Proc. Colloq. on Object-Orientation in Databases and Software Engineering
16.-17.5.94, Montreal (Canada)
V.S. Alagar and R. Missaoui (eds.), World Science Publishing, pp. xxx–yyy (1995)

AN ANIMATOR FOR THE OBJECT SPECIFICATION LANGUAGE TROLL *light*[*]

RUDOLF HERZIG and MARTIN GOGOLLA
Arbeitsgruppe Datenbanksysteme, FB 3 Mathematik und Informatik, Universität Bremen
Postfach 330440, D-28334 Bremen, Germany

ABSTRACT

In our 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. One important ingredient of the environment to be described here in more detail is the TROLL *light* animator.

1. Introduction

KORSO is a joint project of fourteen university and one industrial partner, sponsored by the German Ministry of Research and Technology (BMFT). The project aims at the consolidation of existing and the investigation of new methods for the development of correct software [9,4]. Within this framework the focus of the KORSO group at the Technical University of Braunschweig is on the improvement of techniques for the development of reliable information systems. 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 [16].

In the following we will concentrate on the specification phase. This phase involves at least four important tasks (see Figure 1): (1) Find the users' demands on the system in mind (requirements detection), (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* [5,10,12], a dialect of OBLOG [18] and TROLL [14]. Other

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

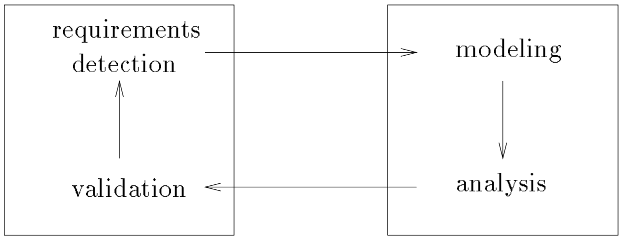

Figure 1: Requirements engineering tasks [6]

similar languages are MONDEL [2], CMSL [20], PI [8], and ALBERT [6]. TROLL *light* is especially appropriate for information system design because it embodies ideas from data type specification, semantic data models, and process theory.

In any case a language for information systems design must be completed by specification tools. Therefore the object description language TROLL *light* comes along with a specification environment, including tools like a parser, a proof support system for verifying certain properties of specifications and an animator allowing the prototypical execution of specifications for validation purposes [19,4].

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.

Next we present our system by going into some details concerning the concepts of TROLL *light* and presenting one specific tool for animation. 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 description 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   subobject slots
  ATTRIBUTES   attribute slots
  EVENTS       event generators
  DERIVATION   rules for derived attributes
  CONSTRAINTS  restricting conditions on object states
  VALUATION    effect of event occurrences on 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 a number of axioms concerning static (`DERIVATION` and `CONSTRAINTS`) and dynamic (`VALUATION`, `INTERACTION`, and `BEHAVIOR`) properties is specified.

2.1. Describing Basic Objects

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, Int;
  ATTRIBUTES   Name:string; DateOfBirth:string;
               SoldBooks(Year:int):int;
  EVENTS       BIRTH create(Name:string, DateOfBirth:string);
               changeName(NewName:string);
               storeSoldBooks(Year:int, Number:int);
               DEATH destroy;
  VALUATION    [create(N,D)] Name=N, DateOfBirth=D;
               [changeName(N)] Name=N;
               [storeSoldBooks(Y,NR)] SoldBooks(Y)=NR;
  BEHAVIOR     PROCESS Author =
                 ( create -> AuthorLife1 );
               PROCESS AuthorLife1 =
```

```
                    ( storeSoldBooks -> AuthorLife1 |
                      changeName -> AuthorLife2 |
                      destroy );
                  PROCESS AuthorLife2 =
                    ( storeSoldBooks -> AuthorLife2 |
                      destroy );
  END TEMPLATE;
```

2.1.1. Data types section

Data types are assumed to be specified with a data type specification language [7,21]. In the KorSo project we use SPECTRUM [3] 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 `Int` means that the data sort `int`, operations like `+ : int x int -> int`, and predicates like `<= : int x int` 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.

2.1.2. Attributes section

Attributes denote observable properties of objects. They are specified in the `ATTRIBUTE` section of a template by $a\,[\,(s_1,\ldots,s_n)\,]\!:\!s$, where a is an attribute name generator, s is a sort expression determining the range of an attribute, and $s_1,\ldots,s_n$ ($n \geq 0$) denote optional parameter sorts. The sort expression s may be built over both data sorts and object sorts by applying predefined sort constructors. We have decided to include the sort constructors `tuple`, `set`, `bag`, `list`, and `union` (of course, further constructors could be added as well). 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:int):int` 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 $a : \mathtt{set}(\mathtt{tuple}(s_1,\ldots,s_n,s))$, but clearly the formerly inherent functional dependency would have to be described by an explicit constraint. The same works for parametrized subobject constructors to be discussed later.

2.1.3. 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 sorts. 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.

2.1.4. 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:

$$[\{precond\}]\ \texttt{[}event\texttt{]}\ attr\_term\ \texttt{=}\ term$$

Such a rule says that immediately after an event occurrence belonging to *event*, 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 being representatives of event parameters. By a concrete event occurrence 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.

2.1.5. 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, which can be represented by a behavior diagram as shown in Figure 2 for the behavior of authors. However, regarding an object together with its attributes we shall in general get an infinite number of possible object states and not a finite one. By behavior patterns event sequences, which an

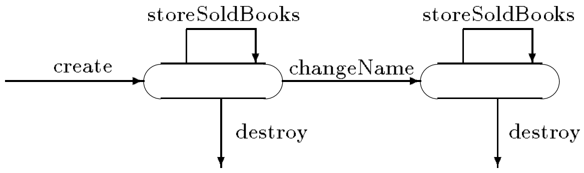

Figure 2: Behavior diagram of authors

object must go through, can be specified as well as event dependent branchings. In addition to the mentioned possibilities, behavior patterns may include preconditions for event occurrences referring to the objects' attributes and subobjects in order to restrict the possible event sequences further. Thus, edges in the behavior diagram may not only be labeled by event names but should also mention the appropriate precondition; in our example in Figure 2 these preconditions are all `true`. 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.

2.2. Describing Composite Objects

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 subobject relationships denoting (exclusive) part-of relationships. The following example gives the format of container objects for authors.

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

2.2.1. 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 having object identifiers as instances. Hence referring to `Author` means that the object sort `author`, and the attributes and the event generators of `Author` are visible in `AuthorContainer`.

2.2.2. Subobjects section

An object of sort `authorContainer` 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 subobject, 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 subobject slots as shown in the example. As with parametrized attributes, a possibly infinite number of logical subobject names like `Authors(42)` are defined by the subobject name declaration for `Authors`, but not the author objects themselves. The declaration only states that in context of an object of sort `authorContainer`, author objects are identified by an integer number. In semantic data models the parameter `No` would be called a key. But unlike in semantic data models the parameters need not be related to any attributes of the objects they identify. Each logical subobject 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 subobjects exists.

2.2.3. Interaction section

Interaction rules are specified in the `INTERACTION` part of the template definition. During the lifetime of an author container there will be events concerning the insertion or deletion of authors. In `AuthorContainer`, 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 `addAuthor(N,S,D) >> Authors(N).create(S,D)`, where `N` denotes an integer number, `S` a string, and `D` a string. The general event calling scheme is expressed as

$$[\{precond\}]\ [src\_obj\,.]src\_event \texttt{ >> } [dest\_obj\,.]dest\_event$$

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 subobject. In this case it is even a requisite that the destination object does not exist already.

2.2.4. Derivation and constraints sections

In addition to the mentioned language features TROLL *light* offers some further, more sophisticated concepts which will be mentioned only briefly: Derived attributes can be specified by stating `DERIVATION` rules which, in fact, are closed terms of the TROLL *light* query calculus, being an offspring of an extended Entity-Relationship calculus [11,13]. In the example we declared the attribute `NumberOfAuthors` of the template `AuthorContainer` 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(int,author))`. So `NumberOfAuthors` does not need to be set by explicit valuation rules. 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. For example, an object of sort `authorContainer` may only be populated by less than 10000 authors (as required above).

2.3. Instantiation of Object Communities

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, for example:

```
TEMPLATE LibraryWorld
  TEMPLATES
    AuthorContainer, ...;
  SUBOBJECTS
    AuthorIndex:authorContainer; ...;
  EVENTS
    BIRTH create;
    ...
END 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 subobjects of this distinguished schema object. More details concerning TROLL *light* can be found in [5,10,12].

3. The TROLL *light* Animator

3.1. 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 subobject 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.

3.2. 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 subobject 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,

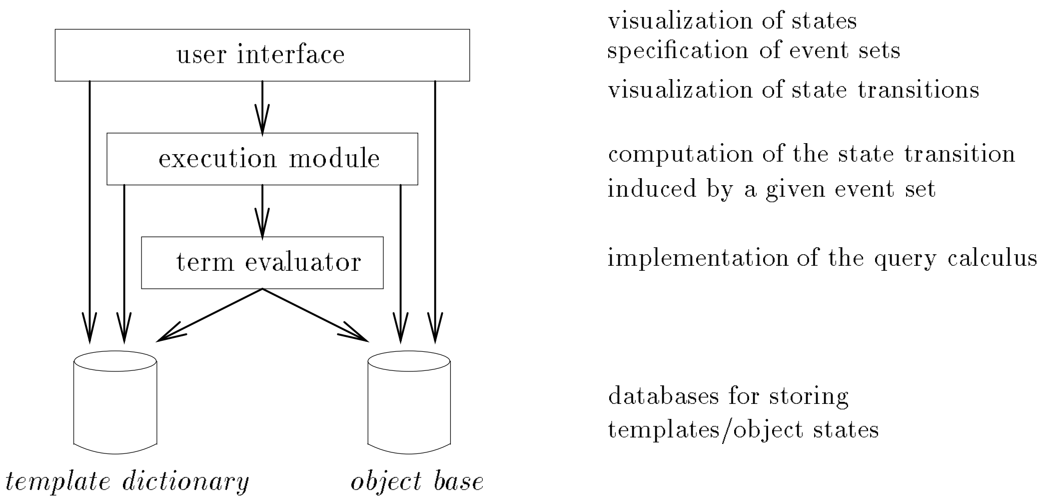

Figure 3: Architecture of the TROLL *light* animation system

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.

3.3. Architecture of the Animation System

The structure of the TROLL *light* animation system is depicted in Figure 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.

3.4. Implementation Apects

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 object-oriented database system ObjectStore [15] 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 [1].

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 [17]. In the prototype version of the animation system object states are visualized in so called *object windows* (see Figure 4 and 5). Object windows resemble the representation of pointers to current subobjects (marking the subobject slot Authors in Figure 5 yields the select box shown of Figure 6, selecting number 23 in this box pops up the object window already shown in Figure 4), 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 (see Figure 7) from which a set of event occurrences may be sent to the execution module.

4. Conclusion and Future Work

We have presented a sketch of our object description language TROLL *light* and an important part of its accompanying specification environment. The environment supports the development of correct information systems in validating the informal view of the real-world fragment to be modeled against the current conceptual schema and 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 subobject relationships, and enhanced visualization of object behavior. Up to now only the first steps towards integration of the different tools have been implemented and much more work could be done here.

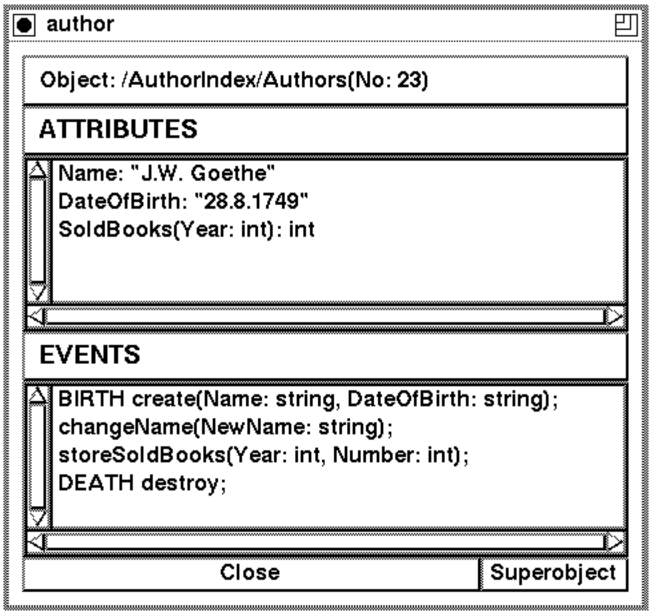

Figure 4: Object window representing an author

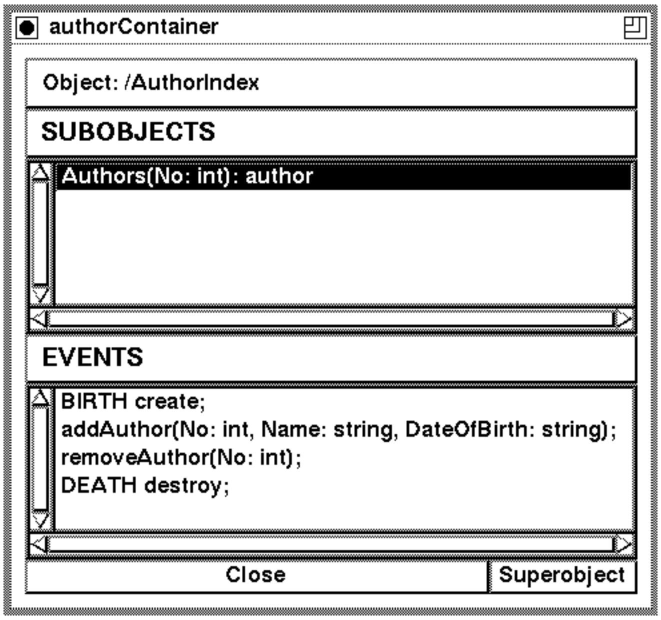

Figure 5: Object window representing an author container

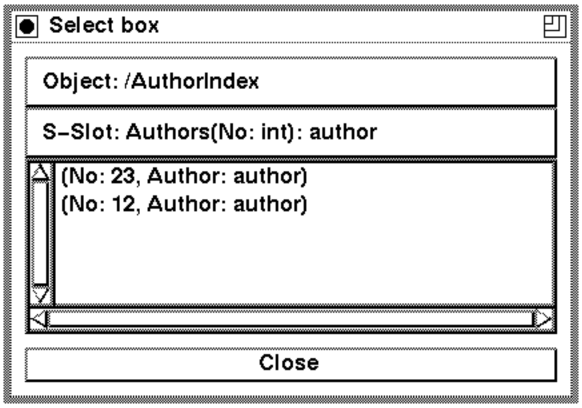

Figure 6: Presentation of a parametrized subobject slot

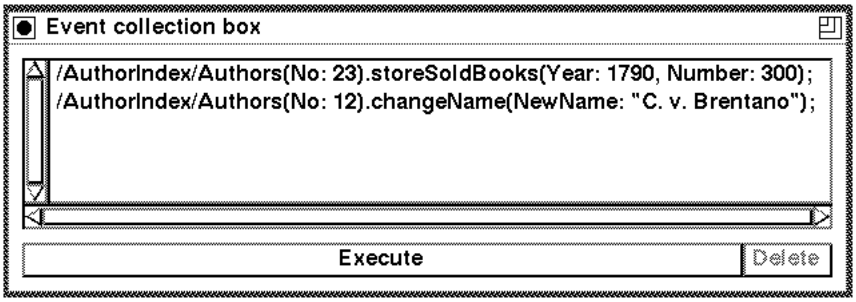

Figure 7: Collection of events to be executed

5. References

1. S. Abiteboul, P.C. Fischer, and H.J. Schek (eds.), *Nested Relations and Complex Objects in Databases*, Springer, Berlin, LNCS 361, 1989.
2. G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams, *Mondel: An Object-Oriented Specification Language*, Publication 748, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1990.
3. M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen, *The Requirement and Design Specification Language SPECTRUM — An Informal Introduction (2 parts)*, Tech. Report TUM–I9311/9312, Technische Universität München, 1993.
4. M. Broy and S. Jähnichen (eds.), *KORSO: Correct Software by Formal Methods*, 1994.
5. S. Conrad, M. Gogolla, and R. Herzig, *TROLL light: A Core Language for Specifying Objects*, Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.
6. E. Dubois, P. Du Bois, and M. Petit, *O-O Requirements Analysis: an Agent Perspective*, ECOOP'93 — Object-Oriented Programming (O.M. Nierstrasz, ed.), Springer, Berlin, LNCS 707, 1993, pp. 458–481.
7. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer, Berlin, 1985.
8. P. Gabriel, *The Object-Based Specification Language* Π*: Concepts, Syntax, and Semantics*, Recent Trends in Data Type Specification — Proc. 8th Workshop on Specification of Abstract Data Types (ADT'91) (M. Bidoit and C. Choppy, eds.), Springer, Berlin, LNCS 655, 1993, pp. 254–270.
9. M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and H.-D. Ehrich, *TROLL light: The Language and Its Development Environment*, In Broy and Jähnichen [4].
10. M. Gogolla, S. Conrad, and R. Herzig, *Sketching Concepts and Computational Model of TROLL light*, Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO) (A. Miola, ed.), Springer, Berlin, LNCS 722, 1993, pp. 17–32.
11. M. Gogolla and U. Hohenstein, *Towards a Semantic View of an Extended Entity-Relationship Model*, ACM Trans. on Database Systems **16** (1991), no. 3, 369–416.
12. R. Herzig, S. Conrad, and M. Gogolla, *Compositional Description of Object Communities with TROLL light*, Proc. Basque Int. Workshop on Information Technology (BIWIT): Information Systems Design and Hypermedia (C. Chrisment, ed.), Cépaduès-Éditions, Toulouse, 1994, pp. 183–194.
13. R. Herzig and M. Gogolla, *A SQL-like Query Calculus for Object-Oriented Database Systems*, Proc. Int. Symp. on Object-Oriented Methodologies and Systems (ISOOMS), Palermo (Italy) (E. Bertino and S. Urban, eds.), Springer, Berlin, LNCS 858, 1994, pp. 20–39.

14. 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.
15. C. Lamb, G. Landis, J. Orenstein, and D. Weinreib, *The ObjectStore Database System*, Communications of the ACM **34** (1991), no. 10, 50–63.
16. P. Loucopoulos, *Conceptual Modeling*, Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development (P. Loucopoulos and R. Zicari, eds.), John Wiley & Sons, New York, 1992, pp. 1–26.
17. J.K. Ousterhout, *Tcl and the Tk Toolkit*, Computer Science Division, University of California at Berkeley, 1993, preprint, to be published by Addison-Wesley.
18. A. Sernadas, C. Sernadas, and H.-D. Ehrich, *Object-Oriented Specification of Databases: An Algebraic Approach*, Proc. 13th Int. Conf. on Very Large Data Bases (VLDB) (P.M. Stocker and W. Kent, eds.), Morgan-Kaufmann, Los Altos (CA), 1987, pp. 107–116.
19. N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich, *Towards Reliable Information Systems: The KORSO Approach*, Advanced Information Systems Engineering, Proc. 5th CAiSE'93 (C. Rolland, F. Bodart, and C. Cauvet, eds.), Springer, Berlin, LNCS 685, 1993, pp. 463–482.
20. R.J. Wieringa, *Equational Specification of Dynamic Objects*, Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG2.6 Working Conference, Windermere (UK) 1990 (R.A. Meersman, W. Kent, and S. Khosla, eds.), North-Holland, Amsterdam, 1991, pp. 415–438.
21. M. Wirsing, *Algebraic Specification*, Handbook of Theoretical Computer Science, Vol. B (J. Van Leeuwen, ed.), North-Holland, Amsterdam, 1990, pp. 677–788.