Safe Derivations in Object Hierarchies

Stefan Conrad[1], **Martin Gogolla**[2], **and Rudolf Herzig**[2]

[1] Universität Magdeburg, Fakultät für Informatik,
Postfach 4120, D–39016 Magdeburg, Germany
[2] Universität Bremen, FB3 Mathematik und Informatik,
Postfach 33 04 40, D–28334 Bremen, Germany
E-mail: conrad@iti.cs.tu-magdeburg.de Fax: +49 391 5592 2020

ABSTRACT. We present a language for specifying structure and behavior of objects in information systems. This language is restricted to a set of core concepts for conceptual modeling. But it includes powerful concepts for specifying constraints and derived data. We present an algorithm for deciding on safe computation of derived information and demonstrate how this algorithm works with an example.

KEY WORDS: object oriented databases, object specification, TROLL light, object hierarchies, derived attributes, safe derivation / safe queries.

1. Introduction

In the last years two important directions for developing databases for non-standard applications have evolved: object-oriented databases and deductive databases.

Object-oriented databases [Ban88, ABD90] aim to support conceptual modeling features known from semantic data models, like SDM [HM81], IRIS [LK86], IFO [AH87], EER [EGH92], and data models for complex objects (e.g., [SS86, PA86, Heu89]). Moreover they come along with object-oriented programming languages, thereby overcoming the well-known impedance mismatch in relational database systems (an example of a popular object-oriented databases systems is O_2 [Deu90]).

In principle, deductive databases are based on the relational data model. Relations are treated as predicates in order to apply the machinery of logic programming. The basic declarative language DATALOG [CGT89, CGT90] has been extended for capturing sets and functions in many different ways. Well-known examples are LDL [NT89, BNST91], LPS [Kup90], and COL [AG91].

The integration of object-orientation and deduction promises to unify the advantages of both areas [Abi90, Bee90]. First efforts resulted, e.g., in F-Logic [KL89, KLW90] and LOCO [STV91, STVV91], two deductive languages incorporating object-oriented features like object identity and inheritance. Although principle problems in integrating object-oriented and deductive databases were stated in [Ull91], there seem to be several meaningful ways for their combination. For instance, object-oriented features can be embedded into declarative languages (e.g., as in IQL [AK89] and OIL [Zan90]). More references on object-orientation, deduction, and their integration can be found in [CG92].

In this paper we present a core language for specifying objects in information systems [CGH92, GCH93, HCG94]. This language called TROLL *light* is a dialect, but not a subset of TROLL [JSH91]. Because our aim in the KorSo project is to study aspects of validating and verifying object specifications we have restricted TROLL *light* to a small number of concepts for modeling structure and behavior of object communities. Nevertheless our language includes a powerful sublanguage for specifying derived data and static integrity constraints in a declarative way. This sublanguage is an offspring of a calculus for an Extended Entity-Relationship model [GH91] and marks one of the mayor differences between TROLL and TROLL *light*.

With our language we specify derived data, not derived objects. This is in contrast to approaches where derivation rules may create new objects (F-Logic [KL89], HiLog [CKW89] or Living in a Lattice [HS91]). Usually, derived objects represent a special form of views. The problem occurring in those approaches is how to classify the derived objects with regard to an existing class structure [HS91]. In our comprehension the result of a query is a complex value [AFS89] which, of course, may contain object identifiers. We consider values like `42` or { `flight(HAJ,JFK)`, `flight(JFK,DAL)`, `flight(HAJ,FUE)` } as time-independent things. In contrast, objects are persistent items showing (apart from their identity) time-dependent features which can be observed by values [EGS90].

But we can specify derived attributes for objects. The question arising here is whether we can ensure that the values of these attributes can always be computed. For this problem we present an algorithm which gives a sufficient criterion for the safety of a large class of derivations. This algorithm is based on the principle of stratification (cf. [ABW88, CGT89, etc.]) which however had to be modified due to the fact that we use a language conceptually richer than Horn-clause languages and due to the fact that our specification language requires to organize objects into object hierarchies. This is why we will talk about hierarchical stratification.

In the sequel we briefly introduce TROLL *light* and the sublanguage which we will call query calculus for convenience (Sect. 2). In Sect. 3, we first present our algorithm for deciding on safe computation of derived information, then we demonstrate this algorithm with an example, and finally we give a short example for safe computation which our algorithm cannot detect. We conclude with some remarks on the presented algorithm and on future work.

2. Concepts of TROLL *light*

In the sequel we present the concepts of TROLL *light*, a dialect of TROLL [JSH91]. In our project, we need a simple but powerful language for investigating aspects of validating and verifying specifications. Therefore, TROLL *light* offers only a small number of core concepts for specifying objects and object communities in information systems. However, with regard to certain aspects, TROLL *light* is richer than TROLL. For example, we integrated a well-founded query calculus. This offers the possibility to formulate (static) integrity constraints and to query objects. Furthermore, this calculus is used to describe derived information, i.e., derived attributes. We will consider this query calculus in more detail after having given a brief overview of TROLL *light*.

2.1 Concepts for modeling object structure and behavior

In TROLL *light* objects are described by templates which have the following structure.

```
TEMPLATE name of current template
  DATA TYPES   data types used in current template
  TEMPLATES    other templates used in current template
  SUBOBJECTS   slots for subobjects
  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;
```

In the following we will explain each section which may appear in a template. First, the `DATA TYPES` and `TEMPLATES` sections are explained because they provide the context for object descriptions. Then, the `ATTRIBUTES`, `CONSTRAINTS`, `EVENTS`, `VALUATION`, and `BEHAVIOR` sections are considered. With these concepts we are able to describe single objects. Afterwards, we deal with the `SUBOBJECTS` and `INTERACTION` section which we need for forming object communities. The `DERIVATION` section will be treated in more detail in Sect. 2.2.

In the `DATA TYPES` section we have to list which data types are used in the current template, i.e., sorts, operations, and predicates specified for these data types are made known. Please note that TROLL *light* is not intended for specifying data types. Instead we assume data type specifications to be already given. For this, we may use other specification languages, like SPECTRUM [BFG93] (used in the KorSo project), ACT ONE [EFH83], PLUSS [Gau84], or OBJ3 [GW88].

Analogously, other templates used in the current template must be declared in the `TEMPLATES` section. We employ a certain convention concerning data types, templates, and associated sorts. Names starting with an upper case letter denote data types and templates, i.e., complete modules, whereas the corresponding sort names start with a lower case letter. For example, `Nat` refers to the complete specification of the natural numbers including the sort `nat` and constants and operations like `0 : -> nat` or `+ : nat nat -> nat` . The sorts corresponding to templates are called object sorts.

Observable properties of objects are attributes which are specified in the `ATTRIBUTES` section. The general form of an attribute declaration is a [`(`$[p_1:]s_1, \ldots, [p_n:]s_n$`)`] $: d$, 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). To describe the intuitive meaning of parameters, optional parameter names p_i might be added. The sort expression d may be built over both data and object sorts by applying the predefined type constructors `TUPLE`, `LIST`, `BAG`, `SET`, and `UNION`. This allows to specify arbitrary complex attributes which can be data-valued, object-valued, multi-valued, composite, and alternative. We allow attribute names to be provided with parameters, e.g., the attribute `SoldBooks(Year:nat):nat` in a template `Author`. Thereby, an infinite number of attribute names, like `SoldBooks(1993)`, can be introduced. However, only a finite number of attributes of an object can have a defined value.

Static integrity constraints restricting the allowed object states, in particular restricting the values which attributes may take, are specified in the `CONSTRAINTS` section. Here, we omit details on constraints because in Sect. 2.2 the construction of terms and formulas for specifying derivation rules of derived attributes is described. In principle, the same kind of terms and formulas may be used for constraints.

Events are specified in the `EVENTS` section of a template by e[`(`$[p_1:]s_1, \ldots, [p_n:]s_n$`)`], where e denotes an event generator and $s_1, \ldots, s_n$ represent optional parameter sort expressions. In order to give an intuitive meaning to parameters, optional parameter names p_i may be added. Events are used for modeling incidents which may happen in an object's life. Special events are birth and death events (marked by `BIRTH` and `DEATH`, resp.). Several birth or death events may be specified for the same object. In any case there must be at least one birth event description in a template.

The effect of events on attribute values is described in the `VALUATION` section by valuation rules having the following form:

[`{`*precondition*`}`] `[`*event_descr*`]` *attr_term* = *term*

Such a valuation rule describes that after the occurrence of an event matching *event_descr* the attribute denoted by *attr_term* has the value given by *term* provided the optional *precondition* is fulfilled. Both terms 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. Of course, derived attributes cannot directly be manipulated by valuation rules.

In the `BEHAVIOR` section admissible life cycles may be 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 as well as event dependent branchings may be described here. Additional preconditions for state transitions allow to forbid event occurrences, i.e., if in an o-machine 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. Life cycles must start with a birth event and possibly end with a death event. Omitting the `BEHAVIOR` section in a template allows arbitrary event occurrences after a birth event and before a possible death event.

For relating objects in an object community we can specify subobject relationships in the `SUBOBJECTS` section. A subobject declaration $u\,[\,(\,[p_1:]s_1,\ldots,[p_n:]s_n\,)\,]:o$ resembles an attribute declaration. Here, however, the sort expression o is restricted to object sorts. Such a subobject declaration provides possible subobject names. Subobject relationships organize the objects of an object community into a hierarchy.

Communication between objects is specified in the `INTERACTION` section by event calling schemes of the following form:

[{*precondition*}] [*source_object*.]*source_event* >> [*destination_object*.]*destination_event*

Such an event calling scheme expresses the following dependency: Provided the precondition is fulfilled the occurrence of an event matching *source_event* in an object matching *source_object* causes the simultaneous occurrence of the event described by *destination_event* in the object described by *destination_object*. If *source_object* or *destination_object* are missing, we assume that the corresponding event refers to the object for which this interaction is specified.

2.2 Employing the query calculus for derivation of attributes

Now, we will put the focus on derived attributes. TROLL *light* offers this concept because it enables the specification of information which can be computed from other information already present. Thereby, redundant storage of data can be avoided. Furthermore, it is possible to query objects. Here, we concentrate on derived attributes, but the presented calculus can also be used for general query purposes.

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 right hand side.

Example:

```
TEMPLATE Node
  DATA TYPES  Nat;
  SUBOBJECTS  Child(nat) : node;
  ATTRIBUTES  Content : nat;
              DERIVED  Total : nat;
  ...
  DERIVATION  Total = SUM ( SELECT Total(PRJ(X,2)) FROM X IN Child )
                      + Content;
  ...
END TEMPLATE;
```

The above template describes objects which can appear as nodes in a tree. Each node has a finite number of private subobjects. For instance binary trees could be described as objects with at most two subobjects, say `Child(0)` and `Child(1)` giving the left and right subtree, respectively. Objects described by this template have a non-derived attribute `Content` and a derived attribute `Total`. In each object state the value of the derived attribute is given by the defining term

```
SUM( SELECT Total(PRJ(X,2)) FROM X IN Child )
+ Content;
```

which is a term of our query calculus. In the `SELECT` expression, `Child` is used as a term of sort `SET(TUPLE(nat,node))`. This will be explained below. The term describes that we have to sum up the values of the attributes `Total` of all existing subobjects of the considered object and to add the value of the attribute `Content`. In the sequel, we discuss this query calculus in more detail in order to give an impression of its expressiveness. First, we have to distinguish terms and formulas.

Terms may be built with variables and operation symbols as usual. As already mentioned in the previous section we assume a data specification including a data signature to be given. So, data sorts and operation symbols are known. Additional sorts are object sorts which correspond to templates.

Attribute and subobject symbols are special operation symbols. Recall that attributes are specified by a declaration $a\!:\!d$ for non-parameterized attributes and by $a'(s_1,\ldots,s_n)\!:\!d$ for parameterized attributes. Now, we have to distinguish where we want to use attribute symbols for constructing terms. If they shall refer to the object itself, then we may use a respectively $a'(t_1,\ldots,t_n)$ as terms of sort d (with $t_1,\ldots,t_n$ being terms of suitable sorts), e.g., we use `Content` for referring to the object's own attribute. If we want to refer to another object we have to add a parameter denoting this object, e.g., $a(\mathrm{o})$ respectively $a'(o,t_1,\ldots,t_n)$. For example `Total(PRJ(X,2))` denotes the attribute `Total` of an object referred to by `PRJ(X,2)`. Additionally, there is a further use of parameterized attributes. a' (resp. $a'(o)$) is a term of sort `SET(TUPLE(`$s_1,\ldots,s_n,d$`))` denoting the set of all tuples which can be constructed from parameter values for which the attribute has a value different from undefined. Subobject symbols can be used in the same way. For a parameterized subobject declaration $u(s_1,\ldots,s_n)\!:\!s_o$ the term u (resp. $u(o)$) is of sort `SET(TUPLE(`$s_1,\ldots,s_n,s_o$`))` and includes all tuples for which the corresponding subobject (of object sort s_o) is currently existing. For instance, we have used the term `Child` in the defining term of the derived attribute `Total`.

Two additional constructs are offered: The conditional `(IF...THEN...ELSE...)` and the query construct `(SELECT...FROM...[WHERE...])`. A term `(IF` φ `THEN` t_1 `ELSE` t_2`)` is admissible if φ is a formula and t_1, t_2 are terms of the same sort. This sort also determines the sort of the complete term. The query construct requires some more explanation. The general form of a query term is as follows:

$$(\texttt{SELECT}\ t_1,\ldots,t_n\quad \texttt{FROM}\ \ decl_1,\ldots,decl_k\ [\texttt{WHERE}\ \varphi\ ]\,)$$

In the `FROM`-part there is a list of variable declarations. Each declaration has the form var `IN` *range_term* where var is a variable and *range_term* is a multi-valued term giving a finite range to var. In *range_term* there may occur other variables which must be declared before. Thereby, dependent ranges can be specified. In the `SELECT`-part a list of terms $t_1,\ldots,t_n$ is given in which the variables introduced in the `FROM`-part may occur. The sort of such a query term is given by `BAG(TUPLE(`$sort(t_1),\ldots,sort(t_n)$`))` where $sort(t_i)$ denotes the sort of term t_i. The value of a query term is the least multi-set computed as follows. For each instantiation of the variables (according to their declarations) the tuple consisting of the values obtained by evaluating the terms $t_1,\ldots,t_n$ is an element of the result provided the optional selection condition φ is fulfilled. If the `WHERE`-part is missing the selection condition is assumed to equal `TRUE`.

To round off the construction of terms some additional operations are used. For converting multi-sets or lists into sets the operations `BTS` or `LTS`, respectively, can be used (Bag To Set or List To Set). The operation `PRJ` is necessary for projecting components from a tuple. Moreover, there are several aggregate functions like `CNT` (for counting elements) and `SUM` which can be applied to sets, lists, or multi-sets.

Next, we briefly sketch the possibilities to build formulas. Because we assume a data signature containing predicate symbols to be given, these predicate symbols applied to terms yield atomic formulas (e.g., $p(t_1,\ldots,t_n)$ for some predicate symbol p; of course, infix notation for binary predicates could be used for convenience). Furthermore, there are pre-defined predicate symbols: `IN` for testing whether a value is an element of a set (list, multi-set), and `DEF` for testing whether a term yields a value different from the undefined value.

Formulas may be connected by the connectives `AND`, `OR`, and `IMPLIES`. Furthermore, formulas can be built by applying `NOT`. For quantifying variables we have a universal quantifier `FORALL` and an existential one `EXISTS`. A variable declaration as explained for query terms must follow each of these quantifiers. Thereby, quantifiers bind variables to finite ranges only. The above assumptions guarantee that formulas can be evaluated in each state of an object community.

Now, we close this section by explaining the example for a derived attribute given above. In the query term the variable `X` is given the finite range `Child`. `Child` is a term of sort `SET(TUPLE(nat,node))` yielding a set of pairs consisting of a natural number and an object of sort `node`. As explained above only those pairs are elements of `Child` where the second component is a living subobject of the considered object, and the first component is the number n contributing to the subobject's name `Child(`n`)`. In consequence, `X` ranges over these pairs. This is why we have to project the second component of such pairs by `PRJ(X,2)` in order to get the subobject itself. The application of `Total(...)` yields the value of the attribute `Total` of the respective subobject. The aggregation function `SUM` adds up all these attribute values which are in the result of the query term. Finally, the value of the object's own attribute `Content` is added. For objects having no subobjects the `SELECT` term evaluates to the empty set and the aggregation function `SUM` yields `0` as result.

The usage of other attributes for defining the value of derived attributes imposes the problem that cyclic dependencies can be specified such that it might be impossible to safely compute the values of derived attributes. In the next section we deal with this problem by presenting an algorithm for checking TROLL *light* specifications whether their derived attributes can be safely computed.

3. Ensuring safe computation of derived attributes

In the sequel we develop an algorithm for checking a TROLL *light* specification for safe derivation. For this we have to analyze the defining terms of derived attributes and to construct a dependency graph for derived attributes. If this dependency graph fulfills certain conditions the computation of derived attributes is guaranteed to be safe. However it should be noted that there may be specifications which guarantee safe computation of derived attributes where the dependency graph does not fulfill these conditions.

Next we present our algorithm for ensuring safe derivations. Afterwards we will apply our algorithm to a small example, and, finally, we will give an example which shows that our algorithm cannot cover any case in which the computation of derived attributes is safe. Therefore, we have a sufficient but not necessary criterion.

3.1 The algorithm

The algorithm for ensuring safe computation of derived attributes is based on a syntactical analysis of the defining terms of derived attributes. The result we want to obtain is a graph which reflects the computation of derived attributes. If this graph fulfills a certain condition which we call the *stratification condition* then the computation of derived attributes is guaranteed to be safe.

The main idea of the following algorithm is that the computation of a derived attribute is unsafe if its computation may become cyclic. Then, there is a directly or indirectly self-recursive definition of that derived attribute. In consequence, we have to check whether such self-recursive derivations have been specified. Therefore, the algorithm constructs a graph from the defining terms of derived attributes. Nodes are attribute and subobject symbols specified in the object descriptions. Edges reflect computational order, i.e., a path starting at a symbol of a derived attribute goes to symbols of attributes and subobjects which are used for computations of that derived attribute. Going to a subobject symbol means that the following symbols (on the path) refer to attributes or subobjects of a subobject. This is the positive case because we are moving downwards in the object hierarchy. The negative case is going to a symbol of a non-derived, object-valued attribute. Here, we cannot make a general statement to which object consequent symbols on a path are referring. Because this might result in self-recursive computations we cannot guarantee safe computation.

The graph construction algorithm:

Input:	a collection of TROLL *light* templates as a specification of an object community
Output:	a directed graph with attribute and subobject symbols as nodes and edges denoting access dependencies provided by the specification of derived attributes

Execute the following steps for each derived attribute:

1. *Build the syntax tree of its defining term.*
2. *Eliminate variables from the syntax tree.*

   Two cases may occur:

   - The considered variable is free in the defining term. This may only occur for parameterized derived attributes because for such an attribute the defining term may have free variables referring to parameters of the attribute. In consequence, when evaluating such an attribute these variables are substituted by the actual parameters. Therefore, these variables cannot introduce dependencies on other attributes, and we may ignore these variables.
   - The considered variable is bound by a variable declaration in which a multi-valued term describing the range for the variable is given. This term may involve attributes and subobjects. Therefore, we must substitute each occurrence of the variable by the syntax tree of that term. The occurrence of the variable in its own declaration may be totally ignored, because, here, the substitution does not affect the further steps.

3. *Eliminate all operation symbols from the modified syntax tree which are neither attribute symbols nor subobject symbols.*

   Thereby, we remove operation symbols from the tree which do not affect the safe computation of derived attributes. Among these symbols there are symbols given by data types (e.g., symbols like `2` and `+`), symbols standing for predefined operations (e.g., `PRJ`) or for special constructs of TROLL *light* (e.g., `IF`, `SELECT`). The edges of the resulting graph have to be constructed as follows: There is an edge between two remaining symbols if and only if there was a path from the root to a leaf in the former tree and both symbols lie on that path.
4. *Interpret the edges in the sense of computational order and construct the dependency graph.*

   Each edge in the graph obtained in the previous step can be interpreted as a computational dependency. This is quite easy to see because syntax trees can be used for determining computational orders. The dependencies given by a syntax tree are that each term represented by a node n in the syntax tree can only be evaluated if all its subterms (represented by nodes which are successors of n in the syntax tree) have been evaluated before. Therefore we orientate the edges in a way that they are going upwards, i.e., from subterms to root terms.

   Now, we know that the derived attribute we are considering depends on those attributes and subobjects for which there is a node with the corresponding symbol in the obtained graph. But, if there is a chain $n_1 \to \ldots \to n_k$ in that graph then the dependency on $n_2, \ldots,$ and n_k is not a "direct" one. We have first to evaluate n_1, then $n_2, \ldots,$ and finally n_k. Thereby, the edges reflect the navigation through templates necessary for evaluating a derived attribute. And this navigation starts at the considered derived attribute itself. In consequence, we build the final dependency graph by adding a node with the symbol of the considered derived attribute and by drawing edges from this new node to each starting node of a chain. Please note that isolated nodes are considered to be trivial chains and, thereby, they are also starting nodes of chains. Now, nodes labeled by the same symbol are collapsed – as well as multiple edges from one node to another.

After having constructed a dependency graph for each derived attribute, we can bring together these graphs by collapsing nodes labeled with the same symbol and by eliminating multiple edges. Next, we have to find all (minimal) cycles in the resulting graph. For this, standard graph algorithms can be applied. Then the following condition must be checked for fulfillment:

Stratification condition:

The computation of derived attributes is safe if in the dependency graph each cycle, which contains a symbol of a derived attributes, also contains at least one subobject symbol and does not contain any symbol of a non-derived attribute.

This property ensures that the computation can be hierarchically stratified. This is because a cycle containing a subobject symbol represents the fact that the computation of a derived attribute is based recursively on the computation of the same derived attribute but the computation is done for subobjects which are strictly lower in the object hierarchy. This recursion makes no problems because the objects in an object community form an object hierarchy in each instant of time. Moreover, in each state this hierarchy is finite. In consequence, such a recursive computation is safe.

However, if the cycle contains a symbol of an object-valued attribute (or a complex-valued attribute where an object value occurs as part of the complex value) we have a problem. Because object-valued attributes of an object may have for instance the object's own superobject as values such a cycle could be an indication for a non-terminating computation.

3.2 Applying the algorithm

In this section we will give a concrete example for the application of the algorithm presented before. For this we will use the following TROLL *light* specification of persons which may have other persons identified by a name as direct offsprings (modeled as subobjects):

```
TEMPLATE Person
  DATA TYPES String, Nat, Real;
  SUBOBJECTS DirOffspring(string):person;
  ATTRIBUTES Age:nat;
             DERIVED Offsprings:SET(person);
             DERIVED OffspringNames:BAG(string);
             DERIVED OffspringCount:nat;
             DERIVED SumAge:nat;
             DERIVED AvgAge:real;
  ...
  DERIVATION
    Offsprings =
       BTS(SELECT PRJ(X,2) FROM X IN DirOffspring)
       +
       BTS(SELECT Y FROM X IN DirOffspring, Y IN Offsprings(PRJ(X,2)));
    OffspringNames =
       SELECT PRJ(X,1) FROM X IN DirOffspring
       +
       SELECT Y FROM X IN DirOffspring, Y IN OffspringNames(PRJ(X,2));
    OffspringCount =
       CNT(Offsprings);
    SumAge =
```

```
        SUM(SELECT SumAge(PRJ(X,2)) FROM X IN DirOffspring) + Age;
      AvgAge =
        SumAge / (OffspringCount+1);
    ...
  END TEMPLATE
```

There are several derived attributes for persons.

`Offsprings`: The set of direct and indirect offsprings of a person.

`OffspringNames`: The bag of all names of direct and indirect offsprings of a person.

`OffspringCount`: The number of direct and indirect offsprings of a person.

`SumAge`: The sum of ages of all direct and indirect offsprings of a person plus the age of the person itself.

`AvgAge`: The average age of all direct and indirect offsprings of a person including the person itself.

The four steps for constructing the dependency graph for a derived attribute will be demonstrated by the derivation rule for `Offsprings`:

```
BTS(SELECT PRJ(X,2) FROM X IN DirOffspring)
+
BTS(SELECT Y FROM X IN DirOffspring, Y IN Offsprings(PRJ(X,2)));
```

Step 1:

The syntax tree of the considered term is depicted in Fig. 1.

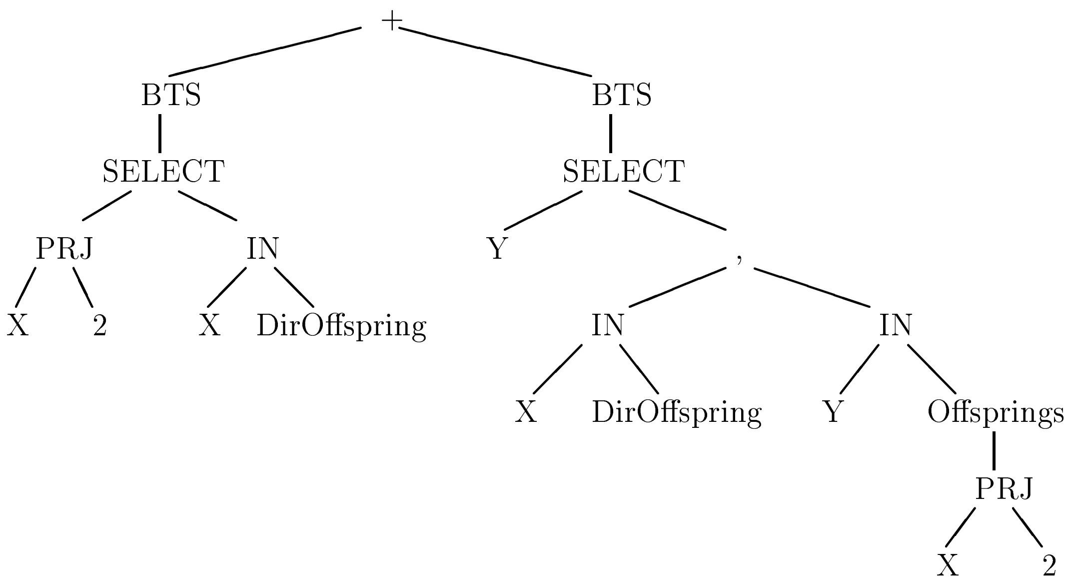


Figure 1: The syntax tree of the defining term.

Step 2:

The elimination of variables from the syntax tree results in a modified tree given in Fig. 2.

Step 3:

By eliminating all symbols from the modified syntax tree which are neither attribute nor subobject symbols we obtain the following graph:

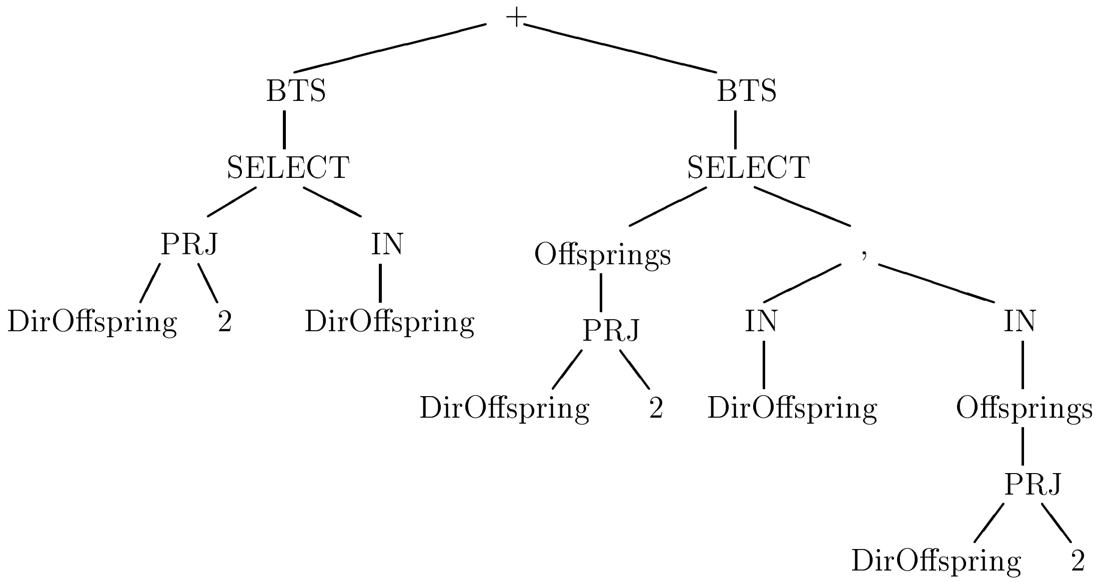
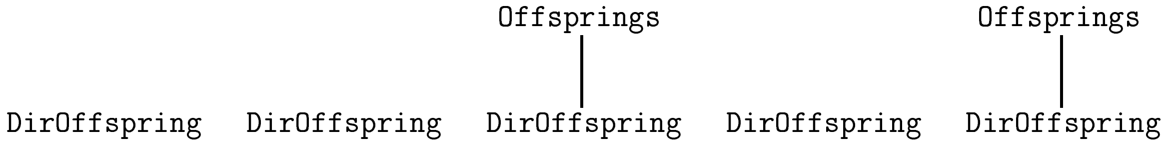
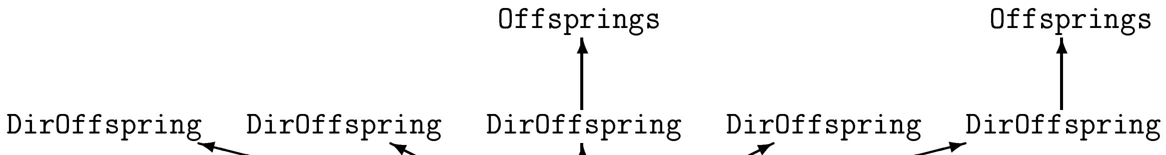


Figure 2: The modified syntax tree.



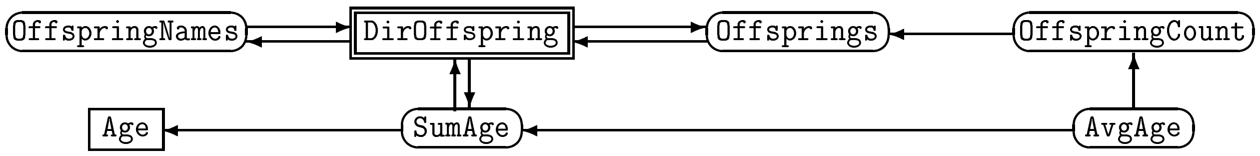
Step 4:

Giving an orientation to the edges as explained, adding the symbol of the considered derived attribute as a node, and connecting this node by directed egdes to the starting nodes of chains yields the following graph:



Collapsing nodes labeled by the same symbol and eliminating multiple edges lead to the dependency graph for the considered derived attribute `Offsprings`:

Now these four steps must be repeated for the other derived attributes. We skip this here and give the final dependency graph in which all dependency graphs of derived attributes are embedded:



In the graph we have marked the different kinds of symbols by different kinds of frames: double frames for subobject symbol, frames with rounded corners for symbols of derived attributes, and square frames for symbols of non-derived attributes. It can easily be checked that each cycle in that graph goes through a subobject symbol, i.e., goes through `DirOffspring`. Furthermore, there is no symbol of a non-derived (and object-valued) attribute in any cycle. In consequence, the graph fulfills the stratification condition and the computation of derived attributes is guaranteed to be safe.

3.3 Safe computations not detected by the algorithm

The algorithm described in the last section provides only a sufficient criterion, not a necessary one. Take for example the following template.

```
TEMPLATE Strange
  ATTRIBUTES  DERIVED A: nat;
              DERIVED B: nat;
              C: nat;
  ...
  DERIVATION  A = (IF C > 100 THEN B ELSE C );
              B = (IF C < 100 THEN A ELSE C );
  ...
END TEMPLATE
```

Here, the dependency graph has a cycle which violates the stratification condition. Nevertheless, we can safely evaluate the values of the attributes `A` and `B` in each possible state. For this we have to consider the value of attribute `C`:

`C = 100:` `A` as well as `B` have the same value as `C`.

`C > 100:` `A` must have the same value as `B`, and `B` gets the value of `C`. In consequence, `A` gets the value of `C`, too.

`C < 100:` `B` must have the same value as `A`, and `A` gets the value of `C`. In consequence, `B` gets the value of `C`, too.

In any case `A` and `B` have the same value as `C`. Thus investigating the semantics of if-terms one may verify the derivation to be safe.

On the other hand it is possible to construct derivation rules where safety totally depends on the current state of an object community. For example replace the subobject relationships of `person` by object-valued attributes. Then it is no longer guaranteed that offspring relationships establish a tree structure, and hence the derivation process may lead into cycles.

4. Conclusions

We have presented TROLL *light*, a core language for specifying structure and behavior of objects in information systems. This language includes a powerful query calculus which can be used to define derived attributes. We have given an algorithm which guarantees safe computation for a class of syntactical decidable derivation rules. The algorithm is based on the idea of hierarchical stratification as a modification of standard stratification.

However, as we have shown there are cases in which safe evaluation could be guaranteed, but which cannot be found by our algorithm. This is due to the fact that in those cases we need semantical knowledge for deciding the safety of computation.

Our algorithm is designed for hierarchies of objects as they are specified in TROLL *light*. It would be interesting whether this algorithm could be transferred to more comfortable specification languages which offer more or other structuring concepts for object communities, for instance, hierarchies of classes together with inheritance. In our specification language we do not offer inheritance because it is often used in a way which contradicts usual declarative approaches. For instance, verifying object properties becomes nearly impossible in the presence of inheritance because, in general, inheritance causes non-monotonicity. As far as we know, the approaches of [BL91, BL92] and [LSV90, LV90, LV91] are the only ones which try to overcome this problem.

However, these and several further questions are lying outside the current scope of our project. Instead we focus our work on a development environment for reliable information systems [VHG93]. For this environment we implement an animation system which can be used for validating object specifications. Furthermore, for verifying TROLL *light* specifications we have developed a verification calculus [Con94] which is realized with the help of the generic proof system Isabelle [Pau90]. The algorithm presented in this paper is thought to be the basis for automatically checking consistency of specifications. Thereby, we hope to support the development of reliable information systems.

Acknowledgement

Work reported here has been partially supported by the German Ministry for Research and Technology (BMFT) under Grant No. 01 IS 203 D (KorSo = Korrekte Software) and by the CEC under Grant No. 6112 (COMPASS). This work originates in Braunschweig where the authors had worked together.

References:

[ABD90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, eds., *Deductive and Object-Oriented Databases*, pages 223–240. Elsevier (North-Holland), 1990.

[Abi90] S. Abiteboul. Towards a deductive object-oriented database language. *Data & Knowledge Engineering*, 5(2):263–287, 1990.

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowlewdge. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Kaufmann, 1988.

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

[AG91] S. Abiteboul and S. Grumbach. A Rule–Based Language with Functions and Sets. *ACM Trans. on Database Systems*, 16(1):1–30, 1991.

[AH87] S. Abiteboul and R. Hull. IFO — A Formal Semantic Database Model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[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.

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

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990.

[BFG93] 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). Technical Report TUM–I9311/9312, TU München, 1993.

[BL91] S. Brass and U.W. Lipeck. Semantics of Inheritance in Logical Object Specifications. In C. Delobel, M. Kifer, and Y. Masunaga, eds., *Deductive and Object-Oriented*

Databases (DOOD'91), pages 411–430. Springer, LNCS 566, 1991.

[BL92] S. Brass and U.W. Lipeck. Generalized Bottom-Up Query Evaluation. In A. Pirotte, C. Delobel, and G. Gottlob, eds., *Advances in Database Technology – EDBT'92*, pages 88–103. Springer, LNCS 580, 1992.

[BNST91] C. Beeri, S. Naqvi, O. Shmuelli, and S. Tsur. Set Constructors in a Logic Database Language. *Journal of Logic Programming*, 10(3):181–232, 1991.

[CG92] S. Conrad and M. Gogolla. An Annotated Bibliography on Object-Orientation and Deduction. *ACM SIGMOD RECORD*, 21(1):123–132, 1992.

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

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What You always wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transaction on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer New York, 1990.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. HiLog as a Platform for Database Languages (or why predicate calculus is not enough). In D. Stemple R. Hull, R. Morrison, ed., *Proc. Int. Workshop on DB Programming Languages*, pages 315–329. Kaufmann, 1989.

[Con94] S. Conrad. On Certification of Specifications for TROLL *light* Objects. In F. Orejas, ed., *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (WADT/Compass'92*. Springer, LNCS, 1994.

[Deu90] O. Deux. The Story of O_2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with Two Levels of Semantics. Technical Report 83-03, Technische Universität Berlin, 1983.

[EGH92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, 9(2):157–204, 1992.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. deBakker, W.P. deRoever, and G. Rozenberg, eds., *Proc. REX/FOOL Workshop*, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin.

[Gau84] M.-C. Gaudel. A First Introduction to PLUSS. Technical Report, Université de Paris-Sud, Orsay, 1984.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, ed., *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.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Research Report SRI-CSL-88-9, SRI International, 1988.

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

[Heu89] A. Heuer. A Data Model for Complex Objects Based on a Semantic Database Model and Nested Relations. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, eds., *Proc. Nested Relations and Complex Objects in Databases*, pages 297–312. Springer, LNCS 361, 1989.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database

Model. *ACM Trans. on Database Systems*, 6(3):351–386, 1981.

[HS91] A. Heuer and P. Sander. Classifying Object–Oriented Query Results in a Class/Type Lattice. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, eds., *Math. Fundamentals of Database Systems MFDBS*, pages 14–28. Springer, LNCS 495, 1991.

[JSH91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[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.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object–Oriented and Frame–Based Languages. Informatik-Manuskript 3/1990, University of Mannheim, 1990.

[Kup90] G. Kuper. Logic Programming with Sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.

[LK86] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In K.R. Dittrich and U. Dayal, eds., *Proc. of the Int. Workshop on Object-Oriented Database Systems (California)*, pages 6–17, 1986.

[LSV90] E. Laenens, D. Sacca, and D. Vermeir. Extending Logic Programming. In *ACM SIGMOD Conf. on Management of Data*, pages 184–193, 1990.

[LV90] E. Laenens and D. Vermeir. A Fixpoint Semantics for Ordered Logic. *Journal of Logic and Computation*, 1(2):159–185, 1990.

[LV91] E. Laenens and D. Vermeir. On the Relationship between Well–Founded and Stable Partial Models. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, eds., *Math. Fundamentals of Database Systems MFDBS*, pages 59–73. Springer, LNCS 495, 1991.

[NT89] S.A. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[PA86] P. Pistor and F. Andersen. Designing a Generalized NF2 Model with an SQL-Type Language Interface. In W. Chen, G. Gardarin, and S. Ohsuga, eds., *Int. Conf. on Very Large Data Bases VLDB*, pages 278–285. Kaufmann, 1986.

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

[SS86] H.-J. Schek and M.H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[STV91] F. Staes, L. Tarantino, and B. Verdonk. A Logic Approach for Supporting Queries in Object Oriented Databases. In F. Van Assche, B. Moulins, and C. Rolland, eds., *Proc. IFIP Working Conf. on the Object–Oriented Approach in Information Systems*, pages 193–208. North–Holland, 1991.

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

[Ull91] J.D. Ullman. A Comparison of Deductive and Object-Oriented Database Systems. In C. Delobel, M. Kifer, and Y. Masunaga, eds., *Deductive and Object-Oriented Databases (DOOD'91)*, pages 263–277. Springer, LNCS 566, 1991.

[VHG93] 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, eds., *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, eds., *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD'89)*, pages 7–24. North–Holland, 1990.