Proc. Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)
A. Miola (Ed.), Springer, Berlin, LNCS (1993)

Sketching Concepts and Computational Model of TROLL *light*

M. Gogolla[*], S. Conrad[*], and R. Herzig[*]
TU Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D-38023 Braunschweig, GERMANY
e-mail: `gogolla@idb.cs.tu-bs.de`

Abstract

The specification language TROLL *light* is intended to be used for conceptual modeling of information systems. It is designed to describe the Universe of Discourse (UoD) as a system of concurrently existing and interacting objects, i.e., an object community.

The first part of the present paper introduces the various language concepts offered by TROLL *light*. TROLL *light* objects have observable properties modeled by attributes, and the behavior of objects is described by events. Possible object observations may be restricted by constraints, whereas event occurrences may be restricted to specified life-cycles. TROLL *light* objects are organized in an object hierarchy established by sub-object relationships. Communication among objects is supported by event calling.

The second part of our paper outlines a simplified computational model for TROLL *light*. After introducing signatures for collections of object descriptions (or templates as they are called in TROLL *light*) we explain how single states of an object community are constructed. By parallel occurrence of a finite set of events the states of object communities change. The object community itself is regarded as a graph where the nodes are the object community states reachable from an initial state and the edges represent transitions between states.

1 Introduction

Formal program specification techniques have been receiving more and more attention in recent years. Several approaches to specification are studied nowadays, for example: Specification of functions (VDM, Z [Jon86, BHL90]), abstract data types [EM85, EGL89, EM90, Wir90, BKL$^+$91], predicate logic and extensions like temporal and modal logic, semantic data models [HK87], and process specification (CCS [Mil80], CSP [Hoa85], petri nets [Rei85]). But for different reasons, none of the above approaches seems to meet in isolation all the requirements needed for the conceptual modeling of information systems: The specification of functions and abstract data

[*]Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS) and BMFT under Grant No. 01 IS 203 D (KorSo).

types does not adequately support persistent objects or the interactive nature of information systems, predicate logic seems to be too general, semantic data models do not take into account the evolution of information systems, and process specifications do not reflect structural aspects.

The paradigm of object-orientation seems to offer more natural ways to model systems and to develop modular software. The two areas formal specification and object-orientation are brought together by object-oriented specification. In particular, object-oriented specification tries to take advantage of several specification techniques by combining their capabilities. For this extremely current research field semantic foundations of object-oriented specifications are given for instance in [EGS90, SE91, Mes92b]. But in addition to semantic foundations, concrete specification languages for objects are needed as well. As far as we know OBLOG [SSE87, CSS89, SSG$^+$91, SGG$^+$91, SRGS91] was the first proposal. Based on experience with this language and on results achieved in the ESPRIT BRWG IS-CORE the language TROLL [JSHS91, JSS91] was developed. Other specification languages for objects are ABEL [DO91], CMSL [Wie91], MONDEL [BBE$^+$90], OS [Bre91], and Π [Gab93].

We did not want to re-invent the wheel, and therefore took the specification language TROLL as a basis for our language. The richness of concepts in TROLL allows us to model the Universe of Discourse (UoD) as adequately as possible. With this aim in mind even some partially redundant language concepts are offered by TROLL. Thus it was necessary to evaluate which concepts are redundant and could be disregarded for our language. Other concepts of TROLL, like temporal logic, have been excluded for pragmatical reasons (so as to simplify animation and to make verification manageable). Finally, we obtained a language with a small number of concepts and hence called it TROLL *light*.

However, TROLL *light* is not just a subset of TROLL. Some details have been added or modified in order to round off TROLL *light*. This was necessary because we needed a clear and balanced semantic basis for our specification language. In particular we want to stress the fact that in TROLL *light* classes are understood as composite objects having the class extension as sub-objects. Therefore in contrast to TROLL an extra notion of class is not needed in TROLL *light*. This leads to a more orthogonal use of object descriptions. Over and above that concepts like class attributes, meta-classes, or heterogeneous classes are inherent in TROLL *light* while they had to be introduced in TROLL by additional language features. Second TROLL *light* incorporates a query calculus providing a general declarative query facility for object-oriented databases. For instance, terms of this calculus may be used in object specifications to describe derivation rules for attributes, or to query object communities in an ad hoc manner.

The paper is organized as follows. Section 2 shortly introduces the various language concepts offered by TROLL *light*. A more detailed presentation of TROLL *light* can be found in [CGH92]. In Section 3 a simplified computational model for TROLL *light* is outlined. The last section discusses future work and gives some concluding remarks.

2 Concepts of TROLL *light*

TROLL *light* is a language for describing static and dynamic properties of objects. As in TROLL object descriptions are called templates in TROLL *light*. Because of their pure declarative nature templates may be compared with the notion of class found in object-oriented programming languages. In the context of databases however, classes are also associated with class extensions so that we settled on a fresh designation. Templates show the following structure.

```
TEMPLATE name of the template
  DATA TYPES   data types used in current template
  TEMPLATES    other templates used in current template
  SUBOBJECTS   slots for sub-objects
  ATTRIBUTES   slots for attributes
  EVENTS       event generators
  CONSTRAINTS  restricting conditions on object states
  VALUATION    effect of event occurrences on attributes
  DERIVATION   rules for derived attributes
  INTERACTION  synchronization of events in different objects
  BEHAVIOR     description of object behavior by a CSP-like process
END TEMPLATE;
```

To give an example for templates let us assume that we have to describe authors. For every author the name, the date of birth, and the number of books sold by year have to be stored. An author may change her name only once in her life. An appropriate TROLL *light* specification would hence be:

```
TEMPLATE Author
  DATA TYPES    String, Date, Nat;
  ATTRIBUTES    Name:string; BirthDate:date;
                SoldBooks(Year:nat):nat;
  EVENTS        BIRTH create(Name:string, BirthDate:date);
                      changeName(NewName:string);
                      storeSoldBooks(Year:nat, Number:nat);
                DEATH destroy;
  VALUATION     [create(N,D)] Name=N, BirthDate=D;
                [changeName(N)] Name=N;
                [storeSoldBooks(Y,NR)] SoldBooks(Y)=NR;
  BEHAVIOR      PROCESS authorLife1 =
                  ( storeSoldBooks -> authorLife1 |
                    changeName -> authorLife2 |
                    destroy -> POSTMORTEM );
                PROCESS authorLife2 =
                  ( storeSoldBooks -> authorLife2 |
                    destroy -> POSTMORTEM );
                ( create -> authorLife1 );
END TEMPLATE;
```

Data types are assumed to be specified with a data specification language. In the KorSo project we use SPECTRUM [BFG$^+$92] as a reference language, but other proposals like ACT ONE [EFH83], PLUSS [Gau84], Extended ML [ST86], or OBJ3 [GW88] will do their job just as good. With the `DATA TYPES` section the signature of data types is made known to the current template. For example, referring to `Nat` means that the data sort `nat`, operations like `+ : nat x nat -> nat`, and predicates like `<= : nat x nat` are visible in the current template definition. Note that we employ a certain convention concerning the naming of data types, templates and associated sorts. We use names starting with an upper case letter to denote data types and templates whereas the corresponding sort names start with a lower case letter.

Attributes denote observable *properties* of objects. They are specified in the `ATTRIBUTE` section of a template by $a\,[\,\texttt{(}[p_1\texttt{:}]s_1,\ldots,[p_n\texttt{:}]s_n\texttt{)}\,]\texttt{:}d$, where a is an attribute name generator, d is a sort expression determining the range of an attribute, and $s_1,\ldots,s_n$ $(n \geq 0)$ denote optional parameter sorts (data or object sorts). To stress the meaning of parameter sorts, optional parameter names p_i might be added. The sort expression d may be built over both data sorts and object sorts by applying predefined type constructors. We have decided to include the type constructors `tuple`, `set`, `bag`, `list`, and `union`. Of course other choices can be made. Thereby, *complex* attributes can be specified, e.g., data-valued, object-valued, multi-valued, composite, alternative attributes, and so on. The interpretation of all sort expressions contains the undefined element $\perp$, and therefore all attributes are *optional* by default. Attribute names may be provided with parameters. For example, by the declaration `SoldBooks(Year:nat):nat` a possibly infinite number of attribute names like `SoldBooks(1993)` is introduced. We demand that in a given state of an object only a finite number of attributes takes values different from $\perp$ such that only these attributes have to be stored. A parametrized attribute $a(s_1,\ldots,s_n) : s$ can also be viewed as an attribute $a : \mathrm{set}(\mathrm{tuple}(s_1,\ldots,s_n,s))$, but clearly the formerly inherent functional dependency would have to be described by an explicit constraint. The same works for parametrized sub-object constructors to be discussed later.

Incidents possibly appearing in an object's life are modeled by *events*. Events are specified in the `EVENT` section of a template by $e[\texttt{(}[p_1\texttt{:}]s_1,\ldots,[p_n\texttt{:}]s_n\texttt{)}]$, where e denotes an event generator and $s_1,\ldots,s_n$ $(n \geq 0)$ represent optional parameter sort expressions. To underline the meaning of parameters, optional parameter names p_i may be added. Event parameters are used to define the effects of an event occurrence on the current state of an object (see the explanation of the `VALUATION` section below), or they are used to describe the transmission of data during communication (see the explanation of the `INTERACTION` section below). Special events in an object's life cycle are the `BIRTH` and `DEATH` events with which an object's life is started or ended. Several birth or death events may be specified for the same object. A template may have no death event, but we require a it to have at least one birth event.

The effect of events on attribute values is specified in the `VALUATION` section of a template definition by *valuation rules*, having the following form.

$$[\texttt{\{}\textit{precondition}\texttt{\}}]\ \texttt{[}\textit{event\_descr}\texttt{]}\ \textit{attr\_term}\ \texttt{=}\ \textit{term}$$

Such a rule says that immediately after an event occurrence belonging to the event descriptor *event_descr*, the attribute given by the attribute term *attr_term* has the value determined by destination *term*. The applicability of such a rule may be restricted by a *precondition*, i.e., the valuation rule is applied only if the precondition is true. It is important to note that the precondition as well as both terms are evaluated in the state *before* the event occurrence. Thereby all these items may have free variables which however must appear in the event descriptor. An *event descriptor* consists of an event generator and a (possibly empty) list of variables representing the parameters of an event. By a concrete occurrence of such an event, these variables are instantiated with the actual event parameters in such a way that the precondition and the other terms can be evaluated.

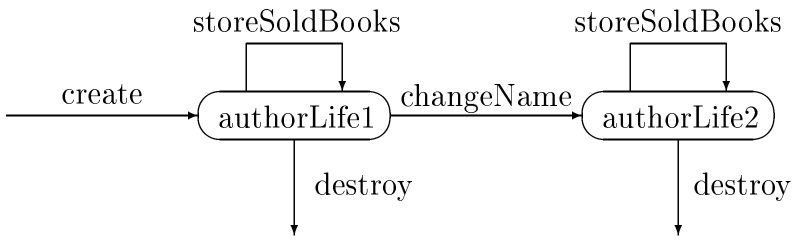

Figure 1: Behavior of authors

To summarize we can say that a valuation is a proposition stating that after the occurrence of a certain event some attributes shall have some special values. The following frame rule is assumed: Attributes which are not caught by a suitable valuation rule of a given event remain unchanged. Before the birth event, all attributes of an object are assumed to be undefined. Thus if the event in question is a birth event, some attributes may remain undefined. It is important to note that an attribute can only be affected by local events, i.e., events which are specified in the same template in which the attribute is specified.

In the `BEHAVIOR` section the possible life-cycles of objects are restricted to admissible ones by the means of *behavior patterns*. Using an abstract characterization, behavior patterns are given by an event-driven machine, which will be called *o-machine* (o for object) in the sequel. This machine has a finite number of *behavioral* states (cf. [SM92]). However, regarding an object together with its attributes we shall generally get an infinite number of object states. To achieve a compact notation within the `BEHAVIOR` section such an o-machine is represented by process descriptions. By process descriptions *event sequences*, which an object must go through, can be specified as well as *event dependent branchings*. Event sequences always lead back into processes, and event alternatives, which produce the same o-machine state transitions, can be listed separated by commas. The possibility of providing every state transition with a precondition allows for *guarded events* (cf. CSP [Hoa85]). We have used the keyword `POSTMORTEM` within the behavior specification to denote that the object vanishes. In Figure 1 the behavior of authors is visualized by the corresponding o-machine representation. Within an object description the behavior section may be missing. In that case life-cycles are unrestricted, i.e., it is only required for life-cycles to start with a birth event and possibly end with a death event.

After dealing with the TROLL *light* features for simple objects we now turn to *composite objects*. In order to combine several authors in a higher-level object, *classes* are usually introduced as containers in object-oriented databases. Here, TROLL *light* does not support an explicit class concept. Classes are viewed as composite objects instead, and therefore classes have to be described by templates as already mentioned in the last section. However, the means of describing the relationship between a container object and the contained objects must be added. This is done by introducing sub-object relationships denoting (exclusive) *part-of* relationships. The following example gives the format of container objects for authors.

```
TEMPLATE AuthorClass
  DATA TYPES   String, Date, Nat;
  TEMPLATES    Author;
  SUBOBJECTS   Authors(No:nat):author;
```

```
    ATTRIBUTES   DERIVED NumberOfAuthors:nat;
    EVENTS       BIRTH createClass;
                       addObject(No:nat,Name:string,BirthDate:date);
                       removeObject(No:nat);
                 DEATH destroyClass;
    CONSTRAINTS  NumberOfAuthors<10000;
    DERIVATION   NumberOfAuthors=CNT(Authors);
    INTERACTION  addObject(N,S,D) >> Authors(N).create(S,D);
                 removeObject(N) >> Authors(N).destroy;
  END TEMPLATE;
```

Within the TEMPLATES section, other (existing) *templates* can be made known to the current template. We assume templates to induce corresponding object sorts. Hence referring to Author means that the object sort author, and the attributes and the event generators of Author are visible in AuthorClass.

An object of sort authorClass will hold finitely many author objects as private components or *sub-objects*. In order to be able to distinguish several authors from a *logical* point of view, an explicit *identification mechanism* is needed. One way to install such a mechanism would be to assign a unique name for each sub-object, e.g., MyAuthor, YourAuthor, ... Indeed, such a name allocation could be expressed in TROLL *light* as follows

```
   SUBOBJECTS  MyAuthor:author; YourAuthor:author; ...;
```

But clearly, in the case of a large number of authors such a procedure is not practicable. A solution is given by parametrized sub-object constructors as shown in the example. As with parametrized attributes, a possibly infinite number of logical sub-object names like Authors(42) can be defined by the sub-object name declaration for authors, but not the author objects themselves. The declaration only states that in context of an object of sort authorClass, author objects are identified by a natural number. In semantic data models the parameter No would be called a *key*. But the parameters need not be related to any attributes of the objects they identify. Each logical sub-object name corresponds to an object of the appropriate object sort. Analogously to attributes, we demand that in each state only a finite number of defined sub-objects exists.

Interaction rules are specified in the INTERACTION part of the template definition. During the lifetime of an author class there will be events concerning the insertion or deletion of authors. In AuthorClass, the insertion of an author should always be combined with the creation of a new author object. In the template definition this is expressed by an event calling rule addObject(N,S,D) >> Authors(N).create(S,D), where N denotes a natural number, S a string, and D a date. The general event calling scheme is expressed as

[{*precondition*}] [*src_obj_term* .]*src_event_descr* >> [*dest_obj_term* .]*dest_event_term* .

Such a rule states that whenever an event belonging to the source event descriptor *src_event_descr* occurs in the source object denoted by *src_obj_term*, and the *precondition* holds, then the event denoted by the destination event term *dest_event_term* must occur simultaneously in the destination object denoted by *dest_obj_term*. If one of the object terms is missing, then the corresponding event descriptor (respectively event term) refers to the current object. The source object term is not allowed to have free

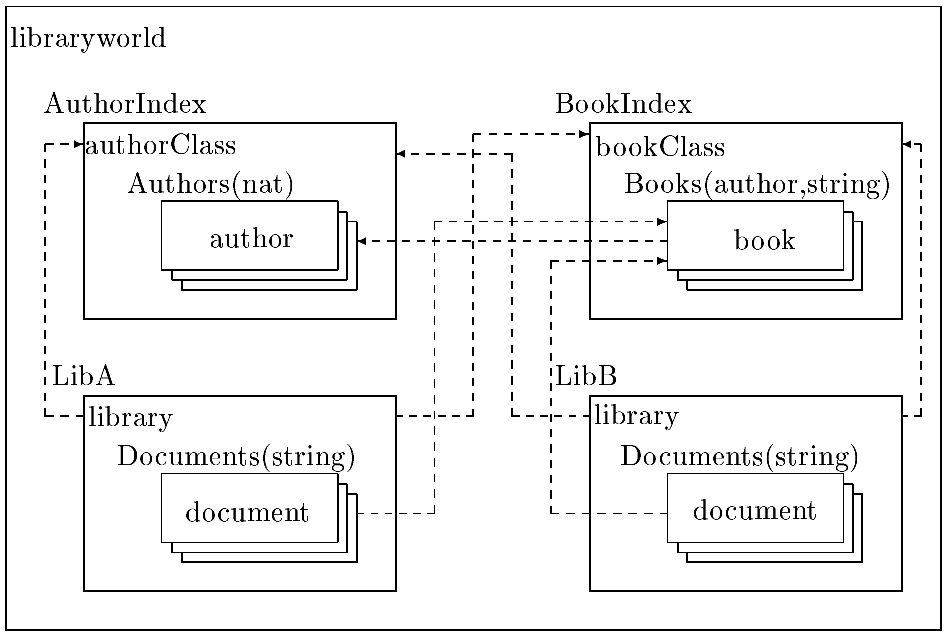

Figure 2: Instance of sort `libraryworld`

variables, but the destination object term, the destination event term, and the precondition are allowed to have free variables, which however have to occur in the source event descriptor. As already mentioned in the explanation of valuation rules, these free variables are instantiated by an event occurrence corresponding to the event descriptor in such a way that the precondition and the other terms can be evaluated. In almost all cases the objects to be called for must be alive. The one and only exception is when a parent object calls for the birth of a sub-object. In this case it is even a requisite that the destination object does not exist already.

In addition to the mentioned language features TROLL *light* offers some further, more sophisticated concepts which will be mentioned only briefly: Possible object states can be restricted by explicit *constraints*, which are specified in the `CONSTRAINTS` section of a template. Constraints are given by closed formulas. For example, an author container of sort `authorClass` may only be populated by less than 10000 authors. Derivable attributes can be specified by stating *derivation rules* which, in fact, are closed terms. In the example we declared the attribute `NumberOfAuthors` of `authorClass` as derived. We employed the function `CNT` which counts the elements of a multi-valued term. Please remember that in the derivation rule `Authors` is a term of sort `set(tuple(nat,author))`. So `NumberOfAuthors` does not need to be set by explicit valuation rules.

Up to now we have only dealt with the description of objects (i.e., templates). In order to attain object instances we must choose one template as the *schema template*, and one fixed object of the corresponding object sort as the *schema object*. Because the schema object is not a sub-object of any other specified object it has no logical object name. Therefore, we give the special name INIT to this object. Then we can make use of templates to derive all other object names by means of a term algebra construction. This will be shown in the next section.

In Figure 2 we give an example of a possible embedding of an object of sort `authorClass`. The considered world consists of an object representing a *library world* which has two libraries, an author index and a book index as sub-objects. The author index and the book index are modeled as class objects having authors respectively books as sub-objects. The libraries have (disjoint sets of) documents as sub-objects. In the figure, objects are represented by rectangles with the corresponding object sorts standing in the upper left corner and the logical sub-object name standing outside. Thereby, sub-object relationship is represented by putting the rectangles of the sub-objects into the rectangle of their common superobject. A similiar approach for representing object structures graphically can be found [KS91].

Within the figure dashed lines are used to state another kind of relationship. In the specification these relationships are given by *object-valued* attributes. For instance, each document refers to a corresponding book, and each book refers to (a list of) authors. Thereby, *object sharing* is possible because several books can have common authors. Furthermore, object-valued attributes can change in the course of time whereas sub-object relationships are fixed. Therefore, object-valued attributes provide a flexible mechanism to model arbitrary relationships between objects.

3 Simplified Computational Model for TROLL *light*

In this section we will outline a simplified computational model for TROLL *light*. Indeed this model is just one of many possible models. Unluckily it is beyond the scope of this paper to discuss a general notion of model for specification of templates. Our computational model serves as a basis for an animation system of the language. For this reason, we use concrete set- and graph-theoretical concepts to describe it. We will not go into detailed technicalities, but we will give an overview and a general taste how the computational model works by means of examples. We assume that a collection of TROLL *light* templates together with one distinguished (schema) template is given. We first define the data part of our model.

Definition 3.1: Data signature and data algebra

A *data signature* $\Sigma_D = (S_D, \Omega_D, \Pi_D)$ is given by

- a set S_D of data sorts,
- a family of sets of operation symbols $\Omega_D =< \Omega_{D,ws} >_{ws \in S_D^* \times S_D}$, and
- a family of sets of predicate symbols $\Pi_D =< \Pi_{D,w} >_{w \in S_D^*}$.

We assume a data signature is interpreted by one fixed term-generated data algebra $DA \in \mathrm{ALG}_{\Sigma_D}$. The set $E_{DA} = \{t_L = t_R \mid t_L, t_R \in T_{\Sigma_D}, t_L^{DA} = t_R^{DA}\}$ denotes the equations induced on Σ_D-terms by DA. □

The algebra DA is assumed to be specified with a data specification language. Next we define signatures for template collections where the names of object sorts, and the names and parameters of sub-object constructors, attributes, events, and o-machine states are given.

Definition 3.2: Template collection signature

A *template collection signature* $\Sigma_O = (S_O, SUB_O, ATT_O, EVT_O, OMS_O)$ is given by

- a set S_O of object sorts — from now on we assume $S := S_D \cup S_O$ —,
- a family of sets of sub-object symbols $SUB_O =< SUB_{O,ows} >_{ows \in S_O \times S^* \times S_O}$,
- a family of sets of attribute symbols $ATT_O =< ATT_{O,ows} >_{ows \in S_O \times S^* \times S}$,
- a family of sets of event symbols $EVT_O =< EVT_{O,ow} >_{ow \in S_O \times S^*}$, and
- a family of finite sets of o-machine states $OMS_O =< OMS_o >_{o \in S_O}$.

The notation $u : o \times s_1 \times ... \times s_n \ \rightarrow s$ stands for $u \in SUB_{O,os_1...s_ns}$. We assume an analogous notation for attribute symbols, event symbols, and o-machine states. □

Example 3.3:

For our running example we have the following template collection signature.

$$\begin{array}{lcll}
S_O & = & \{ & \text{author, authorClass} \quad \} \\
SUB_O & = & \{ & \text{Authors} : \text{authorClass} \times \text{nat} \rightarrow \text{author} \quad \} \\
ATT_O & = & \{ & \text{Name} : \text{author} \rightarrow \text{string}, \\
 & & & \text{BirthDate} : \text{author} \rightarrow \text{date}, \\
 & & & \text{SoldBooks} : \text{author} \times \text{nat} \rightarrow \text{nat}, \\
 & & & \text{NumberOfAuthors} : \text{authorClass} \rightarrow \text{nat} \quad \} \\
EVT_O & = & \{ & \text{create} : \text{author} \times \text{string} \times \text{date}, \\
 & & & \text{changeName} : \text{author} \times \text{string}, \\
 & & & \text{storeSoldBooks} : \text{author} \times \text{nat} \times \text{nat}, \\
 & & & \text{destroy} : \text{author}, \\
 & & & \text{createClass} : \text{authorClass}, \\
 & & & \text{addObject} : \text{authorClass} \times \text{nat} \times \text{string} \times \text{date}, \\
 & & & \text{removeObject} : \text{authorClass} \times \text{nat}, \\
 & & & \text{destroyClass} : \text{authorClass} \quad \} \\
OMS_O & = & \{ & \text{authorClasslife} : \text{authorClass}, \\
 & & & \text{authorLife1, authorLife2} : \text{author} \quad \}
\end{array}$$

□

After having fixed the syntax of our template collections we can define a name space for possible objects to be considered.

Definition 3.4: Universe of possible object names

We construct the *universe of possible object names* by considering the term algebra generated by data operations in Σ_D, sub-object operations in SUB_O, and a special constant INIT for the object sort of the distinguished (schema) template. Additionally we factorize this term algebra by the equations E_{DA}:

$UNIV :=< T_{\Sigma_D + SUB_O + \text{INIT}, E_{DA}, o} >_{o \in S_O}$. □

Example 3.5:

For our running example template `AuthorClass` is the schema template and therefore INIT is a constant for object sort `authorClass`.

$UNIV_{authorClass} = \{$ INIT $\}$

$UNIV_{author} = \{$ ..., Authors(INIT,23), Authors(INIT,42), ... $\}$ □

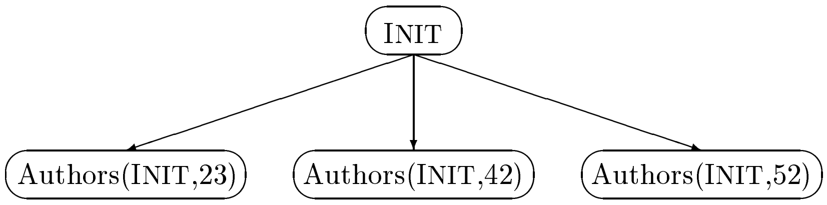

Figure 3: Objects in example object community state

Up to now we have only considered the template signatures. Now we will assume that families of constraints, valuation formulas, derivation rules, interaction rules, and behavior descriptions constitute the template specifications. We first define how object community states, i.e., snapshots describing the shape of the object community after a number of events occurred, look like.

Definition 3.6: Object community state

An *object community state* is a finite tree (N, E) with root INIT (provided $N \neq \emptyset$) such that the conditions given next are satisfied.

- The nodes N are given as an S_O-indexed family of finite sets with $N_s \subseteq UNIV_s$.
- The edges are given by $E = \{(t, u(t, t_1, ..., t_n)) \mid t, u(t, t_1, ..., t_n) \in N\}$.
- With the above it is possible to define $I_s := \begin{cases} DA_s & \text{if } s \in S_D \\ N_s & \text{if } s \in S_O \end{cases}$.
- A sub-object symbol $u : o \times s_1 \times ... \times s_n \ \rightarrow s$ is interpreted as a mapping $u_I : N_o \times I_{s_1} \times ... \times I_{s_n} \rightarrow N_s \cup \{\perp\}$.
  $u_I : (t_o, t_1, ..., t_n) \mapsto \begin{cases} u(t_o, t_1, ..., t_n) & \text{if } u(t_o, t_1, ..., t_n) \in N_s \\ \perp & \text{otherwise} \end{cases}$
  Each u_I is already fixed by the choice of the nodes N.
- An attribute symbol $a : o \times s_1 \times ... \times s_n \ \rightarrow s$ is interpreted as a mapping $a_I : N_o \times I_{s_1} \times ... \times I_{s_n} \ \rightarrow I_s \cup \{\perp\}$. In contrast to u_I the interpretation of a non-derived attribute is free.
- Each object has a fixed o-machine state determined by a family of functions $< \delta_o >_{o \in S_O}$ with $\delta_o : N_o \rightarrow OMS_o$ for each $o \in S_O$.
- The above interpretation must obey the definitions given for derived attributes and must fulfill the specified static constraints.

The initial object community state S_0 is defined by $S_0 = (\emptyset, \emptyset)$. The set of all object community states is denoted by STATES. □

Example 3.7:

Consider the object community state for the running example depicted in Figure 3. Here only the sub-object relationships and their tree structure but not the attributes are shown. More general graph structures with shared objects can be realized with object-valued attributes.

`author`	Authors(INIT,52)	Authors(INIT,23)	Authors(INIT,42)
Name_I	'Mailer'	'Mann'	'Sartre'
BirthDate_I	(01.04.1929)	(06.06.1875)	(21.06.1905)
SoldBooks_I	$1981 \mapsto 300$ $1983 \mapsto 600$	$1979 \mapsto 400$ $1980 \mapsto 700$	$1985 \mapsto 600$
δ_{author}	authorLife2	authorLife1	authorLife1

`authorClass`	INIT
NumberOfAuthors_I	3
$\delta_{authorClass}$	authorClasslife

Figure 4: Attributes and o-machine states in example object community state

The attribute and o-machine state values for the example state are characterized by the table in Figure 4. In the state there is an author which has already changed his name from 'Kundera' to 'Mailer' and consequently his o-machine state is 'authorLife2' disallowing a second name change.

If we had a constraint restricting authors to be born in the 20th century (for example by requiring `CONSTRAINTS BirthDate>=(01.01.1900)`), then the above would not be a valid object community state. On the other hand we even get a valid object community state if δ_{author} maps Authors(INIT,52) to 'authorLife1'. □

So far we have not considered events. The occurrence of events changes the state of the object community. Due to our mechanism for event calling the occurrence of one event may force other events to occur as well. Therefore state transitions will in general be induced by sets of events.

Definition 3.8: Event and closed event set

A possible *event* in an object community state (N, E) consists of an event symbol $e\ :\ o \times s_1 \times ... \times s_n$ together with appropriate actual parameters $(t_o, t_1, ..., t_n) \in (N_o \times I_{s_1} \times ... \times I_{s_n})$ where the first parameter — the object for which the event takes place — is written in dot notation before the event symbol: $t_o.e(t_1, ..., t_n)$. A finite set of events $\{\underline{e}_1, ..., \underline{e}_n\}$ is called *closed* with respect to event calling, if there does not exist another event $\underline{e}$ and an interaction rule such that an event $\underline{e}_i$ calls for $\underline{e}$.

The set of all possible closed events sets is denoted by EVENTS. □

Definition 3.9: Object community state transitions

The specification of the templates determine a relation

TRANSITION $\subseteq$ STATES $\times$ EVENTS $\times$ STATES

given as follows. Let an object community state S and a closed event set $\{\underline{e}_1, ..., \underline{e}_n\}$ be given. (S,$\{\underline{e}_1, ..., \underline{e}_n\}$,S') $\in$ TRANSITION, if the following conditions are satisfied.

- The object community state S' differs from S only by modifications forced by evaluating valuation rules of events from the event set (modification of attribute functions) and by birth and death events from the event set (modification of the sub-object structure).

- There is at most one event per object in the event set.
- Each event fits into its object's life-cycle determined by the behavior patterns.
- In the state S' the o-machine states have changed in accordance with the behavior patterns.

Recall, if S' is a valid object community state, then the constraints are satisfied by S'. □

Example 3.10:

We give examples of transitions for the object community state in Example 3.7 and also examples for event sets which do not induce such transitions. One possible transition is induced by the closed event set:

{ INIT.addObject(37,'Mann',(27.03.1871)),
  Authors(INIT,37).create('Mann',(27.03.1871)) }.

The state belonging to the transition is characterized as follows:

- $N'_{authorClass} = N_{authorClass}$
- $N'_{author} = N_{author} \cup$ { Authors(INIT,37) }
- Name$'_I$ = Name$_I$ $\cup$ { Authors(INIT,37) $\mapsto$ 'Mann' }
- BirthDate$'_I$ = BirthDate$_I$ $\cup$ { Authors(INIT,37) $\mapsto$ (27.03.1871) }
- SoldBooks$'_I$ = SoldBooks$_I$
- NumberOfAuthors$'_I$: INIT $\mapsto$ 4
- $\delta'_{author} = \delta_{author} \cup$ { Authors(INIT,37) $\mapsto$ authorLife1 }

After this, another transition may be performed by executing the closed event set:

{ INIT.removeObject(52), Authors(INIT,52).destroy }

As examples for events which cannot be elements of event sets inducing a state transition for the state in Example 3.7 we mention the following events.

- Authors(INIT,52).changeName('Kundera')
- Authors(INIT,52).create('Kundera',(01.04.1929))
- INIT.addObject(23,'Mann',(06.06.1875))
- INIT.removeObject(13)
- INIT.createClass

A sequence of event sets — indeed, this is one sequence of many possible ones — leading to the state in Example 3.7 would start with { INIT.create } and then one could sequentially create the authors 'Kundera', 'Mann', and 'Sartre'. Afterwards one could have the five events updating the `SoldBooks` attribute. The last step could be the event changing the `Name` attribute of author 'Kundera' to 'Mailer'. This sequence of event sets has a rather non-parallel nature. But parallel occurrence of events is supported as well. Consider a situation where authors 'Kundera' and 'Mann' already exist and have not changed their name. In this situation the following closed event set could induce a state transition.

{ INIT.addObject(42,'Sartre',(21.06.1905)),

Authors(INIT,42).create('Sartre',(21.06.1905)),

Authors(INIT,52).changeName('Mailer'),

Authors(INIT,23).storeSoldBooks(1980,700) }

These four events occur concurrently in different objects. □

We now come to our last and most important notion, namely the object community. An object community is the collection of all object community states reachable from the initial state together with the transitions between these states. Thus an object community reflects structure and behavior of the described system.

Definition 3.11: Object community

The semantics of a template collection specification, i.e., the specified *object community*, is a directed, connected graph where the nodes are the object community states and edges represent closed event sets. All nodes must be reachable from the initial state S_0. □

In general, the object community will be an arbitrary directed graph and not a tree or dag structure. From the initial state subsequent states are reached by performing a birth event for the (schema) object INIT. Not all possible object community states contribute to the object community but only those which are reachable from the initial state. Our computational model implies for instance severe restrictions on the interpretation of attributes: $a_I(t_o, t_1, ..., t_n)$ yields values different from $\perp$ only for a finite number of arguments, because only a finite number of events lies between the initial state and a reachable object community state.

4 Conclusion and Future Work

Although we have now fixed the language, a lot of work remains to be done. Special topics for our future investigations will be formal semantics of TROLL *light*, animation, certification, and an integrated development environment.

Here, we have presented a computational model for TROLL *light* serving as a basis for implementation. But what we desire is an abstract *formal semantics* for TROLL *light* which is compositional. The approach presented in [Mes92a, Mes92b] seems to us very promising since our computational model already reflects the idea to have systems with structured states and transitions between these states. On the other hand the entity algebra approach of [Reg90] or the Berlin projection specifications [EPPB$^+$90] may be suitable tools as well. A detailed overview on algebraic semantics of concurrency can be found in [AR93].

Animation is another aspect of our future studies. Formal specifications cannot be proved to be correct with regard to some informal description of the system to be specified. Only validation can help to ensure that a formally specified system behaves as required. Animation of specifications seems to be an important way to support validation in an efficient manner. Therefore, we develop and implement an animation system for TROLL *light* specifications supporting designers in validating object communities against their specification.

Another aspect we work on is *certification*. *Consistency* is an essential prerequisite for specifications to be used. For instance, animation, as well as proving properties of specifications, requires consistent specifications. Therefore, requirements for specifications to be consistent have to be worked out. *Verification* is needed to prove the properties of specifications. Writing all intended properties of some objects into their template specification seems to be unrealistic, because in this way specifications would become larger and larger and finally nobody would be able to read and understand such specifications.

We also design an *integrated development environment* [VHG+93] for our specification language. In principle, this environment will support all phases of software development: Information analysis, formal specification, animation, certification, and transformation into executable code. Of course we do not expect that at the end of our project there will be a complete development environment. Consequently we are going to implement the important parts of such a system.

References

[AR93] E. Astesiano and G. Reggio. Algebraic Specification of Concurrency. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification — Proc. 8th Workshop on Specification of Abstract Data Types*, pages 1–39. Springer, Berlin, LNCS 655, 1993.

[BBE+90] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An Object-Oriented Specification Language. Département d'Informatique et de Recherche Opérationnelle, Publication 748, Université de Montréal, 1990.

[BFG+92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM — An Informal Introduction (Version 0.3). Technical Report TUM–I9140, Technische Universität München, 1992.

[BHL90] D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z — Formal Methods in Software Development.* Springer, LNCS 428, 1990.

[BKL+91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. The Compass Working Group: *Algebraic System Specification and Development.* Springer, Berlin, LNCS 501, 1991.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer, LNCS 562, 1991.

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

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. OBL-89 Users Manual (Version 2.3). Internal report, INESC, Lisbon, 1989.

[DO91] O.-J. Dahl and O. Owe. Formal Development with ABEL. Technical Report 159, University of Oslo, 1991.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with Two Levels of Semantics. Technical Report 83-03, Technische Universität Berlin, 1983.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen – Eine Einführung in die Theorie.* Teubner, Stuttgart, 1989.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (Proc. REX/FOOL Workshop, Noordwijkerhood (NL))*, pages 203–228. Springer, LNCS 489, 1990.

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

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints.* Springer, Berlin, 1990.

[EPPB$^+$90] H. Ehrig, F. Parisi-Presicce, P. Boehm, C. Rieckhoff, C. Dimitrovici, and M. Große-Rhode. Combining Data Type and Recursive Process Specifications Using Projection Algebras. *Theoretical Computer Science*, 71:347–380, 1990.

[Gab93] P. Gabriel. The Object-Based Specification Language Π: Concepts, Syntax, and Semantics. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification — Proc. 8th Workshop on Specification of Abstract Data Types*, pages 254–270, Berlin, 1993. Springer, LNCS 655.

[Gau84] M.-C. Gaudel. A First Introduction to PLUSS. Technical Report, Université de Paris-Sud, Orsay, 1984.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Research Report SRI-CSL-88-9, SRI International, 1988.

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

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

[Jon86] C.B. Jones. *Systematic Software Developing Using VDM.* Prentice-Hall, Englewood Cliffs (NJ), 1986.

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

[JSS91] R. Jungclaus, G. Saake, and C. Sernadas. Formal Specification of Object Systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91, Brighton*, pages 60–82. Springer, Berlin, LNCS 494, 1991.

[KS91] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proc. 7th Int. Conf. on Data Engineering, Kobe (Japan)*, pages 530–539, 1991.

[Mes92a] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegener, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* MIT Press, 1992. To appear.

[Mes92b] J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–156, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, Berlin, 1980.

[Reg90] G. Reggio. Entities: An Institution for Dynamic Systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, pages 246–265. Springer, LNCS 534, 1990.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer, Berlin, 1985.

[SE91] A. Sernadas and H.-D. Ehrich. What is an Object, after all? In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 39–70. North-Holland, 1991.

[SGG$^+$91] C. Sernadas, P. Gouveia, J. Gouveia, A. Sernadas, and P. Resende. The Reification Dimension in Object-Oriented Database Design. In D. Harper and M. C. Norrie, editors, *Proc. Int. Workshop on Specification of Database Systems*, pages 275–299. Springer, 1991.

[SM92] S. Shlaer and S.J. Mellor. *Object Life Cycles: Modeling the World in States.* Yourdon Press computing series, Prentice-Hall, Englewood Cliffs (NJ), 1992.

[SRGS91] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large Object-Oriented Design of Information Systems. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Proc. Object-Oriented Approach in Information Systems*, pages 209–232. North Holland, 1991.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stocker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Data Bases (VLDB)*, pages 107–116. Morgan-Kaufmann, Palo Alto, 1987.

[SSG$^+$91] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. OBLOG — Object-Oriented Logic: An Informal Introduction. Technical report, INESC, Lisbon, 1991.

[ST86] D.T. Sannella and A. Tarlecki. Extended ML: An Institution-Independent Framework for Formal Program Development. In *Proc. Workshop on Category Theory and Computer Programming*, pages 364–389. Springer, LNCS 240, 1986.

[VHG$^+$93] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich. Towards Reliable Information Systems: The KORSO Approach. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Advanced Information Systems Engineering, Proc. 5th CAiSE'93 Paris*, pages 463–482. Springer, LNCS 685, 1993.

[Wie91] R. Wieringa. Equational Specification of Dynamic Objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, 1991.

[Wir90] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. Elsevier, North-Holland, 1990.