Proc. 9th Int. Symposium on Computer and Information Science (ISCIS'94)
S. Kuru, M.U. Caglayan, E. Gelenbe, H.L. Akin, C. Ersoy (Eds.)
Bogazici University Printhouse, Istanbul, pp. 423-430 (1994)

From Object Specification to Enhanced Logic Programming

Martin Gogolla, Sedat Güler, Rudolf Herzig*
TU Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D-38023 Braunschweig, Germany
e-mail: {gogolla|gueler|herzig}@idb.cs.tu-bs.de

Abstract

We combine the area of object-oriented specification and object-oriented logic programming by indicating the translation of a specification language into an enhanced logic programming language. The translation shows that in comparison to the programming language the specification language offers richer concepts. The background of our work is the development of rapid prototyping systems for object-oriented specification languages.

Key words: Software Engineering, Programming Languages, Object-Orientation, Formal Specification, Logic Programming, Rapid Prototyping.

1 Introduction

Object-orientation seems to be a promising paradigm for the development of large software systems. Up to now, object-orientation has been intensively used mainly in the programming world. But recently, object-oriented ideas have been successfully applied in the realm of database systems. An even newer area where ideas concerning objects are applied nowadays is the specification and requirement phases in the software life cycle.

In this paper, we combine the area of object-oriented specification and object-oriented programming by indicating the translation of a specification language into an enhanced logic programming language. The translation shows that in comparison to the programming language our specification language offers richer concepts. The aim of our work is to bring the two fields object-oriented specification and object-oriented programming closer together by developing rapid prototyping systems for object-oriented specification languages.

The specification language we employ is TROLL *light* [CGH92, GCH93, HCG94]. The language has an accompanying development environment [VHG+93], and the connection between TROLL *light* and Entity-Relationship modelling [GHC+94] and TROLL *light*

*Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS).

and specification animation [HG94] seems to be well-understood. The programming language which we use is L&O [McC92]. We have chosen L&O for a number of reason: It is available because it is an extension of standard Prolog, it has powerful features, and there is an excellent textbook for the language [McC92]. Other choices for logic programming languages enhanced by object-oriented features are Polka [Dav88], Hoops [Con88], Life [AK91] or Oz [SHW93]. Applications of object-oriented concepts in the context of deductive databases are studied in the approaches for IQL [AK89], OIL [Zan90], F-Logic [KL89] and LOCO [STVV91].

2 A Brief Look at 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. Object descriptions are called templates in TROLL *light*. Templates show the following structure.

`TEMPLATE` name of the template

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

`END TEMPLATE`

To give an example for templates let us assume that we have to describe authors. For every author the name, the date of birth, and the number of books sold by year have to be stored. 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; 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(S,D)] Name=S, DateOfBirth=D;
               [changeName(S)] Name=S;
               [storeSoldBooks(Y,N)] SoldBooks(Y)=N;
  BEHAVIOR     PROCESS authorlife1 =
                 ( storeSoldBooks -> authorlife1 |
                   changeName -> authorlife2 |
                   destroy -> POSTMORTEM );
```

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

Data types section. Data types are assumed to be specified with a data type specification language. 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. 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. 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 take defined values such that only these attributes have to be stored.

Events section. Incidents possibly appearing in an object's life are modeled by events. Event names are given in the `EVENT` section. 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. Such a rule says that immediately after the event occurrence the attribute has the value determined by the right-hand side term. The applicability of such a rule may be restricted by a formula precondition, i.e., the valuation rule is applied only if the precondition is true. The precondition as well as left- and right-hand side term are evaluated in the state before the event occurrence. 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. A valuation is a proposition stating that after the occurrence of a certain event some attributes will 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

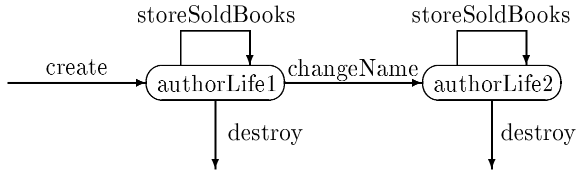

Figure 1: Behavior of authors

determine a finite non-deterministic machine called o-machine (o for object) in the sequel. However, regarding an object together with its attributes we will 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.

The following example gives the format of container objects for authors.

```
TEMPLATE AuthorClass
  DATA TYPES   String, Date, Nat;
  TEMPLATES    Author;
  SUBOBJECTS   Authors(No:nat):author;
  ATTRIBUTES   NumberOfAuthors:nat;
  EVENTS       BIRTH createClass;
                     addObject(No:nat,Name:string,
                               DateOfBirth:date);
                     removeObject(No:nat);
               DEATH destroyClass;
  VALUATION    [createClass] 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
```

Templates section. Within the `TEMPLATES` section, other 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. Such a rule states that whenever a left-hand side event occurs, then the right-hand side event must occur simultaneously. As already mentioned in the explanation of valuation rules, free variables are instantiated by an event occurrence corresponding to the event term in such a way that the preconditions and 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` and derived attributes can be specified by stating `DERIVATION` 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].

3 From TROLL *light* Specifications to L&O Programs

Here, we use elements of the programming language L&O. Due to space limitations we cannot give an overview on the language. We can only refer to the respective textbook [McC92] instead. However, we hope that the meaning of L&O features we use are clear from the context. We indicate how TROLL *light* object descriptions can be implemented in L&O. But we refrain from giving a formal translation. Instead we explain the basic idea by means of the above introduced TROLL *light* example. The principle structure of the TROLL *light* specification is also reflected by the L&O program. Thus, the two TROLL *light* example templates are implemented by two L&O objects. Let us first give the translation of the `author` template.

```
author:{
```

```
    name:=bottom.
    dateOfBirth:=bottom.
    soldBooks:=bottom.
    oMachineState:=0.
  create(Name,Date) :-
    oMachineState=0, name:=Name, dateOfBirth:=Date,
    soldBooks:=[], oMachineState:=1.
  changeName(NewName) :-
    oMachineState=1, name:=NewName, oMachineState:=2.
  storeSoldBooks(Year,Number) :-
    (oMachineState=1;oMachineState=2),
    soldBooks:=[(Year,Number)|soldBooks].
  destroy :-
    (oMachineState=1;oMachineState=2), name:=bottom,
    dateOfBirth:=bottom, soldBooks:=bottom, oMachineState:=0.
        }.
```

As indicated above, attributes in TROLL *light* correspond to so-called dynamic variables in L&O and events to predicate definitions. The second TROLL *light* template `AuthorClass` is implemented in an analogous way. For both templates, we have to introduce a special dynamic variable `oMachineState` which represents the object machine state.

```
authorClass:{
    numberOfAuthors:=bottom.
    members:=bottom.
    oMachineState:=0.
  createClass :-
    oMachineState=0, numberOfAuthors:=0, members:=[],
    oMachineState:=1.
  addObject(Author,Name,Date):-
    oMachineState=1, Author:create(Name,Date),
    numberOfAuthors:=numberOfAuthors+1,
    members:=[Author|members].
  removeObject(Author) :-
    oMachineState=1, Author:destroy,
    members:=delete(Author,members),
    numberOfAuthors:=numberOfAuthors-1.
  destroyClass :-
    oMachineState=1, numberOfAuthors:=bottom,
    members:=bottom, oMachineState:=0.
             }.
```

In the above L&O program, we add one special feature which is not present in the TROLL *light* descriptions. This is the dynamic variable `members` which represents the current members of `authorClass`. In our translation, we have benefited mainly from the following L&O features: L&O object definitions (e.g., the L&O object `author`), dynamic variables (e.g., the L&O variable `dateOfBirth`), and calls to predicates in other objects (e.g., the call `Author:create(Name,Date)` in the right hand side of `addObject` in the L&O object `authorClass`).

In order to summarize, we compare the different sections a TROLL *light* template offers with the corresponding features in L&O.

- In TROLL *light* data types are assumed to be specified in an equational data specification language. In principle, these specifications can be implemented in L&O by interpreting the equations as directed rewrite rules.
- The templates section in TROLL *light* corresponds to the feature of L&O to define objects.
- The subobjects section in TROLL *light* has no direct counterpart in L&O because within L&O objects new objects cannot be introduced. Therefore we have to model the name space objects are allowed to have within TROLL *light* directly by L&O instantiations.
- The TROLL *light* attributes can be directly modelled in L&O by the use of dynamic variables like `soldBooks`.
- TROLL *light* event declarations correspond to the introduction of predicates in L&O objects like `changeName(NewName)`.
- Constraints in TROLL *light* templates restrict the possible values the attributes may take. This can be expressed in L&O by introducing additional literals on the right hand side of rules checking the validity of the respective constraint.
- Valuations in TROLL *light* are implemented in L&O by assignments to dynamic variables like `numberOfAuthors:=numberOfAuthors+1`. These assignments occur in the defining clauses of the respective events.
- The derived attributes in TROLL *light* can also be implemented in L&O by dynamic variables employing the rich structure L&O offers for the definition of functions and predicates.
- The interaction rules in TROLL *light* synchronizing events in different objects are realized in L&O with predicate calls to other objects, like `Author:create(Name,Date)` in predicate `addObject` of `authorClass`.
- The TROLL *light* behaviour part must be implemented in L&O by the introduction of an additional variable `oMachineState` and by respective checks like `oMachineState=1` and assignments like `OMachineState:=2`.

4 Conclusion

We have sketched the specification language TROLL *light* and have roughly indicated how this language could be implemented in the enhanced logic programming language L&O. Not all features of TROLL *light* could be directly implemented but the very basic idea of TROLL *light* objects could be expressed in L&O. The background of our work is the development of rapid prototyping systems for object-oriented specification languages. With our approach, we hope to have made a contribution for bridging the gap between object-oriented specification and object-oriented programming.

References

[AK89] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *ACM SIGMOD Conf. on Management of Data*, pages 159–173, 1989.

[AK91] H. Ait-Kaci. An Overview of Life. In J.W. Schmidt and A.A. Stagny, editor, *Proc. 1st Int. East-West Database Workshop, Next Generation Information System Technology*, pages 42–58. Springer, Berlin, LNCS 504, 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.

[Con88] J.S. Conery. Hoops-user's Manual. Technical Report CIS-TR-88-12, University of Oregon, Department of Computer and Information Science, 1988.

[Dav88] A. Davison. Polka: A Parlog Object-Oriented Language. Internal Report, Imperial College, London, 1988.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*. Springer, Berlin, LNCS, 1993.

[GHC+94] M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the ER Approach in an OO Environment. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. 12th Int. Conf. on the Entity-Relationship Approach (ER'93)*, pages 376–389. Springer, Berlin, LNCS 823, 1994.

[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): Information Systems Design and Hypermedia*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994.

[HG94] R. Herzig and M. Gogolla. An Animator for the Object Specification Language TROLL *light*. In V.S. Alagar and R. Missaoui, editor, *Proc. Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*. Universite du Quebec a Montreal, 1994.

[KL89] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme. In *ACM SIGMOD Conf. on Management of Data*, pages 134–146, 1989.

[McC92] F.G. McCabe. *Logic and Objects*. Prentice Hall, Hertfordshire, 1992.

[SHW93] G. Smolka, M. Henz, and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. Technical Report RR-93-16, DFKI, Kaiserslautern, 1993.

[STVV91] F. Staes, L. Tarantino, B. Verdonk, and D. Vermeir. Supporting User Interaction with OODB's: A Declarative Approach. In D. Karagiannis, editor, *Proc. Int. Conf. on Database and Expert System Applications (DEXA'91)*, pages 210–215. Springer, 1991.

[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.

[Zan90] C. Zaniolo. Object Identity and Inheritance in Deductive Databases — an Evolutionary Approach. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD'89)*, pages 7–24. North–Holland, 1990.