Software Architecture

OCL2Trigger: Deriving active mechanisms for relational databases using Model-Driven Architecture [☆]

Harith T. Al-Jumaily [*], Dolores Cuadra, Paloma Martínez

Computer Science Department, Carlos III University of Madrid, Av. Universidad 30, Leganés, 28911 Madrid, Spain

Keywords:
Integrity constraints
Model-Driven Architecture
CASE tools
Active database systems
OCL
Sequence diagrams

A B S T R A C T

Transforming integrity constraints into active rules or triggers for verifying database consistency produces a serious and complex problem related to real time behaviour that must be considered for any implementation. Our main contribution to this work is to provide a complete approach for deriving the active mechanisms for Relational Databases from the specification of the integrity constraints by using OCL. This approach is designed in accordance with the MDA approach which consists of transforming the specified OCL clauses into a class diagram into SQL:2003 standard triggers, then transforming the standard triggers into target DBMS triggers. We believe that developing triggers and plugging them into a given model is insufficient because the behaviour of such triggers is invisible to the developers, and therefore not controllable. For this reason, a DBMS trigger verification model is used in our approach, in order to ensure the termination of trigger execution. Our approach is implemented as an add in tool in Rational Rose called OCL2Trigger.

1. Introduction

The introduction of the MDA approach (Model Driven Architecture) (OMG, 2007) in Software Engineering has provided a good support and consolidation for the automatic generation of code for application development. MDA focuses on using models as approaches to cover the life cycle of software development. The heterogeneity and interoperability between systems with different implementation platforms are resolved by using this approach.

In the context of databases, we believe that MDA is very ambitious because we find that when developing databases, conceptual models used in methodologies such as Elmasri and Navathe (2000) and Teorey (1999) are more efficient in expressing the semantics of the real world. Nevertheless, in the transformation of a conceptual model to a logical model, a development problem arises because conceptual elements do not have similar logical elements, that is, the semantics associated to one conceptual element cannot correspond to one relational element. Multiplicity constraints are a clear example of this.

As part of the solution to the development problem, a Computer Assisted Software Engineering (CASE) tool is used to support software design practices (Budgen and Thomson, 2003). In this work, we are interested in database development CASE tools such as Objecteering/SQL (2007) and Rational Enterprise Edition (2003). These platforms try to help database developers in different phases of design. Nevertheless, these tools are frequently simple graphical interfaces and do not completely carry out the design methodology that they are supposed to support.

Therefore, in our approach, the MDA approach has been adapted to implement the OCL2Trigger tool, which is used to enhance the transformation rules of the conceptual into the relational schema. The Relational model is considered because most database methodologies agree that it is useful for transforming the conceptual into a logical schema. The OCL2Trigger is plugged into the Rational Rose CASE tool, and can automatically transform the OCL/UML constraints into target DBMS triggers.

Although OCL is an object constraint language, we are interested in converting these constraints into relational database language as many organizations still use this type of database system. In addition, most important commercial object oriented database systems, such as ORACLE use relational tables for saving their objects. Thus, we believe that it is worth doing more work related to relational databases, especially in the context of the active mechanisms because in order to achieve best performance, these mechanisms are implemented as part of the database schema rather than in the application. Additionally, embedding integrity constraints in the database schema rather than in external applications is better for preserving logical data independence (Türker and Gertz, 2000).

☆ This work is part of the "*Software Process Management Platform: Modelling, reuse and measurement*". TIN2004/07083 project.

* Corresponding author.
E-mail addresses: haljumai@inf.uc3m.es (H.T. Al-Jumaily), dcuadra@inf.uc3m.es (D. Cuadra), pmf@inf.uc3m.es (P. Martínez).

Nevertheless, because trigger implementation is more complicated than procedural implementation, we have detected that transforming integrity constraints into triggers is insufficient because the behaviour of triggers is invisible and needs to be verified. Therefore, besides the transformation of integrity constraints into triggers, our OCL2Triggers tool creates UML sequence diagrams to verify the interaction of these triggers with themselves and with the other elements in the schema. These contributions make our approach useful, practical and intuitive in managing triggers.

The trigger system is specified according to the recent SQL:2003 standard that revises all parts of SQL99 and incorporates new features (ISO/IEC 9075 Standard, 2003). A trigger is a named event condition action rule that is activated by a transition in the database state. Every trigger is associated with a table and is executed whenever an event occurs to modify that table. An event is a DML statement (Delete, Insert, and Update (Attribute)) issued against a table. Once a trigger is activated and its condition is evaluated as true, the predefined actions are automatically executed. In this work, we denote the triggers of the SQL:2003 standard as Trigger:2003.

The rest of this work is organized as follows: in Section 2, related works are presented; in Section 3, the MDA adaptation for our approach is explained. Section 4 explains how the OCL2Trigger tool is designed and implemented. In Section 5, some conclusions and areas for future development are presented.

2. Related work

Different approaches have been applied to transform integrity constraints into active rules. Some of these approaches reject updates when the violation occurs, and the initial state before updates is restored (Decker et al., 2006). Another approach is that in which inconsistency states are detected first but consistency is restored by issuing corrective actions that depend on the particular constraint violation. Such works are Ceri and Widom (1990) and Ceri et al. (1994) in relational, and Ceri and Fraternalli (1997) in object oriented databases.

In some works, such as Ceri and Widom (1990), a general framework is described for transforming constraints into active rules for constraint maintenance. They define a general language for expressing integrity constraints, and transformation rules are used to convert integrity constraints into active rules. The contribution in Türker and Gertz (2000) is very important in our work because they issued some simple rules that are independent of a particular commercial system. These rules implement triggers based on constraint specifications by using Tuple Relational Calculus (Elmasri and Navathe, 2000). In this work, we will specify the transformation rules based on constraint specifications by using OCL. Our approach agrees with the proposal in Olivé (2003) that introduced OCL as a method for facilitating the definition of integrity constraints in conceptual schemas as constraint operations. In this proposal, constraints have been introduced in the conceptual schema as operations without defining any rules to specify the transformation of these operations into a logical schema.

On the other hand, a trigger is a SQL procedure that is automatically invoked by the DBMS to respond a specified event. During the invocation process, a trigger can cascade the activation of one or more other triggers, including itself. The final database state should not depend on the order in which these triggers are chosen for the execution. This problem becomes more complicated when users need to add new triggers to existing ones because adding new triggers may produce undesired and unexpected behaviour. Therefore, when working with triggers we always need to verify trigger execution. The verification of trigger execution is a major problem that makes application development a difficult task, i.e. database developers need to make an additional effort to verify the behaviour of triggers. The objective of this verification is to guarantee the termination and the confluence of trigger execution (Paton and Díaz, 1999).

Termination means that the execution of any set of active rules must terminate. This is needed to avoid cycling in the execution. A set of activated rules is confluent if the final state of the database is independent of the order in which activated rules are executed (Baralis and Widom, 2000). The non termination state is a major problem that produces an error causing the execution of the transaction to abort.

Many works have been done in the area of static termination analysis. Most of these works such as Paton and Díaz (1999) use the concept of a Triggering Graph (TG) as approach to detect the non termination state. The triggering graph was introduced in Baralis et al. (1993) to detect the non termination state of a set of activated rules. The termination analyses themselves focus on identification and elimination rules, which cause infinite execution in a triggering graph (Hickey, 2000). Redefining this rule and reconstructing the triggering graph is a good solution for verifying the termination state of the set of activated rules.

In the last decade, diverse efforts have been developed to resolve the problem of database development. One of these efforts is to use the CASE tools for database development. The main contribution of these tools is to provide automatic processes for developing all phases supported in a database methodology. Nevertheless, the current situation of these tools shows that they provide automatic and graphical user interfaces to reduce manual work and to make decision making easier. However, the total support of a database development methodology is missing and the code generated needs to be modified to complete the requirements. Therefore, tool support is needed to reduce the development, testing and maintenance effort (Verheecke and Straeten, 2003).

One of the most important phases that should be supported in these tools is the transformation of integrity constraints into integrity maintaining mechanisms. Although, most commercial CASE tools support the definition of some integrity constraints in the conceptual schema, not all of these tools provide the support of maintaining mechanisms to preserve these constraints in the logical phase. Some CASE tools have been studied and the most relevant results are shown in Table 1. All of these tools support both UML modelling and SQL code generation for maintaining integrity constraints in a relational database. Most of these tools support only the generation of SQL maintenance mechanisms to enforce the attribute constraints such as not null, primary key, etc. The Objecteering/SQL (2007) tool allows the trigger system to map multiplicity constraints, but only can be used for UML associations. To be exact, generated triggers conserve only the definition of maximum multiplicity for insertions, ignoring deletions and updates. The OCL22SQL tool generates SQL tables and views from a given UML/OCL model.

Table 1
Generated maintenance mechanisms for enforcing integrity constraints

Tools	Maintenance mechanisms type
(ArgoUML, 2007)	Attribute constraints included (e.g. not null, primary key, unique)
(MetaEdit+, 2007)	Attribute constraints included
(Objecteering/SQL, 2007)	Allows use of a trigger system to map only the multiplicity constraints for the UML associations of the insertions, ignoring deletions and updates
(OCL22SQL, 2007)	Generates views to support maintaining integrity
(Rational Enterprise Edition, 2003)	Attribute constraints included
(Visual Case Tool, 2007)	Attribute constraints included

In general, the current CASE tools have no strategy for trans forming integrity constrains into SQL code, and in particular trig gers. If there is one that has this strategy, it does not have any way of verifying the interaction of the generated triggers with themselves and with the other elements of the schema.

One of our objectives is to implement a tool following the phases proposed in MDA software development which transforms the OCL constraints into triggers. The MDA phases are denoted as follows: specifying OCL constraints in the UML class diagram, transforming the OCL constraints into SQL:2003 standard triggers, and transforming the standard triggers into target DBMS triggers. In addition, this tool can represent and verify the trigger execution by using UML sequence diagrams. This tool is considered as a pro posal for filling some of the gaps that commercial CASE tools leave during database development.

We are conscious that there are many publications and tools in the field of integrity constraints and active database systems. Nev ertheless, the main contributions of our work are:

- Our approach joins the UML aspects that have been widely accepted and supported by many CASE tools with the relational database aspects that have ample propagation in the commer cial DBMS. The approach is developed within a relational data base framework because many proposals in the context of object oriented databases respond completely to the active sys tem development such as that demonstrated in the previously related works.
- Certain drawbacks have been detected with the development tools of active mechanisms. In the case of commercial CASE tools, most of them do not give a complete support for develop ing active mechanisms within the relational database. By com plete support, we mean the development and the verification of the execution of these mechanisms.

Therefore, our approach not only provides technical support, it could be considered as a complete solution for generating active mechanisms from the specification of integrity constraints. More over, the UML sequence diagram is used to verify the active behav iour and avoiding non termination which is an added value with respect to the other CASE tools. In addition, using the Rational Rose commercial tool makes our approach more accessible and common than other approaches and research prototypes.

3. Description of the approach

MDA aims to obtain complete (semi)automatic software devel opment phases. It specifies two principle models; PIM and PSM (OMG, 2007). The PIM (platform independent model) focuses on high level business logic without considering the features of the implementation technology of the system. The PSM (platform spe cific model) represents the detail of using a specific platform for a system.

There are three phases for the adaptation of MDA to our ap proach (see Fig. 1). The first one, called PIM represents the logical view of the specification of integrity constraints using OCL/UML into a class diagram. The second one called PSM describes the tech nology used to build database applications. SQL:2003 is used as a standard technology to describe the active mechanisms of Rela tional databases. Finally, the third phase is called Codes which de scribes the active mechanism technology related to the commercial DBMS. As well as these phases, we use two transfor mation models to transform integrity constraints defined by OCL/ UML to trigger code related to a target DBMS; Transforming OCL to Trigger:2003 Model, and Transforming Trigger:2003 to DBMS Triggers Model.

In the other hand, as we said in the previous section, the devel opment of active mechanisms always needs to be verified to guar antee the termination state of the execution of triggers. Therefore, the DBMS trigger verification model is used to detect any non ter mination state in the execution of the generated DBMS triggers. The sequence diagram is used to help understanding and visualiza tion of trigger behaviour in the database design phase. According to our approach, the necessary transformation rules of these three models are carried out automatically. In the following we will fo cus on the description of these three models.

3.1. Transforming OCL to Trigger:2003 Model

There are many approaches to constraint classification in data base literature. Integrity constraints have been classified as static and dynamic constraints in the Relational model (Elmasri and Nav athe, 2000). The approach presented by Türker and Gertz (2000) divides integrity constraints according to the types of enforcement granularity, i.e. row, table, inter table, and transition constraints. Our constraints framework initially establishes two types of constraints:

- Inherent constraints are imposed by the data model itself. These constraints are not defined by the user; they are defined by the nature of the model. A schema is well constructed if it fulfils the inherent constraints of the corresponding model.
- Integrity constraints, also called semantic constraints or user constraints, are used to define in a systematic manner the busi ness rules of the discourse universe. Database modifications are rejected whenever the database final state does not fulfil the corresponding constraint. Sometimes, predefined actions are performed to fulfil business rule constraints. Most relational database systems support procedural mechanisms (such as pro

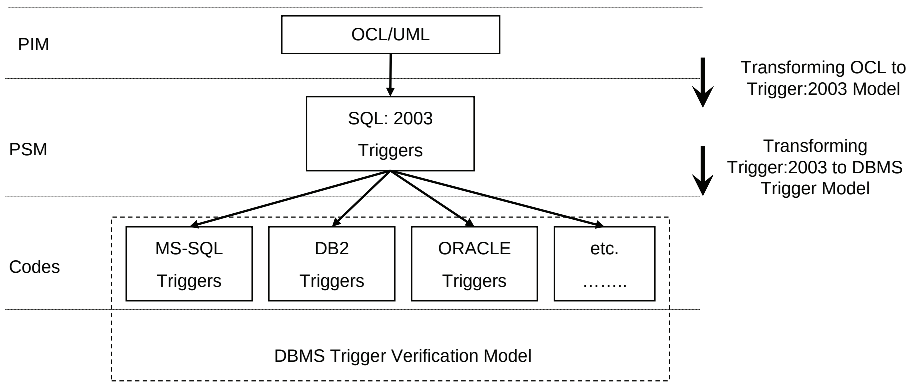

PIM
OCL/UML
Transforming OCL to Trigger:2003 Model
PSM
SQL: 2003 Triggers
Transforming Trigger:2003 to DBMS Trigger Model
Codes
MS-SQL Triggers
DB2 Triggers
ORACLE Triggers
etc. ……..
DBMS Trigger Verification Model

Fig. 1. Applying MDA for our approach.

cedures and triggers) to maintain the integrity constraints of the database. The syntax of these mechanisms depends on the particular characteristics of each DBMS.

Our approach is focused on integrity or semantic constraints which are divided into two types:

1. Integrity constraints which are expressed directly in a UML class diagram as relationships between classes (such as, associations, aggregations, generalization, etc.) because they offer more semantics than others and a greater effort for their implementation.
2. Integrity constraints which are expressed in a UML class diagram using OCL clauses. For example business rules such as "Married people are of age >= 18", or "The salary of an employee must be less than the salary of a manager". Although an OCL expression specifies invariant, pre condition, post condition, and other types of constraints, in this work the invariant constraints type is considered. An invariant constraint is a Boolean expression that can be associated with a class, a type or interface in a UML class diagram. OCL invariant constraints are used also to specify relationship constraints in the corresponding class although these constraints are already included in a UML class diagram (Cabot and Teniente, 2006).

In addition, we use an OCL invariant to define relationship constraints because the OCL2Trigger tool motor processes only OCL expressions, and sometimes it is impossible to fix the real values of these constraints without using an OCL expression, as we will show in this paper.

According to our approach most integrity constraints can be transformed to trigger but until now our OCL2Trigger tool supports only the transformation of OCL invariant constraints to a target DBMS triggers. We hope in future work to be able to include other types of OCL expression such as pre and post conditions.

Although, triggers are available in most DBMS, unfortunately the specification of these triggers changes from one DBMS to another. There are common components that are valid for almost database systems. These components usually do not change. To transform OCL specification to a target DBMS trigger, Trigger:2003 components are used in this work as common components (see Fig. 2).

Although the current DBMS allow us to use Java for trigger definitions we consider that the transformation of OCL to SQL is still necessary because in the context of the Relational model using SQL, no more effort is needed to use another programming language, such as Java. In addition, the transformation of OCL expression to a SQL trigger is a straightforward process (1 1) while the transformation of an OCL invariant to Java requires us to also create a Java class to represent the OCL invariant, but this class needs to be invoked using a SQL trigger (Oracle, 2007).

In general, the transformation model of an OCL expression to Trigger:2003 is a straightforward process. An integrity constraint contains three basic components which are the same as the three basic components of a trigger (event, condition, and action). An event is an operation which is used to modify a database object. A condition is a logical expression which specifies logical conditions on one or several elements of a schema and needs to be fulfiled. An action is an executable behaviour that is invoked when the corresponding constraint is violated. Besides the basic components, a trigger has two other components related to its dynamic behaviour (activation time and granularity). An activation time specifies when a trigger should be fired, before, or after a triggering event is produced. A trigger is executed according to its granularity level. A statement level trigger executes once for each triggering event, and a row level trigger executes once for each row in the modified row set.

In the next sections these components will be presented and how they are derived from the specification of OCL invariants. A more extensive version of our approach is detailed in Al Jumaily (2006).

3.1.1. Mapping an OCL invariant to SQL:2003 query

An OCL invariant expression must be true for all instances of element type at any time. An invariant is a type of Boolean (OMG, 2007). The formal definition of an OCL invariant is shown in the following:

```
Context <class_name> inv <constraint_name>:
  <OCL_expression (self)>
```

Self is an instance of a type (e.g. Company). The *context* specifies the class in which an OCL expression is defined. The *constraint_name* is a name of an OCL constraint and it will be converted to a trigger name. The *OCL_expression* is a logical expression that describes a relationship between two expressions. This expression is evaluated as true if the relationship is fulfiled or false in the opposite case.

The mapping of the OCL invariant expression to a query is done directly by using the following SQL:2003 template:

```
(SELECT * FROM context_table SELF
    WHERE NOT OCL_expression)
```

The context_table of the query is substituted for the class_name of the OCL. The OCL_expression is mapped using the logical and mathematical operators of SQL. For example, the mapping of mathematical operators (, +, *,/) is performed directly. The logical operators (and, or, not, xor) are mapped using counterpart expressions of SQL.

An OCL invariant clause is used to specify a condition on objects so if this condition cannot be satisfied, we need to abort the transaction which leads to the inconsistency in the database state. The logical negation 'NOT' of the original OCL expression is used in the SQL query because a trigger should be fired whenever the corresponding OCL expression is not satisfied. For example, a business rule might be "the number of employees of a company must al

Trigger name	`CREATE TRIGGER <`*`trigger name`*`>`
Activation time	`[`*`before`*` | `*`after`*`]`
Critical operation	`[`*`insert`*` | `*`delete`*` | `*`update`*` [`*`OF`*` <`*`trigger column name`*`>]]`
Related table	`ON <`*`table name`*`>`
Granularity	`[`*`FOR EACH`*` {`*`ROW`*` | `*`STATEMENT`*`}]`
SQL condition	`[WHEN <`*`condition`*`>]`

Action	`BEGIN ATOMIC`
	`{<`*`SQL procedure statement……`*`>}...`
	`END;`

Fig. 2. The SQL:2003 standard trigger syntax.

ways exceed 50". Here, if the number of employees of that com pany is equal or less than 50, then the corresponding trigger should be fired to satisfy this constraint.

To convert OCL clauses into SQL queries, some related ap proaches are considered, for example, Birgit and Heinrich (1999) where OCL constraints are transformed into SQL Assertion although current assertions cannot be defined in most commercial database systems. The similarities between our approach and the work mentioned above are that we use the OCL invariant main and basic type patterns and their transformation to SQL queries in the same way. The differences between the two works, in addi tion to the use of different mechanisms to enforce OCL constraints, is that in our work, the patterns are adapted according to our con straints framework while in the cited work the patterns were spec ified according to the result type of the OCL expressions. We think that our study makes the task of classifying the OCL constraints easier, as well as facilitating the transformation and implementa tion tasks.

In addition to this, we considered the atomic constraint patterns used (Wahler et al., 2006) to restrict the fundamental concepts of a model, e.g., attribute values or relationships between objects. We adopt some of these constraint patterns to transform PIM to PSM although in that work these are used to transform the computation independent model (CIM) into PIM.

In the following sections, we define the transformation patterns which are used to convert OCL invariant into SQL:2003 query. All these patterns are derived from the abovementioned formal defini tion of an OCL invariant:

3.1.1.1. AttributeValueConstraint pattern. This pattern is applied when a constraint is defined to specify a logical predicate on a tar get attribute in a class. The formal definition of this OCL pattern is shown below:

```
Context <class_name> inv <constraint_name>:
    <self.attribute.op. term>
```

The transformation to SQL query is shown below:

```
(SELECT * FROM context_table self
    WHERE NOT self.attribute.op. term)
```

The transformation is performed directly. The context_table of the above query is substituted by the class_name of the corresponding OCL. The self.attribute and the operator.op. are substituted by the target attribute name and the counterpart operator of SQL, respec tively. The principal rules for mapping 'term' are shown in the following:

(a) If the term is a basic type value (e.g. Boolean, Integer, Real and String), then the term is substituted by the value. For example, the following expression would specify that the number of employees of a company must always exceed 50:

```
Context Company inv numberOfEmployees:
    self.numberOfEmployees > 50
```

According to the above OCL expression the corresponding trig ger should be fired whenever the number of employees is less than 50. Applying this rule and the logical negation of the con straint, the OCL invariant is transformed into the following SQL query:

```
(SELECT * FROM Company self
    WHERE self.numberOfEmployees <=50)
```

Although this type of constraint can be enforced using triggers as well as by using declarative constructors, triggers tend to consume fewer resources and are much faster than some declarative constructors such as Check and Assertion (Decker et al., 2006).

(b) If the term contains associated tables then the mapping is done by joining these tables by the related keys (Primary Key = ForeignKey). As shown in the following constraint: "The budget of a project must not exceed the budget of the controlling department".

```
Context Project inv BudgetDept:
    self.budget <=self.department.budget
```

The transformation to SQL query is shown below:

```
(SELECT * FROM Project P JOIN Department D
    ON P.Deptno = D.Deptno
      WHERE p.budget > d.budget)
```

(c) The operator 'implies' does not have a counterpart in SQL so that it can be mapped using the logical operator AND, and the logical negation of a constraint is applied to the atomic expression which becomes directly after 'implies'. The fol lowing example shows how a constraint is used to ensure that every employee over 50 years age gets at least 3000. Here, the corresponding trigger should be fired whenever the age of an employee is greater than 50 years and the sal ary is less than 3000.

```
Context Person inv EmployeeSalary:
    self.age > 50 implies
      self.contract.salary >=3000
```

By applying the previous rule (b) this OCL constraint is trans formed to a SQL query by joining the associated classes (Person and Contract) by the related keys. According to (c) the operator 'implies' is transformed into AND, and the logical operator '>=' of the atomic expression self.conract.salary >= 3000 is replaced by its negation '<', the following SQL query shows the transfor mation of the above OCL:

```
(SELECT * FROM Person P JOIN Contract C
    ON P.Psn = C.Psn
      WHERE self.age > 50 AND C.salary < 3000)
```

3.1.1.2. MultiplicityConstraint pattern. A correct transformation of a conceptual schema and its constraints to a Relational model is nec essary to preserve business rules of the discourse universe. Multi plicity or cardinality constraint is one of the constraints that can be established in a conceptual schema. Since its introduction by Chen (1976), the cardinality constraint consists of the minimum and maximum numbers of entity instances associated in a relationship. UML multiplicity constraints follow Chen's style because verifying the multiplicity constraints is needed to fix an object of a class *A* and to see how many objects are related to it in another class: *B* (Cuadra et al., 2003). That is to say, the multiplicity constraint is the number of instances of one class related to one instance of the associated class.

Although these constraints are defined between the classes in many UML commercial CASE tools, most of these tools do not gen erate any mechanisms with which to enforce them in the Rela tional model. The formal definition of the OCL invariant which specifies a multiplicity constraint is shown below:

```
Context <class_name> inv <constraint_name>:
  <self.navigation >size().op. term>
```

Although associations are bi directional, the arrowhead (>) is added to restrict the direction of the navigation. OCL allows navi gating from an association class itself to objects which participate in that association.

The mapping of multiplicity constraints is done using nested queries and depends on the association type Our work supports the three types of multiplicity (one to one, one to many, many to many).

Let us consider two classes: *A* and *B*. *R* is an association between these classes, mapping this association to the Relational model is shown as the following:

- If *R* is a one to many association then the associated class *A* and *B* become tables in the Relational model, and the foreign key in the table *B* must match an existing primary key in the context table *A*.
- If *R* is a many to many association, a new related table *R* is created and the foreign keys in *R* must match existing primary keys in *A* and *B*.
- One to one association is a particular case of the associations one to many or many to many.

In general, nulls are allowed in optional multiplicities but are not allowed when using mandatory multiplicities. Mandatory multiplicities always need triggers to enforce relationships between the associated classes. The principal rules for mapping multiplicity constraints into SQL query depend on the minimum and maximum bounds of these constraints, as shown below:

(a) If the minimum multiplicity of an association is equal or more to (1) then the following OCL clause is used:

```
Context <class_name> inv <constraint_name>:
    <self.navigation >size () >=1>
```

Here, we need to navigate from a context class towards an associated class. To map this OCL to an SQL query the class_name and the associated class are transformed to context_table and related_table respectively. In addition, a definition of the related keys *pk*, *fk* between the two previous tables is needed. The logical negation of the constraints is applied to the operator IN of the subquery. The following SQL query is true whenever an object in the context_table does not have any associated objects in the related_table.

```
SELECT * FROM Context_table self
    WHERE self.pk NOT IN
      (SELECT fk FROM Related_table)
```

(b) If the maximum multiplicity of an association is known then the following OCL clause is used:

```
Context <class_name> inv <constraint_name>:
    <self.navigation >size() <= Max>
```

The transformation is performed as in the pervious rule (a). The logical negation of the constraint is applied to the atomic expression which restricts the maximum multiplicity value, as shown in the following SQL query:

```
SELECT COUNT(*) FROM Context_table self
    WHERE self.pk IN
      (SELECT fk FROM Related_table
        GROUP BY fk HAVING COUNT(*) > Max)
```

Although the aggregation and the composition relationships are not implemented in this work, the MultiplicityConstraint pattern can be extended to include this type of constraints. These relationships are special forms of an association and specify the possible existence of links between objects of associated classes. The types of these links are well specified in (Gogolla and Richters, 1998). Rational Rose, the framework of our approach, maps these relationships into two types of aggregations. The first type is aggregations by value, also named *composite aggregations* are used when the part cannot exist without the aggregate. The second type is aggregations by reference, also named *aggregations* are used when we have multiple objects of an aggregate class owning a part class. For these two types of relationships we need to use the previous multiplicity pattern query (a) in order to restrict the mandatory multiplicity of the part class, as follows:

```
SELECT * FROM Aggregate_table self
    WHERE self.pk NOT IN
      (SELECT fk FROM Part_table)
```

3.1.1.3. GeneralizationConstraint pattern. A generalization is another type of relationship constraint that has dynamic aspects which need to be verified. It is a set of relations which is produced when a generic entity is disjointed into supertype entity and subtype entities. A supertype entity contains the generic entity key and all other common attributes. A subtype entity contains the generic key and only the specific attributes of the subtype. Although there are four possible types of these constraints: Disjoint Total, Overlapping Total, Disjoint Partial, and Overlapping Partial (Teorey, 1999), in this work only the Disjoint Total constraint is considered because it offers more semantics than the other ones and it needs more effort to be implemented. A Total constraint specifies that an object of a supertype can be a member of at most one of the subtype. A Disjoint constraint specifies that the objects in a different subtype from the same supertype are completely different.

According to our approach, the transformation of the generalization into Relational model produces one table for each supertype and subtype class. It is necessary to include an attribute for partitioning objects (not null) in the supertype table.

As for multiplicity constraints, although generalization constraints are defined between classes in many UML commercial CASE tools, most of these tools do not generate any mechanisms to enforce them.

The formal OCL definition of a generalization relationship is specified according to the constraint type (Total or Disjoint), as shown next:

1. Total constraint: An object of the supertype must be a member of at most one of subtypes.

```
Context <supertype> inv <constraint_name>:
    <self >forAll(c|oclIsKindOf(subtypeA) OR
        oclIsKindOf(subtypeB))>
```

According to this definition, to verify a Total constraint of the generalization, the corresponding OCL invariant is specified in the supertype class. This may be useful in an OO language but in SQL it is useless because, if a trigger is defined in the supertype table this trigger is activated only when the supertype table is modified.

Let us consider the case when the supertype table contains a generic key value, denoted by *id*, which has an instance in the subtypeA, and a trigger *T* is defined in the supertype table to enforce the Total constraint. If the value *id* is deleted from subtypeA table then the Total constraint is lost because the instance *id* in the supertype table is not a member of any one of the subtype. In this case, although an instance was deleted from the relationship, the defined trigger *T* on the supertype table was not fired because the deletion action was produced in the table subtypeA and not in the table supertype.

In addition, in order to map the above OCL to SQL query, we need to study the semantics of the generalization relationship and how it can be transformed to the Relational model (Al Jumaily, 2006). To enforce Total constraints, the following rule is used:

(a) A SQL query is used to ensure that each instance in the supertype table is a member of at most one of the subtype tables. The following SQL query is defined for each subtype table in a generalization. The corresponding trigger should be fired whenever an instance in a supertype table does not have a related instance in the subtype tables.

```
(SELECT pk FROM supertype
    WHERE pk NOT IN (SELECT pk FROM subtype)
```

2. Disjoint constraint: Objects in a different subtype from the same supertype are completely different.

```
Context <supertype> inv <constraint_name>:
    <self >forAll(c|not(oclIsKindOf(subtypeA)
AND
        oclIsKindOf(subtypeB)))>
```

As it has been shown in the Total constraint case, defining a trigger on a supertype table is useless for enforcing a Disjoint constraint. Let us consider that the supertype table contains a generic key value (*id*) which has an instance in subtypeA, and a trigger *T* is defined on the supertype table to enforce the Dis joint constraint. If a new *pk* instance is inserted into the subty peB table then the Disjoint constraint is lost because the *pk* instance in the supertype table is a member of all subtypes. To enforce Disjoint constraints, the following rule is used:

(b) A SQL query is used to ensure that each instance in the supertype table can be a member of only one subtype table. This SQL query is defined for each subtype table in a gener alization. Each SQL query must ensure that every instance of the related subtype table cannot be a member of another subtype table. The following SQL clause is used to define this constraint. The corresponding trigger should be fired when ever an object in the subtypeB table can be a member of the other subtypeA.

```
(SELECT pk FROM subtypeA
    WHERE pk IN (SELECT pk FROM subtypeB)
```

3.1.2. Deriving the remainder of trigger components

Once the OCL to SQL query is mapped, the next step of the pro posal is how to derive the other components of a trigger such as critical operations, activation time, granularity, and actions. These components are derived from the SQL queries obtained in the pre vious section.

3.1.2.1. Critical operations. A critical operation is an event that may violate an integrity constraint, these events are: Insert, Delete, and Update. If an event violates an integrity constraint, it is necessary to implement a mechanism for enforcing that constraint. However, using a mechanism to control an event that does not violate any constraint would not make sense and would lead to a negative ef fect on the system performance.

Once the SQL:2003 queries have been established in the previ ous section, the critical operations are derived from them, as the following rules:

(a) *AttributeValueConstraint critical operations*:

- If a SQL query defines a logical predicate to specify an attribute value in a table then the critical operations are the insertions into that table, and the updates of that table. For example, a business rule might be "the salary of an employee must be >= 1200 and <= 1800". In this case, an insertion or an update of employees needs to ensure that their salary must be in the spec ified range. Normally, a Check mechanism is used to enforce this type of constraint.
- If a SQL query specifies a logical predicate between attributes in various rows in the same or in a different table, then the critical operations are: insert into the tables, and the update of these attributes. For example, "The salary of employees must be less than the salary of the manager". In this case, insert "a new man ager needs to ensure that his/her salary must be greater than the salary of their employees" and also insert "a new employee needs to ensure that his/her salary must be less than the salary of his/her manager". The update is considered as a deletion of an old value and an insertion of a new value, so that it needs to ver ify that the constraint is the same as for the insertion.

(b) *MultiplicityConstraint critical operations*:

Table 2
Critical operations of multiplicity constraints

Multiplicity	Critical operations
Many-to-many	Insert(A,B), Delete(A,B), Delete(R), Update(R.a,R.b)
One-to-many	Insert(A), Delete(B), Update(B.a)
One-to-one	Insert(A), Delete(B), Update(B.a)

- If a SQL query defines a minimum or maximum multiplicity con straint between associated tables, critical operations are shown in Table 2. Let us consider *A* and *B*: associated entities in a binary relationship *R*. Let us suppose that the referential actions in this relationship are On Delete Cascade and On Update Cascade. The transformation of this relationship to the Relational model is performed according to the relationship type (see **Multiplicity-Constraint pattern**). For example, the one to many relationship between **Department** and **Employee** may specify a constraint such as "Every department has at least one employee". In this case, critical operations are the insertion of a new department, *Insert(A)*, deletion of an employee, *Delete(B)*, and updating a for eign key of Department in Employee, *Update(B.a).*

(c) *GeneralizationConstraint critical operations*:

- If a SQL query defines a Total Disjoint constraint, critical opera tions are the deletion from a subtype table, and the insertion into a subtype table. For example, a person can be only a student or a professor. The transformation of this constraint to the Rela tional model is shown in **GeneralizationConstraint pattern**. If an instance is deleted from the table Student, the Total con straint of the generalization maybe lost because the related instance in Person is not being a member of any one of the sub type. If a new instance related to a Person is inserted into the Professor table, the disjoint constraint maybe lost if there is a related instance to that Person being a member in the table Student.

3.1.2.2. Activation time. An activation time defines whether a trig ger execution must be produced before or after a related event. Although some commercial database systems have some limita tions when activation time is defined, the SQL:2003 standard al lows two types (Before and After trigger) to be used without any limitation. Since the Before trigger is executed and verified immediately before the finishing of the transaction which leads to that trigger to be fired, SQL:2003 recommends using Before trig gers to read from a database or to correct an error produced in the processing of data input. For example, "The salary of an employee must be less than 2000" this constraint can be transformed into an insert trigger with an activation time of Before trigger. In the ac tion of this trigger, the salary can be assigned to 2000 if its new va lue is greater than 2000.

In a Relational database, when an event takes place, the consis tency of the database state must be verified after all cascade events that may have been produced by the original event. In addition, using After trigger allows during triggers execution to reach the old and new transition values. Therefore, in general, After triggers are useful to update other tables, or to invoke functions to perform tasks inside or outside the database.

The activation time is derived in our approach according to the following rules:

(a) Before trigger activation time is used when a SQL query defines a logical predicate to specify an attribute value in a table. It is used to correct an error produced in a processing of data input such as the above example. Our work will be extended to introduce more OCL constraints types such as the pre condition constraints.

(b) After trigger activation time is used when the SQL query defines other type of logical predicate such as, a logical pred icate between attributes in various rows in the same or dif ferent tables, multiplicity constraints, and generalization constraints. These constraints may produce cascade events; therefore its verification should be produced only after the current transaction is finished. Most of the commercial data base systems support only an After trigger activation time.

Let us consider an example, a constraint like "the salary of an employee must be less than 2000" could be defined in OCL as:

```
Context Employee inv EmpSalary:
        self.salary<1200
```

Applying the AttributeValueConstraint Pattern this OCL con straint is transformed to the following SQL query:

```
SELECT * FROM Employee self
   WHERE self.salary >= 1200
```

That is to say, the trigger should be fired wherever the salary of an employee is greater than or equal to 1200. According to our transformation rules (see section 'Critical operations') the critical operations of this type of pattern are: Insertion into Employees or Update Employees.salary. In this case, the Before and After trig ger can be used as activation time for this constraint although in the standard SQL:2003, this type of constraint can be enforced by using Before triggers activation time.

3.1.2.3. Granularity. There are two levels of granularity. The state ment level trigger is executed once for each triggering event and the row level trigger is executed for each row in the modified set. Integrity constraints can usually be established in one row or a combination of rows. When the constraint only affects one row, it is converted into a row level trigger, whilst when it affects a combination of rows, the statement level trigger is needed.

In general, all types of constraint can be verified by using state ment level triggers. For example, a constraint such as "the salary of an employee must be >= 1200 and <= 1800" can be verified using a statement level trigger but the verification will be to the salary of all the employees', and not only to the modified salary. This will consume more resources and therefore lead to inefficient databases.

For performance reasons, it is preferable to use row level trig gers because they allow the verification of conditions to be adapted to the modified rows only. Furthermore, SQL:2003 standard and most commercial DBMS allow the use of the WHEN clause only in the row level triggers.

The granularity is used in our approach according to the follow ing rules:

(a) Row level granularity is used if a SQL query defines a logical predicate to specify an attribute value constraint, multiplic ity constraints, and a generalization constraint.
(b) Statement level granularity is used if a SQL query includes aggregate functions such as SUM, AVG, MIN, MAX and COUNT. An example, "The total salary of employees working in a department must not exceed the department's budget". This constraint needs to be verified by a statement trigger.

3.1.2.4. Action. Normally, when an integrity constraint is violated, a specific reaction is fired to reject the actual transaction or to trigger corrective actions depending on the business rule semantics. Two types of reaction are applied in the proposal:

(a) Corrective actions are used to enforce the generalization relationship. These actions are shown below:
    ◦ Total constraint is reflected through two triggers. One of them is the trigger which ensures that each instance in the supertype table can be a member of at most one of the subtype tables. The other one is the trigger for each subtype table which are used to delete the related instance from the supertype table whenever an instance is deleted from a subtype table.
    ◦ Disjoint constraint is the capture by a trigger in order to ensure that each instance in the supertype table can be a member of only one subtype table. There are also other triggers for each subtype table which is used to delete an insertion into a subtype whenever the other subtype contains the same key value.

(b) Rollback action is used if a SQL query defines other types of constraints, such as attribute values and multiplicity constraints.

In addition, to make our approach more flexible, we allow the user to modify a trigger body in order to include more operations according to the required semantics, as we will explain in Section 4.1.1.

3.2. Transforming trigger:2003 to DBMS triggers model

Although, most Relational DBMS trigger systems have the same components, the transformation of the trigger:2003 to a target DBMS trigger should take into account the specific characteristics of each one. There are some differences between the specific char acteristics of triggers in these DBMS. These differences create the issue that triggers of one system cannot be used directly with an other system without modification. A comparison of the more important trigger aspects of some Relational systems is shown in Table 3.

According to the study, all database systems allow the execu tion of multiple triggers at the same time. The execution of multi ple triggers can sometimes lead to a non termination problem. Only disjunctive composite events are allowed in Oracle. The exe cution of triggers before, or after the event, is performed by all the database systems that were studied, except SQL Server 2005 which only allows execution after the event and only using the statement type granularity. This is to say, all OCL constraints types such as invariant, pre condition, post condition constraints are mapped using the After triggers and Statement trigger in SQL Server 2005.

There are some limited strategies which are used to treat the cascade execution of triggers in the database system such as Oracle 11g, DB2 and SQL Server 2005. In Oracle 11g trigger syntax now in cludes the *Follows* clause to guarantee the order of execution for triggers defined with the same timing point. DB2 contains the op tion 'No Before Cascade' which specifies that the trigger's action cannot cause the activation of other triggers. In SQL Server 2005, it is possible to use the function *SP_SETTRIGGERORDER* to specify the trigger which is fired first or last. The After triggers that are fired between the first and last triggers are executed in undefined order. The termination strategy is used to prevent the non termi nation problem in most Relational database systems. For example, when a trigger in Oracle causes another trigger to be fired, Oracle allows up to 32 triggers to cascade at any one time. However, this limit can be changed using the initialization parameter OPEN_CURSORS. Because Trigger:2003 does not support any limi tation or cascade execution strategy, our approach does not con sider any strategy to limit the execution of the triggers.

Once the Trigger:2003 components are derived according to the previous section, these components are mapped to a target DBMS

Table 3
A comparison among some trigger aspects

	Trigger:2003	ORACLE 11g	DB2	SQL SERVER 2005
Multiple triggers (N)	N	N	N	N
Event type	Insert, Delete, Update	Insert, Delete, Update	Insert, Delete, Update	Insert, Delete, Update
Composite events	No	Yes (only OR)	No	No
Activation time	Before/After	Before/After	Before/After	After
Granularity	Row/Statement	Row/Statement	Row/Statement	Statement
Cascade strategy	No	Yes	Yes	Yes
Termination strategy	No	Yes	Yes	Yes

triggers. The mapping is performed directly, that is, a Trigger:2003 is mapped into one trigger in a target DBMS (1 1). To do this map ping, DBMS trigger templates are used. A trigger template is a gen eric trigger in which some values are established as parameters so that different particular triggers can be derived by giving different values to the parameters (Domínguez et al., 2002). The values of trigger templates are obtained from the execution of the following function:

```
Trigger2003Components(OCL, Constraint_name, SQL
query, Context_table(), Related_table, Event(), Time,
Granularity, Action)
```

This function requires the specification of the string value of an OCL which is submitted to a syntax analysis to obtain the following values:

Constraint_name is the name of the OCL constraint.
SQLquery is a trigger:2003 condition which will mapped to a target DBMS trigger condition.
Conext_table() is the name of a table in which the trigger is cre ated. It is an array type parameter because in some case we can obtained more than one context table to map some constraint type, for example for a Total constraint of a generalization we need to create a trigger for each subtype table.
Related_table: in some types of constraints we need to calculate a related table or an associated table. As an example, in an asso ciation between two classes: the context table and the related table is introduced to map the corresponding trigger.
Event() is the type of critical operation that may violate the OCL constraints. It is an array type parameter because in some cases we can predict that more than one event will map some con straint type. For example for a multiplicity constraint we need to create a trigger for each event (see Table 1).
Time is the activation time of the corresponding trigger (Before, or After).
Granularity is the granularity of the corresponding trigger (Row or Statement).
Action is a specific reaction to be activated when the constraints is violated.

Once these values are calculated, template functions are called in order to derive the required trigger. There is one template func tion for each DBMS and for each OCL pattern considered in this work. For example, we use one template to transform the Multi plicityConstraint pattern to an Oracle trigger.

Although Oracle has the same components as Trigger:2003, it is a less efficient system, because in Oracle the mutating tables prob lem needs to be solved (Oracle, 2007). Since Oracle 11g was intro duced, this problem has been solved using Compound Triggers. A Compound Trigger allows code for one or more triggers for a spe cific table to be combined into a single trigger.

In this section, we present only one template as an example to illustrate a generic template of the multiplicity constraint to illus trate the transformation of the Trigger:2003 to Oracle 11g. The function *OracleTemplate4Multiplicity* is shown in Fig. 3. The calling and the execution of this function is done according to the follow ing algorithm:

(a) Calling *OracleTemplate4Multiplicity* with Trigger:2003 parameters which are calculated from the previous function *Trigger2003Components*. This function is executed once for each pair (context_table, event). For example, for an aggre gation constraint between two classes $C1$ and $C2$, such as "every object in class $C1$ has at least one object in $C2$" one trigger needs to be generated for each pair (table_C2, Delete), (table_C2, Update), and (table_C1, Insert) (Al Juma ily, 2006). Although Oracle allows using composite events to optimize the number of triggers, in this paper we do not use this type of events although it will be tackled in future works. The following parameter values are used when the function is executed to generate triggers for the pair (table_C2, Delete):
```
Constraint_name: table_C2_multiplicity
SQLquery : "(SELECT * FROM table_C2 self WHERE
self.id_C1
             NOT IN (SELECT self.id_C1 FROM
table_C1))"
Context_table: "table_C2"
Event: "DELETE"
Time: "AFTER"
Granularity: "FOR EACH ROW"
Action: "Raise_Application_Error"
```
Using compound triggers in Oracle obliges us to use the two types of activation time and the two types of granular ity although in the above values, only one value for the activation time and one value for the granularity is shown.
(b) Substituting the parameters of the template with the corre sponding specific values. For example, in order to generate the trigger for controlling the aggregation constraint shown in (a) we need to substitute the parameters shown in *Oracle Template4Multiplicity* by the corresponding values. The tem plate parameters are shown as italic and bold text (see Fig. 3). Although the SQL standard considers a complex query in the WHEN clause of triggers, most commercial DBMS have the limitation of considering a complex query in this clause. So that in the trigger template, we have included the SQL query of a constraint in the trigger body with some adaptation to fit in with the specific characteris tics of Oracle.
(c) Printing the generated trigger. All generated triggers are saved in a text file *.sql which can be executed directly in any related DBMS.

```
Sub OracleTemplate4Multiplicity(Constraint_name, SQLquery, Context_table,
                         Event, Time, Granularity, Action)
C_trigger:= "CREATE OR REPLACE PACKAGE constraint_name IS
              FOR event ON context_table
              COMPOUND TRIGGER

              -- Global declaration.
              TYPE context_table_PK IS TABLE OF context_table.PK%TYPE
              INDEX BY BINARY_INTEGER;
              old_PK   context_table_PK;
              counter    NUMBER:=0;
              var      NUMBER:=0;

              BEFORE EACH ROW IS
              BEGIN
              counter:= counter + 1;
              old_PK(counter):=:Old.PK;
              END BEFORE EACH ROW;

              AFTER STATEMENT IS
              BEGIN
              FOR I IN 1 .. counter LOOP
                   IF EXISTS (SQLquery) THEN
                   Action;
                   END IF;
              END LOOP;
              END STATEMENT;
END constraint_name;"
End OracleTemplate4Multiplicity;
```

Fig. 3. Oracle template for Multiplicity Constraints.

3.3. *DBMS trigger verification model*

The objective of the verification model is to guarantee the termination of the execution of triggers. Termination indicates that the execution of any set of active mechanisms must terminate. This is needed in order to avoid cycles in the execution. The Triggering Graph (TG) (Baralis et al., 1993) is used to detect non termination states. The TG is a straightforward graph where each node T_i corresponds to a trigger and a direct arc between T_1 and T_2 is the event which belongs to T_1's action and causes the activation of T_2. A cycle or non termination is produced in the TG when a T_i may trigger itself or when T_i triggers the same initial subset. In Fig. 4, the subset of triggers $S = \{T_1, T_2, T_3\}$ is a cycle when T_1 is fired again by the event e_3. The termination analysis itself focuses on identifying and eliminating arcs that could introduce cycles into the TG (Hickey, 2000). Redefining rule T_3 and reconstructing the graph TG is a good solution for verifying the termination state of the subset.

On the other hand, the most important aspects of the SQL:2003 standard are the interactions between the triggers and the referential constraint actions (Kulkarni et al., 1998). In Relational databases, the tables are represented by sets of rows, and there are relationships between tables, which are represented by the foreign key definition. Referential constraints are predicates on the database state that must be evaluated. If these restrictions are violated, the database is in an inconsistent state. To maintain the referential integrity of the database, the SQL:2003 standard and most commercial DBMS use referential constraint actions such as On Delete Cascade (DC) and On Update Cascade (UC).

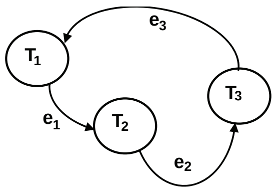


Fig. 4. The TG of rule execution.

A relational table R′ is a Referencing_Table if it has one or both referential constraint actions (DC and UC) which are defined within the specification of Referenced_Table R. The triggers which are defined on R' are activated as a consequence of modifying R (Delete or Update(Attribute)).

The interactions between triggers and these referential actions make the detection of non termination more difficult because two types of events can activate triggers in a Relational database. We sort the events in Direct such as Insert, Delete, and Update and Indirect such as the previous referential actions. Therefore, in this work, we take these two types of events into account when mapping the trigger execution.

Our approach considers that a *non termination problem arises when a trigger T is triggered twice in the same activated set*. To demonstrate this, the following steps are checked for all trigger sets that can be activated by any possible event:

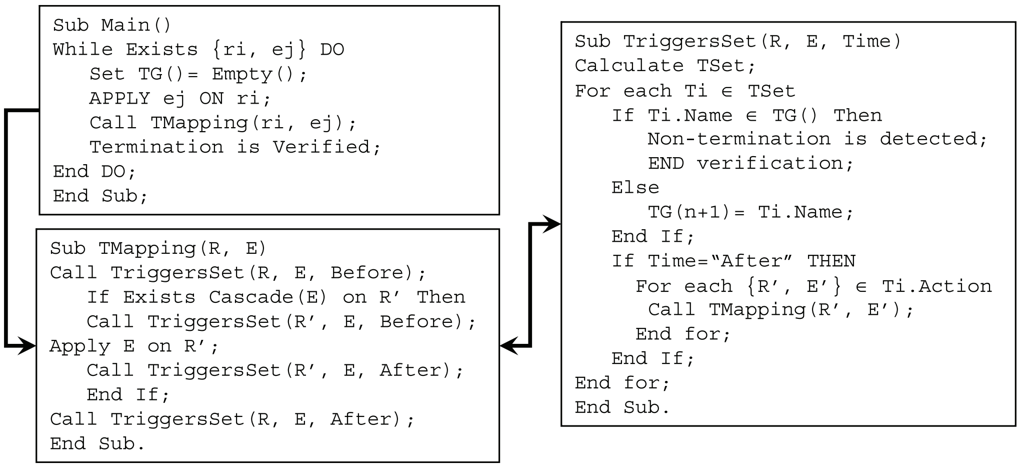
1. Check the triggers which are defined in the table r and which are activated directly by the event *e*.
2. If *r* is a referenced table, then check the triggers which are activated by the indirect event (referential actions) on the referencing table r′.
3. If the action of any trigger produces a new event then check the triggers which are activated by that event.

By applying the previous three steps for any set of $\{r, e\}$ the termination of any trigger execution scenario is verified. For the implementation of these steps, the generic algorithm shown in Fig. 5 is used:

where

$\{r_1, r_2, r_3, \ldots, r_n\}$ is the database schema with $r_1,\ldots,r_n$ relational tables.

$\{e_1, e_2,\ldots,e_m\}$ is the set of direct events (Insert, Delete, Update(Attribute).



```
Sub Main()
While Exists {ri, ej} DO
   Set TG()= Empty();
   APPLY ej ON ri;
   Call TMapping(ri, ej);
   Termination is Verified;
End DO;
End Sub;
```

```
Sub TMapping(R, E)
Call TriggersSet(R, E, Before);
   If Exists Cascade(E) on R' Then
   Call TriggersSet(R', E, Before);
Apply E on R';
   Call TriggersSet(R', E, After);
   End If;
Call TriggersSet(R, E, After);
End Sub.
```

```
Sub TriggersSet(R, E, Time)
Calculate TSet;
For each Ti ∈ TSet
   If Ti.Name ∈ TG() Then
      Non-termination is detected;
      END verification;
   Else
      TG(n+1)= Ti.Name;
   End If;
   If Time="After" THEN
     For each {R', E'} ∈ Ti.Action
      Call TMapping(R', E');
     End for;
   End If;
End for;
End Sub.
```

Fig. 5. The generic algorithm for detecting non-termination.

TG() is the vector for representing the triggering graph. Once an event *e* is issued to modify *r*, the TG() vector is reinitiated.

Cascade(e) is the referential action (indirect event) which is produced by the original event. For example, if the original event is deleted from any referenced table then Cascade(e) is the action "On Delete Cascade" against the referencing table, and so on.

TSet = {T_1, T_2, ..., T_x} is the set of Before or After triggers which are activated by the event *e* on the table *r*.

T_i.Action: If the trigger action issues any new event e' to modify any table r' then the algorithm is replied to, in order to map the trigger execution which is activated by that event. In this work, only After triggers can include actions which modify another table because the SQL:2003 standard recommends that Before triggers be used to read from the database, or to verify the attribute values before applying them to the database. Another issue which must be taken into account is that the Before triggers are executed and verified immediately before the transaction is finished. Nevertheless, in order to obtain the required semantics, it must be verified that all modifications will be made by the original statement. While, the AFTER triggers are useful to update other tables, or invoke functions to make tasks inside or outside the database.

Termination is verified: if the algorithm execution touches this point, it means that the execution of triggers which are activated by e_j to modify r_i is correctly verified.

Non termination state is detected when there are two instances of the same object in TG(). In this case, a message is sent to warn the user about the existence of this problem and the mapping is finished immediately.

UML sequence diagrams are used to implement the above algorithm and to show the interaction between triggers and objects in a sequential time order depending on their occurrence. Sequence diagrams allow users to create a visual representation of a scenario. It is a two dimensional diagram, where the vertical dimension is the time axis, and the horizontal dimension shows the interaction of object roles. In Section 4.3.2 we will explain how UML sequence diagrams have been adopted for our approach.

4. OCL2Trigger tool design

Now that we have presented our approach we will explain how it has been implemented as a tool. This tool is called OCL2Trigger, and it aims to carry out the necessary rules of the transformation models of our approach automatically. The tool has been added to Rational Enterprise Edition (2003), one of the most important commercial CASE tools in the market. We have chosen to incorporate our tool into this package, because this commercial tool has the potential for adding modules to support software development needs. OCL2Trigger can be accessed from the Rational Rose Tools menu.

The architecture of the OCL2Trigger tool is shown in Fig. 6. It consists of three phases: the OCL Constraints Specification Phase, the Transformation of OCL clauses into DBMS Triggers Phase, and the Trigger Adaptation and Verification Phase. A brief for each phase is presented below.

4.1. OCL constraints specification

To specify business rules in a UML class diagram as OCL clauses, OCL2Trigger tool provides two modules. The first one is used for editing and checking OCL constraints that cannot be expressed more easily in the graphical model. The second one is used to generate OCL clauses in those constraints which are expressed in the graphical model such as multiplicity constraints and generalizations. All OCL constraints of this phase are plugged into the Rational Object Model in the corresponding classes. This phase contains the following modules.

4.1.1. Edit/Check OCL constraints module

For editing and checking OCL constraints that are used in this module, users can introduce the integrity constraints of any class diagram as OCL clauses. To do that, Oclarity tool (EmPowerTec, 2006) is applied. It is an add in for Rational Rose which offers a comprehensive support for OCL editing and verifying. According to the current OCL 2.0 specification, Oclarity tool provides full syntactic and semantic checking. For example, let us consider the constraint "Married people are of age >=18". It is impossible to specify this constraint directly in the graphical model without using OCL. The user can introduce this type of constraint into the corresponding class by using the Oclarity editor and he/she can verify their syntax. Fig. 7 shows the previous constraint in an OCL clause as well as the syntax verification.

4.1.2. Converting relationship constraints to OCL module

According to our approach, relationship constraints include multiplicity and generalization constraints. Because these constraints are defined in the class diagram, the user does not need to redefine them using OCL. OCL2Trigger is able to transform relationship constraints already included in an object model to OCL clauses automatically and, save them in that model. This facilitates

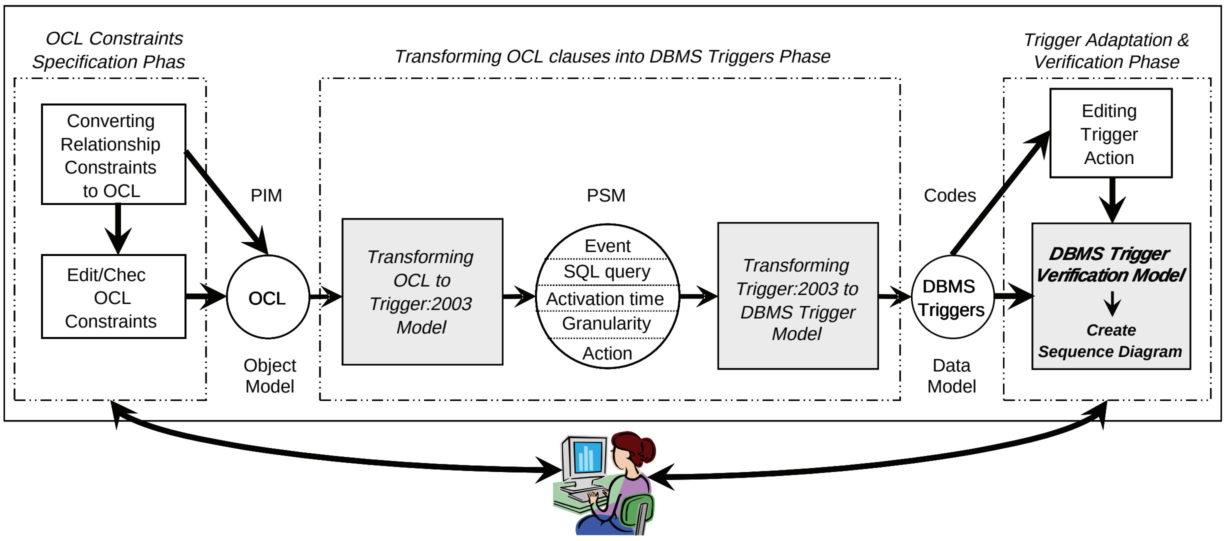
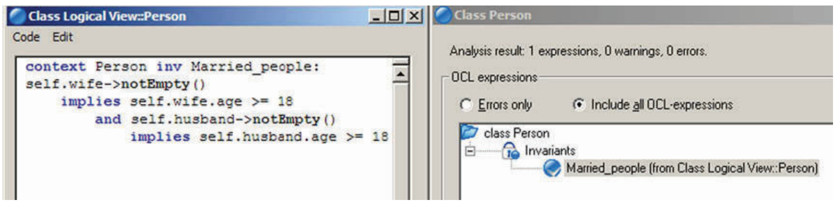

OCL Constraints Specification Phas
Transforming OCL clauses into DBMS Triggers Phase
Trigger Adaptation & Verification Phase
Converting Relationship Constraints to OCL
Edit/Chec OCL Constraints
PIM
OCL
Object Model
PSM
Transforming OCL to Trigger:2003 Model
Event
SQL query
Activation time
Granularity
Action
Transforming Trigger:2003 to DBMS Trigger Model
Codes
DBMS Triggers
Data Model
Editing Trigger Action
DBMS Trigger Verification Model
Create Sequence Diagram

Fig. 6. OCL2Trigger Architecture.


Class Logical View::Person
Code Edit
Class Person
Errors only
Include all OCL-expressions
class Person
Invariants
Married_people (from Class Logical View::Person)

```
context Person inv Married_people:
self.wife->notEmpty()
    implies self.wife.age >= 18
        and self.husband->notEmpty()
            implies self.husband.age >= 18
```

Analysis result: 1 expressions, 0 warnings, 0 errors.

OCL expressions: Errors only [✓ Include all OCL-expressions]

Fig. 7. Edit/Check OCL constraints interface.

the task of analysing and processing these clauses using our trans formation models. In addition, most of these constraints require the use of OCL to define them, because it is sometimes impossible to establish the real values of these constraints, as we will show in the following example:

Fig. 8 shows the constraint associates of the class Flight which defines that “the number of passengers is less than or equal to the number of seats on the airplane that is associated with the flight”.

In this example, only the maximum multiplicity should be en forced because the minimum multiplicity is optional (0), so OCL2Trigger converts this constraint as follows:

```
context Flight inv:
passengers >size() <=*
```

Now, by using the Oclarity editor the user can replace the sym bol (*) with the real value of the maximum multiplicity, as shown below:

```
context Flight inv:
passengers >size() <= plane.numberOfSeats
```

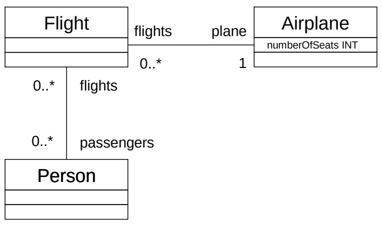


Fig. 8. UML class diagram example.

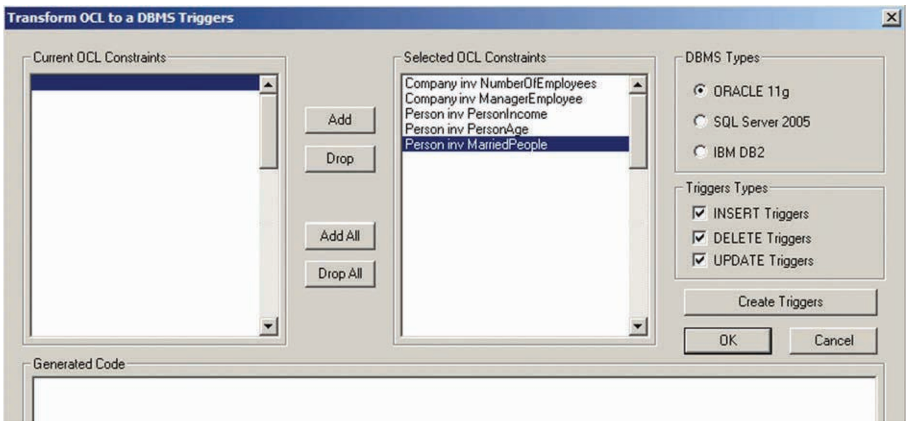
4.2. Transforming OCL clauses into DBMS triggers

Before this phase is done, Rational Rose automatically main tains the mapping between Rational Object Model and Rational Data Model where each class is mapped into a Relational table. The generalization relationship is mapped by using one table per class (Rational Software, 2000).

To generate the target DBMS trigger the interface shown in Fig. 9 is implemented. It is able to detect the specified OCL con straints in the Rational Object Model, and shows them in the list box “**Current OCL Constraints**”. The detection of the specified OCL constraints in a given model is performed according to the fol lowing algorithm:

```
CurrentOCLset() : Empty; n=0;
Set
AllClasses = RoseApp.CurrentModel.GetAllClasses();
For i = 1 to AllClasses.Count
Set theClass = AllClasses.GetAt(i);
   For j = 1 to theClass.AllOperation.Count
   Set theOperation = AllOperation.GetAt(j);
      If theOperation.Stereotype = "inv" Then
      CurrentOCLset(n + 1)=theOperation;
      End if;
Next j; Next i;
```

CurrentOCLset is a set of collection objects to represent the current OCL constraints in a class diagram. This set is initiated to empty when the algorithm is started. When it is finished, *Cur rentOCLset* contains all the OCL constraints included in the dia gram. An OCL clause is represented in Rational Rose as an operation with a stereotype. According to our approach, all OCL clauses are assigned to the stereotype <<inv>>. When *Curren*


Transform OCL to a DBMS Triggers

Current OCL Constraints

Selected OCL Constraints
- Company inv NumberOfEmployees
- Company inv ManagerEmployee
- Person inv PersonIncome
- Person inv PersonAge
- Person inv MarriedPeople

DBMS Types: [✓ ORACLE 11g] SQL Server 2005 IBM DB2

Triggers Types: [✓ INSERT Triggers] [✓ DELETE Triggers] [✓ UPDATE Triggers]

Generated Code

Fig. 9. Transforming OCL clauses into DBMS Triggers interface.

tOCLset is calculated a new set of chosen OCL constraints. *UserOCLset* is created since users can choose any constraints of a class diagram to be enforced. The list box "**Selected OCL Constraints**" of the interface shows the selected constraints list. It contains one or more constraints. The "**DBMS Types**" shows the target commercial database systems included in our approach (ORACLE 11g, MS Server 2005, and DB2). The "**Trigger Types**" check box represents the trigger types to be required, where the user can choose to generate triggers for controlling one or more types of events. For example, if the user chooses only the DELETE Trigger option this means that OCL2Trigger will generate only triggers for deleting events.

Once *UserOCLset* list is chosen, a target DBMS is specified, and triggers type events are selected the user can obtain the generated triggers by clicking on the '**Create Triggers**' button. The following algorithm shows how can UserOCLset is processed to obtain the corresponding triggers.

```
For i = 1 to UserOCLset.Count
Set theOCL = UserOCLset.GetAt(i);
Trigger2003Components(theOCL,    Constraint_name,
SQLquery, Context_table(), Related_table, Event(),
Time, Granularity, Action);
TriggersTemplates(DBMStype, TriggersTypes());
Next i;
```

Once the parameters values are calculated by the *Trigger2003Components* function (see Section 3.2), these values, the selected DBMS, and the required trigger events are submitted to the *TriggersTemplates* function. This function is used to call the corresponding trigger template to be executed. Finally, the generated triggers are plugged into their corresponding class into the Rational Data Model, and are then saved in a text SQL file, which is ready to be directly submitted to the corresponding DBMS.

4.3. Trigger Adaptation and Verification

Although sometimes users need to issue a predefined action to enforce constraints, currently OCL invariants do not support the specification of any action in a declarative manner. In this paper, we propose that incorporating actions into the generated triggers will make our approach more user efficient. Incorporating actions to triggers by users has advantages, but also adds complexity. The advantage is that it makes our approach more flexible by having the ability to incorporate more semantics, whilst the complexity is produced because the execution of these actions may leads to undesired and uncalculated behaviours. Therefore, in this section, we show how the user can incorporate actions into the generated triggers and how he/she can verify the execution of these triggers. This phase contains the following modules.

4.3.1. Editing trigger action module

To incorporate actions to a trigger body we use IBM Rational XDE Developer (IBM Rational, 2007) that provides a Table Specification for triggers to allow users to create user defined triggers or to modify an existing trigger body to enforce business rules in the database. For example, let us consider the constraint "the salary of an employee must be >= 1200 and <= 1800". According to our approach, OCL2Trigger transforms these constraints into a trigger to reject any value that is less than 1200 or more than 1800. Nevertheless, by using this module, the user is able to add or replace the rejection action by another one such as fixing the salary at 1200. In this case, the user is able to access the trigger body and introduce an action to fix the salary of the employee to the specified value.

4.3.2. Create sequence diagram module

In Section 3.3, the generic algorithm for detecting non termination problem in triggers execution has been explained. Now in this section we will explain how this algorithm has been mapped to the sequence diagram in OCL2Trigger tool. To verify trigger execution, UML sequence diagrams are used to show the interaction between objects and events in a sequential time order depending on their occurrences. Sequence diagrams allow users to create a visual representation of a scenario. These are two dimensional diagrams, where the vertical dimension is the time axis, and the horizontal dimension shows object roles in their interactions. In this approach, a scenario diagram is created for each event that may be issued to modify a table, and the cascade events that may follow after that event. Therefore, for each table in the data model three scenario diagrams are created, one for each DML statement.

If large database schemata with many tables are considered in the proposal then the most interesting sequence diagrams to the user are those in which a non termination problem is detected. Therefore, our OCL2Trigger tool detects these diagrams and shows them to the user to make the task of triggers development easier.

Applying UML sequence diagram notations to map trigger execution is explained as follows:

- *Tables*: Tables are represented in Rational Rose as a stereotype of an object instance. The scenario diagram contains one or more object instances that have the behaviour shown in the diagram.

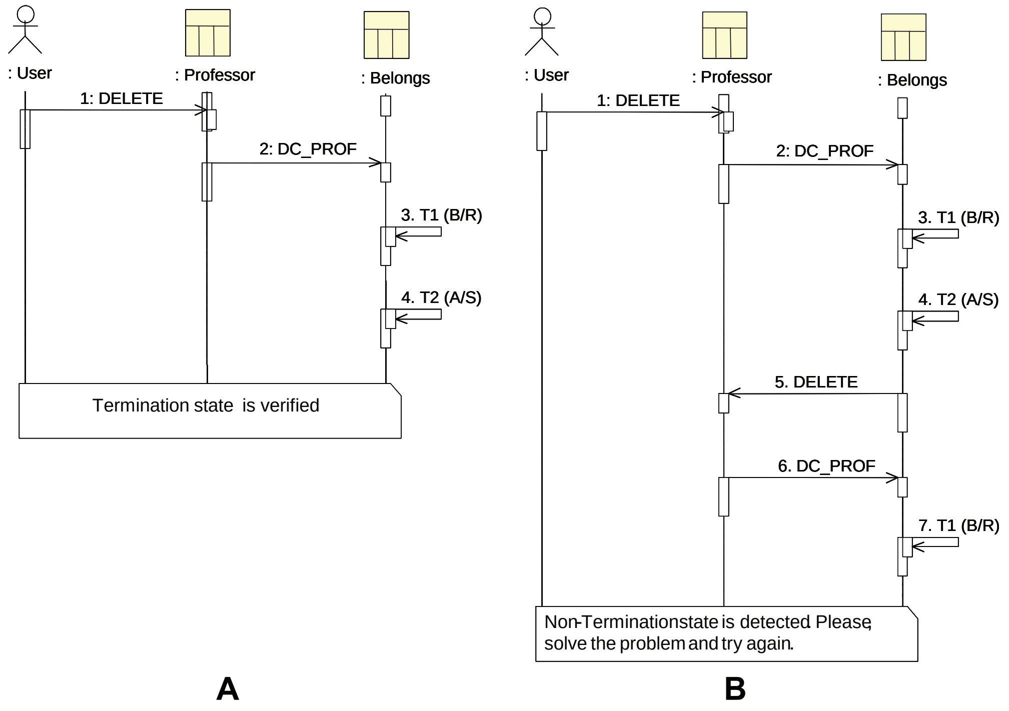


Fig. 10. Sequence diagrams to detect non-termination problem.

A table has three relevant basic behaviours for static analysis of termination: these behaviours are Insert, Delete, and Update. An object instance has a lifeline, which represents the existence of the object over a period of time.

- *Trigger/Message*: Messages in a sequence diagram are methods or operations, which are used to illustrate the object behaviour. A message is a communication carried between two objects to define the interaction between them. A message is represented in the sequence diagram by using the message icon connecting a sender object lifeline together with a receiver object lifeline. The message icons appear as solid arrows with a sequence num ber and a message label. The first message always starts at the top of the diagram and other messages follow it. When **theSender** is equal to **theReceiver**, this means that the **theSender** object is sending a message to itself, **MessageToSelf**. Each message is associated with an integer number that shows the relative posi tion of the message in the diagram. For example, if **theSequence = 3**, this message is the third message in the diagram. On the other hand, triggers that are associated to a table fire when that table is modified. When a trigger queries or modifies the related table, it is exactly the same as when an object sends a message to itself. Therefore, in our approach a trigger is repre sented in a sequence diagram as a message from the sender to itself, i.e. **theSender = theReceiver**, which means that this trig ger is represented as **MessageToSelf**. The trigger name is included into the message icon. Before triggers and After trig gers are represented by using the same notation **MessageToSelf**. The notation Message is used for other operations related to the behaviour of triggers, as shown in the following:
  - Direct events such as (Insert, Delete, and Update (Attribute)) in this case, **theSender** $\neq$ **theReceiver**.
  - Indirect events (referential constraint actions). These actions are represented in the sequence diagrams as messages from a referenced table (**theSender**) to a referencing table (**theReceiver**). The event types are indicated on the message icon.
- ***Note***: We use notes to warn users about the results of the veri fication. Our tool represents two types of notes to users. The first is "***Termination state was correctly verified***" which is sent when the execution of a given scenario is correctly terminated. The second note is "***Non termination state was detected. Please, solve the problem and try again***". This note is sent when the verification of a scenario detects a non termination state in the execution of triggers.

To show the mapping of the generic algorithm (Fig. 5) to se quence diagrams, an example is presented. In this example, **Professor** is a referenced table while **Belongs** is a referencing table. Each table is represented in the sequence diagram as an object instance with a lifeline see Fig. 10. The object instance User is used to represent the point at which the initial event is started. The referential action between these two tables is On Delete Cascade (**DC_PROF**). **T1** and **T2** which are two triggers for delet ing are defined on **Belongs**. In this example, we will apply only the DELETE event as an example in two scenarios illustrated below.

4.3.2.1. Scenario 1: (Fig. 10A). The scenario is begun when the *Actor* issues a direct event to delete from the table *Professor*. The map ping is started by calling the function *Sub Main*() for the pair (*Pro fessor,DELETE*). In this function the triggering graph vector *TG*() is reinitiated and the *DELETE* event is applied to *Professor*. This event is represented in the sequence diagram as a solid arrow with a message label (1:*DELETE*). The function *Sub TMapping*(*Professor, DELETE*) is called. It immediately calls the other function *Sub TriggersSet*(*Professor,DELETE,Before*) to calculate *TG*(). Because there is not any Before trigger applied to *Professor* the function *Sub Trigger Set* is finished, and *TG*() is returned empty. At this point, the algo rithm needs to check whether there is a cascade event produced by (1:*DELETE*) or not. In this case, the referential action *DC_PROF* is the cascade event on *Belongs*. It is represented in the sequence diagram as a solid arrow with a message name (2:*DC_PROF*). The function *TriggerSet*(*Belongs*, *DELETE*,*Before*) is called again to calculate the new *TG*(). In this case, *Belongs* has one Before trigger which is rep resented in the sequence diagram as a *MessageToSelf* with a mes sage label (3:*T*1(*B*/*R*)). If *TG*() has an instance of *T*1(*B*/*R*) then a non termination state is detected and a message is sent to the user. In any other case, a new instance in *TG*(1) = {*T*1(*B*/*R*)} is created. According to the SQL:2003 recommendation Before triggers is used to read from a database or to correct an error produced in the processing of data input (see section, **Activation Time**) there fore, we do not need to check whether the action of *T*1(*B*/*R*) may produce new events. The event *cascade*(*DELETE*) is applied to *Be longs*. Then the function *TriggersSet*(*Belongs*,*DELETE*,*After*) is called to calculate the new *TG*(). There is only one After trigger defined on *Belongs:* (4:*T*2(*A*/*S*)) represents this trigger. If *TG*() has an in stance of *T*2(*A*/*S*) then a non termination state is detected, other wise a new instance in *TG*(2) = *T*2(*A*/*S*) is created. If a new event is issued from the action of *T*2 the algorithm should be repeated again calling the function *TMapping* with the pair (new event, ta ble) as parameters. If there is not any cascade event in the trigger action then the algorithm is finished and the termination is verified by sending a message to the user.

4.3.2.2. Scenario 2: (Fig. 10B). In this scenario the action of *T* is mod ified to incorporate the event *DELETE* from *Professor*. In order to avoid repetition, in this scenario, the sequence of operations is sim ilar to the previous one until it reaches the message (4:*T*2(*A*/*S*)). Until now, the trigger set *TG*() has two instances {*T*1(*B*/*R*), *T*2(*A*/*S*)}. Because the action of *T*2 has the new event *DELETE* from *Profes sor,* the function *Sub TMapping*(*Professor*,*DELETE*) is called again which immediately calls *Sub TriggersSet*(*Professor*,*DELETE*, *Before*) to calculate the new instances in *TG()*. Because there is not any Be fore trigger applied to *Professor* the function *Sub TriggerSet* is fin ished, and *TG*() is returned with only the previous instances. At this point, the algorithm checks again whether there is a cascade event produced by *(5:DELETE)* or not. The referential action (6:*DC_PROF*) is executed on *Belongs*. Then the trigger (6:*T*1(*B*/*R*)) is fired and added to the triggers set as *TG*(3) = *T*1(*B*/*R*). Now, the *TG*() has three instances {*T*1(*B*/*R*), *T*2(*A*/*S*),*T*1(*B*/*R*)} this means that there are two instances which have been applied to the same ob ject. In this case, a non termination state is detected and a message is sent to the user to warn him about the existence of this problem. When a non termination is detected the mapping is finished immediately.

5. Conclusions

Although the database CASE tools have been developed to re solve the database modelling problem and to provide automatic processes to develop all phases supported in a database methodol ogy, the current state of these tools is that they provide conceptual models with more abstraction and are concerned with expressing the semantics of the real world more accurately. However, the move from the conceptual level to the logical level is not supported by these tools, and the generated code needs to be modified to comply with the requirements of the real world.

It is true that various studies have lead to important results such as the creation of the current commercial CASE tools and some research prototypes to support maintaining mechanisms to preserve integrity constraints in the logical models. Nevertheless, in the context of Relational databases we consider that current practice is below the needs of the requirements of active technol ogy. These requirements need to have a verification process which is considered as important as development.

On the other hand, although the Relational database has been widely used in the commercial DBMS and the most important commercial Object Oriented database systems utilize the Rela tional tables to store objects, we consider that most proposals have been developed to respond to the needs of Object Oriented dat abases development.

Therefore, to fill in some of the gaps that the current CASE tools leave during the development of active Relational Databases, we present the OCL2Trigger tool as a support to the theoretical ap proach which follows the phases proposed in the MDA software development, by completely transforming the OCL constraints into triggers. These phases are as follows: specifying OCL constraints in the UML class diagram, transforming the OCL constraints into SQL:2003 standard triggers, transforming the standard triggers into target DBMS triggers. In addition, this tool can represent and verify trigger execution by using UML sequence diagrams. Thus, this work unites the UML aspects that are widely accepted and is supported by many CASE tools for aspects of Relational databases that have wide presence in commercial DBMS.

Our approach has some limitations which are explained as fol lows: (a) although we believe that applying MDA makes the trans formation of any type of OCL constraints to triggers easier, currently the OCL2Trigger tool supports only the OCL invariant constraints. Specifically, three patters have been proposed: attri bute value constraints, multiplicity constraints and generalization constraints. Other types of constraints such as aggregations and compositions, pre conditions, and post conditions will be included in future work; (b) Including complex OCL expressions in which many relations are involved may result a difficult task to generate triggers. We think that this limitation could be solved by incorpo rating more patterns to our approach to cover such expressions. The article presents a first effort to check the viability of this ap proach through three of the most widely used constraints in the conceptual model; (c) The triggers execution analysis focuses only on detecting the non termination problem and the user himself needs to redefine and reconstruct triggers definition to verify the termination. We think that this could be a limitation especially for users without experience in triggers implementation. Thus, a part of our future work will be apart from detecting the non termi nation problem trying to provide some alternatives for the solu tion. (d) The user needs to define the OCL constraints, which can not be directly specified in the graphical model, manually into the corresponding class by using the Oclarity editor. This task re quires experienced users in OCL although the Oclarity editor could perform syntactic verification. Therefore, we think as future work incorporating a new module to make easier the transformation of the CIM (Computation Independent Model) of the constraints spec ification to PIM.

Our approach makes it easier for the database developer to gen erate maintaining mechanisms directly from the generation of the schema in question. Moreover, when the integrity constraints of this schema are modified, the corresponding triggers are also auto matically modified. Using this approach, the developers will obtain both the best system performance because active mechanisms are implemented as part of the database schema rather than in the application, as well as the best data independence because the integrity constraints are also embedded in the database schema rather than in external applications.

Furthermore, we will design experiments to validate our tool. These experiments focus on showing the usefulness of using it to facilitate maintenance and design tasks. Therefore, we propose two kinds of experiments: the first concerns the usefulness of checking semantics with triggers. The second is concerned with the user interface showing triggers and sequence diagrams. We want to know whether the designer understands the proposed dia grams and detects what each one does.

References

Al-Jumaily, H.T., 2006. Active technology application to control constraints in database development. Ph.D. Thesis, Carlos III University of Madrid, Spain.

ArgoUML, 2007. <http://argouml.tigris.org/>.

Baralis, E., Widom, J., 2000. An algebraic approach to static analysis of active database rules. ACM Transactions on Database Systems 25 (3), 269–332.

Baralis, E., Ceri, S., Widom, J., 1993. Better termination analysis for active databases. In: Proceedings of the First International Workshop on Rules in Database Systems, Edinburgh, Scotland, pp. 163–179.

Birgit, D., Heinrich, H., 1999. Using OCL constraints for relational database design – The unified modeling language. In: Proceedings of the Second International Conference, LNCS 1723. Springer, pp. 598–613.

Budgen, D., Thomson, M., 2003. CASE tool evaluation: experiences from an empirical study. Journal of systems and software 67 (2), 55–75.

Cabot, J., Teniente, E., 2006. Constraint Support in MDA Tools: A Survey, Model Driven Architecture – Foundations and Applications, LNCS, pp. 256–267.

Ceri, S., Fraternalli, P., 1997. Designing Database Applications With Objects and Rules: The IDEA Methodology. Addison-Wesley.

Ceri, S., Widom, J., 1990. Deriving production rules for constraint maintenance. In: Proceedings of VLDB Conference IBM Almaden Research Center.

Ceri, S., Fraternali, P., Paraboschi, S., Tanca, L., 1994. Automatic generation of production rules for integrity maintenance. ACM Transaction on Database Systems 19 (3).

Chen, P., 1976. The entity-relationship model – toward a unified view of data. ACM Transactions on Database Systems 1 (1).

Cuadra, D., Nieto, C., Castro, E., Martinez, P., Velasco, M., 2003. Preserving relationship cardinality constraints in relational schemata. Database Integrity: Challenges and Solutions. Ed: Idea Group Publishing.

Decker, H., Martinenghi, D., Christiansen, H., 2006. Integrity checking and maintenance in relational and deductive databases and beyond. In: Intelligent Databases: Technologies and Applications, Idea Group, pp. 238–285.

Domínguez, E., Lloret, J., Zapata, M.A., 2002. Integrity constraint enforcement by means of trigger templates. In: Second International Conference, Advances in Information Systems, ADVIIS, 2002.

Elmasri, R., Navathe, S., 2000. Fundamentals of Database Systems, Third ed. Addison-Wesley.

EmPowerTec, 2006. <http://www.empowertec.de/products/rational-rose-ocl.htm/>.

Gogolla, M., Richters, M., 1998. Transformation rules for UML class diagrams. The Unified Modeling Language, UML'98 – Beyond the Notation. In: First International Workshop, Mulhouse, France.

Hickey, T., 2000. Constraint-based termination analysis for cyclic active database rules. In: Proceedings of the Sixth International Conference on Rules and Objects in Databases, LNAI, vol. 1861. Springer, pp. 1121–1136.

IBM Rational, 2007. XDE Developer. <http://www-306.ibm.com/software/awdtools/developer/rosexde/>.

ISO/IEC 9075 Standard, 2003. Information Technology – Database Languages – SQL:2003 International Organization for Standardization.

Kulkarni, K., Mattos, N., Cochrane, R., 1998. Active Database Features in SQL3, Active Rules in Database Systems. Springer-Verlag, New York. pp. 197–218.

MetaEdit+, 2007. <http://www.metacase.com/>.

Objecteering/UML, 2007. Objecteering/SQL Designer User Guide Version 5.2.2. <http://depinfo.u-bourgogne.fr/docs/Objecteering522/SQLDesigner.pdf>.

OCL22SQL, 2007. Dresden OCL Toolkit. <http://dresden-ocl.sourceforge.net/>.

Olivé, A., 2003. Integrity constraints definition in object-oriented conceptual modeling languages. In: Conceptual Modeling – ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA.

OMG, 2007. Object Management Group, Inc. <http://www.omg.org/mda/>.

Oracle, 2007. Oracle® Database SQL Developer User's Guide <http://download.oracle.com/docs/cd/B19306_01/appdev.102/b31695/dialogs.htm#BACIGFBJ>.

Paton, N., Díaz, O., 1999. Active Database Systems. ACM Computing Surveys 31 (1).

Rational Enterprise Edition, 2003. <www-306.ibm.com/software/rational/>.

Rational Software, 2000. Mapping Object to Data Models with the UML Mapping Objects to Relational Databases. <http://www.uml.org.cn/oobject/tp185.pdf>.

Teorey, T.J., 1999. Database Modeling and Design, third ed. Morgan Kaufmann Series in data management systems.

Türker, C., Gertz, M., 2000. Semantic Integrity Support in SQL-99 and Commercial (Object) Relational Database Management Systems. UC Davis Computer Science Technical Report CSE-2000-11, University of California.

Verheecke, B., Straeten, R., 2003. Specifying and implementing the operational use of constraints in object-oriented applications. In: Proceedings of TOOLS PACIFIC 2002, vol. 10, p. 23.

Visual Case Tool, 2007. <http://visualcase.com/index.htm/>.

Wahler, M., Koehler, J., Brucker, A., 2006. Model-Driven Constraint Engineering, Workshop on OCL for (Meta-)Models in Multiple Application Domains (OCLApps), Models 2006.

Harith T. Al-Jumaily. Since 1999, he has worked at the Advanced Databases Group in the Computer Science Department at the Universidad Carlos III of Madrid. In 2006, he obtained a Ph.D. in Information Science from the Universidad Carlos III of Madrid. He is currently teaching File Structure and Database Design. His research interests include Advanced Database Technologies, Information Retrieval and Software Engineering.

Dolores Cuadra received the M.Sc. in Mathematics from the Universidad Complutense of Madrid in 1995. Since 1997, she has worked as assistant lecturer at the Advanced Database Group in the Computer Science Department of the Carlos III University of Madrid. In 2003 she obtained a Ph.D. in Computer Science from the Carlos III University of Madrid. She is currently teaching File Organization, Database Design and Data Modelling. Her research interests include data models, conceptual and logical modelling and Advanced Database CASE environments. She has been working in the Computer Science Department at Purdue University of West Lafayette (Indiana) for nearly a year, where she has applied her research in Spatio-Temporal databases.

Paloma Martínez Fernández obtained a degree in Computer Science from the Universidad Politécnica of Madrid in 1992. Since 1992, she has been working at the Advanced Databases Group in the Computer Science Department at Universidad Carlos III of Madrid. In 1998 she obtained a Ph.D. in Computer Science from the Universidad Politécnica of Madrid. She is currently teaching Database Design and Advanced Databases in the Computer Science Department at the Universidad Carlos III de Madrid. She is working in several European and National research projects on Natural Language Processing, Information Retrieval, Advanced Database Technologies and Software Engineering.