M. Broy and S. Jähnichen, editors, KORSO: Methods, Languages, and Tools for the Construction of Correct Software, Final Report, pages 205–220. Springer, 1995. LNCS 1009.

TROLL *light* — The Language and Its Development Environment

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

Abstract. In our sub-project we are concerned with techniques for the development of reliable information systems on the basis of their formal specification. Our work focuses on the specification language TROLL *light* which allows to describe the part of the world to be modeled as a community of concurrently existing and communicating objects. Our specification language comes along with an integrated, open development environment. The task of this environment is to give support for the creation of correct information systems. Two important ingredients of the environment to be described here in more detail are the animator and the proof support system.

1 Introduction

The information system development process can be roughly split up into two important phases. The aim of the requirements engineering (or specification) phase is to obtain a first formal description of the system in mind. Since this formal description should still abstract from most implementation details it is usually called a conceptual schema. Based on the conceptual schema and by further consideration of nonfunctional requirements a working system is developed in the design engineering (or implementation) phase [Lou92]. We will concentrate in the following on the requirements engineering phase. This phase involves at least four important tasks [DDP93]: (1) Find the users' demands on the system in mind (elicitation), (2) describe a conceptual model of the system in mind (modeling), (3) test whether the conceptual model satisfies formally described quality criteria (analysis), (4) test whether the conceptual model meets the informal user requirements (validation).

As a first formal description of real-world entities we start with the object specification language TROLL *light* [CGH92, GCH93, HCG94], a dialect of OBLOG [SSE87] and TROLL [JSHS91]. TROLL *light* is especially appropriate for information system design because it embodies ideas from data type specification, semantic data models, and process theory. But an attractive language for information system design must be completed by specification tools. Therefore the object description language TROLL *light* comes along with a development environment offering special tools for verification and validation purposes in order to support the user during the design process according to tasks (3) and (4) from above:

- For us, the analysis task (3) consists in verifying properties, i.e., tackling the *formal correctness problem*. But in contrast to program verification where

a program, i.e., an implementation, is proved to satisfy its specification, we want to verify properties of objects at the level of specification. This is necessary in order to check whether the specification meets the intended requirements. Verifying such properties directly from the specification can help to avoid misdevelopments based on inadequate specifications.
In order to support this kind of verification the TROLL *light proof support system* solves verification tasks given by the user. It checks whether the desired properties are fulfilled by the specified TROLL *light* object descriptions.

- A specification of a formal conceptual schema must be validated against the informal system requirements in order to meet task (4). This is known as the *informal correctness problem.* One possible way to assure informal correctness of a conceptual schema consists in rapid prototyping which means to construct an experimental version of a system on a quick and cheap basis. Hence a prototype will often illustrate only some important aspects of a required behavior, thereby neglecting others like questions of performance or security. Nevertheless, by observing the behavior of a prototype the clients of a system can better judge the usefulness of a conceptual schema than by reading specification texts only. Typical tools supporting prototyping may be screen painters, report generators, program generators, animation systems, etc.
The TROLL *light animator* is designed to simulate the behavior of a specified object community. By this the informal view of the real-world fragment to be modeled is validated against the current specification.

In the following we present our system by going into some details concerning the concepts of TROLL *light*, giving comments on the idea of an open development system and presenting the two specific tools for animation and verification. We concentrate on these topics because case studies and application domains are discussed in [Ehr95]. Finally we give some concluding remarks.

2 The Language TROLL *light*

TROLL *light* is a language for describing static and dynamic properties of objects. This is achieved by offering language features to specify object structure as well as object behavior. The main advantage of following the object paradigm is the fact that all relevant information concerning one object can be found within one single unit and is not distributed over a variety of locations. As in TROLL object descriptions are called templates in TROLL *light*. Because of their pure descriptive nature templates may roughly 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 used a different notion. 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
```

Roughly speaking, the DATA TYPES and TEMPLATES sections are the interfaces to other templates, the SUBOBJECTS, ATTRIBUTES, and EVENTS sections constitute the template signature, and in the remaining sections axioms concerning static (CONSTRAINTS and DERIVATION) and dynamic (VALUATION, INTERACTION, and BEHAVIOR) properties are specified.

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. For the dynamic part we require that 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 section. Data types are assumed to be specified with a data type specification language. In the KorSo project we use SPECTRUM [BFG+93] as a reference language, but other proposals like ACT ONE, PLUSS, Extended ML, or OBJ3 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 section. Attributes denote observable properties of objects. They are specified in the `ATTRIBUTE` section of a template by $a\,[\,(s_1,\ldots,s_n)\,]:d$, 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). 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(1994)` is introduced. To underline the meaning of attribute parameters, optional parameter names like `Year` in `SoldBooks` may be added. 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. Therefore, a parametrized attribute $a(s_1,\ldots,s_n) : s$ can also be viewed as an attribute $a : \texttt{set}(\texttt{tuple}(s_1,\ldots,s_n,s))$, 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.

Events section. Incidents possibly appearing in an object's life are modeled by events. Event names are given in the `EVENT` section by $e[(s_1,\ldots,s_n)]$, where e denotes an event generator and $s_1,\ldots,s_n$ ($n \geq 0$) represent optional parameter sort expressions. Event parameters are used to define the effect 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 it to have at least one birth event.

Valuation section. The effect of events on attribute values is specified in the `VALUATION` section of a template definition by valuation rules, having the following form: [{*precond*}] [*event_descr*] *attr_term* = *term*. Such a rule says that immediately after an event occurrence belonging to the event term *event_term*, 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 formula *precond*, i.e., the valuation rule is applied only if the precondition is true. 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 term. An event term 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.

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. 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.

Behavior section. In the `BEHAVIOR` section the possible event sequences in an object's life are restricted by means of behavior patterns. Roughly speaking, behavior patterns determine a finite non-deterministic machine called o-machine (o for object) in the sequel. However, regarding an object together with its attributes we shall in general get an infinite number of object states and not a finite one. By behavior patterns event sequences, which an object must go through, can be specified as well as event dependent branchings. We have used the keyword `POSTMORTEM` within the behavior patterns to denote that the object vanishes. In Fig. 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 event sequences are unrestricted, i.e., it is only required for event sequences to start with a birth event and possibly end with a death event. In addition to the mentioned possibilities, behavior patterns may include preconditions for event occurrences referring to the objects' attributes and sub-objects in order to restrict the possible event sequences further. Thus, edges in the o-machine may not only be labeled by event names but should also mention the appropriate precondition; in our example these preconditions are all `true`.

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. In TROLL *light* an explicit class concept is not needed. Classes are viewed as composite objects instead, and therefore classes are described by templates as already mentioned before. However, 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.

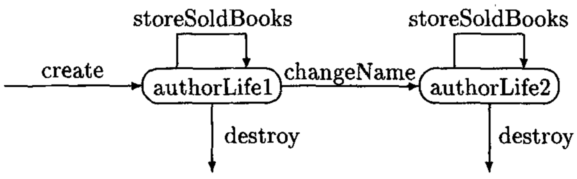


Fig. 1. Behavior of authors

```
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;
```

Templates section. 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`.

Subobjects section. An object of sort `authorClass` will hold finitely many author objects as private components or `SUBOBJECTS`. 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)` are 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 section. 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 [{*precond*}] [*src_obj*.]*src_event* >> [*dest_obj*.]*dest_event*. Such a rule states that whenever an event belonging to the source event *src_event* occurs in the source object denoted by *src_obj*, and the formula *precond* holds, then the event denoted by the destination event *dest_event* must occur simultaneously in the destination object denoted by *dest_obj*. If one of the object terms is missing, then the corresponding event term refers to the current object. The source object term is not allowed to have free 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 term. As already mentioned in the explanation of valuation rules, these free variables are instantiated by an event occurrence corresponding to the event term 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.

Constraints and derivation sections. 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 in the TROLL *light* query calculus, an offspring of an extended Entity-Relationship calculus [GH91]. For example, an author container of sort `authorClass` may only be populated by less than 10000 authors (as required above). Derived attributes can be specified by stating `DERIVATION` rules which, in fact, are closed terms of the TROLL *light* query calculus. 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. In the derivation rule `Authors` is regarded as 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 obtain object instances we must choose one template as the schema template, and one fixed object of the corresponding object sort as the schema object. All other objects in an object community will be sub-objects of this distinguished schema object. More details concerning TROLL *light* can be found in [CGH92, GCH93, HCG94]. In particular, [GCH93] discusses the semantics of the language as a state transition system in which a state transition is accompanied by a finite set of event occurrences which has to be closed against synchronization. Furthermore, a detailed comparision with the language Maude is given in [DG93].

3 The Development Environment

The TROLL *light* development environment is an open system which allows developers to integrate new tools and to adapt and extend it to their own requirements. Integration requirements concern the adaptation of tools, i.e., new or adapted tools should be able to work with documents (specifications, source code, etc.) created by other tools. This is a very important aspect because documents have certain structure and semantics and are stored in a structured way in a repository.

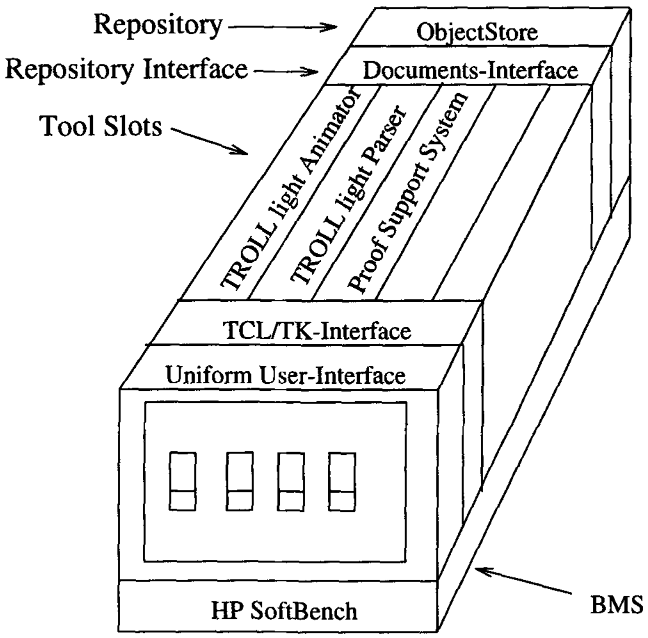


Fig. 2. Architecture of the TROLL *light* development environment

For preserving the openness of the environment tools have to be as independent as possible. For this the TROLL *light* development environment supports a loose coupling of tools. This is achieved on the one hand by a kind of tool communication which is based on interchanging messages and notifications (HP SoftBench [Cag90]) and on the other hand by an object-oriented documents repository (ObjectStore [LLOW91]). The latter supports the possibility of storing design documents in a structured way whereas the former provides mechanisms for tool integration.

The core of the HP SoftBench is the *Broadcast Message Server (BMS)* responsible for the indirect message exchange protocol. Tools can decide themselves whether they react to certain messages. During the starting phase they inform the BMS about the messages they are interested in. The BMS forwards messages only to interested tools. If, for instance, the TROLL *light* editor closes the creation of a specification it will inform its environment through a notification message about this event. The environment, i.e., other interested tools, may then decide on their own what to do next. The TROLL *light* parser could start parsing the new specification, and after this it could inform its environment about the results. Again other tools may take actions on their own. Since tools do not depend on each other it is quite easy to add, replace, or remove a tool from the environment.

The TROLL *light* development environment can be seen as an instantiation of the *ECMA* Reference Model for software development environments [Ear90] (see Fig. 2). It is built on the depicted integration frames BMS, ObjectStore, and TCL/TK user interface manager [Ous90]. A further discussion of the TROLL *light* development environment can be found in [VHG+93].

3.1 The Animation System

Animating templates. A template generally describes structural and dynamic aspects of a prototypical object. Structural properties are centered around the specification of possible attribute states, dynamic properties around the specification of possible event sequences. Looking at a template with sub-object slots the prototypical object described by this template is in fact an object community where events in different objects may be synchronized by interaction rules.

Speaking in more technical terms the model of a template is a state transition system in which a state transition is accompanied by a finite set of event occurrences which has to be closed against synchronization. TROLL *light* object descriptions abstract from the causality or initiative of event occurrences. Hence making a state transition system to move means to indicate certain event occurrences from the outside of the system. This is what we call *animation* of templates.

Animation of templates may help to assure that the specified behavior of objects or object communities matches the required behavior. Of course, animation of a conceptual model shows the same problems like testing an implementation. From observations that the observed behavior agrees with the requirements we would like to draw the conclusion that a conceptual schema (an implementation) is correct with respect to the requirements (the conceptual schema). This, however, cannot be done, because there still may be some traces in the animation (in an implementation) which have not been tested and which may show the opposite effect. Hence animation (testing) is only useful to falsify informal correctness.

Requirements for a tool supporting animation. A software system with the aim to support the animation of templates should meet the following requirements. It should support (1) the exploration of actual states of an object community, (2) the specification of event occurrences for initiating state transitions, and (3) the visualization of state changes.

To be more precise exploration of states should be assisted by means to illustrate the actual global structure of an object community, to show the attribute state (and optionally the behavior state) of a single object, to traverse sub-object relationships or object-valued attributes, and eventually to formulate ad hoc queries against object states.

With respect to the specification of event occurrences it must be possible to indicate event occurrences for possibly more than one object at a time. The system should help to find event sets being closed against synchronization. Hence, when a specified event occurrence provokes a second event occurrence in another object, this event occurrence should be added automatically to the current event set.

State changes provoked by a given closed event set, as there might be insertions and deletions of objects or attribute updates, should be made visible to the user, for example by appropriate messages in a specific window. When a desired state transition cannot be carried out, for instance because the resulting state violates some integrity constraints, these conditions should also be reported to the user.

Architecture of the animation system. The structure of the TROLL *light* animation system is depicted in Fig. 3. First of all the animation system consists of a *template dictionary* which is a persistent store for object descriptions. The template dictionary is not an exclusive part of the animation system but an integral part of the whole development environment, for instance shared by the parser and the proof support system. The second basic component of the animation system is the *object base* which is a persistent store to hold object states. In principal, the object base could also be a transient store but in the system used for implementation there was only a slight distance between transient and persistent structures so that we preferred persistent structures. Hence it is possible to stop a current animation session at one time and start it again later on.

Object descriptions contain terms and formulas of the TROLL *light* query calculus which are evaluated by the third component of the animation system, the *term evaluator*. The evaluation of terms and formulas generally depends on the current state of an object community so that the term evaluator must be able to access the object base. The *execution module* is the heart of the animation system. Its task is to compute for a given set of event occurrences a successor state or to report errors if such a state cannot be determined for different reasons. Finally the *user interface* establishes communication with the human operator.

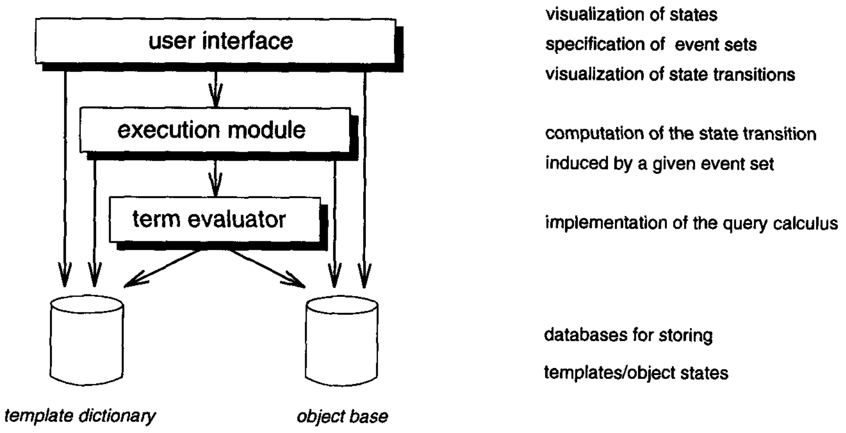

user interface
execution module
term evaluator
template dictionary
object base
visualization of states
specification of event sets
visualization of state transitions
computation of the state transition
induced by a given event set
implementation of the query calculus
databases for storing
templates/object states

Fig. 3. Architecture of the TROLL *light* animation system

Implementation aspects. A prototype version of the animation system covering basic requirements has been completed with the end of the KORSO project. Work was done on the basis of three diploma thesis.

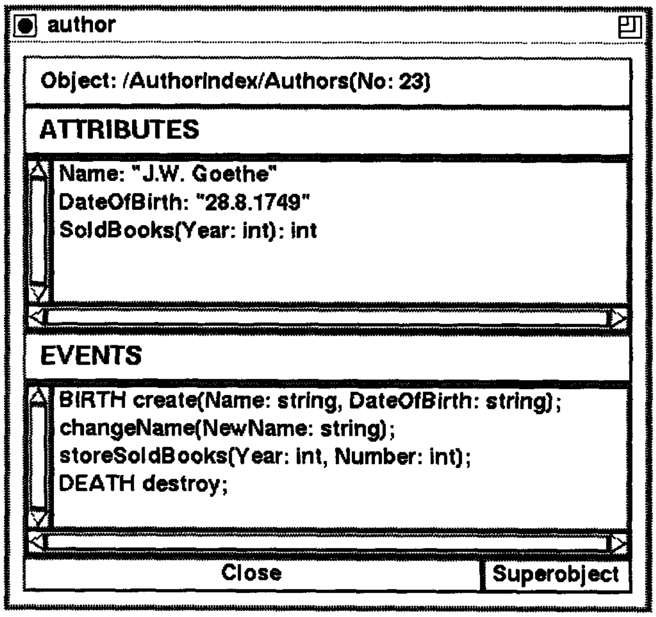
The template dictionary and the object base were implemented using the OODBMS ObjectStore [LLOW91] on the basis of C++. The object base is designed for storing values of any complexity, even a whole NF^2 or complex object model database [AFS89].

The term evaluator and the execution module were implemented in C++. Term evaluation generally results in a complex value which is stored using the same structures as found in the object base. It is noteworthy that terms that are to be evaluated are not limited concerning their complexity. For instance, it has been shown that the term evaluator was even capable of evaluating 1000 nested select-from-where queries.

Finally the user interface was realized by the help of TCL/TK [Ous90]. In the prototype version of the animation system object states are visualized in so called *object windows* (see Fig. 4). Object windows resemble the representation of pointers to current sub-objects, the visualization of current attribute values as well as a depiction of the list of possible event generators. Event occurrences can be specified by marking an event generator and optionally entering required parameter values. Specified event occurrences are collected in a separate window from which a set of event occurrences may be sent to the execution module.

3.2 The Proof Support System

Verification calculus. The development of the verification calculus was driven by pragmatic considerations. Since the simplicity of the proof system was of great importance for our purposes (i.e., verifying object properties), we decided


author

Object: /AuthorIndex/Authors(No: 23)

ATTRIBUTES

```
Name: "J.W. Goethe"
DateOfBirth: "28.8.1749"
SoldBooks(Year: int): int
```

EVENTS

```
BIRTH create(Name: string, DateOfBirth: string);
changeName(NewName: string);
storeSoldBooks(Year: int, Number: int);
DEATH destroy;
```

Fig. 4. Object window

to choose a simple structure of formulas. We only allow Gentzen-formulas with a conjunction of propositions as antecedent and a disjunction of propositions as conclusion, i.e., formulas do not have other formulas as subformulas. The decision for this kind of formulas was motivated by the fact that thereby we can easily adopt existing proof systems of various sequent calculi, e.g., [FM91]. Furthermore, we have found a generic theorem prover (ISABELLE [Pau90]) which allows for a realization of such sequents.

In order to give an impression of the verification calculus we present the main concepts followed by an example. The most important concept is the concept of formulas. As already mentioned we allow only a special form of clauses as formulas: $P_1, \ldots, P_n \rightarrow Q_1, \ldots, Q_m$ where P_i and Q_j are so-called propositions. Such a formula must be understood as follows: in each state (of an object community) in which all the propositions $P_1, \ldots, P_n$ are fulfilled there is at least one of the propositions $Q_1, \ldots, Q_m$ which is also fulfilled in that state.

Next, we have to introduce our notion of proposition. Propositions are mainly given by predicate expressions $p(t_1, \ldots, t_n)$ (with p a predicate symbol and t_i appropriate terms). Furthermore we allow negation (i.e., $\neg P$) and the use of positional operators (i.e., $[t_o.t_e]P$, where t_o is a term denoting an object and t_e a term describing an event for this object). A proposition $[t_o.t_e]P$ could be read as "if the event occurring in object t_o is t_e then P holds afterwards". We distinguish between two kinds of predicate symbols: rigid and non-rigid ones (i.e., state-independent and state-dependent predicate symbols, resp.). For instance, rigid predicate symbols are given by the data type specification used (e.g., $\leq$ for integers), whereas attributes of objects are modeled by non-rigid predicates. Furthermore we have non-rigid predicates for dealing with enabledness and occurrence of events, namely the predicates *enable* and *occur*. For local reasoning about a single object the predicate *enable* would be sufficient. However, taking synchronous interaction between objects into account we also need the possibility to express that a certain event really occurs in some state. The predicate *occur* enables us to cope with this requirement.

The following formulas express properties of `authorClass` and `author` objects as described above:

$$\begin{aligned}&NumberOfAuthors(AC, C) \rightarrow \\ &\qquad [AC.addObject(N, S, D)]NumberOfAuthors(AC, C+1) \\ &\rightarrow [A.changeName(N_1)]\neg\, enable(A.changeName(N_2))\end{aligned}$$

The first formula states that given a value C for the attribute `NumberOfAuthors` of an `authorClass` object AC this attribute value is increased after the occurrence of an `addObject` event in the object AC. The second formula describes that directly after the occurrence of a `changeName` event in an author object A a further occurrence of a `changeName` event in the same object is not allowed. Thereby, the set of enabled events can be restricted for the subsequent state.

Properties to be proved must be expressed as formulas as well. For instance, the formula

$$\rightarrow [A.changeName(N_1)]\neg\, occur(A.changeName(N_2))$$

can be proved from the set of formulas obtained by transforming the specification of `authorClass` and `author` objects into formulas of our calculus.

Limitations of the calculus. Of course, the question arises which kind of properties can be proved by means of this calculus. At the time being, the expressive power of the calculus and the power of the proof system is quite limited, although we only need skolemization for transforming arbitrary formulas of first-order predicate logic into formulas of our calculus. But we cannot express arbitrary temporal properties for objects because there are only positional operators which can be compared with the next operator of temporal logics. In consequence, we cannot reason about, e.g., liveness properties. We can only consider single state transitions or finite sequences of events. However, later extensions of our calculus are not excluded. Especially the introduction of additional temporal operator for the future like "always" and "sometime" seems to be advisable.

We have been aware of these limitations from the beginning. But our aim has been to develop a basic calculus which can be used as an experimental basis for modifications and extensions. In consequence, this calculus should only cover inevitable aspects, and we have taken care that the calculus goes together with our specification language TROLL *light*.

Verifying properties. After having introduced the verification calculus we can give a brief impression of how the proof support system is embedded in the development environment. In principle we have the following steps in mind which are also depicted in Fig. 5:

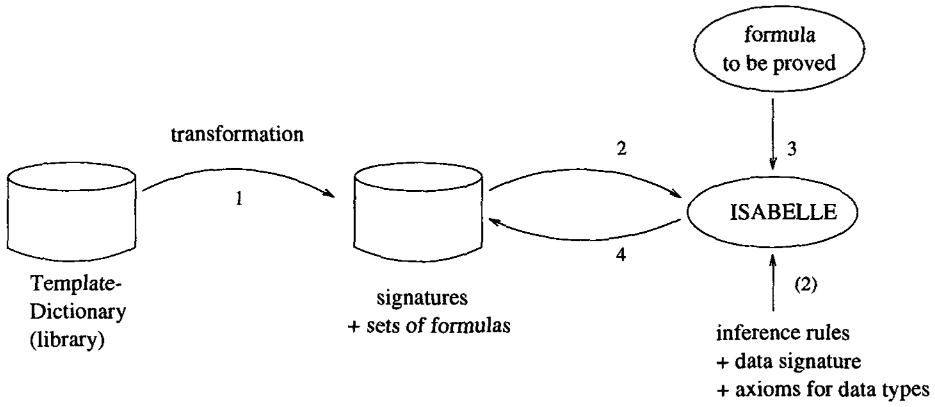

formula to be proved
transformation
1
2
3
4
ISABELLE
(2)
Template-Dictionary (library)
signatures + sets of formulas
inference rules + data signature + axioms for data types

Fig. 5. Working with the proof support system.

1. The (automatic) transformation of a TROLL *light* specification yields a corresponding template signature together with a set of formulas giving a logical description of the specified objects by means of the verification calculus. The results of such transformations are stored in a database.
2. When the transformation of a TROLL *light* specification has been completed, the generic theorem prover ISABELLE [Pau90] can be activated. In order to use the theorem prover for the verification calculus its syntax and inference rules must be loaded into the prover (in fact we can generate an instance of ISABELLE in which these parts are already included). Next, we have to include the signature and the formulas obtained from the TROLL *light* specification which shall be investigated.
3. Now we can formulate a property to be proven as a formula of the verification calculus. Given such a formula we can apply the prover. The proof can be controlled interactively, i.e., we can carry out the proof step by step by applying single inference rules or we can cause ISABELLE to automatically run parts of the proof by choosing suitable proof tactics.
4. If the proof of a property was successful the corresponding formula can be added to the database. Thereby, this formula can be reused later in proving further properties of the same objects.

Of course this is only a first, rough prototype of a proof support system for gaining experience in verifying object properties. This phase has just begun. First examples have already been carried out successfully, however, more and larger example are necessary to be able to judge the adequacy of the calculus and the conception of the proof support system. Afterwards this system can be revised in order to better meet pragmatic and practical requirements. At the time being a bothersome impediment in handling larger proofs is that nearly each proof step must be done by hand. By developing adequate proof tactics and adding them to the theorem prover we will perhaps be able to improve this situation.

4 Conclusion and Future Work

We have presented a sketch of our object description language TROLL *light* and its accompanying environment. The environment supports the development of correct information systems in (1) validating the informal view of the real-world fragment to be modeled against the current conceptual schema and (2) in checking whether desired properties are fulfilled by the specified object descriptions.

However, the system integration and the implemented tools are far from being perfect. For example, the animator does not support user-defined data types, ad-hoc queries, graphical visualization of sub-object relationships, and enhanced visualization of object behavior. The proof support system could be improved by allowing for instance temporal logic formulas to be checked against the specification, and implementation on the basis of ISABELLE would profit from including more proof tactics in order to speed up the inference mechanism. Up to now only the first steps towards integration of the different tools have been implemented and much more work could be done here.

Acknowledgements. Our research activities have been constructively criticized by our Braunschweig and KORSO colleagues. The critical remarks of the referees improved the paper. Most of the implementation work has been done by students. Therefore we would like to thank especially N. Abert, M. Alexander, E. Becker, J. Brinckmann, R. Conrad, H. Fischer, H. Herde, O. Hellmold, M. Meyer, N. Notbohm, C. Paasch, M. Rose, F. Ruloffs, F. Schulten, L. Schulz, J. Siemer, U. Trenkner.

References

[AFS89] S. Abiteboul, P.C. Fischer, and H.J. Schek, editors. *Nested Relations and Complex Objects in Databases*, Springer, Berlin, LNCS 361, 1989.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM — An Informal Introduction (Version 1.0). Technical Report TUM I9311-12, TU München, 1993.

[Cag90] M.R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett Packard Journal*, 41, 1990.

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

[DDP93] E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: An Agent Perspective. In O.M. Nierstrasz, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'93)*, pages 458–481. Springer, Berlin, LNCS 707, 1993.

[DG93] G. Denker and M. Gogolla. Translating TROLL *light* Concepts to Maude. In H. Ehrig and F. Orejas, editors, *Proc. 9th Workshop on Abstract Data Types - 4th Compass Workshop (ADT'92)*, pages 173–187. Springer, Berlin, LNCS 785, 1993.

[Ear90] A. Earl. A Reference Model for Computer Assisted Software Engineering Environment Frameworks. Technical report, Hewlett-Packard Laboratories, Bristol, England, 1990. Version 4.0 ECMA/TC33/TGRM/90/016.

[Ehr95] H.-D. Ehrich. KORSO Reference Languages: Concepts and Application Domains. In this volume.

[FM91] J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic and Computation*, 1(3):357–395, 1991.

[GCH93] 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'93)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[GH91] M. Gogolla and U. Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Trans. on Database Systems*, 16(3):369–416, 1991.

[HCG94] R. Herzig, S. Conrad, and M. Gogolla. Compositional Description of Object Communities with TROLL *light*. In C. Chrisment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94)*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994.

[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.

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

[Lou92] P. Loucopoulos. Conceptual Modeling. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, pages 1–26. John Wiley & Sons, New York, 1992.

[Ous90] J.K. Ousterhout. TK: An X11 Toolkit Based on the TCL Language. Technical Report, University of California at Berkeley, 1990.

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

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

[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, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, pages 463–482. Springer, Berlin, LNCS 685, 1993.