

This document is published in:

*International Journal of Software Engineering and
Knowledge Engineering* 21 (2011) 5, pp. 621–645

DOI: 10.1142/S0218194011005426

© 2011. World Scientific Publishing Company

AN OCL-BASED APPROACH TO DERIVE CONSTRAINT TEST CASES FOR DATABASE APPLICATIONS

DOLORES CUADRA[†], HARITH AL-JUMAILY[‡],
ELENA CASTRO[§] and MANUEL VELASCO[¶]

*Computer Science Department
Carlos III University of Madrid*

Av. Universidad 30, 28911 LEGANES, Madrid, Spain

[†]dcuadra@inf.uc3m.es

[‡]haljumai@inf.uc3m.es

[§]ecastro@inf.uc3m.es

[¶]velasco@ia.uc3m.es

Abstract: The development of database applications in most CASE tools has been insufficient because most of these tools do not provide the software necessary to validate these applications. Validation means ensuring whether a given application fulfils the user requirements. We suggest validation of database applications by using the functional testing technique, which is a fundamental black-box testing technique for checking the software without being concerned about its implementation and structure. Our main contribution to this work is in providing a MDA approach for deriving testing software from the OCL specification of the integrity constraints. This testing software is used to validate the database applications, which are used to enforce these constraints. The generated testing software includes three components: validation queries, test cases and initial data inserted before the testing process. Our approach is implemented as an add-in tool in Rational Rose called OCL2TestSW.

Keywords: Functional testing software; software validation; equivalence class testing; MDA; CASE tools.

1. Introduction

The introduction of the MDA approach (Model-Driven Architecture) [1] in Software Engineering has provided good support and consolidation for automatic code generation. MDA focuses on using models to cover the life cycle of software development. MDA is suitably used to solve the interoperability problem between heterogeneities systems with different implementation platforms.

However, a limitation of MDA is that it does not provide a way to validate the software. Thus, our approach focuses on deriving test cases from the specifications of OCL clauses in a class diagram. These test cases are used to ensure that the database applications employed to enforce these specifications fulfill the user requirements.

In the context of databases, MDA is a very ambitious task and would help make database developers' jobs easier. MDA is adopted in many Computer Assisted Software Engineering (CASE) tools to transform conceptual models into logical models. These tools aim to assist database developers in different design phases. Most of these tools generate applications from the specification of a given schema and its constraints. However, these tools ignore one of the fundamental issues in the software development field, software validation, which aims to ensure that the generated database application fulfils the user requirements exactly.

The validation of generated applications is one of the most important areas in software engineering; this validation checks if the requirements are reflected in the generated applications. Thus, software testing is carried out not only to detect errors but also to show that a program performs its intended functions correctly, as well as to increase confidence that a program is doing what it is supposed to do [15].

A database application consists of a database and a set of operations that modifies and interacts with the former. Database applications are defined in [2] as operations which are developed by integrating domain-specific language such as SQL with general-purpose programming languages. Our research group is interested in making this development task easier through the automatic transformation of Object Constraints Language (OCL) specification into enforcement mechanisms. The OCL is used to specify integrity constraints in the UML class diagram. In the development task we have detected that automatic generation of database applications should be tested to ensure the correctness of the constraints' transformation.

Accordingly, in this paper we present an approach to generate testing software to ensure the correctness of the generated applications. This approach is implemented as an add-in tool in Rational Rose called OCL2TestSW.

The MDA is used in our approach to transform OCL specifications to test software which consists of three mechanisms: test cases, validation queries, and initial data. The functional testing software approach, also called black-box testing, is used. The main objective of this type of testing is to ensure that the tested applications fulfill the user requirements without being concerned about implementation and structure.

The rest of this work is organized as follows. In Sec. 2, related works are presented. In Sec. 3, the MDA adaptation for our approach is explained. Section 4, explains how the OCL2TestSW tool is designed and implemented. Section 5 shows the evaluation results of our approach and discusses obtained results. In Sec. 6, some conclusions and future works are presented.

2. Related Works

Our work deals with database CASE tools to generate testing software components from the specification of UML/OCL constraints associated with a database schema. These components will be used in the validation of the database applications which are used to enforce these constraints. Thus, in this section we are interested in studying some CASE tools that have been widely used in database development and some interesting works related to software validation.

In the past decade, diverse attempts have been made to resolve the database-development problem, one of which is the use of database CASE tools. The main contribution of these tools is that they provide automatic processes to carry out all phases supported in a database methodology. In fact, the current state of these tools shows that they provide automatic and graphical user interfaces to reduce manual work and to make decision-making easier. One of the most important phases that should be supported by these tools is the transformation of integrity constraints into enforcement mechanisms. Some CASE tools such as ArgoUML [3], MetaEdit+ [4], Objectteering/SQL [5], OCL2SQL [6], Rational Enterprise [7], and Visual Case Tool [8] support this type of enforcement mechanisms. However, the total support of development software is missing in these tools because none of the mentioned tools support any way of validating the generated applications. Because of this, tool support is needed to reduce the development, testing and maintenance effort [9].

Although testing generated software by a CASE tool could be contradictory, in general, the CASE tools for the development of databases generate a subset of very simple specifications which must be extended by the designer, or when the specifications that they generate do not work when they are run in a specific DBMS (the SQL code is compiled but it does not work). This is one of the main reasons for our proposal: to show designers that the diagram of classes along with their constraints in OCL is transformed into a correct code.

In recent years a wide range of traditional software testing techniques have been proposed, although relatively little effort has been spent on the testing and analysis of database applications [10]. In [11], the issues which make testing database applications different from other types of software have been identified, and a tool has been developed for populating a database with meaningful data which satisfies constraints. The cited work considers only the constraints which can be expressed in the database schema by using SQL's Data Definition Language. Our work agrees with the issues in testing database applications presented in the cited work. Nevertheless, the difference between the two works is that in our work the testing software is derived from the constraints which are expressed in a UML class diagram. These constraints can be a basic type such as (primary key, foreign key, and unique key) and a specification of OCL clauses. Our approach joins the UML in aspects that have been widely accepted and supported by many CASE tools in the software validation field.

In [12], static analysis techniques (also called white-box techniques) have been used to extract useful information in a web database application. Static analysis testing is a fundamental structural technique that depends on software structure and implementation. That is, the white-box approach has been used to construct an application graph which systematically generates selected paths based on that graph. Each path represents a possible scenario of the use of the application. A test case is organized as an XML file and is automatically executed. In [2, 10, 13], the white-box testing technique is also applied to validate a database application. The main objective is to validate all possible paths in the execution flow of a program. By means of these paths, all the program statements are executed and examined at least once. In [24], a white box method for automatic generation of database instances is proposed. The input of this method consists of an SQL statement, database schema, and assertions to define the user requirements. The output is a set of constraints to validate the desired database instances.

Applying black-box techniques could be more simple and practical, especially when there are an enormous number of interactions with a database. An example of the difficulty in applying white-box techniques is the case of the validation of active database applications. Each trigger has an independent structure, but in execution time it is very difficult to detect all possible paths to be validated. Thus, using the equivalence-class testing technique could make the validation of database applications easier. The equivalence-class testing is a fundamental technique for functional testing which checks software without being concerned about its implementation and structure [14]. In this type of testing, the tester's responsibility is to provide input or initial data and to validate the output results. The main objective is to check whether the tested software fulfills the user requirements or not.

AGENDA [22] is another important work in this area. It is a tool that generates and executes tests for state and transaction constraints. This means that it works with a limited subset of integrity constraints. According to the presented results it is not effective 100% of the time, and temporary tables and triggers need to be created to check if the constraints are verified.

Our approach starts from the specification of constraints in the design phase, which are then implemented according to the chosen database system (SQL Server, Oracle, MySQL). Not only are the basic constraints of the relational DBMS considered (primary keys, unique or foreign key), but also those constraints which are more complex and which can be implemented by triggers (see OCL2Trigger [19]). The database and the generated constraints are tested with the tool presented in this paper. The testing responsibility is to show that the OCL specifications modeled in the UML classes diagram by the user are correctly transformed through the tool OCL2Trigger.

The works related to database applications and software testing, in general, do not consider the integrity constraints which are implemented inside the database.

They analyze the modification operations and the queries which are carried out by external applications and they study if these operations leave the database in a consistency state. Most of the analyzed works do not include testing in the design phase, making it difficult to model both the applications and database.

Our approach is applied in the early phases of development and considers the constraints inside the database (trying to remove the constraints validation from the applications). Once the database is created in our approach, including tools such as AGENDA would be a good complement for analyzing the set of applications and thus achieve a robust database.

Another drawback is that some commercial CASE tools, focusing on the development of databases, generate a code that is not correct. For example, ERWIN [25] provides a set of triggers to validate constraints generated from the transformation of a conceptual schema, such as the primary key in the relational tables, but they cannot be run because the DBMS code is not correct. For this reason, the generated code testing instills confidence in the designers for its use, and that is the principle motivation behind this work.

3. MDA for Deriving Testing Software

The main objective of deriving testing software from the OCL specifications is to ensure that the database applications used to enforce these specifications fulfill the user requirements. The OCL specifications can be enforced by using many types of applications such as SQL procedures, Java application, SQL triggers, Java triggers, etc. There is a clear need for software techniques to test the complicated interaction between applications and database [10]. Because of that, applying the functional software testing approach (also called black-box testing) can help database developers to validate any type of application in an efficient way.

The tester chooses better test cases when their executions contribute a certain confidence in detecting possible defects in the application [15]. For this reason, it is advisable to obtain high coverage for all possible cases in software testing.

In this section, we will explain our approach for deriving functional testing software from the OCL specifications according to the MDA framework. MDA aims to achieve complete (semi-) automatic software development phases [1]. There are three phases for adaptation of MDA to our approach (see Fig. 1).

The first one, called PIM, represents the logical view of the specification of integrity constraints using OCL clauses in a class diagram. The second one, called PSM, describes the technology used to build the necessary components of the testing software. In this phase, the SQL notation is used to refer to the recent 2003 standard [16]. The standard SQL is used to describe these components because we focus on testing relational database applications. The third phase is called Testing Software which describes the software technology which can be applied directly in a target commercial DBMS such as Oracle, DB2, and MS Server. Two models are used to carry out the transformation rules between these phases

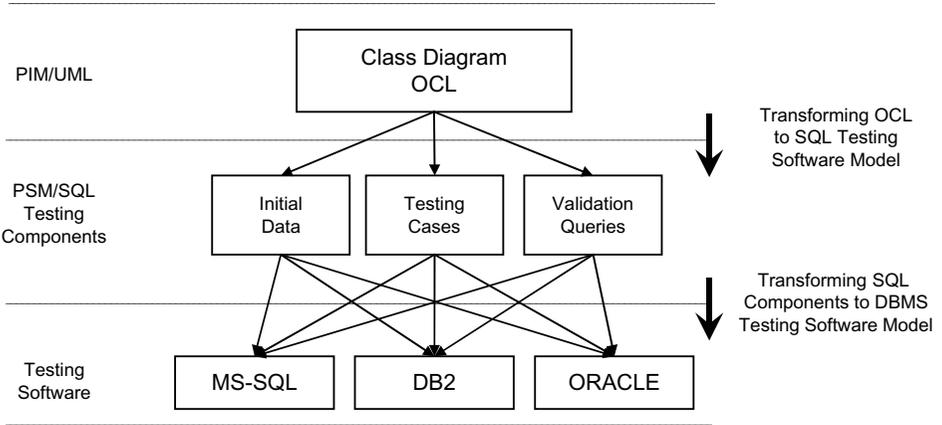


Fig. 1. Applying MDA to our approach.

automatically: *Transforming OCL to SQL components Model*, and *Transforming SQL components to DBMS testing software Model*. In the following we present a brief description of these models.

3.1. Transforming OCL to SQL testing software model

According to Fig. 1, each OCL constraint is transformed into three testing software components. Each one of these components is related to an issue in testing database application. These issues are defined in [11] as specifying database states, applying test cases, and observing database state after test execution. In [2], these issues are defined as test cases generation, test data preparation and test outcomes validation. According to these issues, the testing software components are defined in our work as *initial data* to be inserted before the testing is begun, *test cases* to be applied for testing applications, and *validation queries* to ensure the correctness of the tested application. A detailed description for each one of these components is shown in the next subsection.

3.1.1. Validation queries

The first step of our approach is the OCL clauses transformation into validation SQL queries. These queries are used to check the correctness of the integrity constraints after each test case is run. Because the SQL queries syntax in commercial DBMS depends on the particular characteristics of each system, we transform these clauses into the standard SQL queries before the final commercial DBMS queries are derived.

Integrity constraints, also called semantic constraints or user constraints, are used to define the business rules of the universe of discourse in a systematic manner.

Database modifications are rejected whenever the database final state does not fulfill these constraints. Currently the OCL2TestSW tool considers OCL invariants specification although the MDA framework can be adopted to transform any type of OCL constraints. In future work, our proposal will include other types of OCL expressions such as pre- and post-conditions. An invariant constraint is an OCL expression that can be associated with a class, a type or interface in a UML class diagram [1]. It must be true for all instances of element type at any time. The OCL invariant constraints are also used to specify relationship constraints in a class diagram such as multiplicity, generalization, etc., although these constraints are already included in the diagram [17]. The formal definition of an OCL invariant is shown in the following:

Context $\langle A \rangle$ inv $\langle \text{constraint_name} \rangle$:
 $\langle \text{OCL_expression (self)} \rangle$

A is a class name. Self is an instance of a type (e.g. Company). The Context specifies the class in which an OCL expression is defined. The OCL_expression is a logical expression that describes a relationship between one or more atomic expressions. An atomic expression contains no more than one operator. 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 (Θ) such as \wedge (and), \vee (or), \neg (not) are mapped using counterpart expressions of SQL. An OCL expression is used to specify a condition on objects, so if this condition cannot be satisfied, we need to abort the transaction which leads to inconsistency in the database state. For example, a business rule might be “the number of employees of a company must always exceed 50”. This constraint is transformed to a SQL query to detect whenever the violation has occurred. A simple way to detect the violation of any constraint is applying the negation “ \neg ” of the original constraint when the corresponding SQL query is implemented. If this query returns results then it means that there is at least one instance that does not fulfil the original constraint specification. For example, for the previous constraints, if the number of employees of a company is equal to or less than 50, then it means that the constraint is violated.

Relational Algebra notations are used in this paper to introduce abbreviations to simplify and make the theoretical results of our approach more readable. However, in practice the transformation of an OCL expression into a SQL standard expression is automatically performed. This transformation is a significant phase in our approach because this expression restricts the attribute values and the relationships between them. Hence, we perform this transformation first, and then proceed with the following phases of our approach. It is done according to the OCL expressions syntax where each one of these expressions is transformed into one SQL expression ($\wp_i \Rightarrow \sigma_i$), where \wp is an OCL expression, σ is a negation SQL expression corresponding to \wp , and i is one of the following rules to specify the mapping.

Rule(a) \wp_i is defined on a target attribute in a class.

$$\wp_a(A.x) \Rightarrow \sigma_a(A.x)$$

$$\text{The mapping: } \langle \text{self.x } \Theta V_1 \rangle \Rightarrow \sigma_{(A.x \neg \Theta V_1)}$$

where self.x and Θ are substituted by the target attribute name and the counterpart operator of SQL, respectively. V_i is a basic type value (e.g. Boolean, Integer, Real and String) which is substituted by the real value of the constraint, A is the context relation of the constraint, the attribute $x \in A$, and $\neg \Theta$ is the negation of the original operator. For example, \wp_a : “the age of an employee must be equal to or more than 24”.

Context $\langle \text{EMP} \rangle$

$$\langle \text{self.age} \geq 24 \rangle$$

Rule(b) \wp_i is defined on target attributes in one or more class.

$$\wp_b(A.x, B.z) \Rightarrow \sigma_b(A.x, B.z)$$

$$\text{The mapping: } \langle \text{self.x } \Theta \text{ self.B.z} \rangle \Rightarrow \sigma_{(A.x \neg \Theta B.z)} \text{ OR:}$$

$$\langle \text{self.x } \Theta_1 V_1 \rangle \text{ implies } \langle \text{self.B.z } \Theta_2 V_2 \rangle \Rightarrow \sigma_{((A.x \Theta_1 V_1) \wedge (B.z \neg \Theta_2 V_2))}$$

Here, “implies” can be mapped using the logical operator “AND” and the logical negation is applied on Θ_2 . This is because “implies” does not have a counterpart in SQL.

Rule(c) \wp_i is defined to specify an aggregate function on target attributes in one or more class.

$$\wp_c(A.x, \text{agg}(B.z)) \Rightarrow \sigma_c(A.x, \text{agg}(B.z)) \text{ (agg = \{MAX, MIN, AVG, COUNT, etc.})}$$

$$\text{The mapping: } \langle \text{self.b.z -> agg()} \Theta \text{ self.x} \rangle \Rightarrow \sigma_{(A.x \neg \Theta_{\text{agg}}(B.z))}$$

For example, \wp_c : “the total salary of employees must not exceed the department’s budget”.

Context $\langle \text{EMP} \rangle$

$$\langle \text{allInstances.salary -> SUM()} \leq \text{self.Department.budget} \rangle$$

Rule(d) \wp_i is defined to specify a navigation between two class.

$$\wp_d(A.a, B.a) \Rightarrow \sigma_d(A.a, B.a)$$

$$\text{The mapping: } \langle \text{self.navigation -> size()} \Theta V \rangle \Rightarrow \sigma_{(\text{count}(B.a) \neg \Theta V)}$$

OCL allows navigating from the association class A itself to another class B to specify objects which participate in that association. The arrowhead ($->$) is added to restrict the direction of the navigation. This constraint is true whenever an instance of A does not have any association in B . For example, \wp_d : “every department has at least two employees”.

Context $\langle \text{DEPT} \rangle$

$$\langle \text{self.navigation -> size()} \geq 2 \rangle$$

Rule(e) \wp_i is defined to specify `ocllsKindOf` property to determine whether A is either the direct type or one of the subtypes.

$$\begin{aligned}\wp_e(A.a, B.a, C.a) &\Rightarrow \sigma_{e.Total}(A.a, B.a, C.a) \\ &\Rightarrow \sigma_{e.Disjoint}(A.a, B.a, C.a)\end{aligned}$$

where `A.a` is a direct type instance, `B.a` and `C.a` are subtypes instance. This type of constraint specifies a generalization relationship. Although there are four possible types of generalization: Disjoint-Total, Overlapping-Total, Disjoint-Partial, and Overlapping-Partial [18], in this work only the Disjoint-Total constraint is considered because it needs more effort to be controlled than the other ones do.

- (1) A Total-constraint of a generalization relationship specifies that an object of the supertype must be a member of at least one of the subtypes.
- (2) A Disjoint-constraint specifies that objects in a different subtype from the same supertype are completely different.

The mapping: $\langle \text{self} \rightarrow \text{forall } (A.a | \text{ocllsKindOf}(B.a) \Theta \text{ocllsKindOf}(C.a)) \rangle$

$$\begin{aligned}\Rightarrow \sigma_{e.Total} &= \sigma_{(A.a \rightarrow IN(\pi_{B.a}(B) \wedge \pi_{C.a}(C)))} \\ \Rightarrow \sigma_{e.Disjoint} &= \sigma_{(A.a IN(\pi_{B.a}(B) \wedge \pi_{C.a}(C)))}\end{aligned}$$

3.1.2. Initial data

As in [11] and [12] we generate valid initial data specifically for testing, and an isolated environment is needed to run the test cases. In this section we will describe our approach to generate initial data from the specification of OCL constraints. The standard SQL will be utilized to express the DML operations which are used to insert the generated data in a database.

To insert valid initial data in a database it is necessary to satisfy the basic fundamentals of the database schema definition. These fundamentals are the attribute domain values and the integrity constraints definition. When a class diagram in a CASE tool is defined some basic constraints are defined too. For example, in a given class diagram or an object model of Rational Rose CASE tool, basic constraints such as `IsUnique`, `IsPrimaryKey`, `ForeignKeyConstraint` can be defined. In order to generate valid initial data we will take advantage of the database schema definition to describe relations, attributes, and the basic integrity constraints. For the other types of constraints we will use the OCL specifications which are defined in a given class diagram.

To begin generation of initial data, the tester needs first to define the attribute domain values. Formally, a domain is a set of atomic values, given a set of domain $(D_1, D_2 \dots D_n)$ the Cartesian product of these domains is $DOM = \{D_1 \times D_2 \times \dots \times D_n\}$. Each domain is specified using a data type. According to the standard SQL some basic data types such as `INTEGER`, `NUMERIC`, `CHAR`, `VARCHAR`, `DATE`, etc., are used.

A relation R over the set $(a_1:d_1, a_2:d_2 \cdots a_n:d_n)$ is a subset of the Cartesian product $DOM(R)$, where $d_1 \in D_1, d_2 \in D_2 \cdots d_n \in D_n$. A database schema (SCH) is a set of relations $\{R_1, R_2 \cdots R_k\}$.

We agree with [12] that it will be easier for the human when it comes to checking the test results, to work with meaningful values, rather than with randomly generated gibberish. Therefore, in our approach we use a semi-automated generation approach which allows the tester to define and correct the domains of the attribute values, as shown in the following processes.

3.1.2.1. Generating tuples according to the basic constraints

In this process, a definition text file called `DEF_tablename` is created for each relation in a data model to specify the following schema: $\{\text{attribute.type.constraint list}\}$ and $\{\text{domain values}\}$. The $\{\text{attribute.type.constraint list}\}$ is a set of the attribute names, data types such as (N: numerical, C: Character, D: Date, etc.) and the basic constraints which are defined for each attribute. The $\{\text{domain values}\}$ is a set of domain values of these attributes. The attributes list and the constraints list of a relation are generated automatically in our approach, while the tester needs to manually edit a reduced sample of the domain values for each attribute in the relation. The definition text file is automatically transformed to another text file called `TUP_tablename` which contains all tuple values related to the same table. This process is performed in the following steps:

- (a) **Create a definition text file:** Let us consider A a relation, and that the definition text file `DEF_A` contains the following information: $DEF_A = \{a_1.N.PK, a_2.C, a_3.C\}, \{*, (v_1), (u_1, u_2)\}$. This means that relation A contains three attributes $\{a_1, a_2, a_3\}$, a_1 is defined as a primary key (PK), while there are not any more defined constraints on the attributes a_2 and a_3 . The attribute names and the basic constraints are generated automatically from the data model definition.
- (b) **Add the domain values:** The user needs to edit the file `DEF_A` in order to define the domain values of these attributes. This can be done by any text file editor. Let us consider that the domain values are $\{*, (v_1), (u_1, u_2)\}$. The $(*)$ means that the primary key values of the relation will be generated in an automatic manner. A specific function is used to calculate the values of these keys dynamically. If no $*$ is used, the tester can specify the values of these keys in the file manually. The (v_1) and (u_1, u_2) are the domain values of a_2 and a_3 respectively.
- (c) **Generating the file `TUP_A`:** The `DEF_A` file is automatically transformed into the `TUP_A` which can include all the tuple values of A . The `TUP_A` file represents a reduced sample of the Cartesian product of the domain values DOM of A . This sample may be no more than a few tuples added by the tester. For example, let us consider $TUP_A = \{a_1.PK, a_2, a_3\}, \{(1, v_1, u_1), (2, v_1, u_2)\}$,

the $\{a_1.PK, a_2, a_3\}$ represents the attribute name where a_1 is a primary key. The $\{(1, v_1, u_1), (2, v_1, u_2)\}$ represents the domain values $DOM(A)=\{(v_1) \times (u_1, u_2)\}$. Where $(1, v_1, u_1)$ and $(2, v_1, u_2)$ are two tuples for the relation A. 1 and 2 are the generated primary key values. The same process is applied if a_1 is a unique value. If $DEF_A = \{a_1.N.PK, a_2.N.Unique, a_3.N.FK(B)\}, \{*, *, *\}$. Here, a_1 and a_2 have the same definition as the previous cases, while the definition of a foreign key is used to create a relationship between the referencing relation A and the referenced relation B. If $DEF_B = \{b_1.N.PK\}, \{(u_1, u_2)\}$ then the domain values file of A will contain the following information: $TUP_A = \{a_1.PK, a_2, a_3.FK(B)\}, \{(1, 1, u_1), (2, 2, u_2)\}$.

3.1.2.2. Validating the generated tuples according to the expression rules $\{a, b, c, d, e\}$

Once the tuples are generated for each relation in a database schema, the next process is to ensure automatically that these tuples do not fulfill the expression rules $\{a, b, c, d, e\}$ shown in Sec. 3.1.1 because these rules represent the negation of the original constraints. If there is a tuple which fulfils its rule, this tuple should not be inserted in the related relation. If such a tuple exists, a message will be sent to the tester indicating that.

Let us consider that the constraint σ contains one of the expression rules $\{a, b, c, d, e\}$ that is defined between the two relations A and B. The relations A and B contain the following data: $TUP_A = \{a_1.PK, a_2, a_3\}, \{(p_1, v_1, u_1), (p_2, v_2, u_2)\}$, $TUP_B = \{b_1.PK, b_2, b_3.FK(A)\}, \{(q_1, w_1, p_1), (q_2, w_2, p_2)\}$. The validation rules are performed according to the following algorithms:

Validating initial data — Rule (a)

```
IF EXISTS  $\sigma_a(A.a_2)$  {
  FOR EACH  $a_2.value \in TUP\_A$  DO {
    IF  $a_2.value \neg \Theta \vee \{Send\ a\ warning\ to\ the\ user\}\}$ 
```

i.e. all the related values v_1 and v_2 should not fulfill the expression $a_2.value \neg \Theta \vee$, where \vee is the defined value in σ_a . If any tuple exists, a message will be sent to the tester indicating that fact.

Validating initial data — Rule (b)

```
IF EXISTS  $\sigma_b(A.a_2, B.b_2)$ {
  FOR EACH  $a_1.value \in TUP\_A$  DO{
    AssociationKey =  $a_1.value$ 
    FOR EACH  $b_3.value \in TUP\_B$  AND  $b_3.value = AssociationKey$  DO {
      IF  $a_2.value \neg \Theta b_2.value \{Send\ a\ warning\ to\ the\ user\}$ 
    }}}
```

i.e. v_1 and v_2 should not fulfill the expression $a_2.value \neg \Theta b_2.value$, where w_1 and w_2 are the related values to $b_2.value$.

Validating initial data — Rule (c)

```
IF EXISTS  $\sigma_c(A.a_2, B.b_2)$ {
  FOR EACH  $a_1.value \in TUP\_A$  DO {
    AssociationKey =  $a_1.value$ 
    FOR EACH  $b_3.value \in TUP\_B$  AND  $b_3.value = AssociationKey$  DO {
      Calculate  $agg(b_2.value)$ 
      IF  $a_2.value \neg \Theta agg(b_2.value)$  {Send a warning to the user}
    }
  }
```

i.e. if $agg(b_2)$ be $SUM(b_2)$ then v_1 and v_2 should not fulfill the expression $a_2.value \neg \Theta SUM(b_2.value)$ when $b_3.value = a_1.value$. i.e. if $a_2.value = v_1$ then $SUM(b_2.value) = w_1$, and if $a_2.value = v_2$ then $SUM(b_2.value) = w_2$.

Validating initial data — Rule (d)

```
IF EXISTS  $\sigma_d(A.a_1, B.b_3)$  {
  FOR EACH  $a_1.value \in TUP\_A$  DO {
    AssociationKey =  $a_1.value$ 
    FOR EACH  $b_3.value \in TUP\_B$  AND  $b_3.value = AssociationKey$  DO {
      COUNT ( $b_3.value$ ) = COUNT ( $b_3.value$ ) + 1
      IF COUNT ( $b_3.value$ )  $\neg \Theta V$  {Send a warning to the user}
    }
  }
```

i.e. $COUNT(p_1)$ and $COUNT(p_2)$ into B should not fulfill the expression $COUNT(b_3.value) \neg \Theta V$, where V is the defined value in σ_d .

Validating initial data — Rule (e)

To illustrate the validation of an expression of the rule (e) let us consider that A is a direct type; B and C are the subtypes, where $TUP_A = \{a_1.PK\}, \{(p_1), (p_2)\}$, $TUP_B = \{a_1.FK(A)\}, \{(p_1)\}$, and $TUP_C = \{a_1.FK(A)\}, \{(p_2)\}$. The validation is done according to the following algorithms:

```
IF EXISTS  $\sigma_e(A.a_1, B.a_1, C.a_1)$  {
  FOR EACH  $a_1.value \in TUP\_A$  DO {
    DirectTypeKey =  $a_1.value$ 
    RelatedObj_B = FALSE
    FOR EACH  $a_1.value \in TUP\_B$  AND  $a_1.value = DirectTypeKey$  DO {
      RelatedObj_B = TRUE
    }
    RelatedObj_C = FALSE
    FOR EACH  $a_1.value \in TUP\_C$  AND  $a_1.value = DirectTypeKey$  DO {
      RelatedObj_C = TRUE
    }
    IF NOT RelatedObj_B AND NOT RelatedObj_C {Send a warning to the user}
    IF RelatedObj_B AND RelatedObj_C {Send a warning to the user}
  }
```

i.e. the variables RelatedObj_B and RelatedObj_C are used to ensure that the Direct-TypeKey in question has only one instance in B or C respectively. If there are any

`DirectTypeKey` appearances in the two related subtypes then this may violate the disjoint of the constraint. While if `DirectTypeKey` does not appear either in B or in C then this may violate the total of the constraint.

If any message is sent to the user to indicate an error in the domains, the user must correct the error manually and then the validation process should be repeated again. Once the validation process is successfully finished, the information in the tuple files will be ready as parameters for the following transformation model which converts these parameters into DML statements, taking the specific characteristics of each DBMS into account.

3.1.3. *Test cases*

The functional testing design is based on the definition of the equivalence classes. An equivalence class represents a set of valid or invalid states for certain input conditions [14]. To apply our approach, we first need to specify these input conditions. In database applications, input conditions can be divided into two classifications: *critical events* and the *attribute values* associated to these events. A critical event is an operation that could violate one database constraint; these events are: `Insert`, `Delete`, and `Update`. An event is used with its corresponding attribute values. Although inserting a new tuple in a table means inserting all the defined attributes' values of that table, in this section we consider only the attribute which is defined in the RA expression of a constraint because it is the only attribute that can affect the constraint.

According to the SQL expression rules (Sec. 3.1.1) *critical events* are specified. For example, any expression belonging to type (a) has as critical events: the insertion of one new tuple and the updates of the expression attributes. Table 1 shows the critical events considered in this work. More information about critical events is found in [19], which provides a complete approach for deriving active mechanisms in Relational Databases from the specification of OCL clauses.

Three equivalence classes are defined for these events, one for each event. These equivalence classes specify only the valid states of the input conditions because no DBMS is able to execute any invalid state. It means that if there is any syntax error in the specification of these events, the DBMS itself will reject this operation by raising a compilation error.

As in the case of the critical events, the verification of *attribute values* of the input conditions is also carried out by the DBMS itself. This means that the input conditions must always be valid. For example, if we insert a tuple into a table, the DBMS first verifies the correctness of the inserted tuple before placing it into the corresponding table, or when a tuple is deleted from a table, the DBMS verifies first that the deleted tuple exists in the corresponding table before deleting it. Nevertheless, from the database semantics point of view, it is possible to identify two types of input conditions, valid and invalid classes. For example, the insertion of $t_1(x, y)$ into a table could be valid for the semantic, while the insertion of $t_2(x, z)$

Table 1. The valid and invalid equivalence classes of the rules expression.

Rule	Valid Equivalence Classes		Invalid Equivalence Classes Attribute Values
	Critical Events	Attribute Values	
(a)	(1) Ins(A.x)	(2) A.x Θ V	(3) A.x $\neg\Theta$ V
	(4) Upd(A.x)	(5) A.x Θ V	(6) A.x $\neg\Theta$ V
(b)	(7) Ins(A.x)	(8) A.x Θ B.z	(9) A.x $\neg\Theta$ B.z
	(10) Ins(B.z)	(11) A.x Θ B.z	(12) A.x $\neg\Theta$ B.z
	(13) Upd(A.x)	(14) A.x Θ B.z	(15) A.x $\neg\Theta$ B.z
	(16) Upd(B.z)	(17) A.x Θ B.z	(18) A.x $\neg\Theta$ B.z
(c)	(19) Ins(B.z)	(20) A.x Θ agg(B.z)	(21) A.x $\neg\Theta$ agg(B.z)
	(22) Upd(A.x)	(23) A.x Θ agg(B.z)	(24) A.x $\neg\Theta$ agg(B.z)
	(25) Upd(B.z)	(26) A.x Θ agg(B.z)	(27) A.x $\neg\Theta$ agg(B.z)
	(28) Del(A.x)	(29) A.x Θ agg(B.z)	(30) A.x $\neg\Theta$ agg(B.z)
	(31) Del(B.z)	(32) A.x Θ agg(B.z)	(33) A.x $\neg\Theta$ agg(B.z)
(d)	(34) Ins(A.a)	(35) (count(B.a) Θ V)	(36) (count(B.a) $\neg\Theta$ V)
	(37) Ins(B.a)	(38) (count(B.a) Θ V)	(39) (count(B.a) $\neg\Theta$ V)
	(40) Del(A.a)	(41) (count(B.a) Θ V)	(42) (count(B.a) $\neg\Theta$ V)
	(43) Del(B.a)	(44) (count(B.a) Θ V)	(45) (count(B.a) $\neg\Theta$ V)
	(46) Upd(B.a)	(47) (count(B.a) Θ V)	(48) (count(B.a) $\neg\Theta$ V)
	(e)	(49) Ins(A.a)	(50) A.a IN($\pi_{B.a}(B) \wedge \pi_{C.a}(C)$)
(52) Ins(B.a)		(53) A.a \neg IN($\pi_{C.a}(C)$)	(54) A.a IN($\pi_{C.a}(C)$)
(55) Ins(C.a)		(56) A.a \neg IN($\pi_{B.a}(B)$)	(57) A.a IN($\pi_{C.a}(C)$)
(58) Del(B.a)		(59) A.a IN($\pi_{C.a}(C)$)	(60) A.a \neg IN($\pi_{C.a}(C)$)
(61) Del(C.a)		(62) A.a IN($\pi_{B.a}(B)$)	(63) A.a \neg IN($\pi_{B.a}(B)$)

into the same table could be invalid for the semantic. In these two cases, the DBMS accepts the insertion, but the difference is that the first case produces a correct semantic while in the second a false semantic is produced.

In accordance with what we have stated above, Table 1 specifies one valid class for each critical event and two equivalence classes for each attribute value: one valid class when the attribute value fulfils the corresponding constraint and one invalid class when the attribute value does not fulfill the constraint.

Once the valid and the invalid equivalence classes are defined, in the next step the necessary tests cases, corresponding to each one of these classes, are derived. A test case is defined as a combination of classes from different classifications. For each test case, exactly one class of each classification is considered [20]. In addition to that, to increase the confidence level in our approach we will increase the number of these cases by generating a test case for each tuple in the context relation (the relation in which the testing is performed).

In this work, we will focus on applying the invalid test cases because their executions contribute a certain confidence in detecting possible defects in the tested application. “*Test cases representing unexpected and invalid input conditions seem to have a higher error-detection yield than do test cases for valid input conditions*” [15].

Let us consider that a relation A contains the following schema: $TUP_A = \{a_1.PK, \dots, a_n\}, \{(p_1, \dots), (p_2, \dots), \dots (p_m, \dots)\}$. It has n attributes and m tuples, and a_1 is the primary key of A . The test cases which are defined on A are shown as follows:

$$t_k = \{event(A.a_i), Invalid(a_i.value), (\forall p_j \in A, j = 1, \dots, m)\}$$

where (t_k) is an index number for each test case. The $event(A.a_i)$ is a critical event which modifies an associated attribute in the table. This event can violate the expression rules shown in the previous section. The $Invalid(a_i.value)$ is the associated invalid attribute value for that event. As we previously stated, we apply the invalid test cases only. The $(\forall p_j \in A, j = 1, \dots, m)$ means that the test cases are applied for each primary key value (i.e. for each tuple) in the relation A . According to standard SQL for the Delete and the Update events we can define a condition (WHERE clause) to limit the modified tuples. For example, using the condition (WHERE $a_1 = p_j$) limits the event only on the tuple which has p_j as a primary key value. For the Insert event no such condition is needed. Therefore, the test cases for insertion will be applied only once for each relation, and we will use a new primary key value to apply the insertions. The other attribute values of the inserted tuple are edited by the user.

3.2. Transforming SQL components into DBMS testing software model

Although most Relational DBMS use SQL components, there are some differences between the specific characteristics of these DBMS. These differences make that testing software of one system cannot be used directly with another system without modification, although sometimes these differences are too small to be significant. Once the standard SQL components are derived in the previous section, these components are mapped to target DBMS testing software. The mapping is performed directly, that is, each SQL component is mapped into one related component in a target DBMS (1 to 1). As shown in the following processes:

3.2.1. Mapping standard SQL queries to target DBMS select statement

A standard SQL query is mapped to one Select statement of a target DBMS. To do this mapping, DBMS Select templates are used. A select template is a generic query template in which some values are established as parameters so that different particular constraint situations can be derived by giving different values to the parameters (Domínguez *et al.*, 2002). There is one template function for each DBMS and for each expression rule considered in this work. For each expression rule σ the execution of these templates is done according to the following algorithm:

For each σ
Find all Para $\in \sigma$

As example, for the expression rule⁽³⁵⁾($\text{count}(B.a) \Theta V$), the corresponding parameters $\text{Paras} = (A, B, A.a, B.a, \Theta, V)$.

```

Choose a DBMS
Call DBMS.Select- $\sigma$ ()
Run DBMS.Select- $\sigma$ (Paras)
Print Select
End For;
```

For example, here, we present the function `OracleSelect-d(Paras)` as a generic template of the expression rule (d) to illustrate the transformation of the standard SQL query into Oracle 11g [31]. The function is shown as follows:

```

Sub OracleSelect-d(Context_table, Related_table, PK, FK, Operator, Value, Select)
Select:= "SELECT * FROM Context_table
        WHERE PK IN (SELECT FK FROM Related_table
                    GROUP BY FK HAVING COUNT(*) Operator Value);"
End OracleSelect-d;
```

where (`Context_table`, `Related_table`, `PK`, `FK`, `Operator`, and `Value`) are input parameters while `Select` is the output parameter which returns the generated Oracle select statement. For the following constraint “every department has at least one employee”, applying the expression rule (d) and the previous function template of Oracle this constraint is transformed into the following select statement:

```

SELECT * FROM DEPT
        WHERE deptno IN (SELECT deptno FROM EMP
                        GROUP BY deptno HAVING COUNT(*) < 1);
```

This select statement is used to validate the database state whenever a test case contains one of the following critical events (⁽³⁴⁾`Insert(DEPT.deptno)`, (⁽³⁷⁾`Insert(EMP.deptno)`, (⁽⁴⁰⁾`Delete(DEPT.deptno)`, (⁽⁴³⁾`Delete(EMP.deptno)`, (⁽⁴⁶⁾`Update(EMP.deptno)`). If this statement returns any instance of `DEPT`, it means that there is at least one instance of a department which does not have any related one in `EMP` i.e. the previous constraint has been violated. All other templates are applied in the same way as the previous one.

3.2.2. Mapping test cases into target DBMS DML statement

Once the test cases are calculated according to what has been stated in Sec. 3.1.3, mapping the test case into standard DML statement is performed directly; that is, each SQL statement is transformed into a related statement in a target DBMS (1 to 1). Here, we are also using a template for each statement and for each DBMS. The execution of these templates is done according to a similar algorithm which has been shown in the case of mapping standard SQL queries to a target DBMS Select statement.

The following function MServerUPD-c() is used for mapping the Update event as a DML statement corresponding to the expression rule (c) using the MS SQL Server.

```
Sub MServerUPD-a(Context_table, a, V, id, DML_Upd)
    DML_Upd:= "UPDATE Context_table
        SET a = V WHERE PK = Value;"
End MServerUPD;
```

The other invalid test cases of the expression rules are generated in the same way as the previous one.

3.2.3. Mapping initial data into target DBMS insert statement

In this process the initial data which is generated in Sec. 3.1.2 is transformed into Insert statements related to a target DBMS. This process seems to be very simple since there are few tuples to be inserted. The tester can control the attribute values of these tuples easier than if these tuples had been generated in a huge number. For this reason we recommend that the tester applies the testing by dividing the database schema in parts, with each one of these parts having a few relations and a few test cases to be validated. As we will explain in the OCL2testSW tool section, the interface of this tool allows the tester to choose one or more OCL specification of the model to be transformed into test cases.

To illustrate the mapping let us consider the two Tables TUP_A = {a1.PK,a2,a3}, {(p1,v1,u1),(p2,v2,u2)} and TUP_B = {b1.PK,b2,b3.FK (A)}, {(q1,w1,p1),(q2,w2,p2)}. These two tables A and B are associated with the keys b3=a1. In this case, if we want to generate Insert statements into the DB2 DBMS then the function DB2InitialData(context_table) is called with the previous attribute names and values.

```
INSERT INTO A (a1,a2,a3) VALUES (p1,v1,u1);
INSERT INTO A (a1,a2,a3) VALUES (p2,v2,u2);
INSERT INTO B (b1,b2,b3) VALUES (q1,w1,p1);
INSERT INTO B (b1,b2,b3) VALUES (q2,w2,p2);
```

4. OCL2TestSW Tool Design

Now that we have presented our approach we will explain how it has been implemented as a tool. This tool is called OCL2TestSW, and it aims at carrying out the necessary rules of the transformation models of our approach automatically. The tool has been added to Rational Enterprise Edition [7], one of the most important commercial CASE tools in the market. Like many tools, Rational Rose has the potential for adding modules to support software development needs, so that we used it as a platform to incorporate our tool. OCL2TestSW can be accessed from the Rational Rose Tools menu. The architecture of the OCL2TestSW tool is shown in Fig. 2. It consists of the following modules.

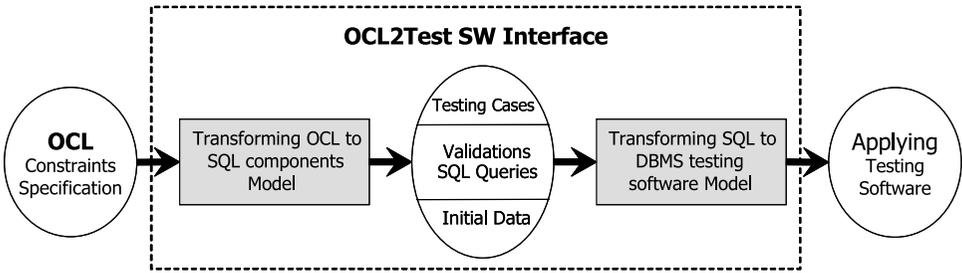


Fig. 2. OCL2TestSW architecture.

4.1. OCL constraints specification module

To specify business rules in a UML class diagram as OCL clauses, the OCL2TestSW tool uses a module for editing and checking OCL constraints. All OCL constraints are plugged into the Rational Object Model in the corresponding classes. 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, the Oclarity tool [21] 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, the Oclarity tool provides full syntactic and semantic checking. For example, let us consider the constraint “Married people are aged ≥ 18 ”. Figure 3 shows this constraint in an OCL clause as well as the syntax verification.

4.2. OCL2Testsw interface

Before this interface is run, Rational Rose automatically maintains the mapping between Rational Object Model and Rational Data Model where each class is mapped into a Relational table. To generate the target DBMS testing software the interface shown in Fig. 4 is implemented, which is able to detect the specified OCL constraints in the Rational Object Model, and shows them in the list box “**Current OCL Constraints.**” The detection of the specified OCL constraints in

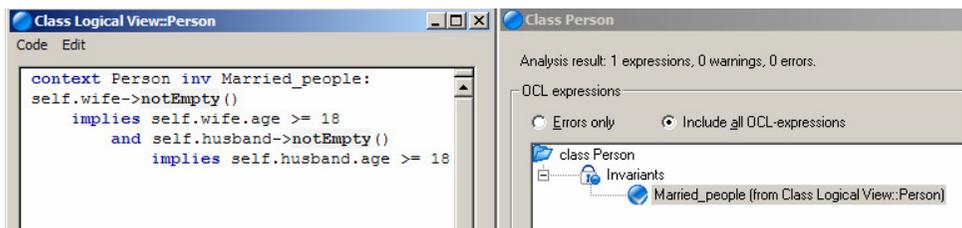


Fig. 3. Edit/Check constraints interface.

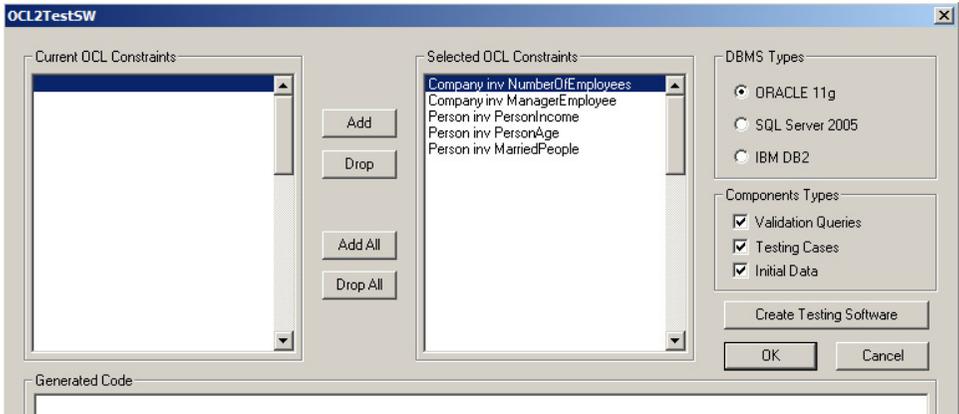


Fig. 4. OCL2TestSW interface.

a given model is performed according to the following 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 which represent the current OCL constraints in a class diagram. This set is initiated to empty when the algorithm is started. When it is finished, CurrentOCLset contains all the OCL constraints included in the diagram. 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 $\ll inv \gg$. When CurrentOCLset 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 “**Component Types**” check box represents the component types to be required, where the user can choose to generate one or more types of these components.

For example, if the user chooses only the initial data option this means that OCL2TestSW will generate only data to be inserted in a schema. This may be useful in the case when the tester needs to correct the error of the domain values manually and needs to repeat the validation process again. Once UserOCLset list is chosen, a target DBMS is specified and the user can obtain the generated components by clicking on the “**Create Testing Software**” button.

5. Evaluation Study

In this section, the applicability of our approach for testing database applications will be shown. According to [26], the applicability is the ability of the approach to be used, given measurable and objective results. For evaluating it, measurements such as test effectiveness can be used. The experimental framework proposed in [26] will be used to validate the test effectiveness presented in this proposal. The evaluation consists of three steps, *preparing code sample with known faults*, *performing the experiment and collect data*, and *evaluating results*. Next, these processes will be explained in detail.

5.1. Prepare code sample with known faults

One point to keep in mind in this section is that the main objective of our OCL2TestSW tool is to generate software components (initial data, test cases and validation queries) from the specification of UML/OCL constraints associated with a database schema. These test components are used to check that the applications, which were implemented to fulfil these constraints, perform correctly its function. For example, if a constraint (φ_1) is defined as “the age of an employee must be equal to or more than 24”, then according to our approach, measures must be taken to ensure that the application which was employed to enforce this constraint fulfils its intent.

Now, having clarified this point, our aim is to ensure the ability of these components to detect inconsistency in a database state when the application under test is run. Inconsistency in a database states means that the application under test includes faults. Here, we investigate a particular type of fault that leads to inconsistency in a database state when one of the defined OCL constraints is violated. A fault is defined as a nonconformance to a constraint definition. This type of faults is not captured by the database compiler.

To provide a realistic and suitable evaluation study, two real database applications have been selected: User Profile Database Application (UPDA) and Employees Database Application (EDA). The selected database schemata consist of 36 and 21 OCL constraints respectively, which cover the five types of the constraint rules that have been considered in the presented proposal (see Sec. 3.1.1). Although it might seem redundant to use more than one constraint per type, we actually did because we have taken into account the evaluation of real applications without any reduction. To enforce the defined constraints the UPDA uses a trigger system, while EDA

uses SQL in Java. It is very important to mention that we have copied the schemata of the two databases without data and we used copies of the two applications. The applications have been validated before the test, i.e. fulfil the user requirements.

Fault seeding is a technique originally proposed to estimate the quality of the software [30]. Therefore, to seed a variety of faults we manually modified the source codes of the two applications. For example, a fault that is seeded in the previous constraint (\wp_i) will convert the same constraint as follows “the age of an employee must be less than 24”. Bearing this in mind, we seeded in each application one fault per constraint (see Table 2).

5.2. Perform the experiment and collect data

Before the test is started, the initial data, test cases, and validation queries are automatically generated according to the definition of the OCL constraints. First, the initial data are inserted into the databases, and then the corresponding validation queries are run to validate the consistency of the databases. After ensuring the consistency of the database states, the test can be started. The number of the test cases (see Table 2) depends on the number of the tuples in each database. In our approach, to increase the confidence level of the test, we generate an invalid and valid test case for each tuple in the databases (see Sec. 3.1.3). Of course, the tester can include in the test all the valid cases or part of them. In this case, a valid test case does not produce an inconsistent database state, while an invalid test case does produce an inconsistent database state. If an inconsistent state has not been detected this means that the corresponding application has a faulty process. To monitor the behavior of the valid test cases, approximately 20% of the total test cases were used as valid cases of each application. When a test case produces an inconsistency database state, this means that a faulty process is executed. So, the test is organized in such a way that when a test case t_i is run the corresponding validation query q_i is run after. If q_i returns the results then we conclude that t_i executes a faulty process.

5.3. Evaluate results

After the test results have been collected, the effectiveness metrics are calculated. The test effectiveness is defined in [11] as the number of faults the test technique will find. In [27] the test effectiveness is the number of faults found per unit of size.

Table 2. Summary of the selected applications.

Application	Number of Faults	Number of Test Cases	
		Invalid	Valid
UPDA	36	103	25
EDA	22	58	18

In this evaluation we adopt the test effectiveness presented in [28] which defines it as the ratio of faults found to the total number of faults. The effectiveness levels of 100% are possible in our experiments when a complete detection is produced of all the faults (available for study) [29]. Our experiments were performed on a PC with Intel Core 2 CPU 1.83 GHz and 2.00 GB RAM. The operative system was Windows XP, and ORACLE 11g was selected as a database environment of the experiments. To calculate the test effectiveness, each application was executed against all its corresponding test cases, the results are shown in Fig. 5. Each point in this figure represents the test effectiveness when a new fault is detected, the dotted curves represent the general tendency of the test effectiveness. In [28] a percentage of 75% was suggested to be an acceptable baseline value of test effectiveness, while in our experiments, the test effectiveness approximates to 100% after the use of almost 90% of the total of test cases. This means that the invalid test cases are distributed in proportion to the number of test cases, although the increase of the effectiveness values at the starting of the test is more that at the end of the test.

In the two applications, the execution of 50% of the test cases has detected almost 80% of the defined faults, while the execution of the 92% of the test cases has detected all the defined faults. This is because when the test cases are generated we give a priority to the invalid test cases then the valid cases. Although, we used valid cases in the testing, the results have shown that these cases did not have a role in the detection of the defined faults. i.e. with only the invalid test cases we can reach complete testing.

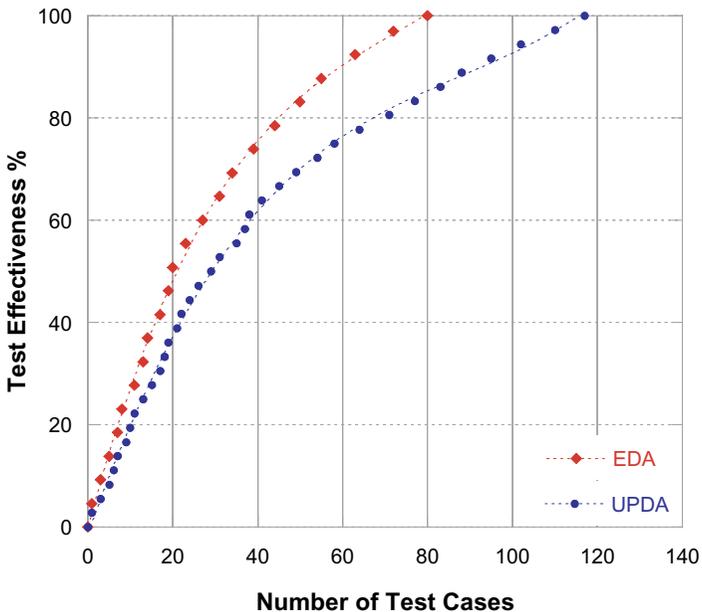


Fig. 5. The test effectiveness study of our experiments.

In our approach, to guarantee the coverage of all the defined faults in the codes, the database state must be consistent before a test case is executed. This is due to the fact that if a test case is specified at the start of the test as an invalid or valid case, this specification may be changed if the database state changes. Therefore, when an invalid case is executed, the database state must be repaired before the next test case is executed. The execution of the two experiments does not need to spend a lot of time; 2.9, 5.1 min were needed for EDA and UPDA respectively.

Finally, it is worth highlighting the importance of deriving test cases from the specification of the database constraints to efficiently detect faults of applications which were generated from the same specification.

6. Conclusions

The database CASE tools have been developed to resolve the database-modeling problem and to provide automatic code generation. Nevertheless, the validation of the generated code is not supported by these tools, so that the generated code is applied without any mechanism to ensure that it complies with the requirements of the real world. Thus, in this work we suggest the validation of the enforcement mechanisms by using the equivalence class test technique. It is a fundamental functional testing technique for checking the software without being concerned about its implementation and structure. In this type of testing the tester's responsibility is to provide input data and to validate the output results. The main objective is to check whether the tested software fulfils the user's requirements or not.

It is true that various studies have led to important results such as the creation of the current commercial CASE tools and some research prototypes to support software testing. However, in the context of Relational Databases we consider that current practice is below the appropriate. Therefore, to fill in some of the gaps that the current CASE tools create during the development of Relational databases, we present the OCL2TestSW tool as a support to the theoretical approach which follows the phases proposed in the MDA software development by completely transforming the OCL specification into testing software. These phases are as follows: specifying OCL constraints in the UML class diagram, transforming the OCL constraints into standard SQL components, transforming the standard SQL components into target DBMS SQL queries. Thus, this work unites the UML aspects that are widely accepted, and it is supported by many CASE tools for aspects of Relational databases that have a wide presence in commercial DBMS.

Our approach has some limitations which are explained as follows: (a) applying MDA makes the transformation of any type of OCL constraints to test software easier; currently the OCL2TestSW tool supports only the OCL invariant constraints. Other types of constraints such as pre-conditions, and post-conditions will be included in future work; (b) Including complex OCL expressions in which many relations are involved may turn out to be a difficult task to generate testing software.

This limitation could be solved by incorporating more transformation rules into our approach to cover such expressions. The article represents a first effort to check the viability of this approach through some of the most widely used constraints in the conceptual model. (c) Until now the generating task of initial data has been considered an incomplete processing because the user must intervene to introduce some type of data such as character type. Therefore, as a future work we would like to incorporate a mechanism to generate these types of data. In addition, we will adopt our tool to generate a huge volume of data in a totally automated way.

Although the tool is implemented using Rational Rose, our approach has been developed taking into account the specification of language (OCL) version 2.0 which meets the specifications of UML 2.0. The Rational Rose CASE tool has served to check the feasibility of the proposal implementation with a significant subset of constraints. One of our future objectives is to use more powerful tools such as IBM Rational Software Architect [23] in which new types of OCL constraints are introduced.

The approach could be considered a solution to motivate database developers to use testing software for validating database applications. This approach makes it easier to generate directly testing software from the specification of OCL constraints in the conceptual schema. Moreover, when the integrity constraints of this schema are modified, the testing software can also be automatically modified.

Acknowledgments

This work has been partially supported by the project Thuban: Natural Interaction Platform for Virtual Attending in Real Environments (TIN2008-02711), and also by the Spanish research projects: MA2VICMR: Improving the access, analysis and visibility of the multilingual and multimedia information in web for the Region of Madrid (S2009/TIC-1542).

References

1. OMG, 2007, Object Management Group, Inc., <http://www.omg.org/mda/>
2. M. Y. Chan and S. C. Cheung, Testing database applications with SQL semantics, in *2nd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'99)*, Wollongong, Australia, 1999, pp. 363–374.
3. ArgoUML, 2007, <http://argouml.tigris.org/>
4. MetaEdit+, 2007, <http://www.metacase.com/>
5. Objectteering/UML, Objectteering/SQL Designer User Guide Version 5.2.2, 2007, <http://depinfo.u-bourgogne.fr/docs/Objectteering522/SQLDesigner.pdf>
6. OCL2SQL, Dresden OCL Toolkit, 2007, <http://dresden-ocl.sourceforge.net/>
7. Rational Enterprise Edition, 2003, www-306.ibm.com/software/rational/
8. Visual Case Tool, 2007, <http://visualcase.com/index.htm/>
9. B. Verheecke and R. Straeten, Specifying and implementing the operational use of constraints in object-oriented applications, *Proc. of TOOLS Pacific 2002*, 2002, p. 23.

10. G. M. Kapfhammer and M. L. Soffa, A family of test adequacy criteria for database-driven applications, *Software Engineering Notes* **28** (2003) 98–107.
11. D. Chays, S. Dan, P. G. Frankl, F. Vokolos and E. J. Weyuker, A framework for testing database applications, *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Portland, Oregon, 2000.
12. Y. Deng, P. G. Frankl and J. Wang, Testing web database applications, in *Workshop on Testing Analysis and Verification of Web Services (TAV-WEB)*, 2004, pp. 1–10.
13. H. W. R. Chan, S. W. Dietrich and S. D. Urban, On control flow testing of active rules in a declarative object-oriented framework, *Proc. of 3rd Intl. Workshop on Rules in Database Systems (RIDS 97)*, Skovde, Sweden, 1997.
14. R. S. Pressman, *Software Engineering: A Practitioner's Approach* (McGraw-Hill Higher Education, 2005).
15. G. J. Myers, *The Art of Software Testing*, 2nd edn. (John Wiley & Sons, 2004).
16. ISO/IEC 9075 Standard, 2003, Information Technology — Database Languages — SQL:2003 *International Organization for Standardization*.
17. J. Cabot and E. Teniente, Constraint support in MDA tools: A survey, in model driven architecture — Foundations and applications, 2006, pp. 256–267.
18. T. J. Teorey, *Database Modeling & Design*, Third Edition, Morgan Kaufmann Series in Data Management Systems, 1999.
19. H. T. Al-Jumaily, D. Cuadra and P. Martínez, OCL2Trigger: Deriving active mechanisms for relational databases using model-driven architecture, *Journal of Systems and Software* **81** (2008) 2299–2314.
20. E. Lehmann and J. Wegener, Test case design by means of the CTE XL, *8th Proc. of European Int. Conf. on Software Testing, Analysis and Review (EuroSTAR 2000)*, Copenhagen, Denmark, 2000.
21. EmPowerTec, 2006, <http://www.empowertec.de/products/rational-rose-ocl.htm/>
22. D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos and E. J. Weyuker, An AGENDA for testing relational database applications, *Software Testing, Verification and Reliability* **14** (2004) 17–44.
23. IBM Rational Software Architect, <http://www-01.ibm.com/software/awdtools/architect/swarchitect/index.html>.
24. Z. Jian, X. Chen, and S. C. Cheung, Automatic generation of database instances for white-box testing, *Proceedings of 25th International Computer Software and Applications Conference on Invigorating Software Development*, 2001, pp. 161–165.
25. AllFusion® ERwin® Data Modeler <http://www3.ca.com/Solutions/>
26. S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson and D. Sundmark, A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques, *TAIC PART*, 2006, pp. 159–170.
27. G. Girauo and P. Tonella, Designing and conducting an empirical study on test management automation, *Empirical Software Engineering* **8(1)** (2003) 59–81.
28. Y. Chernak, Validating and improving test-case effectiveness, *IEEE Software Archives* **18(1)** (2001).
29. V. Debroy and W. E. Wong, Are fault failure rates good estimators of adequate test set size?, in *Proceedings of the 9th International Conference on Quality Software (QSIC)*, Jeju, Korea, August 2009.
30. H. Zhu, P. Hall and J. May, Software unit test coverage and adequacy, *ACM Computing Surveys* **29(4)** (1997) 366–427.
31. ORACLE 11g, <http://www.oracle.com/index.html>.