C. Rolland, F. Bodart, and C. Cauvet, editors, Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93), pages 463–482. Springer, Berlin, LNCS 685, 1993.

Towards Reliable Information Systems: The KorSo Approach*

N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, H.-D. Ehrich

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, W–3300 Braunschweig, Germany
e–mail: vlachant@idb.cs.tu-bs.de

Abstract. Within the compound project KorSo our team is concerned with the research on techniques and methods for the development of reliable information systems on the basis of formal specifications. Our work focuses on the specification language TROLL *light* which allows to describe the part of the world which is to be modeled as a community of concurrently existing and communicating objects by determining their structure as well as their behavior. Moreover we develop and implement a computer aided specification environment for TROLL *light* which permits a prototyping animation as well as the proof of properties of specifications.

1 Introduction

Several approaches for the specification of complex software systems have been put forward in recent years, for example: Specification of functions (VDM, Z [Jon86, BHL90]), abstract data types [EM85, EGL89, EM90, Wir90], predicate logic and extensions like temporal and modal logic, semantic data models [HK87], and process specification (CCS [Mil80], CSP [Hoa85], petri nets [Rei85]).

But in isolation all these approaches are not appropriate for the conceptual modeling of information systems. The aim of the conceptual modeling process is to provide a first precise description of the part of the world which is to be modeled, the so called Universe of Discourse (UoD). By observing a certain UoD one realizes that it consists of complex structured entities with time-varying behavior. These entities exist concurrently and interact with each other. Therefore the idea of object-oriented specification is to represent the real-world entities as formal objects in a specification language. In our project we work with the language TROLL *light* which is a dialect of TROLL [JSHS91]. In TROLL *light* structure as well as behavior of real-world entities can be characterized. Other languages for the specification of objects are for example ABEL [DO91], CMSL [Wie91], GLIDER [CJO92], MONDEL [BBE+90], OBLOG [CSS89], OS [Bre91], and Π [Gab91].

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

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 stress that in TROLL *light* classes are understood as composite objects having the class extension as sub-objects. Therefore, in contrast to TROLL there is no need for an extra notion of class. 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 would have 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. But our specification language is designed as a core language. It is not intended to be a comfortable specification vehicle. However, we kept an eye on maintaining the capability of expressing concepts which we did not include explicitly into our language. By means of concepts included in our language, it should be possible to transform specifications of objects written in a more comfortable specification language into TROLL *light*. Especially, we claim that aspects of inheritance which are not supported directly by the language can be expressed by means of derived attributes and (to some extent) by event calling.

The aim of the KorSo 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 deals with developing reliable information systems on the basis of TROLL *light*. The aim of this paper is to sketch the spectrum of our activities ranging from language design over implementation to questions of logic and semantics.

There are other projects with a similar research purpose like ours. The aim of the compound project STONE [Reh91, UTS+91] is the development of a public domain software engineering environment for educational purposes. One key feature of STONE is that the object-oriented paradigm is applied to all parts of the system developed. Another outcome of this project is the object-oriented database system OBST. The goal of the DAIDA project [JDT89, JMS90] is to build a software engineering environment for designing and implementing information systems. This line of research is continued by the IRIS system [VMK+90] which generates information systems designs from requirement specifications. In [BMHL90] an approach for the translation of semantic data models into object-oriented databases is presented. CADDY [EHH+90, EL92] is a CASE tool which is based on an extended Entity-Relationship model [EGH+92]. It integrates tools for conceptual design of databases and for prototyping of database schemas.

The structure of our paper is as follows. In Sect. 2 we explain how information systems are specified by means of the TROLL *light* development environment. In

Sect. 3 the concepts of our specification language TROLL *light* are described in more detail. Section 4 points out how logic can be employed to prove properties of the specification. Section 5 sketches the semantics of the language. Section 6 closes the paper by reporting the state of the implementation of the TROLL *light* development environment.

2 Developing Information Systems with TROLL *light*

2.1 Conceptual Design

The entire design of an information system requires some separate but interdependent design steps. The first one is the requirements analysis [VB82, SM88, Boo91, SM92] in which the UoD which is the part of the world that is to be modeled must be thoroughly analyzed. After having fixed *what* is to be modeled we must proceed *how* is has to be modeled in terms of a formal model. For convenience, the model used for conceptual design should be as close to the UoD as possible. Here we propose TROLL *light* object descriptions. Further steps may include several transformations of the formal specification in the direction of executable code.

The step from an informal description of the UoD to a first formal specification is very difficult. Therefore we are interested in some guide lines for how to obtain object descriptions which reflect the UoD as adequately as possible. However complex structures need not necessarily be modeled by objects. In particular whenever complex structures represent merely static information, it is also possible to model them as complex values [Hul89, LR89]. After we have fixed the objects to be modeled we can pass through the following design steps.

1. **Building an object hierarchy:** Objects must be organized into an object hierarchy. The topmost object of this hierarchy is the schema object. To fix sub-object relationships we can proceed either top-down or bottom-up.
2. **Modeling object properties:** Observable properties of objects are modeled by attributes.
3. **Modeling of events:** Events are abstractions of state-modifying operations on objects.
4. **Specification of the effect of events on attributes:** The effect of event occurrences on attributes is specified by valuation rules.
5. **Synchronization of events:** The synchronization of events in different objects is specified by interaction rules.
6. **Determining object integrity:** Object integrity is addressed by restricting possible event sequences, by imposing integrity conditions on object states, and by specifying preconditions for the application of valuation or interaction rules.

We cannot expect that one single pass through all these steps yields a satisfactory result. Sometimes we have to go back and repeat some steps until a certain degree of correspondence to the requirements is achieved.

Case studies. The practicability and adequacy of methods and techniques for the development of software is of crucial significance. Therefore a lot of case studies are being undertaken in the KORSO project. With regard to our subproject we are particularly interested in exploring the modeling expressibility of our language TROLL *light*.

First of all TROLL *light* has been used for the specification of a production cell [HV92]. This study showed us that requirements for the specification of active systems are quite different from those for the specification of information systems. For example, questions of liveness are much more urgent in the first case. Currently we are using TROLL *light* for the specification of a system for the processing of medical data. This case study is based on a real application, namely the electronic file of patients at the Herzzentrum Berlin [LCFW92].

2.2 The TROLL *light* Development Environment

An integrated CASE tool is required in order to support the specification of information systems with TROLL *light*. One of the major purposes of the TROLL *light* development environment is the certification of TROLL *light* specifications which includes validation and verification in order to develop reliable information systems. Another future topic is the transformation of TROLL *light* specifications into executable code.

A Methodical View upon the TROLL *light* development environment. Informal work is done especially in the first step (requirements engineering) of the software development process. This step leads to a design phase where a real-world fragment is modeled with an abstract TROLL *light* specification. From this point, the TROLL *light* development environment can be used for the following steps as depicted in Fig. 1 which can be done in a cycle until the desired level of correctness is reached by the specification.

1. **Validation/Animation:** This step consists of prototyping the TROLL *light* specification in a certain environment called animator. An animator provides object windows which enable the user to observe attributes, to open new object windows following sub-object relationships or object-valued attributes, to initiate state transitions by clicking events, to formulate queries against object states, etc. This should help the developer to compare the informal view of the real-world fragment that is to be modeled with the current specification.
2. **Verification:** This means to check the TROLL *light* specification with regard to inconsistencies, deficiencies and assertions given from outside of the specification. Verification is further discussed in Sect. 4.
3. **Modification:** The current specification may be modified after the verification or validation phase.
4. **Transformation:** At last the specification may be transformed, possibly in several steps, towards executable code.

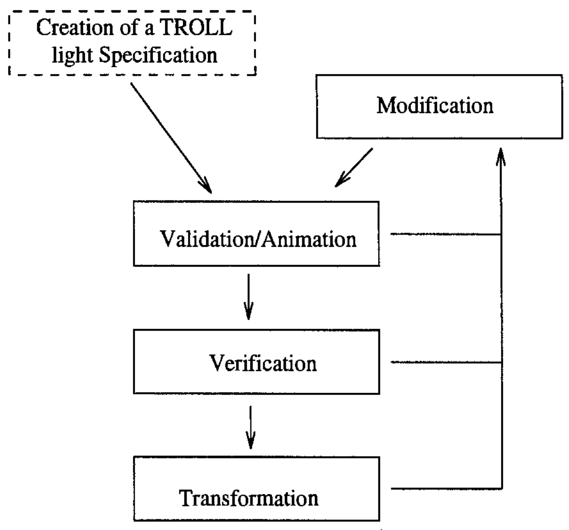

Fig. 1. Methodical view on the TROLL *light* development environment

Technical and Conceptual Aspects. The architecture of the TROLL *light* development environment as depicted in Fig. 2 is an open environment, able to integrate modules, programs, and tools (system units) of any kind in the workspace of a user. All system units run in their own system processes almost independently from each other and can be placed at different machine sites. This distributed architecture was preferred over a monolithic one because the TROLL *light* development environment must be able to integrate existing programs and tools which are written in different programming languages. Other reasons are that monolithic systems are very difficult to handle when they reach a certain extent, they cannot be extended dynamically (always stop, compile, link, and start is required), they have problems in integrating different programming languages, the whole system crashes down in case of an error in a module, etc. So, a distributed architecture seems more adequate for overcoming these problems.

The kernel of the TROLL *light* development environment is the communication manager *HP SoftBench*, a commercial product from the Hewlett Packard Company. It has the task of loosely coupling and functionally integrating the system units. This is done by interprocess communication links of the operating system. The coupling can dynamically be adapted to the requirements of the system units. For this, the communication manager holds information in a repository about which system units belong the environment, which messages they are able to generate, and which messages are relevant to which system units. For instance, a message could be generated by the TROLL *light* editor signaling that a specification has been changed. This message would be broadcasted by the communication manager to each system unit to which this message is relevant. For example, a currently running animation of the same specification would be adapted by the animator by catching the message and by automatically changing to the animation.

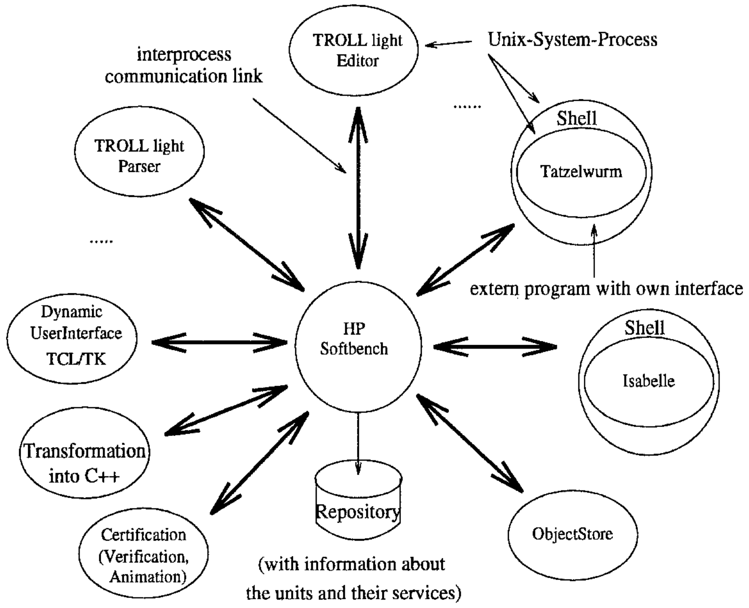
TROLL light Editor
Unix-System-Process
interprocess communication link
TROLL light Parser
......
Shell
Tatzelwurm
.....
extern program with own interface
Dynamic UserInterface TCL/TK
HP Softbench
Shell
Isabelle
Transformation into C++
Repository
Certification (Verification, Animation)
(with information about the units and their services)
ObjectStore

Fig. 2. Architecture of the TROLL *light* development environment

The integration of new modules or programs in the environment can be done in two ways. If the module or program is still under development the message handling can be inserted directly into the source code and the required object libraries have to be linked to the object files of the module. Otherwise, if the module already exists as an executable program with a special interface it has to be encapsulated with a shell which bridges the programs' interface to that of the communication manager.

It should be emphasized that the ability of dynamically including new units in the environment is a very important aspect of the system. This system architecture can be seen as a layer on top of the operating system, which provides a framework for integrating tools in the environment.

Units and their Tasks. The base version of the TROLL *light* development environment will include some units which make up the main parts of the environment. The first unit a user will work with is a special one which is responsible for the graphical user interface. In order to satisfy the requirements of the environment, it will dynamically adapt itself to new started units by requesting which interactive services it offers to the user and by integrating them in the current menu tree of services. Each time a user applies a service, the user interface unit will generate a message which is broadcasted to the appropriate unit which offers that service.

Another special unit is the object-oriented database management system ObjectStore [LLOW91]. It serves as a basis for all data created by any of the units and as a means for communication between other units, especially for interchanging data sets, e.g., parsed TROLL *light* specifications, parser errors, results of theorem provers, etc. The creation of TROLL *light* specifications can be done in a unit which includes an editor and a parser. A parsed TROLL *light* specification will automatically be stored in an ObjectStore dictionary for later use.

In order to prototype a TROLL *light* specification the animator unit will support the validation of a specification. The specified object community will be interactively executable, i.e., the user is able to create objects, execute events in them, observe their behavior, etc. Besides this, other units are responsible for verifying a TROLL *light* specification, namely the theorem provers. For the time being, the theorem provers *Tatzelwurm* [Käu89] and *Isabelle* [Pau90] will be integrated into the environment.

The transformation of TROLL *light* specifications into source code of a programming language is at the time being a future topic. It is planned to implement a tool for doing this task as automatically as possible.

3 Concepts of TROLL *light*

3.1 Overview

TROLL *light* is a language for describing structural and dynamic properties of objects. With respect to the description of structural properties TROLL *light* offers concepts of semantic data models.

- Attributes describe observable properties of objects. Simple attributes are data-valued or object-valued. Over and above that complex attributes are obtainable by the application of predefined sort constructors like *set* or *tuple*.
- TROLL *light* objects are organized into object hierarchies induced by sub-object relationships. Thereby a sub-object relationship must be understood as an exclusive "part-of" relationship.
- Normally attribute values are directly affected by event occurrences. Besides that derived attributes can be specified the state of which depends on other stored or derived information. In order to state derivation rules TROLL *light* incorporates an SQL-like query calculus.
- The query calculus of TROLL *light* also allows to specify static integrity constraints.

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

- Events on objects are described by a finite set of event generators. Each event generator may be accompanied by a list of parameter sorts. An event generator together with actual parameter values denotes an event.

– The effect of events on attributes is described by valuation rules.
– Events in different objects can be synchronized by interaction rules.
– Possible event sequences can be restricted to admissible ones by the means of CSP-like process descriptions.

All in all a template has the following structure:

```
TEMPLATE
  DATA TYPES     declaration of data types used in current template
  TEMPLATES      declaration of templates used in current template
  SUBOBJECTS     sub-object relationships
  ATTRIBUTES     observable properties
  EVENTS         event generators
  CONSTRAINTS    static integrity conditions
  VALUATION      effect of event occurrences on attributes
  DERIVATION     derivation rules for derived attributes
  INTERACTION    synchronization of events in different objects
  BEHAVIOR       description of admissible life cycles
END TEMPLATE
```

In the next section we show by a small example how information systems may be specified by writing templates.

3.2 Example

Let us assume that an information system is to be described which shall contain data about 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 the name only once in the life. An appropriate TROLL *light* specification would be:

```
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 authorlife1 =
                 ( storeSoldBooks -> authorlife1 |
                   changeName -> authorlife2 |
                   destroy -> POSTMORTEM );
               PROCESS authorlife2 =
                   ( storeSoldBooks -> authorlife2 |
                     destroy -> POSTMORTEM );
                 ( create -> authorlife1 );
END TEMPLATE;
```

Process descriptions as used in the BEHAVIOR section can be transformed to event-driven sequential machines (see Fig. 3). This machines will be called *o-machines* (*o* for object) in the sequel. The states of the o-machine denote behavior states of objects.

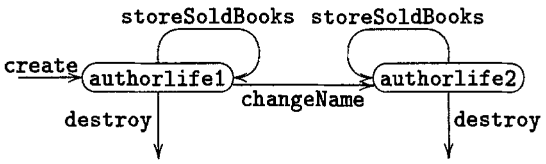

Fig. 3. Behavior of authors

We may assume that the information system will be populated by lots of authors so that these authors must be managed by a higher-level object. To this end usually classes are introduced in many object-oriented approaches. There is some confusion concerning the notion of class in object-oriented programming and object-oriented databases. A class is often used to cover two aspects, namely an *object factory* and an *object container* [Ban88]. In TROLL *light* object factories appear as templates while object containers are merely understood as composite objects, and composite objects have to be described by templates again.

```
TEMPLATE Authorclass
  DATA TYPES    String, Date, Nat;
  TEMPLATES     Author;
  SUBOBJECTS    Authors(No:nat):author;
  ATTRIBUTES    NumberOfAuthors:nat;
  EVENTS        BIRTH create;
                      addObject(No:nat, Name:string,
                                DateOfBirth:date);
                      removeObject(No:nat);
                DEATH destroy;
  VALUATION     [create]       NumberOfAuthors=0;
                [addObject]    NumberOfAuthors=NumberOfAuthors+1;
                [removeObject] NumberOfAuthors=NumberOfAuthors-1;
  INTERACTION   addObject(N,S,D) >> Authors(N).create(S,D);
                removeObject(N)  >> Authors(N).destroy;
END TEMPLATE;
```

Every template implies a corresponding object sort. We use names starting with an upper case letter to denote templates whereas the same name starting with the corresponding lower case letter is used to denote the corresponding object sort.

An object of sort `authorclass` will hold many author objects as private sub-objects (or components). In this example a natural number serves as a key for the identification of sub-objects. Please note that in contrast to many other data models, in TROLL *light* such keys need not be related to attributes of objects they identify. Adding an object to an author class means to create a new sub-object, and removing an object from an author class means to destroy the sub-object.

An object of sort `authorclass` may be sub-object of a higher object again, and so on. For example, a template `Libraryworld` may be specified containing an author index, a book index , and two libraries as sub-objects (see [CGH92]). The structure of an instance of sort `libraryworld` is set out in Fig. 4. The diagram is very similar to the object diagrams presented in [KS91].

Objects are represented by rectangles. The corresponding object sort is noted in the upper left-hand corner of a rectangle. Sub-object relationships are illustrated by embedding rectangles in bigger rectangles. Since the logical name of an object is always defined within a super-object, it appears outside the related rectangle. Dashed lines represent object relationships established by object-valued attributes. For example, libraries refer to an author index and a book index, documents refer to books, and books refer to authors.

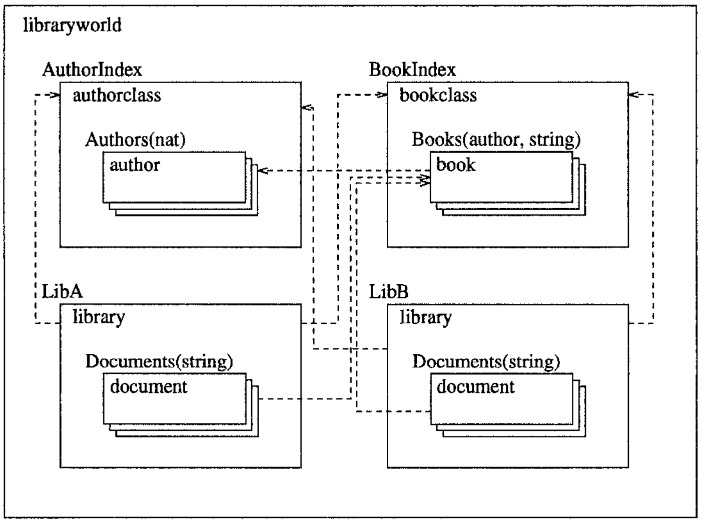

Fig. 4. Instance of sort `libraryworld`

4 Verifying Object Properties

One issue of our project is certification. This notion covers a wide range of different aspects, especially validation, consistency, and verification. In this section, we do not want to discuss all aspects in detail. A brief presentation of some certification aspects has already been given in Sect. 2.2.

Here, we want to deal with verification in more detail. Verification is needed to prove 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. Furthermore, a lot of specified properties would be redundant.

Therefore, it seems better to write "minimal" specifications which only include a small number of properties. From these properties further properties should be derivable. Derivable properties should be formulated outside the specification, but must be stored together with the specification. For these properties a calculus together with a deduction system is needed in order to express such properties and to reason about them. Finally, we also need tools to support the proving of specification properties.

At the moment we are working on a suitable calculus. Therefore, we are extending a calculus in the style of [FM91a, FM91b] by features for dealing with interacting objects (as presented in [SSC92]). The extension of our calculus together with a formal description is in progress. Here, we informally describe our calculus in a form suitable for reasoning about single objects. The main concepts of the calculus are terms, propositions, and formulae.

Terms may be built from variables and function symbols as usual. Furthermore, attribute and event symbols may be used to construct terms in the same way as with function symbols. Terms constructed by event symbols are terms of the event sort. Now, positional terms can be built: given a term t of some sort s (except the event sort) and a term e of the event sort, $[e]t$ is also a term of sort s. $[e]t$ denotes "the value of t after e". For example, $[\mathit{storeSoldBooks}(1992,\ 42)]\,\mathit{SoldBooks}(1992)$ is a positional term. Of course, all these term constructions can be applied inductively.

From terms we can now proceed to propositions. Propositions consist of a predicate symbol applied to an appropriate list of terms. The special predicate symbol Per indicating permission of events can be applied only to event terms. Propositions are state-dependent, i.e., they may be satisfied only in certain object states. Analogously to terms we can prefix propositions by positional operators, i.e., given a proposition P and an event term e, $[e]P$ is also a proposition. If $[e]P$ holds in some object state (for some ground substitution of the occurring variables), this means that P holds in the state reached by the occurrence of e. Thereby, it is possible to refer to future object states.

The next step is to introduce formulae. Formulae are the essential means of our calculus to describe object properties. Such formulae are Gentzen-formulae built from propositions, i.e.,

$$P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$$

is a formula if $P_1, \ldots, P_n, Q_1, \ldots, Q_m$ are propositions ($m \geq 1$). A formula $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ has to be understood such that, if in an object state all propositions $P_1, \ldots, P_n$ hold, then one of the propositions $Q_1, \ldots, Q_m$ holds in the same state, too. In contrast to propositions formulae are not state-dependent, i.e., they are true for every object state.

Now, we can describe objects by a set of formulae. In fact a transformation yields a set of formulae as result for a TROLL *light* template. Together with a collection of specification independent axioms and a sound derivation system we can prove object properties. The following formulae describe properties of `Author` objects. They are part of the result obtained by transforming the `Author` template given in Sect. 3.2:

$$\begin{aligned}
&\rightarrow ([create(N,D)]Name) = N \\
&\rightarrow ([create(N,D)]DateOfBirth) = D \\
&\rightarrow ([changeName(N)]Name) = N \\
&\rightarrow ([storeSoldBooks(Y,NR)]SoldBooks(Y)) = NR
\end{aligned}$$

These formulae describe the effect of events on attributes (specified in the `VALUATION` part). The transformation of the `BEHAVIOR` part is a little bit more complex, but can also be done automatically.

Our certification calculus provides a formal basis for proving object properties from their specifications. It seems to be obvious that the formulae of the calculus received by transforming the specification together with proven properties should be stored by the development system in order to simplify future proofs. Furthermore, the system has to keep a record of successful (and possibly also about unsuccessful) proofs in order to cause the specifier to re-prove properties if the specification has been modified. However, we are aware of the limits of verification. On the one hand there are well-known theoretical limits, e.g., decidability, and on the other hand there are restrictions in practice, e.g., the efficiency of contemporary theorem provers. Therefore, it is obvious that in practice only a small number of crucial properties will really be proved and that for all non-trivial proofs it will be necessary to carry out the proof interactively.

Currently we are working on automating the transformation of TROLL *light* templates, on details of the calculus, and on implementing essential parts of the calculus with the generic proof system *Isabelle* [Pau90].

5 A Semantics for TROLL *light* based on Maude

Because TROLL *light* is our main instrument for several purposes, like developing a certification calculus or testing the practicability and expressibility by means of case studies, it is necessary to give a formal semantics to it.

Up to now there are only few results on semantic fundamentals of the object-oriented paradigm. Investigations about theoretical foundations for object-oriented concepts and constructions are given for example in [ESS92, EGS92, ES91, SE91, SEC90]. Since we regard systems built up by a collection of TROLL *light* templates, in which several objects may exist and communicate concurrently, the work of Meseguer [Mes92b, Mes92c] which joins object-orientation and concurrency seems to be most promising to us to give a semantics to TROLL *light*. Maude's semantic framework is conditional rewriting logic, which is a logic to reason about change in a concurrent system. Since rewriting is naturally concurrent, this logic deals adequately with one intrinsic part of object-oriented systems, namely concurrency.

A direct result of the development of this theory is the logic programming language Maude [Mes92a]. Based on well-defined model-theory Maude offers language features which join essential aspects of object–orientation and concurrency. Since it was not developed for the conceptual design of information systems like TROLL *light*, we try to combine two advantages via a translation of TROLL *light* concepts into the Maude language:

1. Keep offering convenient TROLL *light* language features to model the UoD as adequately as possible
2. Make use of the well-defined semantics of Maude to give a semantics to TROLL *light*

In Maude there are three kinds of *modules*: **functional modules**, **system modules**, and **object-oriented modules**. The semantics of object-oriented modules is reducible to that of system modules; they only provide a notational convenience for object-oriented applications by supporting appropriate language features. In contrast to functional modules system modules provide the specification of non-functional behavior. We will emphasize our main ideas by applying them to the example from Sect. 3. Before we translate each TROLL *light* template to an object-oriented Maude module we have to make notions, like object, message, identity, etc., generally available.

A distributed state of a concurrent object-oriented system, called a system configuration in the sequel, is represented as a multiset of objects and messages. A system configuration is built up by an associative and commutative binary operator, working on two configurations putting them together to one, with 0 (system without objects and messages) as an identity element. Single objects and messages are configurations. All this is expressed in Maude by the following **system module**:

```
mod  CONFIGURATION is
  protecting  ATTRIBUTES ID.
  sorts  Configuration Object Msg .
  subsorts  Object Msg < Configuration.
  op  __: Configuration Configuration -> Configuration
          [assoc comm id: 0] .
  op  <_:_|_>: OId CId Attributes -> Object.
  ...
endm
```

Therefore, a term `<O:C|a_1:v_1,...,a_n:v_n>` represents an object of class `C`, with identifier `O`, and attributes `a_i:v_i`.

In TROLL *light* we specify templates not objects, whereas in Maude classes are described. The specified object sorts `Authorclass` and `Author` will correspond to class types in Maude as we will see in a while. The possible object identities `Authors(No:nat)` as established in the `SUBOBJECTS` part will be reflected by a module `ID` which makes object identifiers available. We yield the following **functional module**:

```
fmod  ID is
  ...
  sorts  Id OId CId AuthorclassId AuthorId IdSet .
  subsorts  OId  CId < Id .
  subsort  AuthorId AuthorclassId < OId .
  op  INIT: -> AuthorclassId .
  op  Authors: AuthorclassId Nat -> AuthorId .
endfm
```

In a given template collection always one template must be distinguished from the others, the so-called schema template. `INIT` is a special operation to generate an instance of the given schema template, in our example the `Authorclass` template.

O-machine states (e.g., `authorlife1` etc.) are essential for the description of object behavior. Maude does not provide a special feature to express constraints on objects behavior as we are used to it in TROLL *light*. In order to give a correct translation we make o-machine states explicit (by defining a sort `OMState` plus adequate generator functions) and use them later in rules in object-oriented Maude modules to express the dependencies between o-machine states and possible object state transitions. O-machine states are also reflected via a functional module:

```
fmod  OMSTATE is
  sorts  OMState AuthorState AuthorclassState .
  subsorts  AuthorState AuthorclassState < OMState .
  op  authorlife1 : -> AuthorState .
  op  authorlife2 : -> AuthorState .
  op  authorclasslife : ->  AuthorclassState .
  op  POSTMORTEM :  OMState .
endfm
```

The above translation, yielding the modules `CONFIGURATION`, `ID`, and `OMSTATE`, is necessary in general for the translation. In the following each TROLL *light* template will be translated to an **object-oriented Maude module** and each Maude module received in this way imports at least these three modules. We give the translation of the `Author` template as an example:

```
omod  AUTHOR is
  extending  CONFIGURATION .
  protecting  ID OMSTATE MAPNAT2NAT STRING NAT DATE .
  class  Author | Name: String, DateOfBirth: Date,
                  SoldBooks: MapNat2Nat, omstate: AuthorState .
  msg  create : AuthorId String Date -> Msg .
  msg  changeName : AuthorId String -> Msg .
  msg  storeSoldBooks : AuthorId Nat Nat -> Msg .
  msg  destroy : AuthorId -> Msg .
  var  AID : AuthorId .
  var  MNN : MapNat2Nat .
  ...
  rl  create(AID,S,D) =>
      <AID:Author|Name:S,DateOfBirth:D,omstate:authorlife1> .
  rl  changeName(AID,S) <AID:Author|Name:S',omstate:authorlife1>
      => <AID:Author|Name:S,omstate:authorlife2> .
  rl  storeSoldBooks(AID,Y,NR) <AID:Author|SoldBooks:MNN>
      => <AID:Author|SoldBooks:MNN U [Y,NR]> .
  rl  destroy(AID) <AID:Author|ATTS> => 0.
 endom
```

First of all we have to import the CONFIGURATION, ID, and OMSTATE modules to each object-oriented module which we receive as a translation from a TROLL *light* template. The essential notions of object, message, identifier, and o-machine state are necessary for each of them. Furthermore, for the author template we need the sort MapNat2Nat, specified somewhere else, in order to express the parameterized attribute SoldBooks(Year:nat):nat, which could also be viewed as an attribute SoldBooks:set(tuple(nat,nat)) with additional constraints expressing the functionality implicit in the former attribute expression. The sort MapNat2Nat comprises the required functional dependency. The sorts String, Date, and Nat which are needed for several attribute and message definitions are protected by module importation.

By means of the object-oriented Maude module AUTHOR a class Author is defined, and instances are described via four attributes: Name, DateOfBirth, SoldBooks, and omstate. Obviously we have to explicitly model the o-machine state as an attribute in order to adequately express dependencies between o-machine states and possible state transitions in Maude. In contrast to that these states are inherent in TROLL *light* specifications, but not visible.

TROLL *light* events are translated more or less one to one to messages in Maude. We only have to follow the rule that each message requires an object identifier, because we describe a class of objects, where single objects are distinguished via their identifiers. Therefore, all events set one more parameter: an Maude object identifier.

In contrast to TROLL *light* all used variables have to be declared in the vars section in Maude modules. We omit details about the variable declarations in the example.

The remaining TROLL *light* parts to translate, i.e., VALUATION, BEHAVIOR, and, for Authorclass, INTERACTION must all be modeled by rules in Maude modules. So the first rule connected with the BIRTH event not only says that after the occurrence of create(AID,S,D) a new object of type Author with identifier AID, and with attributes Name=S and DateOfBirth=D is generated (something which is expressed in the valuation part of the TROLL *light* template), but also that afterwards the o-machine state is equal to authorlife1 (what is stated in the behavior part of the corresponding TROLL *light* template). Combining the valuation formula for the changeName event of the TROLL *light* specification with the stated behavior, i.e., changeName can only occur if the o-machine state is equal to authorlife1, gives the second rule. Similar considerations lead to the third and fourth rule about the storeSoldBooks event. The last rule states that a system consisting of an author <AID:Author|ATTS> and an event which destroys this author object proceeds to a system without any object and event. Please note that 0 is the identity element of the __ operation in CONFIGURATION.

Similarly we can translate the Authorclass template. New aspects are introduced through the specified sub-objects part in conjunction with the interaction formulas.
We would receive

```
rl  addObject(AcId,N,S,D) <AcId:AuthorClass | NumberOfAuthors:N'>
    => <AcId:AuthorClass | NumberOfAuthors:N'+1>
```

from the valuation rule for addObject. But because of the interaction rule

```
addObject(N,S,D) >> Authors(N).create(S,D)
```

we adapt this rule to

```
rl  addObject(AcId,N,S,D) create(Authors(N),S,D)
    <AcId:AuthorClass | NumberOfAuthors:N'>
    => <AcId:AuthorClass | NumberOfAuthors:N'+1>
       <Authors(N):Author | Name:S, DateOfBirth:D,
                            omstate:authorlife1>.
```

We can summarize the following general translation schema:

object sorts $\rightarrow$	class types
sub-object symbols $\rightarrow$	constructor functions for identities
attributes $\rightarrow$	attributes
events $\rightarrow$	messages (one more parameter)
behavior $\rightarrow$	object machine states + rules
interaction $\rightarrow$	"rule adaption"

6 State of the Project

Parts of the TROLL *light* development environment have been implemented already. The TROLL *light* parser is completed by now and specifications can be stored by means of ObjectStore in a structured way. The theorem provers Tatzelwurm and Isabelle have been taken in consideration as a basis of the verification unit.

At the time being, some 15 students have done or are doing their thesis within the project. Current implementation efforts mainly concern the TROLL *light* animator, the user-interface, and the communication component. Up to the end of the project in March 1994 a lot of work remains to be done, but our experience gained so far allows us to anticipate that the most important parts of the TROLL *light* development environment shall be completed by then.

References

[Ban88] F. Bancilhon. Object-Oriented Database Systems. In *Proc. 7th ACM Symp. Principles of Database Systems*, pages 152–162, 1988.

[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.

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

[BMHL90] M. Bouzeghoub, E. Metais, F. Hazi, and L. Leborgne. A Design Tool for Object-Oriented Databases. In *Proc. Conf. Advanced Information Systems Engineering*, pages 365–392. Springer LNCS series, No. 436, 1990.

[Boo91] G. Booch. *Object-oriented Design with Application.* Benjamin/Cummings, Redwood City, 1991.

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

[CJO92] S. Clerici, R. Jimenez, and F. Orejas. Semantic Constructions in the Specification Language GLIDER. Workshop on Abstract data Types, 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.

[EGH$^+$92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, 9(2):157–204, 1992.

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

[EGS92] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their Specification. In M. Bidoit and C. Choppy, editors, *Proc. 8th Workshop on Abstract Data Types*, pages 40–66. LNCS 655, Springer, Berlin, 1992.

[EHH$^+$90] G. Engels, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, and H.-D. Ehrich. CADDY: Computer-Aided Design of Non-Standard Databases. In *Proc. 1st Int. Conf. on System Development Environments and Factories.* Pitman, London, 1990.

[EL92] G. Engels and P. Löhr-Richter. CADDY - A Highly Integrated Environment to Support Conceptual Database Design. In G. Forte, N.H. Madhavji, and H.A. Müller, editors, *Proc. 5th Int. Workshop on CASE, July 6.-10., 1992, Montreal (Canada)*, pages 19–22. IEEE Computer Society Press, 1992.

[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.

[ES91] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability*, pages 1–24. TU Braunschweig, Informatik Bericht 91-03, 1991.

[ESS92] H.-D. Ehrich, G. Saake, and A. Sernadas. Concepts of Object-Orientation. In *Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Ulm (Germany)*, pages 1–19. Springer IFB 303, 1992.

[FM91a] J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.

[FM91b] J. Fiadeiro and T. Maibaum. Towards Object Calculi. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability, Workshop IS-CORE '91, ESPRIT BRA WG 3023, London*, pages 129–178. Informatik-Bericht 91–03, Technische Universität Braunschweig, 1991.

[Gab91] P. Gabriel. The Object-Based Specification Language *Π*: Concepts, Syntax, and Semantics. In *Proc. 8th Workshop on Specification of Abstract Data Types.* Springer LNCS series, 1991.

[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.

[Hul89] R. Hull. Four Views of Complex Objects: A Sophisticate's Introduction. In S. Abiteboul, P.C. Fischer, and H.J. Schek, editors, *Nested Relations and Complex Objects in Databases*, Springer LNCS series, Vol. 361, pages 87–116, 1989.

[HV92] R. Herzig and N. Vlachantonis. Spezifikation einer Fertigungszelle in TROLL light. Interner Bericht, Technische Universität Braunschweig, 1992.

[JDT89] M. Jarke and DAIDA-Team. The DAIDA Demonstrator: Development Assistance for Interactive Database Applications. Technical Report MIP-8927, University of Passau, 1989.

[JMS90] M. Jarke, J. Mylopoulos, and J.W. Schmidt. Information Systems Development as Knowledge Engineering: A Review of the DAIDA Project. Technical Report MIP-9010, University of Passau, 1990.

[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.

[Käu89] T. Käufl. The Program Verifier Tatzelwurm. In H. Kersten, editor, *Sichere Systeme*, pages 219–236. Hüthig, 1989.

[KS91] 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*, pages 121–164. Elsevier (North-Holland), 1991.

[LCFW92] M. Löwe, F. Cornelius, J. Faulhaber, and R. Wessäly. Ein Fallbeispiel für KORSO: Das heterogene verteilte Managementsystem HDMS der Projektgruppe Medizin Informatik (PMI) am Deutschen Herzzentrum Berlin und an der Technischen Universität Berlin. Arbeitspapier, Technische Universität Berlin, 1992.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. 8th ACM Symp. Principles of Database Systems*, pages 360–368, 1989.

[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.

[Mes92c] J. Meseguer. Multiparadigm Logic Programming. Technical Report, SRI Computer Science Laboratory, May 1992.

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

[Pau90] L.C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Reh91] S. Rehm. STONE - Eine strukturierte und offene Umgebung. *Kolloquium Software-Entwicklungs-Systeme und -Werkzeuge, Technische Akademie, Esslingen*, September 1991.

[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. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 39–70, Amsterdam, 1991. North-Holland.

[SEC90] A. Sernadas, H.-D. Ehrich, and J.-F. Costa. From Processes to Objects. *The INESC Journal of Research and Development 1:1*, pages 7–27, 1990.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Prentice-Hall, Englewood Cliffs (NJ), 1988.

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

[SSC92] A. Sernadas, C. Sernadas, and J.F. Costa. Object Specification Logic. Internal report, INESC, University of Lisbon, 1992.

[UTS$^+$91] J. Uhl, D. Theobald, B. Schiefer, E. Sekerinski, S. Rehm, M. Ranft, and W. Zimmer. The Object Management System of STONE – SOS Release 3.2. Project Report No. 027, FZI Karlsruhe, 1991.

[VB82] G.M.A Verheijen and J. Van Bekkum. NIAM: An Information Analysis Method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Proc. "Information Systems Design Methodologies: A Comparative Review"*, pages 537–590, North-Holland, 1982.

[VMK+90] Y. Vassiliou, M. Marakakis, P. Katalagarianos, L. Chung, M. Mertikas, and J. Mylopoulos. IRIS – A Mapping Assistant for Generating Designs from Requirements. In *Proc. Conf. Advanced Information Systems Engineering*, pages 307–339. Springer LNCS series, No. 436, 1990.

[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.