Compositional description of object communities with TROLL *light*[1]

Rudolf Herzig, Stefan Conrad, and Martin Gogolla

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329 - D-38023 Braunschweig - Germany
Tel : (49) 531 391 7442 - Fax : (49) 531 391 3298 - email : herzig@idb.cs.tu-bs.de

ABSTRACT. TROLL light is a language for the modeling of information systems. It is designed to describe the Universe of Discourse (UoD) as a system of concurrently existing and interacting objects. 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 subobject relationships. Communication between objects is supported by event calling. Apart from introducing the various possibilities for the syntactical description of objects, we aim to describe how the state of an object community may be changed by event occurrences.

KEY WORDS. conceptual schema specification, formal methods, object-orientation, state transition system

1. Introduction

1.1. *Object-oriented specification*

Object-oriented approaches to conceptual modeling of information systems promise to improve existing modeling practices. The strength of these approaches is seen in the use of the same concepts in both the requirements analysis and the system design (Rolland/Cauvet, 1992). Object modeling allows for a uniform treatment of structural and dynamic aspects of complex systems, and it also supports a uniform modeling of the system and its environment (Fiadeiro et al., 1992; Ehrich et al., 1993).

The first models used in the system development process should be as close to the application domain as possible. Therefore expressive object-oriented specification languages are needed. As far as we know OBLOG (Costa et al., 1989) was one of the first proposals. Based on experience with this language and on results achieved in the ESPRIT Basic Research Action IS-CORE the language TROLL (Jungclaus et al., 1991) was developed. TROLL allows to write highly declarative specifications so that further transformations towards executable specifications are needed. In this paper we present the language TROLL *light* (Conrad et al., 1992) as a reasonable step in this

[1]in: Proc. Basque Int. Workshop on Information Technology (BIWIT'94): Information Systems Design and Hypermedia, Biarritz, 7.2.-9.2.94, Cépaduès-Éditions, Toulouse (France), pp. 183–194

direction. The following points show some characteristic features in which TROLL *light* is more implementation-oriented than TROLL.

Uniform view of objects: While TROLL supports the conceptual separation of role objects, relationship objects, interface objects, etc., TROLL *light* adopts a uniform view to objects.

No temporal logic: While TROLL provides temporal logic for the declarative specification of object behavior, TROLL *light* is limited to operational specification of object behavior.

Simple process model: While TROLL proposes rather expressive process descriptions concerning the operational specification of object behavior, TROLL *light* process descriptions are at the level of finite behavior diagrams.

1.2. *Classes, schemas, and further meta levels*

A database normally contains lots of formal objects representing real-world entities. Usually many of such formal objects share common structure and common behavior. Therefore in object-oriented databases the notion of *class* is introduced firstly to describe such common properties and secondly to provide a container to collect similar objects. Finally a database *schema* is a collection of classes.

This three-layer structure of object organization is very popular in the database field. For example, it can also be found in relational databases where tuple values represent real-world entities, relations are used as containers for tuple values, and a relational database schema combines relations. However there have been also some attempts to break with this three-level structure by providing additional means for *vertical structuring*. For example, in (De Troyer/Janssen, 1993) *schema object types* are suggested for the modularization of large conceptual schemas, and (Paton et al., 1993) emphasizes the use of *metaclasses* in object-oriented systems. In this paper the language TROLL *light* gives an idea of vertical structuring simply by specifying subobject relationships as part of object descriptions.

Interestingly, TROLL *light* exclusively settles on object descriptions called *templates* for the specification of complex systems. In contrast to many approaches (and even in contrast to TROLL) there is no explicit notion of class or schema in TROLL *light*. As it may be compared with the first role of classes mentioned above, templates describe the common properties of a possibly large class of objects. But with respect to the second role (the container function) classes are viewed as objects in their own right so that separate object descriptions must be given for containers. This leads to a more orthogonal use of object descriptions. Especially concepts like class attributes, meta-classes, or heterogeneous classes are inherent to TROLL *light* while they would have to be introduced in TROLL by additional language features. Importantly all these considerations are also valid for schemas or meta classes.

The remainder of this paper is organized as follows. Object descriptions are built on top of data types. Therefore the specification of data types is briefly discussed in Section 2, before object specification by means of TROLL *light* templates is described in detail in Section 3.1 (structural aspects of objects) and Section 3.2 (dynamic aspects of objects). In Section 3.3 we show how an object community is installed by fixing a certain schema object. The possible evolution of an object community is outlined in Section 3.4. We conclude with some remarks concerning future work in Section 4.

2. Data type specification

A data type normally contains a data sort as carrier set for data values and associated operations and predicates. We assume a certain set of data types like *String*, *Int*, *Real* to be predefined. Besides that arbitrary user-defined data types are possible, either by applying predefined sort constructors or by algebraic specification.

2.1. *Generic data sorts*

For the ad-hoc specification of data types a set of predefined sort constructors is available. More precisely we propose the constructors *set*, *list*, and *bag* for describing multi-valued domains, *tuple* for composite domains,

```
graph = tuple(Nodes:set(int),
              Edges:set(tuple(Start:int, End:int)))
```

and *union*

```
address = union(PA:privateAddress, BA:BusinessAddress)
```

for alternative domains. These constructors can be applied repeatedly to construct domains of any complexity, for example even a so called *complex object model* database (Abiteboul et al., 1989).

Generic sorts are associated with a number of predefined operations and predicates called *aggregate functions*. For example, `CNT` counts the elements in a set (list, bag), `IN` denotes the membership predicate, and `PRJ`i denotes projection on tuple values. For details and further examples see (Gogolla/Hohenstein, 1991).

2.2. Abstract data types

There exists various styles for the specification of data types but in particular the specification of abstract data types by means of algebraic equations (Ehrig/Mahr, 1985) is well-suited to give an *implementation-independent* description of functionality.

```
Stack =   { USES Nat;
            SORT stack;
            OPNS empty: -> stack;  push: stack nat -> stack;
                 pop: stack -> stack;  top: stack -> nat;
            EQNS pop(push(S,N))=S; top(push(S,N))=N; ... }
```

Object descriptions which are called *templates* in TROLL *light* may refer to arbitrary data types whether they are specified in an ad-hoc manner or by means of algebraic equations. This is specified in the `DATA TYPES` clause of a template. Importing data types means to make data sorts and associated operations and predicates public in the current template.

3. Object descriptions

The illustration of our language for describing objects is based on an example concerning the description of a library world. Such a library world consists of two libraries having common access to an author index and a book index. An author index contains author objects, a book index contains book objects, and the libraries manage documents. We give TROLL *light* object descriptions for some of these objects. Therefore we proceed bottom-up.

```
TEMPLATE Author
  DATA TYPES   String, Date, Nat;
  ATTRIBUTES   Name:string; DateOfBirth:date;
               SoldBooks(Year:nat):nat;
  EVENTS       BIRTH create(Name:string, DateOfBirth:date);
                     changeName(NewName:string);
                     storeSoldBooks(Year:nat, Number:nat);
               DEATH destroy;
  VALUATION    [create(N,D)] Name=N, DateOfBirth=D;
               [changeName(N)] Name=N;
               [storeSoldBooks(Y,NR)] SoldBooks(Y)=NR;
  BEHAVIOR     PROCESS Author =
                 ( create -> AuthorLife1 );
               PROCESS AuthorLife1 =
                 ( storeSoldBooks -> AuthorLife1 |
                   changeName -> AuthorLife2 |
                   destroy );
               PROCESS AuthorLife2 =
                 ( storeSoldBooks -> AuthorLife2 |
                   destroy );
END TEMPLATE;
```

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 the name only once in the life.

```
TEMPLATE AuthorContainer
  DATA TYPES   String, Date, Nat;
  TEMPLATES    Author;
  SUBOBJECTS   Authors(No:nat):author;
  ATTRIBUTES   DERIVED NumberOfAuthors:nat;
  EVENTS       BIRTH create;
                     addAuthor(No:nat, Name:string, DateOfBirth:date);
                     removeAuthor(No:nat);
               DEATH destroy;
  CONSTRAINTS  NumberOfAuthors<=10000;
  DERIVATION   NumberOfAuthors=CNT(Authors);
  INTERACTION  addAuthor(N,S,D) >> Authors(N).create(S,D);
               removeAuthor(N) >> Authors(N).destroy;
END TEMPLATE;
```

An author container is an object in which authors can be inserted or from which authors can be removed. Thereby authors are identified by a natural number. An author container must not be populated by more than 10,000 authors.

With further templates for books, book containers, documents, and libraries (see Conrad et al., 1992), the library world can be described as

```
TEMPLATE LibraryWorld
  TEMPLATES   AuthorContainer, BookContainer, Library;
  SUBOBJECTS  AC:authorContainer; BC:bookContainer;
              LibA:library; LibB:library;
  EVENTS      BIRTH create; createContainers; createLibraries;
  INTERACTION createContainers >> AC.create, BC.create;
              createLibraries  >> LibA.create(AC, BC),
                                  LibB.create(AC, BC);
  BEHAVIOR    PROCESS LibraryWorld =
                ( create -> createContainers -> createLibraries );
END TEMPLATE;
```

As demonstrated by the template `AuthorContainer` object descriptions may refer to other object descriptions. This is specified in the `TEMPLATES` section of a template. Informally speaking importing a template means to announce all the things which are known in the corresponding template, i.e., all subobject, attribute, and event symbols (see below) as well as the corresponding *object identifier sort*.

A corresponding object identifier sort is implicitly given for each object description. Instances of this sort provide abstract identifiers (surrogates) for objects satisfying the given description. We employ the following convention. Names for templates always start with an upper case letter while the same names starting with a lower case letter are used to denote the corresponding object identifier sorts. For example, `Author` denotes an object description, while `author` denotes the corresponding object identifier sort.

The description of TROLL *light* concepts is presented in two parts. Structural aspects of objects are discussed in the first part while dynamic aspects are subject to the second.

3.1. *Structural aspects of objects*

3.1.1. *Attributes*

Observable properties of objects are described by attributes. They are specified in the `ATTRIBUTES` section of a template by $a{:}s$ in which a denotes an attribute name and s a sort expression determining the range of an attribute. The sort expression s may be built on both data sorts and object identifier sorts by applying the predefined sort constructors of Section 2. Thereby, *complex* attributes can be specified, e.g., data-valued, object-valued, multi-valued, composite, alternative attributes, etc. (cf. Lécluse/Richard, 1989). Moreover attributes can be provided with parameters, as demonstrated by the attribute `SoldBooks` of `Author`.

3.1.2. *Subobject relationships*

TROLL *light* objects are organized in object hierarchies induced by subobject relationships. A subobject relationship must be understood as *exclusive part-of*, i.e., an object cannot be subobject (component) of two superobjects, and *static*, i.e., a subobject cannot move from one superobject to another. The possible subobjects of an object are specified in the `SUBOBJECTS` section of a template by $u{:}s$ in which u denotes a *subobject slot* and s an object identifier sort determining the sort of object identifiers with which a subobject slot can be filled. Analogously to parameterized attributes, parameterized subobject slots are possible too. This is shown by the subobject slot `Authors` of `AuthorContainer`. In a certain state of an object of sort `authorContainer` this slot might be filled with

`Authors:`

nat	author
3102	$(author)_1$
3273	$(author)_2$
7448	$(author)_3$

Instances of object identifier sorts like `author` are considered as non-printable (abstract) values. Therefore we use $(author)_i$ merely to denote the existence of these values. On the other side subobject slots provide *logical names* for objects, i.e. $Authors(3102)$ is a logical name for $(author)_1$.

3.1.3. *Derived attributes*

Normally attribute values are set explicitly by event occurrences. This will be described in Section 3.2. Besides that, it is possible to specify derived attributes the values of which are computed from other information stored in an object base. For that a corresponding derivation rule must be stated in the `DERIVATION` section of a template. TROLL *light* incorporates a general purpose SQL-like *query calculus*. Every variable-free *term* of this calculus matching the specified range of a derived attribute can serve as a derivation rule. For example, the number of authors of an author container is computed by applying the aggregate function `CNT`.

3.1.4. *Integrity constraints*

In turn every formula of the mentioned query calculus without free variables can be used to specify static constraints on object states in the `CONSTRAINTS` section of a template. For example, the number of authors of an author container may not exceed 10,000.

3.2. ***Dynamic aspects of objects***

The description of dynamic properties of objects is based on the specification of events being abstractions of state-modifying operations on objects.

3.2.1. *Events*

Incidents possibly appearing in an object's life are modeled by events. Events are specified in the `EVENTS` section of a template by $e[\texttt{(}[p_1:]s_1,\ldots,[p_n:]s_n\texttt{)}]$ in which e denotes an *event generator* and $s_1,\ldots,s_n$ $(n \geq 0)$ represent optional parameter sorts. Optional parameter names p_i can be added to emphasize the intention of parameters. An *event occurrence* is given by an event generator together with parameter values of appropriate sorts. 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 template to have at least one birth event.

3.2.2. *Effect of events on attributes*

The effect of events on attribute values is specified in the `VALUATION` section of a template by valuation rules $[\{\varphi\}]$ $[e]$ a = d in which φ is a formula, e is an *event descriptor* which is an event generator together with a list of variables representing its parameters, a is an attribute name, and d is a term the sort of which matches the range of the attribute. Such a rule says that immediately after an event occurrence belonging to the event descriptor e, the attribute a has the value determined by the term d. The applicability of such a rule can be restricted by the precondition φ, i.e., the valuation rule can only be applied if the precondition is true. It is important to note that both the precondition and the term are evaluated in the state before the event occurrence.

3.2.3. *Synchronization of events*

Interaction rules allow to synchronize event occurrences in different objects. They are specified in the `INTERACTION` section of a template. An interaction rule is given by $[\{\varphi\}]$ $[so\,.]se$ >> $[do\,.]de$ in which φ is a formula, so and do are object-valued terms, se is an event descriptor which is an event generator together with a list of variables representing its parameters, and de is an *event term* which is an event generator together with a list of parameter terms. Such a rule states that whenever an event belonging to the source event descriptor se occurs in the source object denoted by so, and the precondition φ holds, then the event denoted by the destination event term de must occur simultaneously in the destination object denoted by do. This form of *event calling* was introduced in (Sernadas et al., 1990). If one of the object terms is missing, then the corresponding event descriptor (or event term respectively) refers to the current object. In almost all cases the objects to be called for must be alive. The one and only exception is when a parent calls for the birth of a child. In this case it is a requisite that the destination object is represented by an unoccupied subobject slot.

3.2.4. *Life cycle specification*

Object life cycles are described by sequences of event occurrences starting with a birth event and possibly ending with a death event of an object. In the `BEHAVIOR` section of a template the possible life cycles can be restricted to admissible ones by means of CSP-like process descriptions (Hoare, 1985). These descriptions allow the specification of *event sequences*, which an object must go through, as well as the specification of *event dependent branchings*. Events can be provided with preconditions (*guarded events*). TROLL *light* process descriptions can always be transformed into *behavior diagrams* as demonstrated in Figure 1. A behavior diagram is a sequential automaton (cf. Shlaer/Mellor, 1992) with a finite number of behavior states and a finite number of event-labeled state transitions which can also be provided with preconditions. The behavior state is part of the whole object state. Hence, when regarding an object together with its attributes we shall generally get an infinite number of possible object states. This is also known as an *augmented state transition system*. Within an object description the behavior section may be missing. In that case life cycles

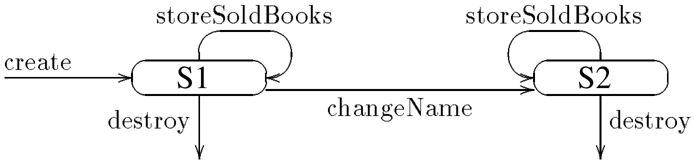

Figure 1: *Behavior diagram of* `Author`

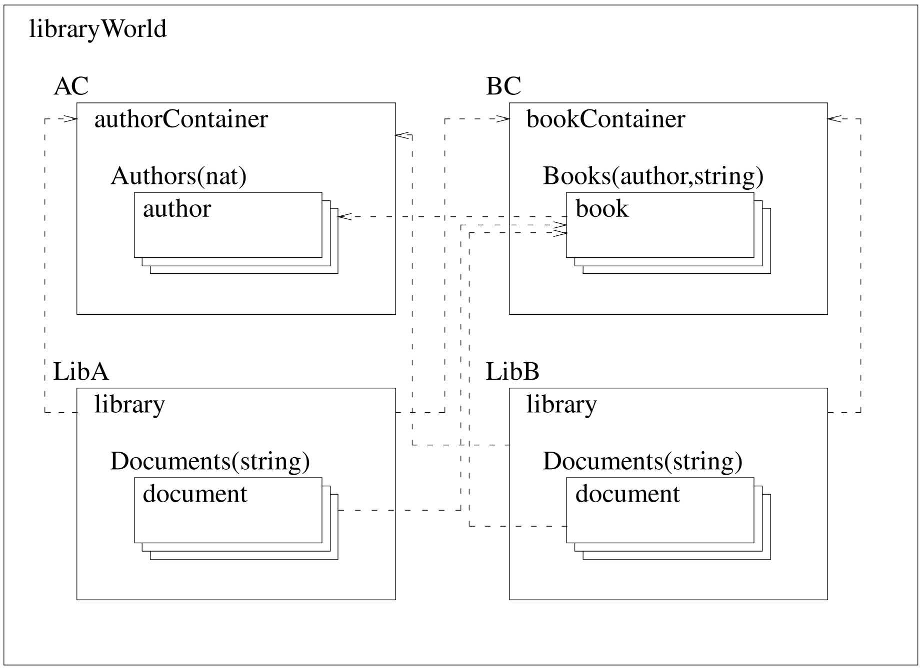

Figure 2: *Structure of an object described by* `LibraryWorld`

are unrestricted, i.e., it is only required for life cycles to start with a birth event and possibly end with a death event.

3.3. *Object communities*

Up to now we only dealt with object descriptions. As mentioned above every object description implies an object identifier sort providing a possibly infinite stock of object identifiers. This also holds for the template `LibraryWorld` given at the beginning of Section 3. However, if we elect `LibraryWorld` as our *schema template*, then we probably focus on exactly one object, the *schema object*, representing the information system in mind. In the following this schema object will be depicted by $\triangledown$ (pronounced *root*). In general a schema object will be the root of a large object community. The structure of an object community induced by an object of sort `libraryWorld` is given in Figure 2.

The diagram is very similar to the object diagrams presented in (Kappel/Schrefl, 1991). In particular it brings to the fore the *modular composition* of TROLL *light* object communities.

Objects are represented by rectangles. The corresponding object sort is noted in the upper left-hand corner of a rectangle. Subobject relationships are illustrated by embedding rectangles in bigger rectangles. Since the logical name of an object is always defined by a subobject slot within the parent object, it appears outside the related rectangle. Dashed lines represent object relationships established by object-valued attributes.

Even the schema object is supposed to be subobject of another object which is in accord with the **principle of object existence** adopted for TROLL *light*: Every object must be subobject of a higher object. Otherwise the very notion of object does not make sense, because an object can only be discerned as an object by a higher object. In particular, this object might be still to be specified (for example, the database management system). Obviously, the birth event of an object of sort `libraryWorld` will be called by such a superobject.

3.4. *An object community as a state transition system*

As demonstrated in the last section an object community can be viewed as a tree the root of which is given by a schema object and all other nodes are subobjects. The object states consisting of attribute and behavior state can be viewed as node information. The state of an object is called *consistent* if all constraints specified in the `CONSTRAINTS` section are fulfilled. For an object community we assume a *global* state so that the state of an object community is consistent if the states of all its members are consistent.

A state transition is induced by a finite set $\mathfrak{E}$ of simultaneous event occurrences in different objects.

$$\sigma \xrightarrow{\mathfrak{E}} \sigma^{new}$$

The event set $\mathfrak{E}$ has to be closed with respect to the specified interaction rules. It is important to note that for a single object only one event can happen at a time (local non-concurrency). Attributes are then modified by applying valuation rules, and the behavior state of an object is altered according to the transition rules described in the `BEHAVIOR` section of a template.

Figure 3 shows a possible evolution of an object community described by the template `LibraryWorld`. State transitions are induced by the occurrence of birth events ($*$), life events (ϟ), and death events ($\dagger$). Filled circles denote objects in life state.

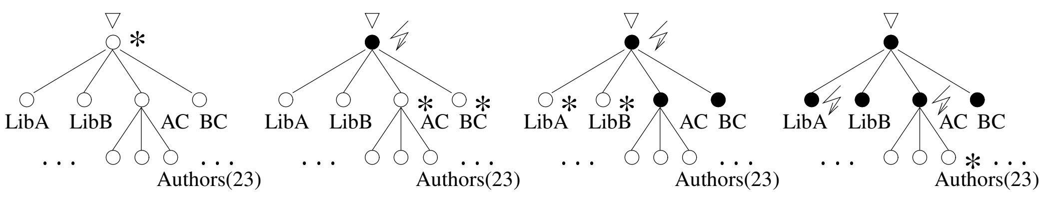

Figure 3: *Evolution of an object community*

1. $\mathfrak{E}^0 = \{ \bigtriangledown.\text{create} \}$

   The life of an object community starts with the occurrence of a birth event for the schema object. Event occurrences are generally depicted by $o.e$ in which o denotes an object surrogate and e an event generator with actual parameters.

According to the behavior patterns specified in object description `LibraryWorld` the state of the schema object turns to a behavior state in which only the event `createContainers` is allowed next.

2. $\mathfrak{E}^1 = \{\ \nabla\text{.createContainers},\ \nabla/AC\text{.create},\ \nabla/BC\text{.create}\ \}$

   The effect of the state transition induced by $\mathfrak{E}^1$ is that the schema object turns to a behavior state in which only the event `createLibraries` can appear next, and that the behavior states of `AC` and `BC` turn to a life state.

   Note that all object surrogates used here turn out to be path expressions rooted at the schema object ∇. There is no constructor for ∇ at the syntactic level. Hence working with concrete surrogates in the conceptual model does not mean to have concrete surrogates in the conceptual schema.

3. Let us assume that the object description `Library` (which is not shown in this paper) contains object-valued attributes `AuthorIndex` and `BookIndex` which are initialized by the birth event of a library. Then it is fixed by the interaction rules of `LibraryWorld` that `LibA` and `LibB` share the same author and book container.

   $$\mathfrak{E}^2 = \left\{ \begin{array}{ll} \nabla\text{.createLibraries}, & \nabla/LibA\text{.create}(\nabla/AC, \nabla/BC), \\ & \nabla/LibB\text{.create}(\nabla/AC, \nabla/BC) \end{array} \right\}$$

   By the state transition induced by $\mathfrak{E}^2$ the schema object turns to a behavior state in which no further events can be accepted.

4. Let us suppose that the object description `Library` (which is not shown here) also contains an event generator `enterAuthor(nat,string,date)` with an interaction rule `enterAuthor(N,S,D) >> AuthorIndex.addAuthor(N,S,D)`.

   $$\mathfrak{E}^3 = \left\{ \begin{array}{l} \nabla/LibA\text{.enterAuthor}(23,\ "J.W.\ Goethe",\ (28.8.1749)), \\ \nabla/AC\text{.addAuthor}(23,\ "J.W.\ Goethe",\ (28.8.1749)), \\ \nabla/AC/Author(23)\text{.create}("J.W.\ Goethe",\ (28.8.1749)) \end{array} \right\}$$

   By the state transition induced by $\mathfrak{E}^3$ a certain author turns to the behavior state S1 (according to the behavior diagram in Figure 1), and the attributes `Name` and `DateOfBirth` of this author are set to the values $"J.W.\ Goethe"$ and $(28.8.1749)$ (according to the valuation rules in the object description `Author`).

These examples can only give a weak taste of how state transitions may look like. Semantic issues are discussed in more detail in (Gogolla et al., 1993).

4. Future work

The language TROLL *light* was developed in the KORSO project which is a joint project of fourteen university and industrial partners, sponsored by the German Ministry of Research and Technology (BMFT). The aim of this project is to improve, develop, and consolidate methods and techniques for the design of high quality software systems. Consequently, several kinds of problems occur like adequate capture of the informal requirements, stepwise development and refinement of the design, and validation of results. Our sub-project especially deals with developing reliable information systems (Vlachantonis et al., 1993). Here TROLL *light* provides the basis for future investigations addressing formal semantics, animation, certification, and an integrated development environment.

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* which includes consistency and verification. 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. Of course, an adequate verification calculus must be based on a compositional proof system respecting the modular structure of TROLL *light* object descriptions.

We also design an *integrated development environment* 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.

Acknowledgement. Work reported here has been partially supported by the German Ministry for Research and Technology (BMFT) under Grant No. 01 IS 203 D (KORSO) and by the CEC under Grant No. 6112 (COMPASS).

References

(Abiteboul et al., 1989) : S. Abiteboul, P.C. Fischer, and H.J. Schek, editors. *Nested Relations and Complex Objects in Databases*. Springer LNCS 361, 1989.

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

(Costa et al., 1989) : J.-F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 Users Manual (Version 2.3)*. Internal report, INESC, Lisbon, 1989.

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

(Ehrig/Mahr, 1985) : H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

(Fiadeiro et al., 1992) : J. Fiadeiro, C. Sernadas, T. Maibaum, and A. Sernadas. *Describing and Structuring Objects for Conceptual Schema Development*. In (Loucopoulos/Zicari, 1992), pp. 117–138.

(Gogolla/Hohenstein, 1991) : M. Gogolla and U. Hohenstein. *Towards a Semantic View of an Extended Entity-Relationship Model*. ACM Trans. on Database Systems, Vol. 16, No. 3, pp. 369–416, 1991.

(Gogolla et al., 1993) : M. Gogolla, S. Conrad, and R. Herzig. *Sketching Concepts and Computational Model of TROLL light*. In A. Miola, editor, Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems DISCO, pp. 17–32. Springer LNCS 722, 1993.

(Hoare, 1985) : C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

(Jungclaus et al., 1991) : 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.

(Kappel/Schrefl, 1991) : G. Kappel and M. Schrefl. *Using an Object-Oriented Diagram Technique for the Design of Information Systems*. In H.G. Sol and K.M. Van Hee, editors, Dynamic Modelling of Information Systems, pp. 121–164. North-Holland, Amsterdam, 1991.

(Lécluse/Richard, 1989) : C. Lécluse and P. Richard. *Modeling Complex Structures in Object-Oriented Databases*. In Proc. 8th ACM Symp. Principles of Database Systems PODS, pp. 360–368, 1989.

(Loucopoulos/Zicari, 1992) : P. Loucopoulos and R. Zicari, editors. *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. John Wiley & Sons, Inc., 1992.

(Paton et al., 1993) : N. Paton, O. Díaz, and M.L. Barja. *Combining Active Rules and Metaclasses for Enhanced Extensibility in Object-Oriented Systems*. Data & Knowledge Engineering, Vol. 10, No. 1, pp. 45–63, 1993.

(Rolland/Cauvet, 1992) : C. Rolland and C. Cauvet. *Trends and Perspectives in Conceptual Modeling*. In (Loucopoulos/Zicari, 1992), pp. 27–48.

(Sernadas et al., 1990) : A. Sernadas, H.-D. Ehrich, and J.-F. Costa. *From Processes to Objects*. The INESC Journal of Research and Development, Vol. 1, No. 1, pp. 7–27, 1990.

(Shlaer/Mellor, 1992) : S. Shlaer and S.J. Mellor. *Object Life Cycles: Modeling the World in States*. Prentice-Hall, 1992.

(De Troyer/Janssen, 1993) : O. De Troyer and R. Janssen. *On Modularity for Conceptual Data Models and the Consequences for Subtyping, Inheritance & Overriding*. In Proc. 9th Int. Conf. on Data Engineering ICDE, 1993.

(Vlachantonis et al., 1993) : 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, pp. 463–482. Springer LNCS 685, 1993.