankCustomer|;
    Account_ID:|Account|;
  template
    including
      B in BankCustomer where B.id = Account_ID.Holder as Customer;
      A in Account where A.id.Holder = BankCustomer_ID as Account;
    ...
end object class JoinAccountHolder_Semantics
```

Since we have not defined conditions between different including clauses, we need including clauses for every participating object class thus splitting the original join condition into two parts. Therefore we construct a class with instances identified by the relevant object identifiers of the participating objects (according to the join predicate) and describe the inclusion in terms of these identifiers according to the selection condition.

7.2.4 Interfaces with Derived Properties

In general we do not only want to restrict the external object interface to some subset of events and attributes, but also be able to present derived properties of objects. We may specify views of an object where some information is explicitly computed from existing attributes.

An example interface for the ATM class may be used for service personnel only. Suppose that we want to indicate machines that must be refilled to avoid a dispensed condition. Therefore this view defines a derived bool-valued attribute `please_refill` to be true for ATM's with `CashOnHand` below a threshold value of 1000 :

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

Attribute derivation is expressed in the **attributes** part of the **derivation** section. Technically, an interface with derived properties consists of a formal implementation step [SE90, ES89] and an explicit projection interface. That is, the semantics is defined by a new object class including the appropriate instance of `ATM` and locally defining the derivation relationships.

More general we can also look at specialized operations as a view on the dynamic part of objects. For example we may have users of the ATM that have to pay an extra charge for each bank transaction. Suppose that the ATM has an additional (may be constant) attribute `extraCharge:money`, denoting the amount of money to be withdrawn from the account :

```
interface class ATMToExtraUser
  encapsulating ATM;
  data types money,bool;
  attributes
    extraCharge:money;
    dispensed:bool;
  events
    issue_TA_Extra(nat,money);
    ...
  derivation
    variables n:nat, m:money;
    process ITE(n,m) = issue_TA(n,m) -> issue_TA(n,extraCharge);
    end process;
    events issue_TA_Extra(n,m) is < ITE(n,m) >
end interface class ATMToExtraUser;
```

The process `ITE` is introduced as a seqence of two `issue_TA` events occurring in sequence, one that denotes the withdrawal of the original requested amount of money, and one that denotes the withdrawal of the extra charge. The process derivation is expressed in the **events** part of the **derivation** section. Note that the derived event can only be formulated as a transaction since events must be *atomic* actions.

To describe the semantics of interfaces with derived properties, the semantic objects include object instances of the classes denoted as encapsulated and then define new attributes and events like for example `dispensed` or `issue_TA_Extra(n,m)`. New attributes are identified with data terms over included events, new events are identified with processes over the included events. This way object interfaces with derived events can be seen as interfaces for special object *usage*. Operations seen at the interface induce (atomic) traces of events in the underlying objects.

7.3 Society Specification

The specification of imported data types, templates, single objects, class types, roles, specializations and generalizations, interfaces for object instances and object classes and the interconnection of all these parts make up a society of interacting objects, the specification of a system as a whole.

An object society is described with a proper name and encloses an arbitrary number of components like introduced in this paper :

```
object society societyName
  specification of
      included data types
      templates
      objects
      class types
      interfaces
      relationships
end object society societyName
```

Note that the including clause lists the externally specified data types used for this society specification, for the detailed syntax see the production rules below.

The language version described in this paper currently supports only this very simple specification of an object society. Extensions to be included in the future may contain :

- Reuse of previously specified societies: resembling the including clauses used for the including of object instances into other instances, here we may specify the inclusion of object societies as a whole [SRGS91].

- Interface definitions for societies: defining an interface, we may restrict the visible properties of a society to some proper subset. For example we can restrict the view of a society to some *user interfaces.*

- Parameterized object societies: to support the reuse of object societies, here we may introduce parameters for society definitions. The choice of suitable specification entities to be used as parameters is still an open problem (even for object specification) [SRGS91].

7.4 Syntax of System Specification

For the language version introduced in this paper, the production rules to generate relationships, interfaces and the society are listed below. Please note that *<society_spec>* is also the start symbol of the grammar :

<rel_def>	::=	**relationship** [*<rel_id>*] **between** *<item_list>*';' [*<data_type_section>*] [*<constraints>*] [*<interaction>*] **end relationship** [*<rel_id>*]
<item>	::=	*<class_id>* [*<ovar_id>*]
<constraints>	::=	**constraints** [*<variable_declaration>*] *<constr_seq>*';'
<constr>	::=	*<fut_formula>* | **initially** *<formula>*
<interaction>	::=	**interaction** [*<variable_declaration>*] [*<process_def_seq>*';'] *<interaction_def_seq>*';'
<interaction_def>	::=	'{'*<past_formula>*'}' '==>' *<simple_interaction>* | *<simple_interaction>*
<simple_interaction>	::=	*<event_term>* '==' *<event_term>* | *<event_term>* '>>' *<event_term>* | *<event_term>* '>>' *<process_term>*
<object_interface>	::=	**interface** *<interface_id>* **encapsulating** *<item_list>*';' [*<data_type_section>*] [*<attributes>*] [*<events>*] [*<derivations>*] **end interface** *<interface_id>*

<class_interface> ::= **interface class** <interface_id>
  encapsulating <item_list>';'
  [**selection where** <formula>';']
    [<attributes>]
    [<events>]
    [<derivations>]
end interface class <interface_id>

<derivations> ::= **derivation**
  [<variable_declaration>]
  [<process_def_seq>';']
  [**attributes** <derivation_rule_seq>';']
  [**events** <event_der_seq>';']

<derivation_rule> ::= '{'<formula>'}' '==>' <simple_derivation>
  | <simple_derivation>

<simple_derivation> ::= <attribute_term> '=' <data_term>

<event_der> ::= <event_term> **is** <event_term>
  | <event_term> **is** '<'<process_term>'>'

<society_spec> ::= **object society** <society_id>
  [**including** <domain_list>';']
    <spec_item_seq>
end object society <society_id>';'

<spec_item> ::= <template_spec>
  | <object_spec>
  | <class_spec>
  | <gen_class_spec>
  | <object_interface>
  | <class_interface>
  | <rel_def>

8 Conclusions and Outlook

In this report we have presented the language features of TROLL, a specification language especially suited to be used in the early stages of information systems design. We have pointed out the importance of modeling the concepts in the problem domain or UoD. A language used for this purpose should offer a large number of abstraction concepts to enable convenient natural descriptions of UoD aspects.

The concept of object grew out of a number of approaches towards UoD modeling like semantic data models, knowledge representation, process languages, and object-oriented languages. Object-oriented conceptual modeling supports the organization of a system description in terms of object descriptions that integrate the description of static and dynamic aspects local to an entity. A previous approach towards a textual language in the same context is [CSS89]. A related graphical formalism is described in [SSG+91].

TROLL is based on a number of formal sublanguages which are the tools to describe properties and evolution in object descriptions. We have describe the sublanguages for data terms, predicate logic formulas and temporal logic formulas in great depth from a syntactic and semantic point of view.

The basic building blocks of specifications are template specifications. We described the sections of a template specification in a detailed way. Templates are anonymous descriptions of instances of classes and thus are a component of a class type. Class types describe the potential extensions of classes in that they consists of a template and a set of possible identifiers for instances of that class.

Aspects are abstractions of one object that allow us to look at an object from different angles. The abstraction mechanisms provided by TROLL for aspects are specialization, roles, and generalization.

In TROLL, objects that are composed from other objects are called complex objects. We explained three different language features to compose objects. Static aggregation means that we describe the composition of objects by predicates over the static parts of templates meaning that aggregation is described declaratively. Dynamic aggregation emphasizes the operational description of object aggregation. The composition of aggregations may be altered by special events that are implicitly declared with a composite object. The aggregation of disjoint complex objects can be looked at as a structuring primitive of our language. Disjoint complex objects can be used to describe private parts of objects which can be manipulated only in the context of the aggregation.

A system is regarded to be a collection of interacting objects. TROLL offers language features to state relationships between separately defined objects. Relationships are interactions or dependencies and are used to put local specifications in a global system context. To support cooperative design, we introduced the concept of additional interfaces for objects.

8.1 Assessment

TROLL offers a large number of language features that are very often redundant. We think that during UoD modeling the designer/specifier should not be forced to transform natural descriptions into a small number of non-redundant language features but should be able to describe aspects in a natural way. Aspects of the real world are seen differently by different people and thus are described differently.

The choice of using logical sublanguages may seem strange but we are of the opinion that logics provide a firm semantic ground for being *precise* and *unambiguous*. We are following a more formal approach to systems design. The semantics of logical formulas is defined precisely over mathematical models and thus gives no rise for ambiguity and inaccuracy.

TROLL is a textual language and specifications of large systems are not readable any more. We know of that problem. But we are of the opinion that many things cannot be express adequately in a graphical manner like e.g. integrity constraints. We think, however, that a *method* for object-oriented conceptual modeling must include a graphical representation for at least the basic features of the language (which could serve as a basis for such a method). The current approaches, however, use different graphical representations for static and dynamic aspects and these graphical representations are sometimes restrictive. Graphics are an essential tool but a textual language should provide the basis.

A complete approach to information systems design should provide support for many more issues like e.g. reusability, modularization above the object level, and schema modification. For the moment, these issues are not addressed by TROLL.

8.2 Future Work

Of course the language version described in this report is far from being ready. Thus, we plan to put a large amount of work into the further development and the "polishing" of our language.

Currently, a colleague of ours is working on a high-level graphical notation for TROLL. The goal is to support the early stages of developing a system model where it is very useful to have a good overview about the structure of the system. For the dynamics, it is also quite useful to be able to specify things graphically. We think, however, that a graphical and a textual language should be used together in the system development process and therefore both languages must be strongly related.

Another direction of work in the future is support for in-the-large development. For this we plan to define a higher language level with constructs to support reuse, modularization, and parameterization.

Specifications are the basis for implementations. Thus, another line of work is the transformation of high-level, application-oriented specifications into low-level, implementation-oriented specifications like the ones described in [HJ91]. These implementation-oriented specifications shall be executable in a run-time environment that supports distribution of objects in a network of workstations. Currently, work has started on developing a transaction model. Further work is needed on communication protocols and object management.

Additional References

For further reading about the issues addressed in the ESPRIT II BRA WG IS-CORE refer to [ESS88, SFSE89, FM90, JSS90, FSMS90, FM91, SF91, SGG+91].

The modeling of database applications is addressed in [Gri82, Vv82, KMS85, HNSE87, DGL87, KL88, EHH+89, Saa91].

Dynamic modeling of database behavior is the topic of [ELG84, LEG85, Lip86, LS87].

Object-oriented issues in general are addressed in [Agh86, KC86, SW87, Kin89, KL89, MK90]. Readings in object-oriented databases and their design are among others [DD86, BCG+87, Dit88, ZM89, Bee90]. Formal approaches to object-oriented programming/specification are [GM87, Mes90].

Related approaches in object-oriented conceptual modeling are [Wie90, KS91] or the ones presented in [VAMR91].

Bibliography

[Agh86] Agha, G.A.: *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

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

[BCG+87] Banerjee, J.; Chou, H.; Garza, J.F.; Kim, W.; Woelk, D.; Ballou, N.; Kim, H.: Data Model Issues for Object-Oriented Applications. *ACM Transaction on Office Information Systems*, Vol. 5, No. 1, January 1987, pp. 3–26.

[Bee90] Beeri, C.: A Formal Approach to Object Oriented Databases. *Data & Knowledge Engineering*, Vol. 5, No. 4, 1990, pp. 353–382.

[BM86] Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.

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

[Boo90] Booch, G.: *Object-Oriented Design.* Benjamin/Cummings, Menlo Park, CA, 1990.

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

[Che76] Chen, P.P.: The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, pp. 9–36.

[CHJ86] Cohen, B.; Harwood, W. T.; Jackson, M. I.: *The Specification of Complex Systems.* Addison-Wesley, Reading, MA, 1986.

[Cod79] Codd, E. F.: Extending the Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, Vol. 4, No. 4, 1979, pp. 397–434.

[CSS89] Costa, J.-F.; Sernadas, A.; Sernadas, C.: OBL-89 Users Manual (version 2.3). Internal Report. INESC, Lisbon, 1989.

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

[Dat86] Date, C. J.: *An Introduction to Database Systems.* Addison-Wesley, Reading, MA, 1986.

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

[DGL87] Dittrich, K.; Gotthard, W.; Lockemann, P.: Complex Entities for Engineering Applications. In: *Proc. Int. Conf. on the ER-Approach.* North-Holland, Amsterdam, 1987, 1987, pp. 421–440.

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

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

[EGH+90] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: Conceptual Modelling of Database Applications Using an Extended ER Model. Informatik-Bericht 90–05, Technische Universität Braunschweig, 1990.

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

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

[EHH+89] Engels, G.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Ehrich, H.-D.: CADDY: Computer-Aided Design of Non-Standard Databases. In: *Proc. 1st Int. Conf. on System Development Environments and Factories*, Berlin, 1989.

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

[ELG84] Ehrich, H.-D.; Lipeck, U. W.; Gogolla, M.: Specification, Semantics, and Enforcement of Dynamic Database Constraints. In: *Proc. Int. Conf. on Very Large Databases VLDB'84*, Singapore, 1984. pp. 301–308.

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

[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.

[ES89] Ehrich, H.-D.; Sernadas, A.: Algebraic Implementation of Objects over Objects. In: deRoever, W. (ed.): *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proc. REX'89)*, Mood (NL), 1989. LNCS 394, Springer Verlag, Berlin, 1989, pp. 239–266.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Information Systems – Correctness and Reusability.* Informatik-Bericht 91–03, TU Braunschweig, 1991.

[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 Verlag, Berlin, 1988, pp. 144–149.

[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.

[EW86] Ehrig, H.; Weber, H.: Programming in the Large with Algebraic Module Specifications. In: Kugler, H. J. (ed.): *Information Processing 86.* North-Holland, Amsterdam, 1986, 1986, pp. 675–684.

[FM90] Fiadeiro, J.; Maibaum, T.: Describing, Structuring and Implementing Objects. In: Bakker, J. de; Roever, W. de; Rozenberg, G. (eds.): *Foundations of Object-Oriented Languages (Proc. REX School/Workshop)*, Noordwijkerhood (NL), 1990. LNCS 489, Springer-Verlag, Berlin, 1991, pp. 275–310.

[FM91] Fiadeiro, J.; Maibaum, T. S. E.: Towards Object Calculi. In: Saake, G.; Sernadas, A. (eds.): *Information Systems – Correctness and Reusability.* Informatik-Bericht 91–03, TU Braunschweig, 1991.

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

[FSMS90] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In: Meersman, R.; Kent, W. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam, 1991, pp. 243–284.

[GBM86] Greenspan, S.; Borgida, A. T.; Mylopoulos, J.: A Requirements Modelling Language and its Logic. In: Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Base Management Systems*. Springer-Verlag, Berlin, 1986, pp. 471–502.

[GKK+87] Goguen, J. A.; Kirchner, C.; Kirchner, H.; Mégrelis, A.; Meseguer, T., J. Winkler: An Introduction to OBJ3. In: Jouannaud, J.-P.; Kaplan, S. (eds.): *Proc. Conf. on Conditional Term Rewriting*, Orsay (F), 1987. LNCS 308, Springer-Verlag, Berlin, 1988, pp. 258–263.

[GM87] Goguen, J. A.; Meseguer, J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In: Shriver, B.; Wegner, P. (eds.): *Research Directions in Object-Oriented Programming*. MIT Press, 1987, pp. 417–477.

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

[HG88] Hohenstein, U.; Gogolla, M.: A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In: *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, Rome, 1988. North-Holland, Amsterdam, 1988.

[HJ91] Hartmann, T.; Jungclaus, R.: Abstract Description of Distributed Object Systems. In: Tokoro, M.; Nierstrasz, O.; Wegner, P.; Yonezawa, A. (eds.): *Proc. ECOOP91-Workshop on Object-Based Concurrent Computing*, Geneva, 1991. LNCS Series, Springer-Verlag, Berlin, *to appear*.

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

[HNSE87] Hohenstein, U.; Neugebauer, L.; Saake, G.; Ehrich, H.-D.: Three-Level Specification of Databases Using an Extended Entity-Relationship Model. In: Wagner, R.; Traunmüller, R.; Mayr, H. C. (eds.): *Proc. GI-Fachtagung "Informationsermittlung und -analyse für den Entwurf von Informationssystemen"*, Linz, 1987. Informatik-Fachbericht 143, Springer Verlag, Berlin, 1987, pp. 58–88.

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

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

[JSH91] Jungclaus, R.; Saake, G.; Hartmann, T.: Language Features for Object-Oriented Conceptual Modeling. In: Teory, T. (ed.): *Proc. 10th Int. Conf. on the ER-Approach*, San Mateo (CA), 1991, pp. 309–324.

[JSS90] Jungclaus, R.; Saake, G.; Sernadas, C.: Using Active Objects for Query Processing. In: Meersman, R.; Kent, W. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam, 1991, pp. 285–303.

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

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

[Ken78] Kent, W.: *Data and Reality.* North-Holland, Amsterdam, 1978.

[Kin89] King, R.: My Cat Is Object-Oriented. In: Kim, W.; Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications.* ACM Press/Addison-Wesley, New York, NY/Reading, MA, 1989, pp. 23–30.

[KL88] Karl, S.; Lockemann, P. C.: Design of Engineering Databases: A Case for More Varied Semantic Modelling Concepts. *Information Systems*, Vol. 13, No. 4, 1988, pp. 335–358.

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

[KMS85] Khosla, S.; Maibaum, T.; Sadler, M.: Database Specification. In: Steel, T. B.; Meersman, R. (eds.): *Proc. IFIP WG 2.6 Working Conf. on Database Semantics DS-1.* North-Holland, Amsterdam, 1986, 1985, pp. 141–158.

[Kro87] Kröger, F.: *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

[KS91] Kappel, G.; Schrefl, M.: Object/Behavior Diagrams. In: *Proc. Int. Conf. on Data Engineering*, Kobe, Japan, 1991, pp. 530–539.

[LEG85] Lipeck, U. W.; Ehrich, H.-D.; Gogolla, M.: Specifying Admissability of Dynamic Database Bevaviour Using Temporal Logic. In: Sernadas, A. et al. (eds.): *Proc. IFIP Working Conf. on Theoretical and Formal Aspects of Information Systems.* North-Holland, Amsterdam, 1985, 1985, pp. 145–157.

[Lip86] Lipeck, U. W.: Stepwise Specification of Dynamic Database Behaviour. In: Zaniolo, C. (ed.): *Proc. 1986 ACM-SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1986. pp. 387–397.

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

[Lip90] Lipeck, U. W.: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, Vol. 76, 1990, pp. 115–142.

[LS87] Lipeck, U. W.; Saake, G.: Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, Vol. 12, 1987, pp. 255–269.

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

[MBW80] Mylopoulos, J.; Bernstein, P. A.; Wong, H. K. T.: A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, Vol. 5, No. 2, 1980, pp. 185–207.

[McC58] McCarthy, J.: Mechanization of Thought Processes. In: *Proc. Symp. of National Physical Laboratory*, London, 1958. pp. 77–84.

[Mes90] Meseguer, J.: A Logical Theory of Concurrent Objects. In: Meyrowitz, N. (ed.): *ECOOP/OOPSLA'90 Proceedings*, Ottawa, 1990. ACM, 1990, pp. 101–115.

[Mey88] Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mil80] Milner, R.: *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, 1980.

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

[Mil90] Milner, R.: Operational and Algebraic Semantics of Concurrent Processes. In: Leeuwen, J. van (ed.): *Formal Models and Semantics*. Elsevier Science Publishers B.V., 1990, pp. 1201–1242.

[Min75] Minsky, M.: A Framework for Representing Knowledge. In: Winston, P. H. (ed.): *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[MK90] McGregor, J. D.; Korson, T. (Guest editors): Special Issue on Object-Oriented Design. *Communications of the ACM*, Vol. 33, No. 9, 1990.

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

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

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

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

[RM75] Roussopoulos, N.; Mylopoulos, J.: Using Semantic Networks for Data Base Management. In: *Proc. Int. Conf. on Very Large Databases VLDB'75*, 1975. Reprinted in: [MB89].

[Saa90] Saake, G.: Conceptual Modeling of Database Applications. In: Karagiannis, D. (ed.): *Proc. Workshop Information Systems and Artificial Intelligence : Integration Aspects*, Ulm, 1990. LNCS 474, Springer-Verlag, Berlin, 1991, pp. 213–232.

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

[SE90] Sernadas, A.; Ehrich, H.-D.: What Is an Object, After All? In: Meersman, R.; Kent, W. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam, 1991, pp. 39–69.

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

[SF91] Sernadas, C.; Fiadeiro, J.: Towards Object-Oriented Conceptual Modelling. *Data & Knowledge Engineering*, Vol. 6, 1991, pp. 479–508.

[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.

[SGG+91] Sernadas, C.; Gouveia, P.; Gouveia, J.; Sernadas, A.; Resende, P.: The Reification Dimension in Object-Oriented Data Base Design. In: Harper, D. (ed.): *Proc. Int. Workshop on the Specification of Database Systems*, Glasgow, 1991. Springer-Verlag, London, 1991. *In print.*

[SJ91] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL Language. In: Harper, D. (ed.): *Proc. Int. Workshop on the Specification of Database Systems*, Glasgow, 1991. Springer-Verlag, London, 1991. *In print.*

[SJE91] Saake, G.; Jungclaus, R.; Ehrich, H.-D.: Object-Oriented Specification and Stepwise Refinement. In: Meer, J. de; Heymer, V. (eds.): *Proc. IFIP TC6*

Int. Workshop on Open Distributed Processing, Berlin, 1991. North-Holland, Amsterdam, *to appear.*

[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*, Berlin, 1989. Springer-Verlag, pp. 288–300.

[SRGS91] Sernadas, C.; Resende, P.; Gouveia, P.; Sernadas, A.: In-the-large Object-Oriented Design of Information Systems. In: Van Assche, F.; Moulin, B.; Rolland, C. (eds.): *Proc. Object-Oriented Approach in Information Systems*, Ottawa, Canada, 1991. North Holland, Amsterdam, 1991, pp. 209–232.

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

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Hammerslay, P. (ed.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987, pp. 107–116.

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

[ST89] Schmidt, J. W.; Thanos, C. (eds.): *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.

[SW87] Shriver, B.; Wegner, P. (eds.): *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, MA, 1987.

[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.

[VAMR91] Van Assche, F.; Moulin, B.; Rolland, C. (eds.): *The Object-Oriented Approach in Information Systems*, Québec, Canada, 1991. North-Holland, Amsterdam, 1991.

[VHL89] Van Horebeek, I.; Lewi, J.: *Algebraic Specifications in Software Engineering.* Springer-Verlag, Berlin, 1989.

[Vv82] Verheijen, G. M. A.; van Bekkum, J.: NIAM: An Information Analysis Method. In: Olle, T. W.; Sol, H. G.; Verrijn-Stuart, A. A. (eds.): *Information Systems Design Methodologies: A Comparative Review*, Noordwijkerhood (NL), 1982. North-Holland, Amsterdam, 1982, pp. 537–589.

[Weg90] Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, Vol. 1, No. 1, 1990, pp. 7–87.

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

[Win90] Wing, J. M.: A Specifier's Introduction to Formal Methods. *IEEE Computer*, Vol. 23, September 1990, pp. 8–24.

[ZM89] Zdonik, S. B.; Maier, D. (eds.): *Readings in Object-Oriented Database Systems*. Morgan-Kaufmann, Palo Alto, CA, 1989.

Appendix A: The Syntax

The following symbols are meta-symbols for the grammar:

- **keywords of the language**
- ‘`terminal symbols`’
- [optional symbols]
- *<non-terminal symbol>*
- alternatives are separated by |
- parentheses are used wherever needed

We use the following abbreviations (let x be a non-terminal-symbol):

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

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

Formula Sublanguage

Data Terms

<data_term> ::= *<data term>* *<inf_op>* *<primary>*
| *<primary>*

<primary> ::= ‘(’*<data_term>*‘)’
| *<op_id>*‘(’*<data_term_list>*‘)’
| *<variable_id>*
| *<constant_symbol>*
| *<attribute_sel>*

```
<attribute_sel>        ::= <attribute_id>
                         | <attribute_id>‘(’<data_term_list>‘) ’
                         | <selector><attribute_id>
                         | <selector><attribute_id>‘(’<data_term_list>‘)’

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

<select_id>            ::= <class_id>‘(’<data_term_list>‘)’
                         | <subtemplate_id>‘(’<data_term_list>‘)’
                         | <component_id>‘(’<data_term_list>‘)’
                         | <ob_id>
```

FOL-Formulae

```
<formula>              ::= <formula> ‘<==>’ <rel_form>
                         | <rel_form>

<rel_form>             ::= <rel_form> ‘==>’ <s_form>
                         | <s_form>

<s_form>               ::= <s_form> or <m_form>
                         | <m_form>

<m_form>               ::= <m_form> and <a_form>
                         | <a_form>

<a_form>               ::= ‘(’<formula>‘)’
                         | not <formula>
                         | forall ‘(’<decl_list>‘)’ ‘(’ <formula> ‘)’
                         | exists ‘(’<decl_list>‘)’ ‘(’ <formula>‘)’
                         | <data_term>
                         | true | false
```

Past Tense Temporal Formulae

```
<past_formula>         ::= <past_formula> ‘<==>’ <past_rel_form>
                         | <past_rel_form>

<past_rel_form>        ::= <past_rel_form> ‘==>’ <past_s_form>
                         | <past_s_form>
```

<*past_s_form*> ::= <*past_s_form*> **or** <*past_m_form*>
| <*past_m_form*>

<*past_m_form*> ::= <*past_m_form*> **and** <*past_a_form*>
| **alwaysp** <*past_m_form*> **since last** <*past_a_form*>
| **sometimep** <*past_m_form*> **since last** <*past_a_form*>
| <*past_a_form*>

<*past_a_form*> ::= **not** <*past_formula*>
| **after**‘(’<*event_term*>‘)’
| **forall** ‘(’<*decl_list*>‘)’ ‘(’<*past_formula*>‘)’
| **exists** ‘(’<*decl_list*>‘)’ ‘(’<*past_formula*>‘)’
| **sometime** <*past_formula*>
| **always** <*past_formula*>
| **previous** <*past_formula*>
| ‘(’<*past_formula*>‘)’
| <*formula*>
| **true** | **false**

Future Tense Temporal Logic

<*fut_formula*> ::= <*fut_formula*> ‘**<==>**’ <*fut_rel_form*>
| <*fut_rel_form*>

<*fut_rel_form*> ::= <*fut_rel_form*> ‘**==>**’ <*fut_s_form*>
| <*fut_s_form*>

<*fut_s_form*> ::= <*fut_s_form*> **or** <*fut_m_form*>
| <*fut_m_form*>

<*fut_m_form*> ::= <*fut_m_form*> **and** <*fut_a_form*>
| <*fut_m_form*> **until** <*fut_a_form*>
| <*fut_m_form*> **before** <*fut_a_form*>
| <*fut_a_form*>

<fut_a_form> ::= **not** *<fut_formula>*
| **forall** '('*<decl_list>*')' '('*<fut_formula>*')'
| **exists** '('*<decl_list>*')' '('*<fut_formula>*')'
| **sometimef** *<fut_formula>*
| **alwaysf** *<fut_formula>*
| **next** *<fut_formula>*
| '('*<fut_formula>*')'
| *<formula>*
| **true** | **false**

Processes

<process_def> ::= **process** *<process_id>* '=' *<process>*
[**with** *<process_def_seq>*] **end process**
| **process**
<process_id> '('*<data_term_list>*')' '=' *<process>*
[**with** *<process_def_seq>*] **end process**

<process> ::= *<process>* '->' *<process_term>*
| *<process_term>*

<process_term> ::= *<process_id>*
<process_id>'('*<data_term_list>*')'
| *<event_term>*
| *<choice>*
| *<loop>*
| *<guarded_process>*
| '<'*<process>*'>'

<event_term> ::= *<event_id>*
| *<event_id>*'('*<data_term_list>*')'
| *<selector><event_id>*
| *<selector><event_id>*'('*<data_term_list>*')'

<guarded_process> ::= '{'*<past_formula>*'}' *<process>*

<loop> ::= **foreach** *<set_term>* **do** *<process>* **od**

<choice> ::= **case** <guarded_process_seq>
[**else** <process>]
esac
| **case on** <set_term>
<guarded_process_seq>
[**else** <process>]
esac

<set_term> ::= <decl_list>
| <data term>

Template Specification

Declarations

<domain> ::= <data_type_id>
| '|'<class_id>'|'
| <type_constructor_id>'('<domain_list>')'

<decl> ::= <variable_id_list> ':' <domain>

<variable_declaration> ::= **variables**
<decl_seq>';'

<process_decl> ::= **declare** <process_def>';'
| **declare on** <event_term_seq> <process_def>';'

Templates

<template_spec> ::= **template** <template_id>
<template_sig>
<template_body>
end template <template_id>';'

<template> ::= **template**
<template_sig>
<template_body>

<*template_sig*>	::=	[<*data_type_section*>]
		[<*including_section*>]
		[<*components_section*>]
		[<*sub_temp_section*>]
		[<*attributes*>]
		[<*events*>]
<*data_type_section*>	::=	**data types**
		<*domain_list*> ‘;’
<*including_section*>	::=	**including**
		<*including_clause_seq*>‘;’
<*including_clause*>	::=	<*ovar_id*> **in** <*class_id*>
		[**where** <*formula*>] [**as** <*class_id*>]
		[<*renaming*>]
		| **instance** <*data_term*> **in** <*class_id*> [**as** <*ob_id*>]
		[<*renaming*>]
		| **instance** <*ob_id*> [**as** <*ob_id*>]
		[<*renaming*>]
<*renaming*>	::=	**renaming**
		<*renaming_clause_seq*>‘;’
<*renaming_clause*>	::=	<*attribute_id*> **by** <*attribute_id*>
		| <*event_id*> **by** <*event_id*>
		| <*comp_id*> **by** <*comp_id*>
<*components_section*>	::=	**components**
		<*comp_def_seq*>‘;’
<*comp_def*>	::=	<*comp_id*> ‘:’ <*comp_domain*>
		| <*comp_id*>‘(’<*domain_list*>‘)’ ‘:’ <*comp_domain*>
<*comp_domain*>	::=	<*ob_id*>
		| <*class_id*>
		| **LIST**‘(’<*class_id*>‘)’
		| **SET**‘(’<*class_id*>‘)’
<*sub_temp_section*>	::=	**subtemplates**
		<*sub_temp_def_seq*>‘;’

<*sub_temp_def*>	::=	<*subtemplate_id*>':' <*template_id*> | <*subtemplate_id*>(<*domain_list*>) ':'<*template_id*>
<*attributes*>	::=	**attributes** <*att_def_seq*>';'
<*att_def*>	::=	<*attribute_def*> | **constant** <*attribute_def*> | **derived** <*attribute_def*>
<*attribute_def*>	::=	<*attribute_id*> ':' <*domain*> | <*attribute_id*>'('<*domain_list*>')' ':' <*domain*>
<*events*>	::=	**events** <*birth_event_seq*>';' [<*event_seq*>]';'
<*birth_event*>	::=	[**active**][**derived**]**birth** <*event_def*>
<*event*>	::=	[**active**] [**derived**] <*event_def*> | [**active**] [**derived**] **death** <*event_def*>
<*event_def*>	::=	<*event_id*> | <*event_id*>'('<*evt_par_list*>')'
<*evt_par*>	::=	[**in**] <*domain*> | [**out**] <*domain*>
<*template_body*>	::=	[<*derivations*>] [<*constraints*>] [<*valuation*>] [<*behavior*>] [<*interaction*>]
<*derivations*>	::=	**derivation** [<*variable_declaration*>] [<*process_decl_seq*>';'] [**attributes** <*derivation_rule_seq*>';'] [**events** <*event_der_seq*>';']

<*derivation_rule*> ::= '{'<*formula*>'}' '==>' <*simple_derivation*>
| <*simple_derivation*>

<*simple_derivation*> ::= <*attribute_term*> '=' <*data_term*>

<*event_der*> ::= <*event_term*> **is** <*event_term*>
| <*event_term*> **is** '<'<*process_term*>'>'

<*constraints*> ::= **constraints**
[<*variable_declaration*>]
<*constr_seq*>';'

<*constr*> ::= <*fut_formula*>
| **initially** <*formula*>

<*valuation*> ::= **valuation**
[<*variable_declaration*>]
<*val_rule_seq*>';'

<*val_rule*> ::= '{'<*past_formula*>'}' '==>' <*simple_valrule*>
| <*simple_valrule*>

<*simple_valrule*> ::= '['<*event_term*>']' <*effect_list*>

<*effect*> ::= <*attribute_term*> '=' <*data_term*>

<*behavior*> ::= **behavior**
[<*permissions*>]
[<*obligations*>]
[<*commitments*>]
[<*patterns*>]

<*permissions*> ::= **permissions**
[<*variable_decalaration*>]
<*per_rule_seq*>';'

<*per_rule*> ::= '{'<*past_formula*>'}' <*event_term*>

<*obligations*> ::= **obligations**
[<*variable_decaration*>]
[<*process_decl_seq*>]
<*obl_expr_seq*>';'

<obl_expr> ::= ‘{’<formula>‘}’ ‘==>’ <obl_s_expr>
| <obl_s_expr>

<obl_s_expr> ::= <obl_s_expr> **or** <obl_term>
| <obl_term>

<obl_term> ::= <obl_term> **and** <obligation>
| <obligation>

<obligation> ::= ‘(’<obl_expr>‘)’
| <process>
| **exists** ‘(’<decl_seq>‘)’ ‘(’<obl_expr>‘)’

<commitments> ::= **commitments**
[<variable_declaration>]
[<process_decl_seq>]
<obl_expr_seq>‘;’

<patterns> ::= **patterns**
[<variable_declaration>]
[<process_decl_seq>‘;’]
<process_seq>‘;’

<interaction> ::= **interaction**
[<variable_declaration>]
[<process_decl_seq>‘;’]
<interaction_def_seq>‘;’

<interaction_def> ::= ‘{’<past_formula>‘}’ ‘==>’ <simple_interaction>
| <simple_interaction>

<simple_interaction> ::= <event_term> ‘==’ <event_term>
| <event_term> ‘>>’ <event_term>
| <event_term> ‘>>’ <process_term>

Object, Classes and Interfaces

<object_spec> ::= **object** <ob_id>
[<dependencies>‘;’]
<template>
end object <ob_id>

```
<dependencies>        ::= specializing from <ob_id_list>
                          | role of <ob_id_list>

<class_spec>          ::= object class <class_id>
                            <type_header>
                            <template>
                          end object class <class_id>

<type_header>         ::= <data_type_section>
                          identification <id_def_seq>‘;’
                          | specializing from <class_id_list> [where <formula>]‘;’
                          | role of <class_id_list>‘;’

<id_def>              ::= <attribute_id> ‘:’ <domain>

<gen_class_spec>      ::= generalization class <class_id>
                            of <class_id_list>‘;’
                              <gen_template>
                          end generalization class <class_id>

<gen_template>        ::= template
                            [<data_type_section>]
                            [<components_section>]
                            [<attributes>]
                            [<events>]
                            renaming
                              case
                                <selection_seq>

<selection>           ::= ‘SELF’ in <class_id>‘:’
                            <renaming_clause_seq>

<object_interface>    ::= interface <interface_id>
                            encapsulating <item_list>‘;’
                              [<data_type_section>]
                              [<attributes>]
                              [<events>]
                              [<derivations>]
                          end interface <interface_id>
```

```
<class_interface>      ::=  interface class <interface_id>
                              encapsulating <item_list>';'
                              [selection where <formula>';']
                                [<attributes>]
                                [<events>]
                                [<derivations>]
                            end interface class <interface_id>
```

Relationships

```
<rel_def>              ::=  relationship [<rel_id>] between <item_list>';'
                              [<data_type_section>]
                              [<constraints>]
                              [<interaction>]
                            end relationship [<rel_id>]

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

Societies

```
<spec_item>            ::=  <template_spec>
                            | <object_spec>
                            | <class_spec>
                            | <gen_class_spec>
                            | <object_interface>
                            | <class_interface>
                            | <rel_def>

<society_spec>         ::=  object society <society_id>
                              [including <domain_list>';']
                                <spec_item_seq>
                            end object society <society_id>';'
```

Appendix B: Predefined Data Signature

In this appendix, we want to list the data signature we assume to be predefined for TROLL. Please note that for the syntax we adopt the convention of OBJ3 in that the operand positions are marked by _. Note also that most operators are overloaded.

Sorts

`bool`	(Boolean symbols)
`nat`	(Natural numbers with Zero)
`integer`	(integer numbers)
`real`	(rational numbers as implemented in computers)
`char`	(printable symbols)
`string`	(sequences of characters)

Usually, these sorts are implemented on computers. There, we have at hand a large number of constant symbols. In our framework, constants are to be regarded as operations but for obvious reasons we do not want to list all the constants that are usually available.

Operations

Let us now list the signatures of operations over these sorts. We start with operations on the sort `bool`.

`~_` : `bool` → `bool`	(negation)
`_ & _` : `bool`×`bool` → `bool`	(conjunction)
`_ | _` : `bool`×`bool` → `bool`	(disjunction)
`_ ==> _` : `bool`×`bool` → `bool`	(implication)
`_ <==> _` : `bool`×`bool` → `bool`	(equivalence)

Next are other operations that yield a result of sort `bool` (these operations are usually called predicates). For all sorts we assume the equality `=` and inequality `#` predicate to be predefined. Other predicates are:

$$
\begin{aligned}
&\_ \;\texttt{>}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>}\; \_ : \texttt{char}\times\texttt{char} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>}\; \_ : \texttt{string}\times\texttt{string} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>=}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>=}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>=}\; \_ : \texttt{char}\times\texttt{char} \rightarrow \texttt{bool}\\
&\_ \;\texttt{>=}\; \_ : \texttt{string}\times\texttt{string} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<}\; \_ : \texttt{char}\times\texttt{char} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<}\; \_ : \texttt{string}\times\texttt{string} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<=}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<=}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<=}\; \_ : \texttt{char}\times\texttt{char} \rightarrow \texttt{bool}\\
&\_ \;\texttt{<=}\; \_ : \texttt{string}\times\texttt{string} \rightarrow \texttt{bool}
\end{aligned}
$$

Arithmetical operations are assumed to be at least the following:

$$
\begin{aligned}
&\_ \;\texttt{+}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{nat}\\
&\_ \;\texttt{+}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{integer}\\
&\_ \;\texttt{+}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{real}\\
&\_ \;\texttt{-}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{nat}\\
&\_ \;\texttt{-}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{integer}\\
&\_ \;\texttt{-}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{real}\\
&\_ \;\texttt{*}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{nat}\\
&\_ \;\texttt{*}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{integer}\\
&\_ \;\texttt{*}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{real}\\
&\_ \;\textbf{mod}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{nat}\\
&\_ \;\textbf{mod}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{integer}\\
&\_ \;\textbf{div}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{nat}\\
&\_ \;\textbf{div}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{integer}\\
&\_ \;\textbf{div}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{integer}\\
&\_ \;\texttt{/}\; \_ : \texttt{nat}\times\texttt{nat} \rightarrow \texttt{real}\\
&\_ \;\texttt{/}\; \_ : \texttt{integer}\times\texttt{integer} \rightarrow \texttt{real}\\
&\_ \;\texttt{/}\; \_ : \texttt{real}\times\texttt{real} \rightarrow \texttt{real}
\end{aligned}
$$

For strings and character symbols we assume the following operations for concatenation to be given:

$$
\begin{aligned}
&\_ \circ \_ : \texttt{char}\times\texttt{char} \rightarrow \texttt{string}\\
&\_ \circ \_ : \texttt{string}\times\texttt{char} \rightarrow \texttt{string}\\
&\_ \circ \_ : \texttt{string}\times\texttt{string} \rightarrow \texttt{string}
\end{aligned}
$$

The following *generic* or parameterized sorts are assumed to be predefined (where *elem* is an arbitrary predefined sort):

set($elem$)	(Sets)
list($elem$)	(Lists)
tuple($elem_1, \ldots, elem_n$)	(Tuples or records)

For sets, we have the following operations:

emptyset : $\rightarrow$ `set`	(set constructor)
in _ _: $elem \times$ `set` $\rightarrow$ `bool`	(element of?)
include _ _: $elem \times$ `set` $\rightarrow$ `set`	(put element into set)
remove _ _: $elem \times$ `set` $\rightarrow$ `set`	(remove element from set)
card _ : `set` $\rightarrow$ `nat`	(cardinality)

For lists, we have the following operations:

emptylist : $\rightarrow$ `list`	(list constructor)
in _ _: $elem \times$ `list` $\rightarrow$ `bool`	(element of?)
pos _ _: $elem \times$ `list` $\rightarrow$ `nat`	(position of *elem*)
_ [_] : `list` $\times$ `nat` $\rightarrow$ $elem$	(selection)
append _ _: $elem \times$ `list` $\rightarrow$ `list`	(append element to list)
remove _ _: $elem \times$ `list` $\rightarrow$ `list`	(remove element from list)
delete _ _: `nat` $\times$ `list` $\rightarrow$ `list`	(delete i-th element)
len _ : `list` $\rightarrow$ `nat`	(length)

For tuples, we have only selectors and constructors:

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

Appendix C: An Example Specification

```
object society SmallBankingWorld
  including nat,money,bool,date,real,date,update_type;

/* The data type update_type is assumed to be defined as an enumeration
type:  { Deposit,Withdrawal} */


object class Account
  identification
    data types nat;
    No:  nat:
  template
    data types |Customer|,money,bool,update_type; attributes
      constant Holder:|Customer|;
      Balance:money;
      Red:bool;
      CreditLimit:money;
      derived MaxWithdrawal:  money;
    events
      birth open;
      death close;
      accept_update(in type:update_type,in Amount:money);
      withdrawal(in Amount:money);
      deposit(in Amount:money);
      update_failed;
    constraints
      initially Red = false;
      initially CreditLimit = 0;
      alwaysf (Red => (Balance <= CreditLimit));
    derivation
      attributes
        { Red } => MaxWithdrawal = CreditLimit - Balance;
        { not Red } => MaxWithdrawal = Balance + CreditLimit;
    valuation
```

```
    variables m:money;
    { not Red and (m <= Balance)} => [withdrawal(m)]Balance = m - Balance;
    { not Red } => [deposit(m)]Balance = Balance + m;
    { not Red and (m > Balance)} =>
      [withdrawal(m)]Balance = m - Balance, Red = true;
    { Red } => [withdrawal(m)]Balance = Balance + m;
    { Red and (m >= Balance)} =>
      [deposit(m)]Balance = m - Balance, Red = false;
    { Red and (m < Balance)} => [deposit(m)]Balance = Balance - m;
  behavior
    permissions
      variables m,m1,m2:money; t,t1:update_type;
      { Balance = 0 } close;
      { exists(t1,m1,m2)(not sometime (after(accept_update(t1,m1)))
        since last(after(update_failed)
        or after(deposit(m2))
        or after(withdrawal(m2)))
      } accept_update(t,m);
      { exists(t,t1,m,m1)(sometime (after(accept_update(t,m)))
        since last after(accept_update(t1,m1))
        and t = Withdrawal and (M > MaxWithdrawal)
      } update_failed;
      { exists(t,t1,m,m1)(sometime (after(accept_update(t,m)))
        since last after(accept_update(t1,m1))
        and t = Deposit
      } deposit(m);
      { exists(t,t1,m,m1)(sometime (after(accept_update(t,m)))
        since last after(accept_update(t1,m1))
        and t = Withdrawal and (m <= MaxWithdrawal)
      } withdrawal(m);
    commitments
      variables t:opdate_type; m:money;
      { after(accept_update(t,m)) and t = Deposit} => deposit(m);
      { after(accept_update(t,m)) and t = Withdrawal and (m <= MaxWithdrawal) }
        => withdrawal(m);
end object class Account;


object class ATM
  identification
    data types nat; IdentNo:nat;
  template
    data types nat,bool,money,|CashCard|; attributes
      CashOnHand:money;
      derived Empty:bool;
    events
```

```
  birth set_up; death remove;
  ready; cancel;
  refill(in Amount:money);
  read_card(in C:|CashCard|);
  check_card_w_bank(in Acct:nat:, in PIN:nat);
  card_accepted; bad_PIN_msg; bad_account_msg;
  issue_TA(in Acct:nat,in Amount:money);
  TA_failed_msg; eject_card;
  dispense_cash(in Amount:money);
constraints
  initially CashOnHand = 10000;
derivation
  Empty = (CashOnHand < 100);
valuation
  variables m:money;
  [refill(m)]CashOnHand = CashOnHand + m;
  [dispense_cash(m)]CashOnHand = CashOnHand - m;
behavior
  permissions
    variables n:nat; m:money; C:|CashCard|;
    { not Empty } read_card(C);
    { m <= CashOnHand } issue_TA(n,m);
  patterns
    variables n,p:nat; m:money;
    process ATM_USAGE(n,p,m) = read_card -> case
                                             cancel;
                                             check_card_w_bank(n,p) ->
                                             GO_ON
                                             esac -> eject -> ready
    end process
    where process GO_ON = case
                          bad_account_msg;
                          bad_PIN_msg;
                          card_accepted ->
                          case
                          cancel;
                          DO_IT;
                          esac;
                          esac
    end process
    where process DO_IT = issue_transaction(n,m) ->
                          case
                          dispense_cash(m);
                          TA_failed_msg
                          esac
    end process;
    ATM_USAGE(n,p,m);
```

```
end object class ATM;


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


object class CheckingAccount
  specializing Account;
    template
      data types nat,|CashCard|,set(|CashCard|);
      attributes
        constant PIN:nat;
        Cards:set(|CashCard|);
      events
        assign_card(in C:|CashCard|);
        cancel_card(in C:|CashCard|);
        new_credit_limit(in Amount:money);
      valuation
        variables C:|CashCard|; m:money;
        [new_credit_limit(m)]CreditLimit = m;
        [assign_card(C)]Cards = insert(C,Cards);
        { in(C,Cards)} => [cancel_card(C)]Cards = remove(C,Cards);
      behavior
```

```
    permissions
      variables C:|CashCard|;
      { in(C,Cards) }cancel_card(C);
      { not in(C,Cards) }assign_card(C);
end object class CheckingAccount;


object Bank:
  template
    data types nat,|ATM|,update_type,money,|CashCard|;
    components
      Accounts:SET(CheckingAccount)
    events
      birth establish; death close_down;
      open_account(in No:nat);
      close_account(in No:nat);
      verify_card(in Acct:nat,in PIN:nat,in ATM:|ATM|);
      no_such_account(out ATM:|ATM|);
      bad_PIN(out ATM:|ATM|);
      card_OK(out ATM:|ATM|);
      process_TA(in Acct:nat,in Type:update_type,in Amount:money,in ATM:|ATM|);
      TA_failed(out ATM:|ATM|); TA_OK(out ATM:|ATM|);
      card_request(in AcctNo:nat);
      card_return(in C:|CashCard|);
    behavior
      permissions
        variables n,p,p1:nat; m,m1:money; t,t1:update_type; atm:|ATM|;
        { not Accounts.IN(CheckingAccount(n)) } open_account(n);
        { sometime after(open_account(n)) } close_account(n);
        { exists(t,m,atm)(not sometime after(process_TA(n,t,m,atm)) since last
          (after(TA_failed(n,atm)) or after(TA_OK(n,atm)))
          and Accounts.IN(CheckingAccount(n)))} process_TA(n,t1,m1,atm1);
        /* Each Transaction must be completed before the next for acct n can start */
        { exists(p1)(not sometime after(verify_card(n,p1,atm)) since last
          (after(bad_PIN(atm)) or after(no_such_account(atm))
          or after(card_OK(atm))))} verify_card(n,p,atm);
        /* Each verification for a particular account must be completed before
        the next can start */
        { exists(n,p)(sometime after(verify_card(n,p,atm))
          and Accounts.IN(CheckingAccount(n))
          and (Accounts(CheckingAccount(n)).PIN = p))} card_OK(atm);
        { sometime after(verify_card(n,p,atm)) and
          not Accounts.IN(CheckingAccount(n))} no_such_account(atm);
        { exists(n,p)(sometime after(verify_card(n,p,atm))
          and Accounts.IN(CheckingAccount(n))
          and not(Accounts(CheckingAccount(n)).PIN = p))} bad_PIN(atm);
```

```
        commitments
          {exists(n,p)after(verify_card(n,p,atm))} =>
            (bad_PIN(atm) or no_such_account(atm) or card_OK(atm));
      interaction
        variables t:update_type; m:money; n:nat; CC:|CashCard|;
        open_account(n) >> Accounts.INSERT(CheckingAccount(n));
        close_account(n) >> Accounts.REMOVE(CheckingAccount(n));
        { Accounts.IN(CheckingAccount(n)) and not in(CC,CheckingAccount(n).Cards) }
          => card_request(n) >> Accounts(CheckingAccount(n)).assign_card(CC);
        card_return(CC) >>
          Accounts(CheckingAccount(CashCard(CC).ForAccount)).cancel_card(CC);
        process_TA(n,t,m,atm) >> Accounts(CheckingAccount(n)).accept_TA(t,m);
        { after(process_TA(n,t,m,atm) } =>
          Accounts(CheckingAccount(n)).withdrawal(m) >> TA_OK(n,atm);
        { after(process_TA(n,t,m,atm) } =>
          Accounts(CheckingAccount(n)).deposit(m) >> TA_OK(n,atm);
        { after(process_TA(n,t,m,atm) } =>
          Accounts(CheckingAccount(n)).update_failed >> TA_failed(n,atm)
end object Bank;


object class CashCard
  identification
    ForAccount:nat;
    IdentNo:  nat:
  template
    attributes
      ExpiringDate:date;
    events
      birth issue; death destroy;
      renew(in until:date);
    valuation
      variables d:date;
      [renew(d)]ExpiringDate = d;
end object class CashCard;


object class CashierStation
  identification
    data types |Employee|,money,update_type,nat; No:nat:
  template
    attributes
      Cashier:|Employee|;
      CashOnHand:money;
    events
```

```
      birth set_up; death close;
      assign_new_cashier(|Employee|);
      request_transaction(in deposit?:update_type,in Acct:nat);
      wrong_acct; id_ok; issue_transaction(...);
      TA_failed; TA_OK;
    ...
end object class CashierSation;
```

```
generalization class EntryStation
  of ATM, CashierStation;
    template
      events
        birth set_up; death close;
        ...
      renaming
        case
          SELF in ATM:
            ATM.set_up by set_up;
            ...
          SELF in CashierStation:
            CashierStation.set_up by set_up;
          ...
  end generalization class EntryStation;
```

```
object class ATMCustomer
  role of Person:
  template
    attributes
      HasCards:set(|CashCard|);
    events
      birth bc_atm_customer; death cease;
      active insert_card(in C:|CashCard|,in into:|ATM|);
      active enter_PIN(in PIN:nat,in into:|ATM|);
      card_accepted(out by:|ATM|);
      active enter_amount(in M:money,in into:|ATM|);
      active enter_cancel(in into:|ATM|);
      cash_dispensed(out by:|ATM|);
      card_ejected(out by:|ATM|;
      active take_card(in from:|ATM|);
      active take_cash(in from:|ATM|);
      active request_card(in Acct:nat);
      card_assigned(in C:CashCard);
      active give_card_back(in C:CashCard);
```

```
    valuation
      variables C:|CashCard|;
      [card_assigned(C)]HasCards = insert(C,HasCards);
      { in(C,HasCards)} => [give_card_back(C)]HasCards = remove(C,HasCards);
    behavior
      permissions
        variables C,C1:|CashCard|; atm:|ATM|;
        { in(C,HasCards)} insert_card(C,atm);
        { exists(C1)(sometime after(insert_card(C1,atm))
          since last after(insert_card(C,atm)))}
          enter_PIN(p,atm),enter_cancel(atm);
        { exists(C)( sometime after(card_accepted(atm))
          since last after(insert_card(C,atm)))}
          enter_amount(m,atm),enter_cancel(atm);
        { exists(C)(sometime after(cash_dispensed(atm))
          since last after(insert_card(C,atm)))}
          take_cash(atm);
        { exists(C)(sometime after(card_ejected(atm))
          since last after(insert_card(C,atm)))}
          take_card(atm);
      commitments
        { after(card_read(atm))} => enter_PIN(p,atm) or enter_cancel(atm);
        { after(card_accepted(atm))} => enter_amount(m,atm) or enter_cancel(atm);
        { after(cash_dispensed(atm))} => take_cash(atm);
        { after(card_ejected(atm))} => take_card(atm);
  end object class ATMCustomer;
```

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

```
relationship UseATM between ATMCustomer,ATM:
```

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

```
relationship CardBusiness between Bank, CheckingAccount, CashCard, ATMCustomer:
  data types |ATMCustomer|,|CashCard|,nat;
  interaction
    variables C:|ATMCustomer|; CC:|CashCard|; n:nat;
    ATMCustomer(C).request_card(n) >> Bank.card_request(CheckingAccount(n));
    CheckingAccount(n).assign_card(CC) >>
      ATMCustomer(CheckingAccount(n).holder).card_assigned(CC);
    CheckingAccount(n).assign_card(CC) >> CashCard(CC).issue;
    ATMCustomer(C).give_card_back(CC) >> Bank.card_return(CC);
    CheckingAccount(n).cancel_card(CC) >> CashCard(CC).destroy;
end relationship CardBusiness;
```

```
interface class ATMToBank
  encapsulating ATM:
    events
      check_card_w_bank(in Acct:nat:, in PIN:nat);
      card_accepted; bad_PIN_msg; bad_account_msg;
      issue_TA(in Acct:nat,in Amount:money);
      TA_failed_msg;
end interface class ATMToBank;
```

```
interface class ATMToCustomer
  encapsulating ATM:
    attributes
      Empty:bool;
    events
      active ready;
      active read_card(in C:|CashCard|);
      card_accepted; bad_PIN_msg; bad_account_msg;
      active issue_TA(in Acct:nat,in Amount:money);
      active cancel;
      TA_failed; eject_card;
      dispense_cash(in Amount:money);
    derivation
      Empty = (CashOnHand < 100);
end interface class ATMToCustomer;


interface class ATMToService
  encapsulating ATM:
    attributes
      CashOnHand:money;
    events
      refill(in Amount:money);
end interface class ATMToService;


end object society SmallBankingWorld;
```

Index