Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)
M. Nivat, C. Rattray, T. Rus, G. Scollo (Eds.), Springer, London, Workshops in Computing, pp. 361-368 (1993)

An Object-Oriented Design for the ACT ONE Environment*

Martin Gogolla
Abt. Datenbanken, TU Braunschweig
D-38023 Braunschweig, Germany

Ingo Claßen
Fachbereich 20, TU Berlin
D-10587 Berlin, Germany

Abstract

The overall aim of this paper is to stabilize and strengthen the algebraic specification method to software engineering and development. We do not introduce new theoretical results, but define a conceptual model, i.e., an information system schema, for the well-established algebraic specification language ACT ONE and its accompanying specification environment.

This paper gives a formal description of suitable database support for an environment supporting the interactive development of ACT ONE specifications. The object-oriented data model of TROLL *light*, a language developed recently within the KorSo project, is used to present the design of such an environment and its dynamic behavior. However, the concepts used are general enough to support other specification and even programming languages as well. Therefore, we feel the design of a conceptual schema for ACT ONE is mainly a case study in employing an object-oriented data model for database support of specification or programming languages. It is therefore a proposal for the consolidation of environments for algebraic specification languages.

1 Introduction

Existing specification systems like ASSPEGIQUE [1], RAP [10], ACT [9, 5], OBJ3 [8] or OBSCURE [13] provide mechanisms to store and retrieve specifications and to operate on them. But in general they employ no systematic approach to information administration. Usually they rely on storage facilities of the underlying programming language and file system. A solution to the information handling problem is the use of information systems or more specific databases. They are already accepted as being important components of software development systems, and, since specification systems can be regarded as parts of general software development systems, the same arguments apply to them.

The use of databases as important parts of specification environments is especially motivated by the following facts: First, the development of algebraic specifications describing software systems of practical relevance usually results in large sets of related specification units. These units arise from the decomposition of complex specifications into smaller pieces by means of the structuring mechanisms provided by specification languages. Additionally algebraic methods and especially specification languages give rise to a bulk of information like proofs, formal transformation steps, formal relations like signature morphisms, etc., which have to be stored to be accessible by various tools. Second, the

*Work reported here has been partially supported by the CEC under Grant No. 6112 (COMPASS) and BMFT under Grant No. 01 IS 203 D (KorSo).

specification task has (or should have) an interactive nature that is characterized by several stages of incompleteness, i.e., stages where certain information is not available. Such stages change during the development process.

These observations lead to the following requirements on the used database technology: The database must be capable to deal with complex (structured) information in a coherent way. This requirement suggests to use object-oriented rather than relational technology. The database must support different degrees of incompleteness and should provide suitable notification mechanisms if the state of incompleteness changes.

This paper gives a formal description of suitable database support for an environment supporting the interactive development of ACT ONE specifications. The object-oriented data model of TROLL *light* [3], a language developed recently within the KorSo project, is used to present the design of such an environment and its dynamic behavior. However, the concepts used are general enough to support other specification and even programming languages as well. Therefore, we feel the design of a conceptual schema for ACT ONE — an overview of the language can be found in [6, 9, 4] — is mainly a case study in employing an object-oriented data model for database support of specification or programming languages. It is therefore a proposal for the consolidation of environments for algebraic specification languages.

2 TROLL *light*

TROLL *light*, a dialect of TROLL [11] and its predecessor OBLOG [14], allows to represent structure and behavior of objects. It is designed to describe the universe of discourse as a system of concurrently existing and interacting objects. Object descriptions are called templates in TROLL *light*. Because of their purely descriptive nature templates may 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 settled on a fresh designation. 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 CSP-like processes
END TEMPLATE
```

We cannot go into the details explaining how TROLL *light* can be translated to the object-oriented database system [12] used in the Braunschweig KorSo project or into the operational semantics of the language [7]. But we will explain the language features of TROLL *light* by means of an example. If we want to specify binary trees then the following template is appropriate for this task.

```
TEMPLATE Node
  DATA TYPES  String, Nat;
  SUBOBJECTS  Children(Name:string):node;
  ATTRIBUTES  Content:nat;
              DERIVED Total:nat;
  EVENTS      BIRTH create;
                    updateContent(Content:nat);
                    createChild(Name:string);
              DEATH destroy;
  CONSTRAINTS CNT(Children)<=2;
  VALUATION   [ updateContent(C) ] Content=C;
  DERIVATION  Total = SUM ( SELECT Total(PRJ2(C))
                            FROM   C IN Children ) + Content;
  INTERACTION SELF.createChild(N) >> Children(N).create;
  BEHAVIOR    PROCESS nodeLife =
                ( updateContent -> nodeLife |
                  createChild -> nodeLife |
                  { CNT(Children)=0 } destroy -> POSTMORTEM );
              ( create -> updateContent -> nodeLife );
END TEMPLATE
```

The above introduces a template called `Node` and a corresponding object sort `node`. Objects belonging to this template are the nodes of our binary tree.

In the `DATA TYPES` section we have to list which data types are used in the current template. Data types are assumed to be specified in a data specification language.

In the `SUBOBJECTS` section we specify sub-object relationships. A sub-object declaration provides possible sub-object names. Above we have (for each object of sort `node`) an infinite number of such sub-object names, for instance `Children("BlueNode")` and `Children("RedNode")`; however in each state only a finite number of names takes a defined value. Sub-object relationships organize the objects of an object community into a hierarchy. We have decided to use a parameterized sub-object in order to demonstrate this possibility. For the binary tree example a reasonable alternative would be to introduce two non-parameterized sub-objects `left, right:node`.

Observable properties of objects are called attributes and are specified in the `ATTRIBUTES` section. Above we have two non-parameterized, data-valued attributes, namely `Content` and `Total`. But analogously to sub-objects, attributes may have data- and object-valued parameters. We allow attributes to be data- and object-valued and to be multi-valued by means of the predefined type constructors `TUPLE`, `LIST`, `BAG`, `SET`, and `UNION`. Again analogously to sub-objects, only a finite number of attribute names takes a defined value. The difference between sub-objects and attributes is that arbitrary and not only hierarchical relationships may be established by attributes.

Events are specified in the `EVENTS` section of a template. Events represent actions which happen to objects of the current template. In our example nodes can be created and their attribute `Content` can be updated; in nodes children can be created and they can be destroyed. Events can have data- and object-valued parameters.

Integrity constraints restricting the allowed object states, in particular restricting the values which sub-objects and attributes may take, are specified in the `CONSTRAINTS` section. The above constraint requires that nodes can have at most two children. This is established in the following way. Remember that only a finite number of sub-object names take a defined value. Therefore the term `Children` can also be viewed as a term of sort `SET(TUPLE(string,node))` and in each state this term is evaluated to the finite set of (string,node)-tuples representing the children of the current node. The cardinality of this set is restricted above (`CNT` stands for count).

The effect of events on (non-derived) attribute is described in the `VALUATION` section by valuation rules. Such a valuation rule describes that after the occurrence of the event the attribute has the value given by the right-hand side term. Additionally, evaluation rules may have preconditions. The attribute parameters and the right-hand side term as well as the precondition are evaluated in the state before the event occurs. There is a frame rule saying that attributes which are not caught by a suitable valuation rule 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.

Derived attributes are marked by the keyword `DERIVED` in the `ATTRIBUTES` section of a template. The evaluation of a derived attribute is specified in the `DERIVATION` section by an equation with the derived attribute as left-hand side and a defining term as the right-hand side. In our example we even have a recursively defined attribute because `Total` appears on the right-hand side of its defining equation. The inner `SELECT` term is a (bag-valued) term of our query calculus. In the `SELECT` expression, `Children` is used as a term of sort `SET(TUPLE(nat,node))`. The term describes that we have to sum up the values of the attribute `Total` of all existing sub-objects of the considered object and have to add the value of the attribute `Content`.

Communication between objects is specified in the `INTERACTION` section by event calling. In the general case event calling schemes may have preconditions. Such an event calling scheme expresses the following dependency: Provided the precondition is fulfilled, the occurrence of an event matching the left-hand side causes the simultaneous occurrence of the event on the right-hand side. In our example this means that whenever the event `createChild` occurs in the super-object the event `create` occurs in the sub-object. If the object qualifications are missing, we assume that the corresponding event refers to the object for which this interaction is specified. Thus we could have dropped the `SELF.` part in the interaction rule.

In the `BEHAVIOR` section admissible life cycles are specified for objects. Life cycles are described by so-called process patterns. Each object has a finite number of possible o-machine states (o for object). For these o-machine states, state transitions caused by event occurrences are specified in the process patterns. Event sequences (for instance `create -> updateContent -> nodeLife`) as well as event dependent branchings (for instance `updateContent -> nodeLife | createChild -> nodeLife`) may be described here. Additional preconditions for state transitions allow to forbid event occurrences, i.e., if in a state the precondition for an event is not fulfilled the event is not permitted to occur. Thereby, the admissible life cycles can be restricted further. For example the death event `destroy` is only allowed to occur if the node in question has no children. Life cycles must start with a birth event and possibly end with a death event. Omitting the `BEHAVIOR` part allows arbitrary event occurrences after a birth event and before a possible death event.

3 A Design of the ACT ONE Environment

The design of an environment for a specification language like ACT ONE supporting specification development and associated information like specifications, proofs, documentation, etc. in a comprehensive way is a non-trivial task leading to a complex conceptual schema. We focus on some aspects of the design, namely support for the interactive nature of the specification and describe them in TROLL *light*.

To give an indication what is to be modeled let us discuss typical situations of specification development with ACT ONE in an interactive environment. First of all such a development is characterized by the fact that we have certain states of incompleteness. It is, e.g., desirable to simply store the text (provided by some editor) of a type or a parameter without doing any checking. Further steps are to check the context free syntax and the context conditions of the language. A context sensitive check requires, e.g., that all used types are already available.

Second, since a complex specification is a collection of related entities, the modification of one of them may have effects on others. To give an example, if we have a type `NATLIST` which uses types `LIST` and `NAT` and explicitly refers to sort `Nat` and if this name is changed, e.g., to `Number` also the actualization in type `NATLIST` has to be changed. In an batch-oriented environment where the whole specification is checked, this problem is captured by providing appropriate error messages. In an interactive environment a notification mechanism seems to be useful to handle this situation. This means, e.g., that the result of a context sensitive check of a specification unit must be invalidated if imported units have been altered. The following TROLL *light* templates are intended to deal with these situations.

```
TEMPLATE Type
  DATA TYPES  Parser, Analyser;
  TEMPLATES   Schema;
  ATTRIBUTES  Name, Text:string;       UsedList:LIST(string);
              UsingTypes:SET(type);    SyntaxTree:syntree;
              FlatSpec:flatspec;       Super:schema;
              DERIVED IsCfCheckable, UsedTypesAvailable,
                      IsCsCheckable:bool;
  EVENTS      BIRTH create(Name:string,Super:schema,
                           UsingTypes:SET(type));
                    alter(Text:string);
                    cfCheck; mkUsedList; csCheck;
                    invalidate; addUsingType(UsedType:type);
  VALUATION   [ create(N,S,UT) ] Name=N, UsingTypes=UT, Super=S ;
              [ alter(T) ] Text=T, SyntaxTree=BOT, FlatSpec=BOT;
              { DEF(parse(Text)) }
                [ cfCheck ] SyntaxTree=parse(Text);
              { DEF(SyntaxTree) }
                [ mkUsedList ] UsedList=getlist(SyntaxTree);
              { DEF(analyse(SyntaxTree)) }
                [ csCheck ] FlatSpec=analyse(SyntaxTree);
              [ invalidate ] Flatspec=BOT;
              [ addUsingType(T) ] UsingTypes=ADD(UsingTypes,T);
```

```
  DERIVATION  IsCfCheckable = DEF(Text);
              UsedTypesAvailable =
                (LTS(UsedList)=BTS(SELECT PRJ1(T)
                                   FROM   T IN TheTypes(Super)
                                   WHERE  PRJ1(T) IN UsedList));
              IsCsCheckable =
                (UsedTypesAvailable AND DEF(SyntaxTree) AND
                 FORALL N IN LTS(UsedList)
                   DEF(FlatSpec(TheTypes(Super,N))));
  INTERACTION alter(T) >> UsingTypes.invalidate;
              mkUsedList >> Super.triggerUsingType(Name);
  BEHAVIOR    PROCESS typeLife =
                ( alter -> typeLife |
                  { IsCfCheckable } cfCheck ->
                                    mkUsedList -> typeLife |
                  { IsCsCheckable } csCheck -> typeLife |
                  invalidate -> typeLife |
                  addUsingType -> typeLife );
              create -> typeLife;
END TEMPLATE
```

The template `Type` represents structural information of ACT ONE types and introduces the following attributes: `Name:` The name of the type is given by a string. `Text:` Textual representation of the type as provided by some editor. `UsedList:` List of used types represented by a list of names. `UsingTypes:` The types stored in the database that use this type, represented by a set of types. `SyntaxTree:` Syntax tree corresponding to the text of a type as provided by the context free analysis. This is modeled by an appropriate data type `Parser` providing a data sort `syntree` and an operation `parse`. `FlatSpec:` The representation of a type after all structuring mechanisms of the language have been resolved (flattened). This attribute is a result of the context sensitive analysis of a type and modeled by a data type `Analyser` providing a data sort `flatspec` and an operation `analyse`. `Super:` Actually this attribute does not belong to the structural information of an ACT ONE type but is of technical nature as explained below. `IsCfCheckable:` A flag indicating whether the textual representation is available. `UsedTypesAvailable:` A flag indicating whether all used types are actually in the database. `IsCsCheckable:` A flag indicating whether context sensitive analysis is possible.

The possible events and according valuations are (hopefully) more or less self-explanatory. The creation of a new ACT ONE type is triggered by the environment represented in our design by the template `Schema`. Since a type may be used by other types without being available in the database the creation of a new type provides it with a set of using types in the database. How this is realized will be seen when dealing with template `Schema`.

The `DERIVATION` section of template `Type` deals with states of incompleteness. A type is context free checkable if its text is available. All used types are available if for every name in `UsedList` there is an ACT ONE type in the database with this name. Note how this derivation relies on the query calculus provided by TROLL *light*. A type is context sensitive checkable if all used types are in the database and are checked context sensitively and the syntax tree of the type is available.

The `INTERACTION` section deals mainly with the effects of modifications. If the text of a specification is altered, all using types must be notified that their context sensitive check is no longer valid. Note that set-valued events provide a very compact description of this effect. The interaction formula `alter(T) >> UsingTypes.invalidate;` is thereby to be interpreted as follows: If a type performs an `alter` event providing a new text, all using types as given by the set-valued attribute `UsingTypes` must perform an `invalidate` event. The second interaction formula describes the following fact: If the use list of a type is calculated by execution of event `mkUsedList` all types in the database whose name occur in the use list are notified that they are used by this type. The `BEHAVIOR` section finally describes possible sequences of events. After an ACT ONE type has been created it can be altered, checked, etc. Certain events are only possible if appropriate preconditions are satisfied. Now we turn to the template `Schema` providing some kind of frame for interaction in the whole environment.

```
TEMPLATE Schema
  DATA TYPES  String, ...;
  TEMPLATES   Type, ...;
  ...
  SUBOBJECTS  TheTypes(Name:string):type;
  EVENTS      BIRTH createSchema;
                    addType(Name:string);
                    triggerUsingType(Name:string);
  ...
  INTERACTION
    addType(N) >>
      TheTypes(N).create(N,SELF,
                         BTS(SELECT PRJ2(T)
                             FROM   T IN TheTypes
                             WHERE  N IN UsedList(PRJ2(T))));
    triggerUsingType(N) >>
      (SELECT PRJ2(T)
       FROM   T IN TheTypes
       WHERE  Name(PRJ2(T)) IN
              getlist(SyntaxTree(TheTypes(N)))
      ).addUsingType(TheTypes(N));
  ...
END TEMPLATE
```

This template is a result of the *principle of object existence* adopted for TROLL *light* stating that every object can only be discerned as an object by a higher object. In our design this means that we intend only to have one instance of template `Schema`, comprising all ACT ONE types as sub-objects. For this reason the template `Schema` has a parametrized sub-object `TheTypes` where data sort `string` is the type of the parameter. This modeling of sub-objects implies that ACT ONE types must have unique names (according to TROLL *light* conventions).

The template `Schema` is responsible for our notification mechanism by providing appropriate interaction statements. The event `addType` creates a new ACT ONE type by enforcing a `create` event in an object belonging to the template `Type`. The event `triggerUsingType` is forced by the event `mkUsedList` from template `Type` and itself enforces the event `addUsingType`.

4 Conclusion

The main contribution of this paper is a formal description of the dynamic behavior of an environment supporting the interactive development of specifications like this has been done in [2]. Different degrees of incompleteness and an appropriate notification scheme are described. Although the presented TROLL *light* specification captures only some aspects of an envisaged ACT ONE environment, we think that these aspects are quite relevant due to the event driven nature of modern software systems. The description itself exhibits many interesting features of TROLL *light*: Set-valued event calling to model notification, the powerful query calculus to express, e.g., derived attributes, and the interaction section to enforce synchronization between objects. Theses features show the power of TROLL *light* to tackle rather involved situations.

References

[1] M. Bidoit and C. Choppy. ASSPEGIQUE: An Integrated Environment for Algebraic Specifications. In H.Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Proc. TAPSOFT'85*, pages 246–260. Springer, LNCS 186, 1985.

[2] M. Bidoit, F. Capy, and C. Choppy. The Design and Specification of the ASSPEGIQUE Database. In A. Miola, editor, *Proc. DISCO'90*, pages 205–214. Springer, LNCS 429, 1990.

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

[4] I. Claßen. Revised ACT ONE: Categorical constructions for an algebraic specification language. In *Categorical Methods in Computer Science, LNCS 393*, pages 124–141. Springer, 1989.

[5] H. Ehrig, I. Claßen, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development — The ACT Approach.* AMAST Series in Computing. World Scientific Publishing, 1993.

[6] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two level of semantics. Technical Report 83/03, TU Berlin, 1983.

[7] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *DISCO'93*. Springer, Berlin, LNCS, 1993.

[8] J.A. Goguen and T. Winkler. Introducing OBJ3. Research Report SRI-CSL-88-9, SRI International, 1988.

[9] H. Hansen. The ACT-System: Experiences and Future Enhancements. In D.T. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification (WADT'87)*, pages 113–130. Springer, LNCS 332, 1987.

[10] H. Hussmann. Rapid Prototyping for Algebraic Specifications - RAP-System User's Manual. Report MIP 8505, CS Department, Passau University, 1985.

[11] 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.

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

[13] J. Loeckx and M. Wolf. The OBSCURE Manual. Technical Report 91/03, Computer Science Department, University of Saabrücken, 1991.

[14] 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*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.