Update Propagation in Chimera, an Active DOOD Language*

Ulrike Griefahn Rainer Manthey
Department of Computer Science University of Bonn
Römerstr. 164
D-53117 Bonn, Germany
{ulrike,rainer}@informatik.uni-bonn.de

Abstract

Propagating updates is an important task to be performed within many database services such as integrity checking, maintenance of materialized views, and condition monitoring. This paper is concerned with the propagation of updates in an active DOOD language. The approach proposed is to make use of Chimera triggers for computing induced updates. It will be shown how a subset of Chimera’s deductive rules can be compiled to update propagation triggers. In its expressiveness the rule set considered corresponds to that of Datalog with sets and negation. Using triggers for implementing update propagation has the advantage that no special component has to be implemented as a trigger mechanism has to exist anyway. In this paper we will not propose new techniques for computing induced updates but will transfer the techniques — well-known for the relational model — to the object-oriented case.

1 Introduction

Update propagation is one of the key tasks a deductive database system is expected to automatically perform. To propagate an update basically means to determine all logical consequences which a physical update of some explicitly stored (base) data induces on any derived data. A single change of some base data may well affect various rules. Induced updates themselves may in turn affect rules too, so that the process of propagation constitutes a complex multi-stage inference process.

Classically, there are at least two reasons why induced updates have to be computed. On the one hand, certain derived data might be permanently materialized in view of enhancing efficiency of query evaluation. If so, each update affecting a materialized concept has to be propagated by incrementally modifying the materialized data according to the rule(s) involved. On the other hand, in presence of integrity constraints, knowledge about the implicit logical consequences of a physical update have to be computed in order to be able to check constraints involving derived concepts. There are other less central, but nevertheless important and interesting motivations for an update propagation facility. Materialization as well as integrity checking can very conveniently be used as mechanisms for implementing schema

*This work has been supported in part by the Commission of the European Community under ESPRIT project number 6333 (IDEA)
design and manipulation tools (e.g., in order to check model-inherent constraints defining correct application schemas). The analysis of potential relationships and conflicts between deductive rules, triggers, constraints, and methods in a complex application schema might result in the need to perform hypothetical updates on test data, that have to be propagated for analytical purposes as well.

Chimera, the language we deal with in this paper, is a novel database language which has been designed within the context of the ESPRIT project P6333 (IDEA), [CM93]. Chimera is based on a conceptual model which provides object-oriented modeling facilities. It includes data definition commands, declarative queries, procedural primitives for database manipulations, as well as deductive rules for the definition of derived data and constraints, and active rules for reactive processing.

Chimera’s object-oriented data model and data manipulation language introduce a couple of new and non-trivial problems with respect to formulating and implementing update propagation methods. The approach proposed in this paper is to make use of Chimera triggers for implementing update propagation. The motivation for doing so is twofold: on the one hand, each propagation step can be very naturally interpreted as being driven by an update to be propagated, and as causing the creation of a follow-up modification of a rule-defined concept. The specific conditions under which the propagation step is performed make up the condition part of a trigger. On the other hand, using a system-component which exists anyway — in this case a trigger interpreter — for implementing other services of the database system is likely to reduce size and complexity of the resulting overall system.

As far as we are aware, the work reported in this paper is the first approach to update propagation addressing the problem in the context of an object-oriented data model. Up till now, update propagation methods have been proposed for the relational model only, in particular in connection with integrity checking and view materialization. Our approach is independent of any such application as we are computing induced updates in a kind of preprocessing phase resulting in temporary, internal data. The result of update propagation may then be used in subsequent steps for other purposes. The only other approach we know, which makes use of active rules for propagating updates, is the one proposed by Ceri and Widom in the context of the Starburst production rule system, [CW90, CW91, CW92].

Other approaches to relational update propagation compile passive rules into passive propagation rules activated by means of queries, [Küc91, Oli91, VBK91, UO92]. The internal events method by Olivé and Urpi explicitly addresses attribute modifications rather than reducing them to combinations of deletions and insertions as Ceri-Widom do. By doing so, they are already tackling one of the new problems arising in an object-oriented context. A third class of relational methods implements update propagation by means of special interpreters (e.g., [Dec86, HD92]).

Our approach is closely related to all of the cited relational methods, in that we compute fully instantiated, “real” changes which have already been checked for effectiveness. We do not improve or modify the techniques underlying these methods, but transfer them to a trigger environment and adapt them to the object-oriented case.

It should be mentioned that several papers have been addressing a related, but different problem, namely update propagation for efficient condition monitoring in active databases (e.g., [RCB89, HD93]). They are propagating updates in order to support reactive processing, while our goal is to employ active rules for implementing update propagation.

Note, that in this paper update propagation is investigated for a subset of Chimera deductive rules. We are just dealing with the propagation of updates caused by derived
attributes, and additionally, we are restricting the syntax of passive rules to a form which is comparable with that of Datalog with sets and negation. This restriction is motivated as follows: Update propagation approaches based on the relational model are mainly formulated wrt. Datalog with negation. The aim of this paper is not to propose a new technique for update propagation, but is mainly concerned with the transfer of existing techniques to an object-oriented language. However, the extension to the full language of Chimera is one of the main future topics we want to deal with. Thereby we have to consider special problems, which arise in the context of object-oriented languages. For instance, in an object-oriented language the propagation of updates is not only necessary due to the presence of rules, but is also due to inheritance.

The paper is structured as follows. Section 2 gives a short overview of Chimera, specifies the Chimera subset we are dealing with, and presents the example used throughout this paper. Section 3 illustrates how the propagated update is to be represented in the database. Section 4 motivates the construction of update propagation triggers from passive rules by means of an example and Section 5 explains the trigger compilation in general. Finally, Section 6 briefly outlines open issues to be addressed in the future.

2 Overview of Chimera

As Chimera consists of an object-oriented data model, each real world entity should be represented as an individual object. Objects have to belong to an object class, and are associated with attributes, operations (i.e., methods) and constraints. Attributes are used to represent any information which is to be stored about a particular object. They are atomic or structured, i.e., they are associating objects with either single values or a set of values (of a fixed type). Operations are employed to change the state of an object (i.e., to modify the value of an attribute) and constraints are used to restrict the set of proper attribute values. Attributes, operations, and constraints may be associated with each instance of a class as well as with the entire class (class attributes, class operations, and class constraints). Furthermore, Chimera offers values which are used to describe objects. Values are simply representing themselves wherever they occur, and therefore cannot be associated with attributes and operations. They can either be atomic (i.e., any printable symbol or object identifier), or structured (i.e., sets, lists, and records), and may be organized by means of types and classes. User-defined value types are defined in order to associate application-specific names with certain value types. Value classes are user-defined collections of values and are used to restrict the extent of a user-defined value type.

Deductive rules (also called passive rules) are a key concept in Chimera. They are used to define the population of derived classes as well as to implement derived attributes and constraints. Furthermore, Chimera provides the possibility to define views in order to combine information from one or more classes. Views are implemented by means of deductive rules, too.

Active rules (also called triggers) provide the possibility to specify certain reactions in response to particular events such as database updates and queries. There are various motivations for integrating an active rule component into a database language: Many external applications (e.g., stockkeeping, air traffic control, . . .) can be naturally expressed by means of triggers, but they can also be usefully employed for internal applications (e.g., enforcing database integrity, maintaining derived data, . . .).
Triggers as well as constraints may be targeted to a particular class, but may also be defined independently of a particular class definition. Thus, like views, they may refer to one or more classes: constraints may relate information in more than one class, and triggers may react to events over different classes. The distinction between targeted and untargeted concepts is motivated only from the data modeling point of view as in an object-oriented context targeted concepts are easier to understand and should therefore be used as often as possible.

In the following we will not introduce the full syntax of Chimera, but restrict ourselves to the subset required in the remaining sections. A complete specification of Chimera can be found in [CM93]. As this paper is concerned with update propagation due to the presence of derived attributes the Chimera subset required includes essentially object classes, views, and triggers.

2.1 Object Classes and Views

The definition of an object class in Chimera is composed of two sections: signature and implementation. The signature introduces all names and types of the desired concepts. It consists of eight parts where each part may be omitted.

```
define object class  object class name
    superclasses (all direct superclasses of the newly defined class)
    attributes (describing the individual instances of the class)
    operations (changing the state of individual objects)
    constraints (restricting the values of attributes)
    c_attributes (associating values with the entire class)
    c_operations (changing the values of class attributes)
    c_constraints (restricting the extent of the class)
    triggers (specifying reactions to events over the class)
end
```

All concepts introduced in the signature have to be implemented. The population of a base class is implemented by individually creating objects of the class and thereby introducing values for the extensional attributes. However, the implementation of the other concepts can be done collectively for all objects of a class within the implementation section. This section consists of seven parts only, as triggers are defined in a separate trigger definition (cf. Section 2.3).

```
define implementation for  object class name
    population (passive rules defining the extent of a derived class)
    attributes (passive rules defining values of all derived attributes)
    operations (implementations defining all operations)
    constraints (passive rules defining all constraints)
    c_attributes (passive rules defining values of all derived class attributes)
    c_operations (implementations defining all class operations)
    c_constraints (passive rules defining all class constraints)
end
```

Views are independent from a particular class, and are defined by means of a separate view definition. Each view is given by a user-defined record type and implemented by a set of passive rules.
define view view name : record_of(l_1 : t_1, \ldots, l_n : t_n)

〈passive rules defining the view〉

end

where the l_i : t_i are denoting the components of the record, l_i is the label and t_i the type of the component.

2.2 Passive Rules

In order to introduce the syntax of passive rules we have first to define the notions of terms and formulas, from which passive rules are constructed.

Terms are either atomic or complex. Atomic terms are constants coming from the basic value types (boolean, char, integer, real, and string), object identifiers (oids), or variables.\footnote{Syntactically, variable names can be distinguished from other names as they are denoted by strings starting with a capital.} Complex terms are constructed from atomic terms and constructor as well as function symbols. Constructor terms are set terms of the form \{t_1, \ldots, t_n\}, list terms \[t_1, \ldots, t_n\], or records \(l_1 : t_1, \ldots, l_n : t_n\) where the \(t_i\) terms and the \(l_i\) are labels. Access to attribute values is performed via attribute terms of the form \(X.a\) where \(X\) is a variable and \(a\) is the name of an attribute. Attributes are considered as unary functions applied to a variable (or recursively to another attribute term, e.g. \(X.a.b\)) using a postfix dot notation.

(Declarative) formulas are either atomic or complex as well. Atomic formulas (also called positive literals) are constructed from arbitrary terms and predicate symbols. We distinguish three kinds of atomic formulas: class formulas, comparisons, and membership formulas. Class formulas are unary predicates of the form \(c(X)\) where \(c\) is a class name or the name of a basic value type and \(X\) is a variable. Comparisons are written in an infix notation \(t_1 \text{ op } t_2\) where \(t_i\) are terms and \(\text{ op }\) is one of the binary predicate symbols \(<, \leq, >, \geq, =\). Membership formulas \(t_1 \text{ in } t_2\) are constructed from the binary predicate symbol \(\text{ in}\) and two terms \(t_i\) where \(t_1\) is an arbitrary term and \(t_2\) is restricted to set and list terms. Atomic formulas may be negated by means of the prefix operator \(\text{ not}\). Negated atomic formulas are also called negative literals. Complex formulas are built from positive and negative literals and the connector “,” denoting the conjunction of the composed subformulas. Each formula is assumed to be range-restricted. i.e., it has to be guaranteed that the number of bindings satisfying the formula is always finite (wrt. any finite database).

Passive rules are expressions of the form \(\text{Head} \leftarrow \text{Body}\) where \(\text{Head}\) is an atomic formula and \(\text{Body}\) is an arbitrary formula. In the case a passive rule implements a derived attribute, the head of the rule is restricted to equalities and membership formulas where one of the terms is the respective attribute term. Note, that list-valued attribute cannot be defined by means of passive rules as the order of derived list elements cannot be specified within rule bodies.

We assume each passive rule to be safe, i.e., each variable of the rule occurs in at least one positive literal in the body. Furthermore, each rule set belonging to one schema has to be stratified wrt. to both sets and negation.

As we already pointed out in the introductory section, in this paper we consider a subset of Chimera passive rules to be compiled to update propagation triggers. In its expressiveness, the set of deductive rules defined above corresponds to Datalog with sets and negation. Deductive rules in Chimera may, additionally, refer to user-defined value types, include view
and constraint formulas, and the set of terms is augmented by selector and operator terms including arithmetic, set, list, and aggregate operators.

2.3 Active Rules

Like in most active database languages, Chimera triggers are based on the ECA-paradigm. Thus, the definition of triggers is essentially divided into three parts. The event part specifies all operations on which the trigger is to be activated. These operations are restricted to queries and primitive update operations predefined in Chimera: object creation, deletion, modification, migration, etc. The condition is a declarative formula (i.e., a conjunction of literals) which is evaluated in order to provide bindings for the variables in the action part. The action part is a sequence of procedure calls, including calls to predefined procedural primitives performing basic database manipulations. The following shows the syntax for defining a trigger in Chimera:

```
define trigger trigger name
  events (triggering operations)
  condition (declarative formula)
  actions (sequence of procedure calls)
end
```

Like for passive rules, there is one restriction imposed on the use of variables in triggers. Triggers are assumed to be safe as well, which in this context means that all variables occurring as input parameters in the action part must also be contained in some positive literal in the condition.

In comparison to declarative formulas allowed in passive rule bodies, the formulas specified in trigger conditions may additionally include two special kinds of atomic formulas.

To begin with, Chimera triggers are set-oriented, i.e., they are reacting to sets of individual database updates (possibly invoked by one set-oriented database operation). Therefore, the event specifications are only giving details of the triggering operation, the class, and in some cases the attribute on which the operation is performed. Possible event specifications are, e.g.,

```
create(scientist)
modify(scientist.name)
```

They do not contain any information about the actual parameters of the triggering updates. Thus, Chimera provides the possibility to refer to the triggering instances within the condition. This is performed by so-called event formulas which retrieve the object identifiers of all objects affected by the triggering operations. The event formula

```
holds(create(scientist),Scientist)
```

retrieves all newly created objects of the class scientist. Note that holds computes the net effect of event instances, i.e., it excludes all events which are compensated by follow-up modifications. For instance, the creation of an object followed by its deletion has no net effect, several modifications followed by a deletion are considered as a single deletion, and an object creation followed by several modifications is considered as the creation of the modified object.
Furthermore, trigger conditions may include references to the old state of the database. i.e., the state before the update was performed. Within the condition this is expressed by wrapping the respective expressions with the constructor **old**. The atomic formula

\[ X \text{ in } \text{old}(Y, \text{same_generation}) \]

e.g., checks whether the scientist \( X \) was known to be in the same generation as the object \( Y \) in the old state.

The semantics of Chimera triggers can informally be expressed as follows. Corresponding to the three parts discussed above trigger processing can be divided into three phases. A trigger is **activated** if one of its specified events occurs. The activation is performed immediately after the event has been determined. At a certain point in time one of the activated triggers is selected and its condition is evaluated. If the condition is satisfied, the action is executed for each of the variable bindings obtained during condition evaluation. These actions may be events in turn, and may cause further reactions to be triggered.

There are more options that can be defined for Chimera triggers. For instance, it can be specified when trigger processing (i.e., condition evaluation and action execution) is to be performed. Immediate triggers are processed immediately after the execution of the triggering operation while deferred triggers are processed at the end of the current transaction. Another example is that Chimera provides the possibility to specify priorities between triggers in order to control execution of different triggers which are to be processed at the same time. As these options are not yet important for the propagation of updates, we omit further details on this topic. A detailed description of Chimera triggers and their behavior can be found in [FPT94].

### 2.4 Running Example of this Paper

In order to demonstrate the propagation of updates in Chimera, we have chosen a peculiar family tree, the members of which are scientists. Each scientist is essentially described by his name, his supervisor, and the year of his graduation. These data are represented by means of three single-valued, extensional attributes. Additionally, two set-valued, derived attributes are defined. The first one associates with each scientist the set of his “scientific” descendants (which means those scientists he has supervised during their graduation, and those who have, in turn, been supervised by his descendants). The second is the set of all scientists who are in the same generation as the object under consideration. The signature definition of the object class **scientist** is illustrated below.

```datalog
define object class scientist
    attributes
        name : string(30),
        supervisor : scientist,
        year_of_grad : integer,
        same_generation : set_of(scientist) derived,
        descendants : set_of(scientist) derived
    end
```

Within the implementation definition the values of both derived attributes are specified by means of recursive rule definitions, which are well-known in the area of deductive databases. if expressed in Datalog syntax, but look quite different in Chimera.
define implementation for scientist
  attributes
    X in Self.same_generation <-
      scientist(X),
      X = Self;
    X in Self.same_generation <-
      scientist(X),
      Xs = X.supervisor,
      Ys = Self.supervisor,
      Ys in Xs.same_generation;
    X in Self.descendants <-
      scientist(X),
      X.supervisor=Self;
    X in Self.descendants <-
      scientist(Y),
      Y.supervisor=Self,
      X in Y.descendants
  end

The first passive rule for the same generation attribute states that each scientist is in the same generation as himself, while the second describes that two scientists are in the same generation if their supervisors are in the same generation. The first rule of descendants expresses that a scientist is a descendant of his own supervisor, while the second rule states that every descendant of a scientist is also a descendant of the scientist’s supervisor. Note, that the special variable Self implicitly ranges over all instances of the target class scientist.

Both derived attributes are remarkably well qualified to demonstrate how updates on the class scientist may cause changes of derived attributes. For instance, the creation of a new scientist causes the descendants attribute of his supervisor to change. The same applies to all his further ancestors. Moreover, the new scientist is added to the follow-up generation of his supervisor’s generation. Finally, the creation of a new object implies that all derived attributes of the new object itself have to be completely computed.

The deletion of a scientist causes the inverse modifications. The object has to be removed from the descendants and same_generation attributes of the scientists already affected by the insertion. As the object itself completely disappears from the database, there is no need to compute updates of its own derived attributes.

The change of a supervisor attribute, for example the replacement of a scientist’s supervisor causes more or less the twice number of follow-up modifications. On the one hand, the scientist is removed from the descendants attribute of all his former ancestors and also disappears from his generation. On the other hand, he is inserted as a descendant of his new supervisor and all his new ancestors as well as he becomes a member of the follow-up generation of the new supervisor’s generation.

Modifications of the remaining extensional attributes are irrelevant for the values of the derived attributes, e. g., an update of a scientist’s name causes no follow-up modifications on the state of other objects.
3 Delta Classes

Before we start describing the construction of update propagation triggers from deductive rules, we have to discuss how to represent the results of update propagation. As induced updates will not be performed on the database, but serve as a means for analyzing the consequences of an update only, we have to find a way how to protocol them. In order to perform update propagation by means of Chimera triggers we are forced to represent induced updates internally in terms of the Chimera data model. For this purpose we introduce so-called delta classes. For each operation, which can be applied by a user, (i.e., object creation and deletion, as well as attribute modification), and for each class and each attribute a delta class is created.

The delta class for object creation of a class c is called created_c, the one for object deletion deleted_c. For modifications of attributes we distinguish three different kinds of delta classes, one for the update of single-valued attributes a named changed_c_a, and two for modifications of multi-valued attributes a, called inserted_c_a and removed_c_a. A set-valued, derived attribute is called multi-valued, if it is implemented by passive rules whose heads are membership formulas. In this case we are able to have a closer look on the update, namely on the value inserted to or deleted from the set-valued attribute.

Let us first consider the structure of delta classes which are representing object creations or deletions. Objects of such delta classes include all relevant information about a newly created or a deleted object, in both cases the identifier of the new object.

```plaintext
define object class created_c
  attributes
  object : c
end

define object class deleted_c
  attributes
  object : c
end
```

As an example consider the creation of a new scientist with object identifier O1. This update will be “protocollled” by creating another object O2 of the class created_scientist. The value of the object attribute of O2 is O1.

The delta classes for modifications of single-valued attributes are also defined with one attribute describing the modified object. The following class definition illustrates the structure of these classes.

```plaintext
define object class changed_c_a
  attributes
  object : c
end
```

However, for multi-valued attributes we introduce two delta classes: one representing the insertion of new values and one representing the deletion of old values. Thus, both delta classes are defined with two attributes describing the modified object and the inserted resp. the deleted value.
define object class inserted_c_a
  attributes
    object : c
    value  : (type of the elements of a)
end

define object class removed_c_a
  attributes
    object : c
    value  : (type of the elements of a)
end

The reason for introducing delta classes for each class and each attribute individually is that
Chimera does not provide the possibility to abstract from the type of attributes. If Chimera
would be augmented in order to define generic classes we could do with only five delta classes
representing updates independently from a particular class and attribute.

Delta classes are intended to contain complete information about all explicit and implicit
changes of the database. For this reason, the updates actually performed by the user have
to be entered into the resp. delta classes as well. This is automatically done by means of
so-called initial update propagation triggers. With respect to the class scientist of Section
2.4, e. g., we obtain the following five such triggers. (The operation create_tmp is one of the
Chimera update primitives that creates temporarily existing objects).

events  create(scientist)
condition holds(create(scientist),01)
actions  create_tmp(created_scientist, (object:01), 02)

events  delete(scientist)
condition holds(delete(scientist),01)
actions  create_tmp(deleted_scientist, (object:01), 02)

events  modify(scientist.name)
condition holds(modify(scientist.name,01))
actions  create_tmp(changed_scientist_name, (object:01), 02)

events  modify(scientist.supervisor)
condition holds(modify(scientist.supervisor,01))
actions  create_tmp(changed_scientist_supervisor, (object:01), 02)

events  modify(scientist.year_of_grad)
condition holds(modify(scientist.year_of_grad,01))
actions  create_tmp(changed_scientist_year_of_grad, (object:01), 02)

4 Update Propagation by Means of Triggers: An Example

In this section we are giving an informal impression of how passive rules can be compiled
into update propagation triggers. We illustrate the derivation of such a trigger from one
particular rule by means of an example, namely the second rule defining the \texttt{descendants} attribute (cf. Section 2.4).

\begin{verbatim}
X in Self.descendants <-
  scientist(Self),
  scientist(Y),
  Y.supervisor=Self,
  X in Y.descendants
\end{verbatim}

Note, that in a preparation step of the rule compilation process the rule body was augmented with the class formula \texttt{scientist(Self)}. This modification is necessary because — opposed to the deductive rule — the resulting update propagation triggers will not be targeted to the class of the derived attributes. This implies that in the following the special variable \texttt{Self} has to be considered as any ordinary variable like \texttt{X} or \texttt{Y}. We only keep the notation \texttt{Self} in order to make the individual compilation steps more transparent. The reason why the resulting triggers are not targeted to the same class as the respective deductive rule will become clear when we are discussing the derivation of events (cf. Section 5.1).

In order to get an impression of the relationship between a passive rule and the propagation triggers derived from it, let us consider the following trigger computed from the recursive descendants rule.

\begin{verbatim}
events  create_tmp(deleted_scientist)
condition holds(create_tmp(deleted_scientist),E),
  E.object = Y,
  old(scientist(Self)),
  old(scientist(Y)),
  old(Y.supervisor)=Self,
  X in old(Y.descendants),
  not X in Self.descendants,
  not v_removed_scientist_descendants((Self,X))
actions  create_tmp(removed_scientist_descendants,
  (object:Self,value:X),
  Delta)
\end{verbatim}

For each body literal of a passive rule there may be several database updates that affect the evaluation of the rule. For instance, a class formula is influenced if an object of the referenced class has been created or deleted. The above example trigger is derived from the class formula \texttt{scientist(Y)} in the body of the descendants rule. The affecting update is the deletion of an object from the class \texttt{scientist}. This deletion is represented in the delta class \texttt{deleted_scientist}. Thus, the triggering event of the corresponding propagation trigger is the creation of an object in this delta class rather than the deletion itself.

As event specifications in Chimera do not contain any information about the actual parameters of an event, we have to retrieve these parameters within the condition part. Therefore the condition has to include the following literals.

\begin{verbatim}
holds(create_tmp(deleted_scientist),E),
E.object = Y
\end{verbatim}

By means of the event formula \texttt{holds(create_tmp(deleted_scientist),E)} we obtain the identifier of the created delta object \texttt{E}. The second formula accesses the identifier \texttt{Y} of the deleted object.
Next, we have to check whether in the old database state (before executing the deletion) the deleted object Y was involved in any derivation of some descendants value. In order to do so, the body of the passive rule has to be included in the condition as well, whereby the respective expressions have to be wrapped by means of the old constructor.

\[
\text{old(scientist}(\text{Self})) , \\
\text{old(scientist}(Y))) , \\
\text{old}(Y.\text{supervisor})=\text{Self} , \\
X \text{ in old}(Y.\text{descendants}) ,
\]

In case of a deletion it is not sufficient to identify just one derivation disabled by the update, but we have to make sure that no other derivation path remains in the new state. Thus, the condition part has to be augmented by the literal

\[
\text{not}(X \text{ in Self.descendants})
\]

which is derived from the head of the underlying deductive rule and has to be evaluated in the changed database state.

If there is no derivation path in the new state, the identified induced update is known to be effective. However, it is possible that this implicit update was already detected during an earlier trigger execution. Thus, we have to check whether there exists an object in the resp. delta class that describes the same update. As Chimera does not provide the possibility to express negative existential formulas we have to perform this duplicate check by means of a view which is defined as follows:

```ldif
define view v_removed_scientist_descendants:
    record_of(object:scientist,value:scientist)
  v_removed_scientist_descendants((Self,X)) <-
    removed_scientist_descendants(0).
  0.object=Self,
  0.value=X
end
```

Therefore, the condition of the trigger can be completed by the following literal realizing the duplicate check.

\[
\text{not}(v_{\text{removed_scientist_descendants}}((\text{Self},X)))
\]

The action part of an update propagation trigger always corresponds to the head of the respective passive rule. In our example trigger the action part looks as follows:

```ldif
create_tmp(removed_scientist_descendants,
  (object:Self,value:X),
  Delta)
```

The update induced is the modification of the descendants attribute value of the object represented by Self. Thus, we create an object of the delta class removed_scientist_descendants that represents this induced update. The variables of the head literal are associated with the newly created delta object by means of attribute values. The target class of the passive rule and the attribute name correspond to the name of the respective delta class.
Of course, the removal of a descendant may cause some further induced updates. Therefore, we also need a trigger that reacts to the creation of a `removed_scientist_descendants` object. This trigger can be derived from the passive rule in a similar way based on the fourth body literal X in Y.descendants and looks as follows:

```plaintext
events create_tmp(removed_scientist_descendants)
condition holds(create_tmp(removed_scientist_descendants), E),
    E.object = Y,
    E.value = X,
    old(scientist(Self)),
    old(scientist(Y)),
    old(Y.supervisor) = Self,
    X in old(Y.descendants),
    not X in Self.descendants.
    not v_removed_scientist_descendants((Self,X))

actions create_tmp(removed_scientist_descendants,
    (object:Self,value:X),
    Delta)
```

Analogously, for each body literal in the passive rule and for each (induced) update affecting this literal a corresponding trigger can be derived. Altogether, 24 update propagation triggers are obtained from the four passive rules of the example schema. The discussion of the technical details of trigger compilation is the task of the following section.

5 Trigger Construction in General

In the preceding section we have motivated the construction of update propagation triggers from passive rules by means of an example. In the following we are going to explain the trigger construction in general.

In order to simplify the description of the single transformation steps we initially omit negation in rule bodies\(^2\) and assume the passive rules to be in a normalized form. First of all, this normalization requires that attribute terms are restricted to be of length 1, i. e., a literal like X.a.b = Z has to be unfolded to X.a = Y, Y.b = Z. Secondly, it demands that attribute terms are only allowed in membership formulas of the form X in Y.a and equalities of the form X.a = Y. Note, that these requirements do not mean a real restriction as each passive rule can be equivalently transformed into a normalized form. As we have already discussed in Section 4 the body of the passive rule has to be augmented by the class formula \(c_t(Self)\) where \(c_t\) denotes the target class of the passive rule. The addition of this class formula does not change the semantics of the deductive rule as well.

In the following three paragraphs the construction of update propagation triggers is described considering separately the derivation of the single parts composing a trigger. The steps are performed for each passive rule implementing a derived attribute and for each literal occurring in the rule body. Figure 1 shows a pseudo-code procedure which illustrates how the single derivations are connected with each other.

\(^2\)The transformation will be extended to negation in Section 5.4.
for each passive rule \( R \equiv L_0 \leftarrow L_1, \ldots, L_n \)
do for each atomic formula \( L \in \{L_1, \ldots, L_n\} \)
do \( E := \{\text{the set of all events affecting } L\} \)
for each event \( E \in E \)
do \( C := \{\text{the set of all condition parts wrt. } L, E, \text{ and } L_0\} \)
for each condition part \( C \in C \)
do \( A := \{\text{the action wrt. } C\} \)
\( T_{(R,L,E,C)} := \{\text{trigger composed of } E, C, \text{ and } A\} \)
end
end
end

Figure 1: Trigger compilation procedure

5.1 Deriving the Affecting Events

The construction of update propagation triggers starts by computing the set \( E \) of all events possibly affecting the evaluation of \( L \), and therefore possibly changing the derived attribute value. We distinguish the following cases depending on the form of the selected literal \( L \):

1. Let \( L \equiv c(X) \) be a class formula.

   (a) If \( c \) is the name of a basic value type, the evaluation of \( L \) cannot be affected as basic value types cannot be changed by any database update.

   \[
   E := \emptyset
   \]

   (b) If \( L \) is the special class formula \( c_l(\text{Self}) \) where \( c_l \) is the target class of the passive rule, we obtain one event indicating an object creation for the target class \( c_l \).

   \[
   E := \{\text{create}_\text{tmp}(\text{created}_{c_l})\}
   \]

   Creating objects of the target class implies that all derived attributes of the newly created object have to be entirely computed. The deletion of such an object, however, is not considered as the object disappears completely from the database and there is no need to compute the changes of its own derived attributes.

   (c) If \( L \) is any other class formula referring to an object class \( c \), the creation as well as the deletion of objects of the class \( c \) are relevant for the evaluation of \( L \).

   \[
   E := \{\text{create}_\text{tmp}(\text{created}_c), \text{create}_\text{tmp}(\text{deleted}_c)\}
   \]

2. Let \( L \) be an atomic formula containing an attribute term \( X.a \).

   (a) If \( a \) is a single-valued attribute, any modification of this attribute may cause the change of the derived attribute value. Thus, we yield one event referring to arbitrary modifications of \( a \).

   \[
   E := \{\text{create}_\text{tmp}(\text{changed}_{c.a})\}
   \]

   where \( c \) is the class in which the attribute \( a \) is defined.
(b) If \( a \) is a multi-valued attribute, the insertion of values to, as well as the deletion of values from \( a \) may be relevant for the evaluation of the passive rule.

\[
\mathcal{E} := \{\text{create tmp(inserted}_c_a), \text{create tmp(removed}_c_a)\}
\]

where \( c \) is the class in which the attribute \( a \) is defined.

3. Let \( L \) be any other atomic formula. As these do not refer to classes or attributes, they can be evaluated independently from the underlying database. In this case, no relevant events are obtained.

\[
\mathcal{E} := \emptyset
\]

Note, if the event set \( \mathcal{E} \) is empty, no update propagation triggers can be derived wrt. the selected atomic formula and the transformation stops at this point.

We are now able to explain why update propagation triggers will not be targeted to the same class as the underlying passive rule. Passive rules implementing derived attributes are always targeted to the class the attribute is defined for. A trigger, however, can be targeted to one particular class if its triggering events refer to this class only. As the event of an update propagation trigger is derived from one of the literals occurring in the body of the passive rule, the target class of the trigger would have to correspond to the class referenced in this body literal. This might, of course, not be same as the one of the passive rule. For this reason we specify update propagation triggers as untargeted triggers which are independent of a particular class.

### 5.2 Deriving the Condition Part

The condition part of an update propagation trigger always consists of the following four sections.

1. The relevant parameters of the event are retrieved.
2. The body of the passive rule is included in order to determine additional or removed derivation paths.
3. The effectiveness of the identified induced update has to be checked.
4. It has to be checked whether the update has already been detected during an earlier trigger execution.

In the following the individual condition sections \( C_1, \ldots, C_4 \) are derived in the order given by the enumeration above. The set \( C \) of all conditions parts is, finally, obtained as the set of all conjunctions that can be composed of the different sections.

#### 5.2.1 Retrieving the Relevant Event Parameters

The task of the condition is to check whether the triggering update induces further updates on the derived attribute. It is clear that changes of the derived value can only be caused by the given update. Therefore, the first section of the condition is to retrieve the identifier of the modified object in order to restrict the evaluation of the condition to the triggering update. If we are considering a multi-valued attribute occurring in a membership formula, then the value inserted to resp. deleted from the set-valued attribute is retrieved as well.
1. Let $L$ be a class formula $c(X)$ or an equality containing the attribute term $X.a$.

   $C_1 := \text{holds}(E,D), D.\text{object}=X$

2. Let $L$ be a membership formula $Y$ in $X.a$.

   (a) If $a$ is a single-valued attribute, then

   $C_1 := \text{holds}(E,D), D.\text{object}=X$

   (b) If $a$ is a multi-valued attribute, then

   $C_1 := \text{holds}(E,D), D.\text{object}=X, D.\text{value}=Y$

### 5.2.2 Computing the Induced Update and Checking its Effectiveness

As we are currently considering passive rules without negation, the creation of new objects may add new derivations only while the deletion may remove some. The same statement holds for the insertion of values to and their deletion from multi-valued attributes. The modification of single-valued attributes, which can be considered as a deletion of the old attribute value followed by an insertion of the new value, may cause the addition of derivations as well as the deletion.

The new occurrence of a derivation can be discovered by restricting the rule body to the triggering update and checking whether it holds on the new database state. All bindings satisfying the restricted body lead to an induced update, whose effectiveness has to be proved by checking whether it is not derivable in the old state. The effectiveness test is necessary, as attribute values can possibly be computed via more than one derivation path and, therefore, may be already derivable in the old state. The deletion of a derivation is detected analogously. First it is checked whether the modified object leads to an derivation in the old state. Afterwards, the effectiveness test is performed on the new database state in order to check whether the values derived in the old state are absent in the new state.

1. If $E$ indicates the insertion of an object or a new value, i. e.,

   \[
   E \equiv \text{create}\_\text{tmp}(\text{created},c) \quad \text{or} \quad E \equiv \text{create}\_\text{tmp}(\text{inserted},c,a)
   \]

   then it has to be checked whether the update causes a new derivation and whether the induced update is effective.

   \[
   U_3 := L_1, \ldots, L_n
   \]

   \[
   C_3 := \text{not}\ \text{old}(L_0)
   \]

   where in $C_3$ the respective expressions are wrapped by means of $\text{old}$.

2. Let $E$ indicate the deletion of an object or of some value, i. e.,

   \[
   E \equiv \text{create}\_\text{tmp}(\text{deleted},c) \quad \text{or} \quad E \equiv \text{create}\_\text{tmp}(\text{removed},c,a)\]

   \[
   U_3 := L_1, \ldots, L_n
   \]

   \[
   C_3 := \text{not}\ \text{old}(L_0)
   \]
then it has to be checked whether the update causes the deletion of a derivation and whether the induced update is effective.

\[ C_2 := \text{old}(L_1), \ldots, \text{old}(L_n) \]
\[ C_3 := \neg L_0 \]

where in \( C_2 \) the respective expressions are wrapped by means of \text{old}.

3. Let \( E \) indicate the modification of a single-valued attribute, i.e.,

\[ E' \equiv \text{create}\_\text{tmp}(\text{changed}_c.a) \]

then the conditions to be created depend on the structure of the passive rule head \( L_0 \). As the modification of a single-valued attribute may add new derivations as well as remove old ones, both cases should be considered by an update propagation trigger.

However, constructing two conditions (and thus two update propagation triggers) is only sensible, if new derivations as well as removed ones can be represented within the delta classes. This is, of course, possible for multi-valued attributes. If the passive rule implements a single-valued attribute, we are just interested in the identifier of the modified object. The old and new values are not represented in the resp. delta classes as they can be retrieved from the database.

Thus, depending on the head \( L_0 \) we obtain the following conditions:

(a) If the passive rule implements a single-valued attribute, i.e., \( L_0 \equiv \text{Self}.b = Z \), then

\[ C_2 := L_1, \ldots, L_n \]
\[ C_3 := \neg \text{old}(L_0) \]

(b) If the passive rule implements a multi-valued attribute, i.e., \( L_0 \equiv \text{Z in Self}.b \), we obtain two different pairs of \( C_2 \) and \( C_3 \), one for detecting new derivations:

\[ C_2 := L_1, \ldots, L_n \]
\[ C_3 := \neg \text{old}(L_0) \]

and one for detecting removed ones:

\[ C_3 := \text{old}(L_1), \ldots, \text{old}(L_n) \]
\[ C_3 := \neg L_0 \]

5.2.3 Deriving the Duplicate Check

Before the identified induced update is protocollled in the respective delta class, it has to be checked if this update was already detected by an earlier trigger execution. Thus, we have to look whether the delta class holds an object describing the same update. This is performed with the help of a view that allows us to fade out the identifier of the delta object. For each delta class we define a view\(^3\) that retrieves all attributes of the resp. delta class. The structure of these views looks as follows:

\(^3\)These views are created at compile time, just like the delta classes and the update propagation triggers.
define view v_changed_c_a : record_of(object:c)
  v_changed_c_a((Self)) <-
  changed_c_a(0),
  0.object=Self
end

define view v_inserted_c_a : record_of(object:c,value:X)
  v_inserted_c_a((Self,X)) <-
  inserted_c_a(0),
  U.object=Self,
  U.value=X
end

Views on the delta classes removed_c_a are defined analogously to the views inserted_c_a. Note, that we do not consider views concerning object creations and deletions as induced changes, we are dealing with in this paper, are only attribute modifications.

Thus, for each condition constructed so far we have to generate the duplicate check as follows:

1. Let $L_0 \equiv \text{Self}.b = Z$
   
   $C_4 := \text{not } v_{\text{changed_c}_t}_b((\text{Self}))$

   where $c_t$ denotes the type of the variable $\text{Self}$, i. e., the target class of the passive rule.

2. Let $L_0 \equiv Z$ in $\text{Self}.b$.
   
   (a) If a new value for the attribute $b$ is derived, i. e., $C$ includes the literal not old($L_0$), then
   
   $C_4 := \text{not } v_{\text{inserted_c}_t}_b((\text{Self},Z))$

   (b) If a deleted value for the attribute $b$ is derived, i. e., $C$ includes the literal not $L_0$, then
   
   $C_4 := \text{not } v_{\text{removed_c}_t}_b((\text{Self},Z))$

5.3 Deriving the Action

The action $A$ of an update propagation trigger corresponds one-to-one with the duplicate check occurring in the condition. Thus, the action can easily be derived from this atomic formula.

1. If $C$ includes the literal not $v_{\text{changed_c}_t}_b((\text{Self}))$, then
   
   $A := \text{create_tmp}(\text{changed}_c_t_b, (\text{object:Self}), \text{Delta})$

2. If $C$ includes the literal not $v_{\text{inserted_c}_t}_b((\text{Self},Z))$, then
   
   $A := \text{create_tmp}(\text{inserted}_c_t_b, (\text{object:Self}, \text{value:Z}), \text{Delta})$

3. If $C$ includes the literal not $v_{\text{removed_c}_t}_b((\text{Self},Z))$, then
   
   $A := \text{create_tmp}(\text{removed}_c_t_b, (\text{object:Self}, \text{value:Z}), \text{Delta})$
5.4 Negation in Passive Rules

Up till now, we have only considered the compilation of passive rules without negation in rule bodies. In the positive case we have observed that any addition of new objects or values possibly causes new derivations, but never removes any. Analogously, any deletion only causes further deletions. However, if we are considering negative literals we observe the inverse effect. Any new value may only remove derivations while any deleted value possibly adds new derivation paths.

Thus, the main changes of the transformation specified so far concerns the steps introduced in Section 5.2.2. The other steps essentially remain unchanged; those depending on the structure of the selected literal $L$ can simply ignore the negation. The steps listed below are added to those presented in Section 5.2.2. In the following we assume that $L \equiv \text{not } A$ is the selected literal.

1. If $E$ indicates the insertion of an object or a new value, i.e.,

$$E \equiv \text{create}_\text{tmp}(\text{created}_c) \quad \text{or} \quad E' \equiv \text{create}_\text{tmp}(\text{inserted}_c \_ a)$$

then it has to be checked whether the update causes the deletion of a derivation and whether the induced update is effective.

$$C_2 \ := \ \text{old}(L_1), \ldots, \text{old}(L_n)$$

$$C_3 \ := \ \text{not } L_0$$

2. Let $E$ indicate the deletion of an object or of some value, i.e.,

$$E \equiv \text{create}_\text{tmp}(\text{deleted}_c) \quad \text{or} \quad E' \equiv \text{create}_\text{tmp}(\text{removed}_c \_ a)$$

then it has to be checked whether the update causes a new derivation and whether the induced update is effective.

$$C_2 \ := \ L_1, \ldots, L_n$$

$$C_3 \ := \ \text{not old}(L_0)$$

Note, that the steps to be performed wrt. modifications of single-valued attributes occurring in negative literals are the same as for positive literals. The reason for this is, that in both cases the addition as well as the deletion of derivation paths is considered.

As it is well-known, fixpoint semantics applied to stratified rule sets with negation may lead to wrong results, if all rules are considered simultaneously. In order to overcome this problem, the rules have to be applied in a particular order which is given by the stratification of the rule sets.

Obviously, the same problems arise for update propagation approaches, too, if data are to be considered during condition evaluation which are not yet completely derived. However, for the approach presented in this paper, the order of trigger application is irrelevant. The reason is that the computation of induced changes does not (really) depend on the delta classes. During the condition evaluation we perform only two accesses to the delta classes.
The first one retrieves the parameters of the triggering update which are used to restrict the evaluation of the rule body. This is not problematic as the effectiveness of induced updates is always verified wrt. the new database state. The second access is performed during the duplicate check. Depending on the result of the duplicate check, a new delta object is created or not. However, the induced updates described by the resp. delta class are always the same independently of the order in which the triggers are applied. If we would change the order, only the value of the object identifier will be different. But this causes no problems for the propagation of updates, as identifiers of delta objects are not considered during the duplicate check via views.

Problems with stratification will arise, if we employ delta classes within condition evaluation in order to gain more efficiency. Furthermore, if the materialization of derived attributes is intended, the delta classes (in this case the partially materialized attributes) should be applied for condition evaluation. The authors of [CW92] overcome this problem by specifying priorities between triggers (corresponding to the strata of the original rule set) in order to control the sequence of trigger executions.

6 Conclusion

In this paper we present our first results on update propagation in an active DOOD language. Our approach makes use of Chimera triggers for computing induced database changes. Propagating updates by means of triggers has already been proposed by [CW91, CW92] for the relational data model, but is new in the context of object-oriented languages. Using triggers for computing induced updates has the advantage that no new component has to be implemented as the already existing trigger mechanism can be employed for this purpose. Our main topic is to illustrate how passive rules (implementing derived attributes) are compiled into update propagation triggers. Thereby, we do not propose new techniques for computing induced updates, but transfer techniques, well-known for the relational model, to an object-oriented language.

Performing update propagation in an object-oriented context leads to a number of new problems not encountered in this way in a relational framework. First, the notion of an update is more complex in an OO model: objects may be created and deleted, attributes modified in various ways, values may be added and deleted. In contrast, there are only insertions and deletions of tuples in a relational database. Consequently, the notion of an induced update is more complex as well, including modifications of views and constraints not yet addressed in this paper.

If using Chimera triggers for performing update propagation, we are forced to represent induced updates internally in terms of the Chimera data model as well. In this paper, we have chosen to create temporary classes for each kind of induced update. However, this solution is by no means optimal, but rather straightforward and has to be improved and extended. One possible solution could be an extension of Chimera that allows to define generic classes.

Using an OO data manipulation language for representing propagation conditions leads to rather complex expressions resulting from the functional style of Chimera’s CL language. The particular form of event syntax used in Chimera increases complexity due to the need to access event parameters by means of individual subformulas in the condition part of an update propagation trigger. Various optimizations have to be exploited if evaluating such complex conditions over the database efficiently.
The update propagation method requires to evaluate subqueries over both, the state of the database before and after the update. Only one of both states can really be stored. Evaluation over the other state has always to be simulated by additionally considering the given updates. One of the future topics we are planning to investigate is how delta classes can be directly employed in order to improve the evaluation over the simulated state.

The proposed method suffers from one special drawback, which by the way all methods proposed for update propagation till now are sharing. The propagation steps performed for an individual deductive rule are derived from the syntactic structure of the rule only, without any reference to additional characteristics of the application likely to influence efficiency of the propagation process. Many parameters could be taken into consideration here, such as particular integrity constraints, or information about the actual (or at least the estimated) size and distribution of data in the classes addressed, or about the selectivity of the attributes involved. Each of these parameters might lead to a more flexible, better tuned propagation approach, potentially resulting in a considerable gain in performance. Up till now, there is hardly any experience with this kind of more elaborate update propagation methods around. This issue is subject to further research and will be addressed soon.

Another open problem concerns the interaction between update propagation and query evaluation components of the system. Each propagation step involves the evaluation of a query — the condition part of the respective propagation trigger. Thus, a rather complex and frequent interaction between different kinds of inference processes is to be expected, likely to suffer from inefficiency if not very well coordinated. One way out seems to be the attempt to implement query evaluation methods by means of triggers as well, thus leading to a uniform trigger process seamlessly blending propagation and evaluation steps. Whether such an approach is feasible and satisfactory is an open issue as well that might become relevant within the future work of our team. The method presented in this paper has been implemented within the context of the IDEA project based on the ECRC Common Logic Programming System ECLIPS [ECL93], a persistent Prolog system.

References


