Proceedings of the IASTED International Conference
INTELLIGENT SYSTEMS AND CONTROL
November 19-22, 2001, Clearwater (Tampa), Florida, USA

MAINTAINING DERIVED INFORMATION USING DYNAMIC RULE ORDERING

J.A. BAKKER and J.H. TER BEKKE
Faculty of Information Technology and Systems, Delft University of Technology,
P.O. Box 356, 2600 AJ Delft, The Netherlands, {J.A.Bakker, J.H.terBekke}@its.tudelft.nl

ABSTRACT

Concepts of the Xplain DBMS allow the specification of assertions on derived data and their calculation. In case a term must satisfy a restricted value, the assertion specifies a static constraint. Such assertions can support management decisions based on information, derived from operational data. In order to deal with dependencies between assertions, we propose a framework for a correctly ordered calculation of interdependent data, using a dynamical calculation of rule priority. We also show how rule termination can be guaranteed.

1. INTRODUCTION

In order to facilitate decisions based on derived or aggregated information, such as the total costs of claims per insurance category, derivable data can be stored. This introduces the risk of inconsistency between original and derived data. Therefore, guaranteeing the correctness of derived information is an important task for a DBMS.

Many approaches are dealing with the management of derived information, but it is not possible to discuss them all. We only mention two dominant approaches, more specific information can be found via the mentioned literature. First, in a relational DBMS, derived information can be specified through materialized views, which is the basis for deriving efficient production rules dedicated to specific events (Stonebraker *et al*. [17]; Ceri and Widom [9]; Etzion [12]; Gupta *et al*. [14]).

The derivation of information can also be based on an Event Condition Action (ECA) model (Widom and Ceri, [24]; ACT-NET Consortium [1]; Paton and Díaz [15]). Events can lead to some database state transition that in case of a fulfilled condition leads to the execution of an appropriate action. Although events and actions might occur outside the database proper, we confine the discussion to internal database operations, in particular events triggering the derivation of data. Most ECA models have a simple procedural character [24]; they require an explicit specification of conditions and actions (or calculations) per anticipated event.
Procedural approaches, if applied to the derivation of data, share the following characteristics:

- An update event might trigger the calculation of derived information.
- Any action can be specified, but its adequateness cannot be guaranteed (Appelrath *et al*. [2]).
- It is possible to specify rules with a cyclic dependency: the termination problem.
- The evaluation order of rules must be specified per event: the rule priority problem.

Rule termination and rule priority are well-known issues in the field of active database systems [24]. Elmasri and Navathe [13] illustrate the first problem by an example of mutually dependent rules. The second one is that rule priority depends on the triggering event (Paton and Díaz [15]). However, because of the nonprocedural semantic concepts of Xplain, none of the diverse approaches can be incorporated in the Xplain DBMS [22]. It is the objective of this paper to present a solution for these two problems using the concepts of the Xplain DBMS [16, 18]. Other problems can only be mentioned.

2. CONCEPTS OF THE XPLAIN DBMS

In Xplain, data models are stored on the basis of the meta model shown in figure 1. It is based on the following definitions:

type type = name, domain, /composite/.
type attribute = composite_type, type, kind.
type role attribute = [attribute], prefix.
assert type *its* composite = *any* attribute *per* composite_type.

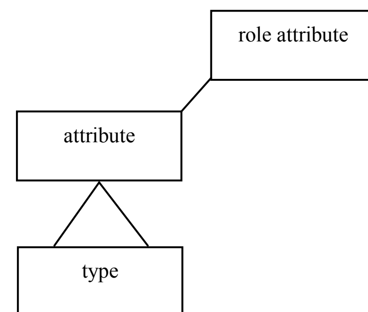

Figure 1: Xplain meta model for data modeling

Each stored type specification consists of four elements: identifier, name, value domain and a Boolean indicating whether or not a type is composite. For example, ‘date’ is a base type and ‘client’ is a composite type (figure 2).

Attributes define a logical connection between two types, which implies inherent structuring; ‘attribute *its* kind’ enables us to distinguish between aggregation and generalization. Generalization is indicated by an attribute between square brackets; see for example the definition of ‘role attribute’, a specialization of ‘attribute’. Role attributes are attributes with a role, indicated by a prefix. The domain of an attribute can be found via ‘attribute *its* type *its* domain’. In our examples, domains are not shown because they are irrelevant for the problems to be discussed. Further, a term between slashes indicates a derived term.

A static restriction can be specified through a derived term and its calculation, possibly accompanied by allowed values. Examples are given after figure 2, which shows an abstraction hierarchy derived from the semantic data model presented after this figure. This model can be used for the registration of data about insurance policies. We also define this model in relational terminology: primary keys are given in **bold** and foreign keys in *italics*. We assume that policies can start every day, but have to be renewed every year on January 1. They always terminate on December 31. We ignore data associated with premium payments.

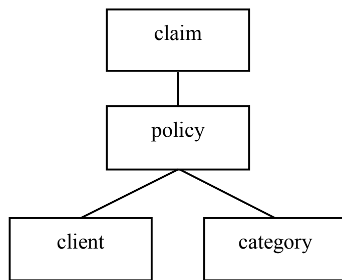

Figure 2: An abstraction hierarchy for insurance policies

We can derive figure 2 from the following relational or semantic specifications:

Relational definition (additional rules are not shown):

client (**client#**, name, address, town, telephone, birth-date, policy-number);
category (**cat#**, name, earnings2001, costs2001, yield2001, excellent2001);
policy (**p#**, *client#*, *cat#*, insured-amount, premium, starting-date);
claim (**claim#**, *p#*, date, description, claimed-amount, reimbursement, pay-date, correctness);

Semantic definition:

type client = name, address, town, telephone, birth_date, /policy_number/.
type category = name, /earnings2001/, /costs2001/, /yield2001/, /excellent2001/.
type policy = client, category, insured_amount, premium, starting_date.
type claim = policy, date, description, claimed_amount, reimbursement, pay_date, /correctness/.

Insert constraints (dynamic rules):
init claim *its* reimbursement = 0.
init claim *its* pay_date = 18990101.

Assertions about derived information (static rules):
{1} *assert* claim *its* correctness (*true*) =
(*yearf*(date) = policy *its yearf* (starting_date)
and date ≥ policy *its* starting_date).
{2} *assert* category *its* earnings2001 =
total policy *its* premium
where yearf (starting_date) = 2001
per category.
{3} *assert* category *its* costs2001 =
total claim *its* reimbursement
where yearf (pay_date) = 2001
per policy *its* category.
{4} *assert* category *its* yield2001 =
earnings2001 – costs2001.
{5} *assert* maxi_yield2001 = *max* category *its* yield2001.
{6} *assert* category *its* excellent2001 =
(yield2001 = maximum_yield2001).
{7} *assert* client *its* policy_number (1..*) =
count policy *per* client.

It is possible to translate Xplain specifications (data models and queries) automatically into relational specifications (De Boer and Ter Bekke [11]).

Assertions specify controlled redundancy if allowed values are included. Assertions specify a derived attribute (assertions 1-4, 6 and 7) or a single derived variable as in assertion 5. A dependency between derived terms occurs if a calculation applies a term derived by another assertion. We can distinguish different kinds of terms. For example, assertion 1 contains terms (‘claim *its* date’ and ‘policy *its* starting_date’) as sub terms of the kind ‘attr’. Assertion 7 shows that a type term ‘policy’ (term with the kind ‘type’) acts as the subject of a set operation (‘*count* policy’). Assertion 2 demonstrates that an attribute can also be the subject of a set operation (‘*total* policy *its* premium’). Assertion 5 specifies a single variable term ‘maxi_yield2001’ (a term of the kind ‘vari’). The assertions 2 and 3 are a preparation for the calculation of yields per category (assertion 4).

3. COMPARING DECLARATIVE AND PROCEDURAL RULES

Assertions can support a kind of integrity, which cannot be enforced by data structure alone [3-8, 18-23]. The present paper investigates the usability of these rules for the maintenance of interdependent, derivable, thus redundant data. As an example, figure 3 shows the seven assertions specified before. Assertion 1 specifies a static constraint through a derived attribute having a particular value. Any update leading to a not-allowed value may not

be executed or has to be accompanied by another operation (action) leading to an allowed value. Assertion 1 states that the date of a claim must be consistent with the starting date of the involved policy. Assertions 2–6 do not specify any constraint; they derive data about the results of each insurance category (category *its* name: “car”, “health”, “life”, etc.). Assertion 7 specifies that the number of policies per client must be at least 1. If an evaluation of assertion 2 or 3 leads to a value modification, this also requires a reevaluation of assertion 4 (‘category *its* yield2001’). If this produces a modified value, then the assertions 5 and 6 are also triggered.

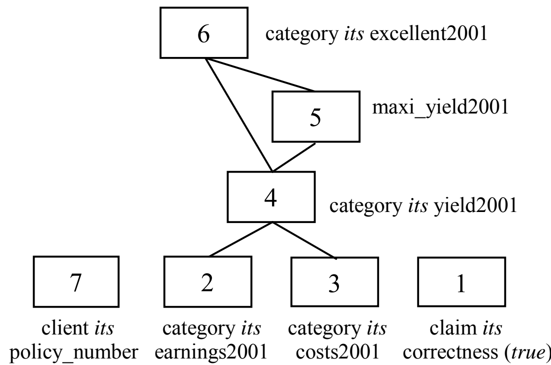

Figure 3: Derived terms and their dependency

The difference between a declarative and a procedural solution can be illustrated by considering a procedural approach mentioned in [13]. For example, dealing with assertion 3 (‘category *its* costs2001’), for each of the following events an adequate procedure, including an operation (calculation) must be specified:

I. Any insertion of an instance of ‘category’.
II. Any insertion of an instance of ‘claim’.
III. Any deletion of an instance of ‘claim’.
IV. Any update of an instance of the attribute ‘claim *its* reimbursement’.
V. Any update of an instance of ‘claim *its* policy’ (part of the link from ‘claim’ via ‘policy’ to ‘category’).
VI. Any update of an instance of ‘policy *its* category’ (also part of this semantic link).
VII. Any update of an instance of ‘claim *its* pay_date’.

Apparently a number of different events can activate a same assertion. As an example we specify a procedure related to event II, the insertion of a tuple of ‘claim’:

```
CREATE TRIGGER category_costs2001
AFTER INSERT ON claim
FOR EACH ROW
WHEN (YEAR(NEW.pay-date) = 2001
      AND NEW.p# IS NOT NULL
      AND NEW.client# IS NOT NULL)
UPDATE category
SET costs2001 = costs2001 + NEW.reimbursement
WHERE cat# IN (SELECT cat# FROM policy
                    WHERE p# IN
                   (SELECT p# FROM claim
                    WHERE claim# = NEW.claim#));
```

This illustrates the versatility and efficiency of a procedural approach to data derivation: an event can trigger any action and (update) actions could control redundancy. Still some problems have to be solved if rules have to deal with controlled data redundancy:

- How to realize a correctly ordered execution of rules? In procedural approaches rule priority has to be specified explicitly, leading to overloaded specifications for programmers who have to design a possibly complex scenario for each triggering event.
- How to guarantee rule termination? In procedural approaches it is possible to specify rules having a cyclic dependency [13, 24].
- How to avoid superfluous calculations? For example, an update of an instance of ‘claim *its* reimbursement’ should only trigger an incremental recalculation of the involved instance of the attribute ‘category *its* costs2001’ instead of the specified set operation.
- How to guarantee that actions produce correctly derived (aggregated) information? In the previous relational example we could specify any update irrespective the (derived) variable and its new value.

In order to discuss rule ordering in more detail, we consider the required consequences of two separate events in the dependency graph of figure 3. The first example is the insertion of an instance of the composite type ‘category’. If we initialize the priority of all assertions to 0, this event should activate the assertions 2, 3, 4 and 6, because they specify derived attributes of ‘category’; they get priority 1. Also the intrinsic attributes of ‘category’ get priority 1 because other terms might depend on them.

The calculation of the derived terms specified by the assertions 2 and 3 should activate assertion 4 again (priority updated to 2), which has to be followed by the activation of assertion 5 and 6 (they get priority 3). Because of the dependency between assertion 6 and 5, assertion 6 finally should get priority 4. The assertions 1 and 7 are not activated; their priority remains 0.

Another example of a trigger is the insertion of an instance of ‘policy’ for the year 2001. Then all attributes of ‘policy’ get priority 1 because they might be involved in a calculation. Via triggering assertion 2 (priority 2) this should lead to activating assertion 4 (priority 3), assertion 5 (priority 4) and assertion 6 (priority 5). Contrary to inserting an instance of ‘category’, assertion 3 (priority 0) is not activated now. Apparently, rule priority depends on the triggering event and not only on the position of rules in a dependency graph. Section 5 presents a solution for dynamic rule ordering.

In the Xplain approach rule termination is not a problem: during the registration of assertions, cyclic dependencies between assertions can be prevented by enforcing that calculations only apply constants or previously registered terms. If a derived term acts as a sub term in the calculation of another term, a dependency between derived terms (thus assertions) exists and their evaluation order is determined by this dependency.

Section 4 presents a meta model for the registration of assertions, (derived) terms and their dependency.

In principle, actions associated with assertions, must lead to derived terms with a correct value.
For example, the following derived attribute initially gets an unallowable value 0:

{7} *assert* client *its* policy_number (1..*) =
count policy *per* client.

Here the required action is the registration of a first policy for each new client. Actions can be predefined and must restore derivable terms into an allowed value. Another solution is to inform a user about a rule violation and to enable the user to specify an action. However, a solution for the adequateness of actions cannot be dealt with here.

4. A META MODEL FOR ASSERTIONS AND TERMS

Before introducing a meta model for assertions, (derived) terms and their dependencies, we mention the possible categories of assertions and terms:

I. Assertions about attributes derived without a set operation, an example is assertion 4.
II. Assertions about single variables derived without a set operation. For example: *assert* year1 = 2001.
III. Assertions about attributes derived with a set operation, an example is assertion 3.
IV. Assertions about single variables derived with a set operation. For example, assertion 8:
assert claimnumber2001 =
count claim *where yearf* (date) = 2001.

The variable 'claimnumber2001' depends on the involved attribute term 'claim *its* date' and the type term 'claim' as well. The subject of a set operation can be a composite type (a type term 'claim' as in assertion 8) or an attribute term ('claim *its* reimbursement' in assertion 3).

Further, a path of attributes may be specified instead of a single attribute. An example is assertion 9 deriving the number of claimed insurance categories per client; two attribute terms are part of the subject: both 'claim *its* policy' and 'policy *its* category'. The other involved attribute term 'policy *its* client' is part of the path from 'claim' to 'client' via 'policy' (in a '*per*' construct):

{9} *assert* client *its* number of claimed categories =
count claim *its* policy *its* category
per policy *its* client.

We propose the following meta model (figure 4) for the registration of assertions, terms and their dependency. This meta model is based on the following definitions:

type type =	name, domain, /composite/, /sequence_number/.
type attribute =	composite_type, type, kind.
type role attribute =	[attribute], prefix.
type range =	preceding_range, type, minimum_value, maximum_value, /correct/, /successor_number/.
type assertion =	[assert_term], calculation_expression, /correct/.
type term =	expression, kind, /derived/, /a_number/, /t_number/, /v_number/, /complete/.
type attribute term =	[term], [spec_attribute], /correct/.
type type term =	[term], [type], /correct/.
type variable =	[term], value, /correct/.
type dependency =	term, involved_term, /correct/.
type allowed value =	term, range, /correct/.

Moreover, we must also apply the following rules:

{10} *assert* range *its* correct (*true*) =
(maximum_value ≥ minimum_value
and (preceding_range = 0
or (*not* preceding_range = 0
and type = preceding_range *its* type
and minimum_value >
preceding_range *its* maximum_value))).
{11} *assert* range *its* successor_number (0..1) =
count range *per* preceding_range.
{12} *assert* type *its* sequence_number (0..1) =
count range *where* preceding_range = 0
per type.
{13} *assert* type *its* composite = *any* attribute
per composite_type.
{14} *assert* term *its* derived =
any assertion *per* assert_term.
{15} *assert* attribute term *its* correct (*true*) =
(term *its* kind = "attr").
{16} *assert* type term *its* correct (*true*) =
(term *its* kind = "type" *and* type *its* composite).
{17} *assert* variable *its* correct (*true*) =
(term *its* kind = "vari").
{18} *assert* dependency *its* correct (*true*) =
(term *its* derived *and not* term = involved_term).
{19} *assert* allowed value *its* correct (*true*) =
(term *its* derived).
{20} *assert* term *its* a_number (0..1) =
count attribute term *per* term.
{21} *assert* term *its* t_number (0..1) =
count type term *per* term.
{22} *assert* term *its* v_number (0..1) =
count variable *per* term.
{23} *assert* term *its* complete (*true*) =
(kind = "attr" *and* a_number =1
or kind = "type" *and* t_number =1
or kind = "vari" *and* v_number = 1).
{24} *assert* assertion *its* correct (*true*) =
(assert_term *its* kind = "attr"
or assert_term *its* kind = "vari").

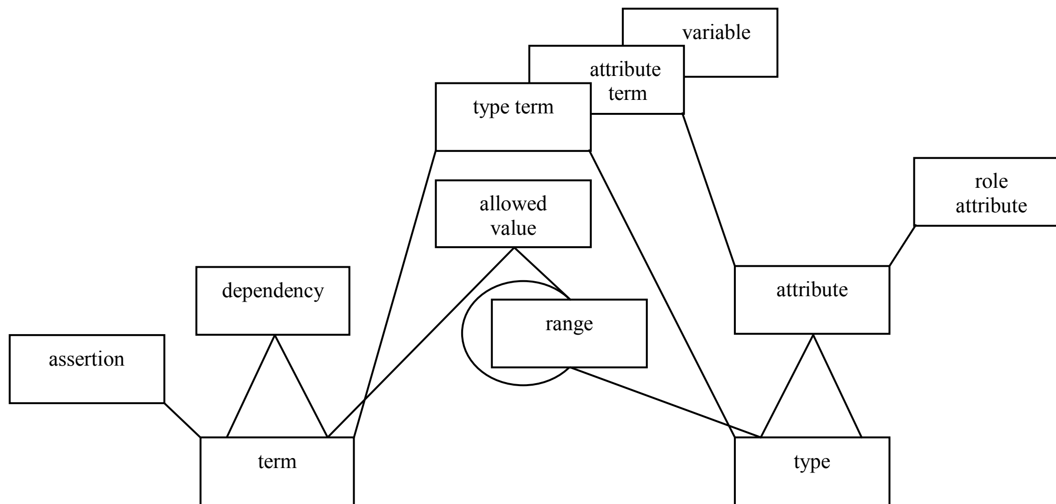

Figure 4: A meta model for assertions, (derived) terms and their dependency

By using different prefixes between brackets, ‘attribute term’ and ‘role attribute’ define overlapping specializations of ‘attribute’. The concept of ‘range’ allows the registration of value ranges or single values. In the last case the minimum equals the maximum value. In case of a value range without a preceding range we can apply: ‘range *its* preceding_range = 0’.
Assertion 10 has to enforce that a range is consistent with the preceding range, whereas assertion 11 has to enforce that a sequence of value ranges is registered. Using assertion 12, a type has at most one sequence of value ranges. Using assertion 14, intrinsic and derived terms can be distinguished. Assertions 15-17 are dealing with the correctness of diverse kinds of ‘term’. The application of assertion 18 guarantees that direct recursive dependencies cannot be specified and that ‘dependency’ really refers to a derived term. Assertion 19 guarantees that instances of ‘allowed value’ refer to a derived term. Assertions 20-23 are dealing with the completeness of term specifications. In the case of ‘term *its* kind = ”attr”’ it is not necessary to require ‘term *its* t_number = 0’ because the meta model already specifies disjoint specialization: a term may not be both a type term and an attribute term. Finally, assertion 24 has to enforce that an assertion either specifies a derived variable or a derived attribute.

The proposed meta model has to be used during the registration and application of assertions. A complete parsing is not required because constants and operators do not affect the logical dependency between terms. This pre-parsing has to perceive, distinguish and register derived terms, involved terms and their dependencies. Calculations must apply previously defined terms or constants in order to prevent cyclic dependencies between assertions.

A brute-force calculation of a derived term could be based on a slight modification of the existing parser (Cheung [10]). A more efficient solution seems possible, but cannot be discussed here. A solution for ordering the evaluation of triggered assertions will be discussed in section 5.

5. DYNAMIC RULE ORDERING

Xplain concepts do not allow us to modify the identifier of an instance of a composite type. A triggering event together with the required calculation and possibly some suitable action must be treated as a single transaction. Consequently, the DBMS must apply immediate coupling between event, calculation and action, even when many assertions are triggered by a single event. Furthermore, because of the required transaction consistency, a DBMS has to register each data operation and its effects in log files before writing to the database. When inserting or deleting an instance of a composite type, this instance and its composite type must be registered. In the case of modifying an attribute instance, this instance, its new value and the involved attribute (if using the deferred update method) must be registered. Using this logging, a correct order for the evaluation of rules can be determined for each possible kind of triggering update event:

- The insertion of an instance of a composite type.
- The deletion of an instance of a composite type, acting in the subject of a set operation.
- The modification of an instance of an attribute, acting as an operand in the derivation of a term.

1. Inserting an instance of a composite type ‘T’.
Two situations can be distinguished. First, type ‘T’ is the composite type of a derived attribute. An example is ‘client’ in assertion 9: ‘client *its* number of claimed categories’. Secondly, type ‘T’ is a type term, acting in the subject of a set operation. An example is the type ‘claim’ in the calculation specified by assertion 9:

assert client *its* number of claimed categories =
count claim *its* policy *its* category
per policy *its* client.

Not each reevaluation of a triggered assertion really triggers the evaluation of other assertions referring to the triggered assertion. For example, the insertion of an instance of ‘claim’ having reimbursement 0, triggers a

recalculation of the involved instance of ‘category *its* costs2001’ (assertion 3), but this recalculation does not change its value. Consequently, assertion 4 should not be triggered. Contrary to that, a value modification of an instance of ‘claim *its* reimbursement’ must trigger the recalculation of the involved instance of ‘category *its* costs2001’ (assertion 3), which leads to a new value and therefore also triggers the recalculation of the involved instance of ‘category *its* yield2001’ (assertion 4). In order to deal with this value-modification dependent trigger propagation, we introduce a Boolean property ‘term *its* modified’, which enables us to register whether the value of a term has been changed.

Xplain also allows the specification of constants, for example: *assert* pi = 3.14. We consider a constant as a special single variable, being independent of other terms. Independent terms with a constant value must have priority 1 because other terms might depend on them. The same applies to derived attributes with a constant value, for example: *assert* policy *its* currency_ratio = 0.899.

We describe the required operations in terms of the declarative language of Xplain in order to illustrate that these operations must respect the structure of the meta model, but the described operations should be considered as run-time code:

1. *extend* term *with* priority = 0. {initialization}
2. *extend* term *with* inserted =
   any attribute term
   where attribute *its* composite_type = “T”
   per term.
3. *extend* term *with* T_related =
   any type term *where* type = “T”
   per term. {a term specified after a set function}
4. *extend* term *with* independent =
   nil dependency *per* term.
5. *extend* term *with* first =
   (inserted *or* T_related *or* independent) .
6. *update* term *its* priority = 1 *where* first.
7. *extend* term *with* second =
   any dependency
   where involved_term *its* priority = 1
   per term.
8. *update* term *its* priority = 2 *where* second.
   {also determine ‘term *its* modified’}
9. *extend* term *with* third =
   any dependency
   where involved_term *its* priority = 2
   and involved_term *its* modified
   per term.
10. *update* term *its* priority = 3 *where* third.
11. *extend* term *with* fourth =
    any dependency
    where involved_term *its* priority = 3
    and involved_term *its* modified
    per term.
12. *update* term *its* priority = 4 *where* fourth. {etc.}

As an illustration we consider the insertion of an instance of the composite type ‘category’. After the operations 1-4 the terms derived by the assertions 1-7 (figure 3) still have priority 0. After operation 5-6 the terms derived by the assertions 2, 3, 4 and 6, which are derived attributes of ‘category’, simply get priority 1, whereas after the steps 7-8 the term derived by assertion 4 gets priority 2. The operations 9-10 lead to priority 3 for the terms derived by the assertions 5 and 6. Finally, the operations 11-12 update the priority of ‘category *its* excellent2001’ (assertion 6) to 4. The DBMS can build a list of triggered assertions for each of the calculated priorities and the derived terms have to be calculated using a correct ordering of these lists.

Another example of a trigger is the insertion of an instance of ‘policy’ with a date in 2001, which initially triggers the assertions 2 and 7, and later, indirectly, the assertions 4-6. But, contrary to the insertion of an instance of ‘category’, assertion 3 is not triggered now. Apparently rule priority depends on the actual triggering event and not only on the relative position of a rule in the dependency graph.

2. Deleting an instance of a composite type ‘T’ acting in the subject of a set operation.

The deletion of an instance of a composite type ‘T’ not acting in the subject of a set operation cannot modify the value of any derived term. For example: the deletion of an instance of ‘client’ does not affect any derived term specified by the assertions 1-7. Only two possible situations must be considered in the case of a deletion:

I. Type ‘T’ is the subject of a set operation. For example, ‘claim’ in:
   assert total_claimnumber2001 =
   count claim *where yearf* (date) = 2001.
II. Type ‘T’ is part of the subject of a set operation. For example, ‘claim’ in:
   assert client *its* number of claimed categories =
   count claim *its* policy *its* category
   per policy *its* client.

In case of a deletion it is not relevant whether or not the type term is subject or is part of a subject. The deletion of a type term does not always lead to a value modification of a derived term. An example is the deletion of a claim not having a date in 2001. In general, we must determine which derived terms are depending on a type term involved in a deletion:

1. *extend* term *with* priority = 0.
2. *extend* term *with* independent =
   nil dependency *per* term.
3. *extend* term *with* deleted =
   any type term *where* type = “T” *per* term.
4. *extend* term *with* first =
   any dependency *where* involved_term *its* deleted
   per term.
5. *update* term *its* priority = 1
   where first *or* (independent *and not* deleted).

6. *extend* term *with* second =
   any dependency
      where involved_term *its* priority = 1
      and involved_term *its* modified
   per term.
7. *update* term *its* priority = 2 *where* second. {etc.}

Because of the occurring deletion, the predicate 'involved_term *its* modified' is superfluous in step 4, but this predicate is still relevant after step 5. As an example we consider the deletion of a claim in 2001. Initially, all derived terms (assertions 1-7) have priority 0. After step 5 only 'category *its* costs2001' (assertion 3) gets priority 1. If the value of the involved instance of 'category *its* costs2001' is changed then the following step is an update of the priority of the term specified by assertion 4, etc.

3. Modifying an instance of attribute 'A', acting as a sub term in a calculation.

An example is the modification of an instance of the attribute 'claim *its* reimbursement'. We only describe the first seven steps:

1. *extend* term *with* priority = 0.
2. *extend* term *with* independent =
   nil dependency *per* term.
3. *extend* term *with* A_related =
   any attribute term *where* attribute = "A"
   per term.
4. *extend* term *with* first = (A_related *or* independent).
5. *update* term *its* priority = 1 *where* first.
6. *extend* term *with* second =
   any dependency
      where involved_term *its* priority = 1
      and involved_term *its* modified
   per term.
7. *update* term *its* priority = 2 *where* second. {etc.}

A complication is that in case of a value violation a required action might act as a trigger. An example is the required insertion of at least one instance of 'policy' associated with the insertion of a new client. This triggers assertion 7:

assert client *its* policy_number (1..*) = *count* policy
per client.

The required insertion of an associated instance of 'policy' for the year 2001 triggers assertion 2, which in its turn activates assertion 4, and so on.

6. DISCUSSION

Procedural ECA approaches to the maintenance of derived information are versatile and efficient because actions and calculations can be dedicated to specific events. They allow - but do not guarantee - adequateness of corrective actions because they do not support a predefinition of allowed values for derived data, which impairs a comparison between calculated and allowed values. Furthermore, rule priority must be specified for each anticipated triggering event because rule priority depends on the event, which could possibly lead to complex scenarios.

Using the proposed meta model, it is not necessary to specify a procedure for each anticipated event because the set of triggered assertions can be inferred from the event. The registered dependencies between terms, enables a DBMS to calculate term priority per event and to maintain the consistency of the set of interdependent assertions in case of deleting an assertion referenced by another assertion.

Moreover, we can prevent cyclic dependencies between derived terms if we require that assertions may only apply constants or previously registered terms. Using assertions, the effect of an action on a derived value can be compared with the allowed value(s), which supports predefined actions and user defined actions as well.

Still some problems must be solved in Xplain: how to specify actions, how to guarantee adequateness of actions and how to infer efficient calculations from set operations and events.

REFERENCES

[1] ACT-NET Consortium, The Active Database Management System Manifesto: A Rulebase of ADBMS Features, *SIGMOD Record 25*(3), 1996, 40-49.

[2] H.-J. Appelrath, H. Berends, H. Jasper and O. Zukunft, Case Studies on Active Database Applications, *Proc. 7th Int. Conf. on Database and Expert Systems Applications DEXA'96*, Zürich, Switzerland, R.R. Wagner and H. Thoma (eds.), *LNCS 1134*, 1996, 69-78.

[3] J.A. Bakker, A semantic approach to enforce correctness of data distribution schemes, *The Computer Journal 37*(7), 1994, 561-575.

[4] J.A. Bakker, Object-Orientation Based on Semantic Transformations, *Proc. 7th Int. Conf. on Database and Expert Systems Applications DEXA'96*, Zürich, Switzerland, R.R. Wagner and H. Thoma (eds.), *LNCS 1134*, 1996, 163-176.

[5] J.A. Bakker, An Extended Meta Model for Conditional Fragmentation, *Proc. 9th Int. Conf. on Database and Expert Systems Applications DEXA'98*, Vienna, Austria, G. Quirchmayr, E. Schweighofer and T.J.M. Bench-Capon (eds.), *LNCS 1460*, 1998, 702-715.

[6] J.A. Bakker, A Semantic Framework for the Design of Data Distribution Schemes, *Proc. 11th Int. Workshop on Database and Expert System Applications DEXA 2000,* Greenwich, London, UK, A.M Tjoa, R.R. Wagner and A. Al-Zobaidie (eds.), IEEE Computer Society, 2000, 653-660.

[7] J.A. Bakker, Semantic Partitioning as a Basis for Parallel I/O in Database Management Systems, *Parallel Computing 26,* 2000, 1491-1513.

[8] B. Bakker and J. ter Bekke, Foolproof query access to search engines, *Proc. 3rd Int. Conf. on Information Integration and Web-based Application & Services iiWAS 2001,* Linz, Austria, W. Winiwarter, St. Bressan and I.K. Ibrahim (eds.), Österreichische Computer Gesellschaft, 2001, 389-394.

[9] S. Ceri and J. Widom, Deriving Production Rules for Incremental View Maintenance, *Proc. 17th Int. Conf. on Very Large Databases*, Barcelona, Spain, R.C.G.M. Lohman and A. Sernadas (eds.), 1991, 577-589.

[10] S.Y. Cheung, *Implementation of a Data Language for the Semantic Data Model*, Masters thesis, Delft University of Technology, Faculty of Information Technology and Systems, 1984 (in Dutch).

[11] B. de Boer and J.H. ter Bekke, Applying semantic database principles in a relational environment, *Proc. IASTED Int. Conf. APPLIED INFORMATICS, Symp.1, Artificial Intelligence and Applications,* Innsbruck, Austria, M.H. Hamza (ed.), ACTA Press, 2001, 400-405. http://www.pobox.com/~berend/xplain2sql/index.html

[12] O. Etzion, PARDES- A Data Driven Oriented Active Database Model, *SIGMOD Record 22*(1), 1993, 7-14.

[13] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, 3rd edition, Addison-Wesley, 2000.

[14] A. Gupta, I.S. Mumick and V.S. Subrahmanian, Maintaining Views Incrementally, *SIGMOD Record 22*(2), 1993, 157-166.

[15] N.W. Paton and O. Díaz, Active Database Systems, *ACM Computing Surveys 31*(1), 1999, 63-103.

[16] F.D. Rolland, *The Essence of Databases*, Prentice Hall, 1998.

[17] M. Stonebraker, A. Jhingran, J. Goh and S. Potamianos, On Rules, Procedures, Caching and Views in Database Systems, *SIGMOD Record 19*(2), 1990, 281-290.

[18] J.H. ter Bekke, *Semantic Data Modeling*, Prentice Hall, 1992.

[19] J.H. ter Bekke, Meta Modeling for End User Computing, *Workshop Proc. 6th Int. Conf. on Database and Expert Systems Applications DEXA'95*, London, UK, N. Revell and A. Min Tjoa (eds.), 1995, 267-273.

[20] J.H. ter Bekke, Can We Rely on SQL?, *Proc. 8th Int. Workshop on Database and Expert Systems Applications DEXA'97*, Toulouse, France, R.R. Wagner (ed.), IEEE Computer Society, 1997, 378-383.

[21] J.H. ter Bekke, Advantages of a Compact Semantic Meta Model, in *Proc. Second IEEE Metadata Conf., Metadata 97*, Silver Spring, Maryland, USA, 1997. http://www.computer.org/conferen/proceed/meta97/list_papers.html

[22] J.H. ter Bekke, *Manual of the Xplain-DBMS, version 5.8*, Delft University of Technology, 1999 (in Dutch). http://is.twi.tudelft.nl/dbs/terBekke.html

[23] J.H. ter Bekke, Semantic requirements for databases in casual environments, *Proc. SAICSIT'99: Prepare for the New Milennium*, Johannesburg, South Africa, P. Machanick (ed.), Electronic extension of the *South African Computer Journal 24* (Nov. 1999). http://www.cs.wits.ac.za/~philip/SAICSIT/SAICSIT~99/papers_ideas.html

[24] J. Widom and S. Ceri (eds.), *Active Database Systems, Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann, 1996.