Providing Explanations for Database Schema Validation

Guillem Rull[1,2], Carles Farré[2], Ernest Teniente[2], Toni Urpí[2]

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
1-3 Jordi Girona, Barcelona 08034, Spain
{grull, farre, teniente, urpi}@lsi.upc.edu

Abstract. We propose a new method for database schema validation that provides an explanation when it determines that a certain desirable property of a database schema does not hold. Explanations are required to give the designer a hint about the changes of the schema that are needed to fix the problem identified. Our method is an extension of the CQC method, which has been shown successful for testing such properties, and its contribution is twofold: Firstly, it is the first method that offers an explanation when the schema is not adequately defined. Secondly, the extension proposed here provides a significant efficiency improvement as far as the run-time performance of the method is concerned.

1 Introduction

Database schema validation is related to check whether a database schema correctly and adequately describes the user intended needs and requirements. The correctness of the data managed by database management systems is vital to the more general aspect of quality of the data and thus their usage by different applications. A well-known approach to this problem is aimed at checking whether the database schema satisfies desirable properties such as schema satisfiability, query liveliness, non-redundancy of constraints, etc.

An important drawback of previous research in this area is that none of the methods proposed to deal with this problem [2,4,10] is able to provide explanations when a certain property does not hold. Therefore, the designer has to consider the full database schema to identify the required schema changes that would fix the problem.

As an example, assume the database schema includes the following two tables:

```
CREATE TABLE Category (
    name      char(10)    PRIMARY KEY,
    salary    real        NOT NULL,
    CONSTRAINT chMinSal  CHECK (salary >= 50000),
    CONSTRAINT chMaxSal  CHECK (salary <= 30000) )
CREATE TABLE Employee (
    ssn       int         PRIMARY KEY,
    name      char(30)    NOT NULL,
```

[1] This work was supported in part by Microsoft Research through the European PhD Scholarship Programme.
[2] This work was supported in part by the Ministerio de Educación y Ciencia under project TIN2005-05406.

```
catName    char(10)     NOT NULL,
CONSTRAINT chCatName CHECK (catName <> 'ceo'),
CONSTRAINT fkCat FOREIGN KEY (catName) REFERENCES Category(name) )
```

The previous schema is not satisfiable since it may not contain any tuple. The reason is that it is impossible to insert a category since it should have a salary lower than 30000 and higher than 50000 as stated by constraints *chMinSal* and *chMaxSal*. Moreover, since employees must always belong to categories (as stated by the *fkCat* constraint) it is also impossible to insert any employee in the previous database.

Previous methods allow determining that the schema is not satisfiable but they do not give any hint about the reasons that motivate this problem. At first glance, and taking into account that this could be just a small part of the schema, it may be very hard for the designer to identify the modification of the schema that would arrange the problem. Therefore, it becomes necessary to define methods able to explain the user why the tested property does not hold. This is the main goal of this paper. We understand an explanation as a set of constraints that is responsible for the non-satisfaction of the property.

A possible solution may be to use *black box* techniques [1,8] to compute an explanation. However, they require executing the method used to test the property as many times as constraints the database schema has. As schemas become larger, a faster way to perform this computation is needed.

In this paper, we follow a *glass box* approach aimed at computing an explanation with a single execution of the method at the same time that it checks whether the tested property holds. We extend the CQC method [3] for this purpose. In the previous example, a single execution of the method we propose in this paper would provide the explanation {*chMinSal*, *chMaxSal*, *fkCat*}. To our knowledge, ours is the first database schema validation method able to obtain an explanation with a single execution and also the first one that follows a glass-box strategy.

In addition to the previous result, the modifications of the CQC method we propose to obtain an explanation result also in a substantial efficiency improvement since they reduce the search space required to find the solution to the tested property. We provide also an experimental evaluation that illustrates the gain of our new method as compared to that of the original CQC method. This is also a significant result since this method is being applied to other areas, such as reasoning on UML class diagrams [7], which directly benefit from this improvement.

In some cases, the explanation provided by our method may be not minimal. An explanation is minimal if there is no proper subset of it that is also an explanation. If we were interested in minimal explanations, we could obtain them through a black-box strategy by executing our method as many times as constraints the non-minimal explanation has. Clearly, since the new method is much more efficient than the original one and the number of constraints taken into account is never greater than the constraints in the schema (being usually much lower), our approach also improves efficiency of previous black-box techniques [1,8] for obtaining a minimal explanation.

This paper is organized as follows. Next section reviews basic concepts. Section 3 describes the CQC_E method, our proposal to draw explanations, while Section 4 presents an experimental evaluation of this method. Conclusions are given in Section 5.

2 Base Concepts

A *database schema* is a tuple (*DR*, *IC*) where *DR* is a finite set of deductive rules and *IC* a finite set of constraints. A *deductive rule* has the form

$$p(\bar{X}) \leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t,$$

and a *constraint* (or *condition*) the denial form

$$\leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t.$$

Symbols p, r_i are *predicates*. Tuples $\bar{X}_i$, $\bar{Y}_i$ contain *terms*, which are either variables or constants. The terms in tuple $\bar{X}$ are distinct variables. Each C_i is a *built-in literal* in the form of $t_1\ \theta\ t_2$, where t_1 and t_2 are terms and operator θ is $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$. Atom $p(\bar{X})$ is the *head* of the rule, and $r_i(\bar{X}_i)$, $\neg r_i(\bar{Y}_i)$ are positive and negative *ordinary literals* (those that are not built-in). Rules and constraints must be *safe*, that is, every variable that occurs in $\bar{X}$, $\bar{Y}_i$, C_i must also appear in some $\bar{X}_j$. Predicates that appear in the head of a rule are *derived predicates* (views and queries), also called *IDB* predicates. The rest are *base predicates* (tables), also called *EDB* predicates.

For the sake of uniformity, we associate an inconsistency predicate Ic_i to each integrity constraint. Then, a database instance *violates* a constraint $Ic_i \leftarrow L_1 \wedge ... \wedge L_k$ if Ic_i is true, i.e. if there is some ground substitution σ that makes $(L_1 \wedge ... \wedge L_k)\sigma$ true.

In this paper, we assume that schema validation properties are expressed in terms of a goal $G = \leftarrow L_1 \wedge \ldots \wedge L_m$ and a set of conditions to enforce $F \subseteq IC$ [2]. In this way, we say that a schema validation property is *satisfiable* if there is at least one database instance that makes G true and does not violate any integrity constraint in F.

An *explanation* for the non-satisfaction of a schema validation property expressed in terms of (G, F) is a set of integrity constraints $E \subseteq F$ such that (G, E) is not satisfiable.

3 The Approach

The main aim of our approach is to perform satisfiability tests for schema validation properties expressed in the formalism stated above, in such a way that (1) if the property is satisfiable, we provide a concrete database instance in which such a property holds; otherwise, (2) if the property is not satisfiable, we provide an explanation.

Reference [2] showed how to use the CQC method [3] to validate highly expressive database schemas (featuring integrity constraints, negations and comparisons). However, the CQC method does not provide any kind of explanation when a schema validation property test fails. In this paper, we propose an extension of the CQC method that provides such an explanation. We refer to this extension as the *CQC_E method*.

Roughly, the original CQC method performs validation tests by trying to construct a database instance for which the tested property holds. The method uses different *Variable Instantiation Patterns* (*VIPs*), according to the syntactic properties of the database schema considered in each test, to instantiate the ground EDB facts to be added in the database. Adding a new fact to the database under construction may cause the violation of some constraints. When a violation is detected, some previous decisions

must be reconsidered to explore alternative ways to reach a solution (e.g., reinstantiate a variable with another constant). In any case, the CQC method does not prescribe any particular execution strategy for the generation of the different alternatives.

The extension we propose in this paper is to define an execution strategy that explores only those alternatives that are indeed relevant for reaching the solution. In order to do this, we need to modify the internal mechanisms of the CQC method to gather the additional information that is required to detect which alternatives are relevant. If none of these alternatives leads to a solution, this same information will be used to build one explanation: the explanation of why this execution has failed.

In addition to allow us the computation of an explanation, using the CQC_E method results in a significant efficiency improvement, as we will show in Section 4.

3.1 Example

Let us consider the example presented in the introduction, expressed here in the logical formalism required by our method. Due to space reasons, we do not consider either the primary keys or the attribute *Employee.name*. However, these modifications do not affect the computed explanation.

$$DR = \{isCat(X) \leftarrow Cat(X, S)\}$$
$$IC = \{Ic_1 \leftarrow Emp(X, Y) \wedge Y = \text{'ceo'},\ Ic_2 \leftarrow Emp(X, Y) \wedge \neg isCat(Y),$$
$$Ic_3 \leftarrow Cat(X, S) \wedge S > 30,\ Ic_4 \leftarrow Cat(X, S) \wedge S < 50\}$$

Suppose we want to check whether schema validation property ($G = \leftarrow Emp(X, Y)$, IC) is satisfiable, that is, whether the *Employee* table may have at least one tuple in its extension. Fig. 1 shows a CQC_E-derivation that tries to construct an EDB to prove that this property is satisfiable. Each row in the figure corresponds to a CQC_E-node that contains the following information (columns): (1) The goal to attain: the literals that must be made true by the EDB under construction. (2) The conditions to be enforced: the set of conditions that the constructed EDB is required to satisfy. (3) The extensional database (EDB) under construction. (4) The conditions to be maintained: a set containing those conditions that must remain satisfied until the end of the CQC_E-derivation. (5) The set of constants used so far.

The transition between an ancestor CQC_E-node and its successor is performed by applying a CQC_E-expansion rule to a selected literal (underlined in Fig. 1) of the ancestor CQC_E-node. We refer to Section 3.2 for a proper formalization of the CQC_E-expansion rules.

The first two steps shown in Fig. 1 instantiate variables X and Y from literal $Emp(X, Y)$ in order to obtain a ground fact to be added to the EDB. The constants used to instantiate the variables are determined according to the corresponding Variable Instantiation Patterns (VIPs) [3] and their data type (int, real or string).

In the first step, X is instantiated with a fresh constant 0. Label 1 is attached to this constant occurrence to keep track of the CQC_E-node in which it was introduced. Similarly, step 2 instantiates variable Y with the constant *ceo* from the set of used constants, and labels it with the CQC_E-node identifier 2.

Step 3 inserts the selected ground literal $Emp(0^1, ceo^2)$ from the goal to the EDB under construction. Label 3 is attached to this new tuple in order to keep record of

Goal to attain	*Conditions to enforce*	*EDB*	*Conditions to maintain*	*Used constants*	*Node ID*
← $\underline{Emp(X, Y)}$	$\{Ic_1, Ic_2, Ic_3, Ic_4\} = C_0$	∅	C_0	{50, 30, ceo}	1
1:A2.1					
← $\underline{Emp(0^1, Y)}$	$\{Ic_1, Ic_2, Ic_3, Ic_4\}$	∅	C_0	{50, 30, ceo, 0}	2
2:A2.1					
← $\underline{Emp(0^1, ceo^2)}$	$\{Ic_1, Ic_2, Ic_3, Ic_4\}$	∅	C_0	{50, 30, ceo, 0}	3
3:A2.2					
[]	$\{Ic_1 \leftarrow \underline{Emp(X, Y)} \wedge Y = ceo, Ic_2, Ic_3, Ic_4\}$	$\{Emp(0^1, ceo^2)^3\}$	C_0	{50, 30, ceo, 0}	4
4:B2					
[]	$\{Ic_1 \leftarrow [Emp(0^1, ceo^2)^3 \wedge] \underline{ceo^2 = ceo}, Ic_2, Ic_3, Ic_4\}$	$\{Emp(0^1, ceo^2)^3\}$	C_0	{50, 30, ceo, 0}	5
5:Failed derivation					

Fig 1. Example of CQC_E-derivation.

which node was responsible for this insertion. After performing this later step, we get a CQC_E-node with an empty goal, i.e. []. However, the work is not done yet, since we must ensure that the four constraints to enforce, Ic_1, Ic_2, Ic_3 and Ic_4, are not violated by the current EDB. In other words, we must make Ic_1, Ic_2, Ic_3 and Ic_4 all false.

Step 4 selects literal $Emp(X, Y)$ from $Ic_1 \leftarrow Emp(X, Y) \wedge Y = ceo$. Since $Emp(X, Y)$ unifies with fact $Emp(0^1, ceo^2)^3$ of the EDB under construction, it cannot be false. That means the literal does not help to make Ic_1 false and, thus, it is marked as "discarded". Therefore, step 5 considers the next literal in the body of the constraint, which is the literal $ceo^2 = ceo$. Since all other literals in the body of Ic_1 have already been selected and this comparison is true, constraint Ic_1 is violated.

The analysis of a violation consists in finding those ancestor CQC_E-nodes in the current derivation that take a decision whose reconsideration may help to avoid, *repair*, the violation. Each one of these CQC_E-nodes is a *repair* for this violated constraint. The set of repairs for Ic_1 is recorded in the failed CQC_E-node 5 where constraint Ic_1 was violated. One way to repair this violation is change the value of constant ceo^2 in order to make $ceo^2 = ceo$ false. The label 2 attached to constant *ceo* indicates that this constant was used in the expansion of CQC_E-node 2 to instantiate a certain variable. Thus, we can backtrack to node 2 and try another instantiation for variable *Y*. This means node 2 is one of the *repairs* for the violation, so node 2 is included in the set of repairs of node 5. Other possible way to repair the violation is avoid the insertion of tuple $Emp(0^1, ceo^2)^3$ to the EDB. Label 3 indicates that this tuple was inserted in order to satisfy the literal $Emp(0^1, ceo^2)$ from the goal of node 3. The only possible way to avoid this insertion is by means of avoiding the presence of this literal in the goal. However, as the literal comes from the original goal (note there is no label attached to it), the insertion of the tuple to the EDB cannot be avoided. Therefore, the set of repairs of node 5 is {2}.

With this information into account, the method will try to construct an alternative CQC_E-(sub)derivation to achieve the initial goal, which will be rooted at CQC_E-node 2 (the repair of node 5). Moreover, in order to keep track of what has happened in the failed derivation, node 2 will record the set of repairs of node 5 together with the *explanation* of why that derivation failed, that is, the set $\{Ic_1\}$.

Goal to attain	*Conditions to enforce*	*EDB*	*Conditions to maintain*	*Used constants*	*Node ID*
← $\underline{Emp(0^1, Y)}$	$\{Ic_1, Ic_2, Ic_3, Ic_4\}$	∅	C_0	{50, 30, ceo, 0}	2
6:A2.1					
← $\underline{Emp(0^1, a^2)}$	$\{Ic_1, Ic_2, Ic_3, Ic_4\}$	∅	C_0	{50, 30, ceo, 0, a}	6
7:A2.2					
[]	$\{Ic_1 \leftarrow \underline{Emp(X, Y)} \wedge Y = \text{ceo}, Ic_2, Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	7
8:B2					
[]	$\{Ic_1 \leftarrow [Emp(0^1, a^2)^6 \wedge] \underline{a^2 = \text{ceo}}, Ic_2, Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	8
9:B5					
[]	$\{Ic_2 \leftarrow \underline{Emp(X, Y)} \wedge \neg isCat(Y), Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	9
10:B2					
[]	$\{Ic_2 \leftarrow [Emp(0^1, a^2)^6 \wedge] \underline{\neg isCat(a^2)}, Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	10
11:B3					
← $\underline{isCat(a^2)^{10}}$	$\{Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	11
12:A1					
← $\underline{Cat(a^2, S)^{11}}$	$\{Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	12
13:A2.1					
← $\underline{Cat(a^2, 50^{12})^{11}}$	$\{Ic_3, Ic_4\}$	$\{Emp(0^1, a^2)^6\}$	C_0	{50, 30, ceo, 0, a}	13
14:A2.2					
[]	$\{Ic_3 \leftarrow \underline{Cat(X, S)} \wedge S > 30, Ic_4, Ic_1, Ic_2\}$	$\{Emp(0^1, a^2)^3, Cat(a^2, 50^{12})^{13}\}$	C_0	{50, 30, ceo, 0, a}	14
15:B2					
[]	$\{Ic_3 \leftarrow [Cat(a^2, 50^{12})^{13} \wedge] \underline{50^{12} > 30}, Ic_4, Ic_1, Ic_2\}$	$\{Emp(0^1, a^2)^3, Cat(a^2, 50^{12})^{13}\}$	C_0	{50, 30, ceo, 0, a}	15
16:Failed derivation					

Fig. 2. An alternative CQC_E-(sub)derivation.

Fig. 2 shows an alternative CQC_E-derivation rooted at node 2. Steps 6, 7, 8 of this new derivation are similar to steps 2, 3 and 4, but step 6 uses a fresh constant '*a*' to instantiate variable *Y*. Step 9 selects literal $a^2 = ceo$. Since such a comparison is false, Ic_1 is not violated now, and so, it is removed from the set of conditions to enforce.

In step 10, the selected literal is $Emp(X, Y)$ from $Ic_2 \leftarrow Emp(X, Y) \wedge \neg isCat(Y)$. Since $Emp(X, Y)$ unifies with the fact $Emp(0^1, a^2)^6$ of the EDB, it cannot be false and, thus, it is marked as "discarded". Step 11 considers the next literal in the body of Ic_2, i.e. $\neg isCat(a^2)$. Since it is the only still-not-discarded literal of the constraint, it must be made false necessarily. Thus, $isCat(a^2)$ becomes a new (sub)goal to achieve and is transferred to the goal part. We must note that this new subgoal is labeled with the corresponding CQC_E-node identifier. Moreover, the integrity constraint Ic_2 is removed (temporarily) from the set of conditions to enforce.

Step 12 unfolds the literal $isCat(a^2)^{10}$ from the goal, getting $\leftarrow Cat(a^2, S)^{11}$ as the new goal to satisfy. Step 13 instantiates variable *S* with 50^{12}, where 50 is one of the values from the set of used constants and 12 the label that identifies the CQC_E-node that contains the instantiated variable. Step 14 inserts tuple $Cat(a^2, 50^{12})^{13}$ to the EDB under construction. Moreover, the set of conditions to maintain is copied to the set of conditions to enforce in order to prevent the new insertion from violating previously discarded constraints, e.g., Ic_1 and Ic_2.

Goal to attain	*Conditions to enforce*	*EDB*	*Conditions to maintain*	*Used constants*	*Node ID*
← $\underline{Cat(\mathrm{a}^2, S)^{11}}$	$\{Ic_3, Ic_4\}$	$\{Emp(0^1, \mathrm{a}^2)^6\}$	C_0	{50, 30, ceo, 0, a}	12
17:A2.1					
← $\underline{Cat(\mathrm{a}^2, 30^{12})^{11}}$	$\{Ic_3, Ic_4\}$	$\{Emp(0^1, \mathrm{a}^2)^6\}$	C_0	{50, 30, ceo, 0, a}	16
18:A2.2					
[]	$\{Ic_3 \leftarrow \underline{Cat(X, S)} \wedge S > 30, Ic_4, Ic_1, Ic_2\}$	$\{Emp(0^1, \mathrm{a}^2)^3, Cat(\mathrm{a}^2, 30^{12})^{16}\}$	C_0	{50, 30, ceo, 0, a}	17
19:B2					
[]	$\{Ic_3 \leftarrow [Cat(\mathrm{a}^2, 30^{12})^{16} \wedge] \underline{30^{12} > 30}, Ic_4, Ic_1, Ic_2\}$	$\{Emp(0^1, \mathrm{a}^2)^3, Cat(\mathrm{a}^2, 30^{12})^{16}\}$	C_0	{50, 30, ceo, 0, a}	18
20:B5					
[]	$\{Ic_4 \leftarrow \underline{Cat(X, S)} \wedge S < 50, Ic_1, Ic_2\}$	$\{Emp(0^1, \mathrm{a}^2)^3, Cat(\mathrm{a}^2, 30^{12})^{16}\}$	C_0	{50, 30, ceo, 0, a}	19
21:B2					
[]	$\{Ic_4 \leftarrow [Cat(\mathrm{a}^2, 30^{12})^{16} \wedge] \underline{30^{12} < 50}, Ic_1, Ic_2\}$	$\{Emp(0^1, \mathrm{a}^2)^3, Cat(\mathrm{a}^2, 30^{12})^{16}\}$	C_0	{50, 30, ceo, 0, a}	20
22:Failed derivation					

Fig. 3. Another alternative CQC_E-(sub)derivation.

Step 15 selects $Cat(X, S)$ from $Ic_3 \leftarrow Cat(X, S) \wedge S > 30$. Since $Cat(X, S)$ unifies with the fact $Cat(a^2, 50^{12})^{13}$ of the EDB, it does not help to make Ic_3 false and is marked as discarded. At step 16, built-in literal $50^{12} > 30$ is selected from Ic_3. Since all other literals in the body of Ic_3 have been selected and this comparison is true, constraint Ic_3 is violated.

As before, the analysis of the violation is performed. One way to repair the violation is change the value of constant 50^{12} in order to make $50^{12} > 30$ false. Label 12 attached to constant 50 indicates that this constant was used in the expansion of node 12 to instantiate a certain variable. Thus, we can backtrack to node 12 and try another instantiation for variable S. This means node 12 is one of the *repairs* for the violation, so this node is included in the set of repairs of the failed node 15. Other possible way to repair the violation is avoid the insertion of $Cat(a^2, 50^{12})^{13}$ to the EDB. This fact was inserted during the expansion of node 13 in order to make the literal $Cat(a^2, 50^{12})^{11}$ from the goal true. The only possible way to avoid this insertion is avoiding the presence of this literal in the goal. The label 11 attached to the literal tells us that it was added to the goal because the expansion of node 11, which performed the unfolding of derived literal $isCat(a^2)^{10}$. As *isCat* is defined by one deductive rule only, there is no point in backtracking to node 11 and trying the unfold using another deductive rule. However, there is another option. We still can try to avoid the presence of $isCat(a^2)^{10}$ itself in the goal. Similarly as before, label 10 indicates that it was added to the goal as result of the expansion of node 10 in order to avoid the violation of referential constraint $Ic_2 \leftarrow Emp(0^1, a^2)^6 \wedge \neg isCat(a^2)$. This means Ic_2 is needed to reach the violation, so we have to add node 10 to the set of repairs of node 15. Since Ic_2 was triggered by the insertion of $Emp(0^1, a^2)^6$ to the EDB, we still could repair the violation by avoiding this insertion. However, the insertion was because the literal $Emp(0^1, a^2)$ from the goal, which has no label attached to it. Having no label means that it already appears in the initial goal, so it cannot be avoided. Summarizing, the set of repairs recorded in CQC_E-node 15 for the violation of $Ic_3 \leftarrow Cat(a^2, 50^{12})^{13} \wedge 50^{12} > 30$ is {12, 10}.

Similarly as before, the method will try to construct an alternative (sub)derivation to achieve the initial goal. It will be rooted at CQC_E-node 12, which is the closest repair to the failed CQC_E-node 15. To keep track of what has happened in the failed derivation, node 12 will record the set of repairs of node 15 together with the explanation for this failed derivation, i.e. $\{Ic_3\}$.

Fig. 3 shows another alternative CQC_E-derivation, this one rooted at CQC_E-node 12. This new derivation fails too. In this case, the explanation for the failure is the violation of integrity constraint Ic_4. As in the previous derivation, the set of repairs for this violation is {12, 10}. Indeed, any derivation starting from node 12 will fail because each possible instantiation for variable S in $Cat(a^2, S)^{11}$ will lead to the violation of either Ic_3 or Ic_4, with {12, 10} as the set of repairs in any case. Therefore, the method marks CQC_E-node 12 as failed, and assume as an explanation for such a failure the union of the explanations of its failed CQC_E-(sub)derivations, that is, $\{Ic_3, Ic_4\}$. Moreover, the set of repairs of CQC_E-node 12 is set to the union of the sets of repairs of its failed CQC_E-(sub)derivations minus the node 12 itself, i.e. {10}.

Still trying to find a successful CQC_E-derivation, the method will visit the closest repair of CQC_E-node 12. In this case, there is only one option: CQC_E-node 10. This node enforces referential constraint Ic_2, and so, leads to the violation of constraints Ic_3 and Ic_4. Since there is not an alternative CQC_E-(sub)derivation rooted at node 10, the method marks this node as failed. The explanation the method assumes for this failure is the explanation of its only CQC_E-(sub)derivation plus the referential constraint Ic_2, i.e. $\{Ic_2, Ic_3, Ic_4\}$. Moreover, the set of repairs of node 10 is set to the one of its child (sub)derivation minus itself, i.e. the empty set. Since there is no repair, there is no point in reconsidering any previous decision, so the method ends without being able of constructing an EDB that satisfies the initial goal, and returns $\{Ic_2, Ic_3, Ic_4\}$ as the set of integrity constraints that explains such a failure (the explanation indicated in the introduction). Note that since CQC_E-node 2 does not belong to the set of repairs of CQC_E-node 10, the explanation for the failed derivation in Fig. 1, recorded at node 2, is discarded and not included in the final explanation.

3.2 Formalization

Let $S = (DR, IC)$ be a database schema, $G_0 = \leftarrow L_1 \wedge \ldots \wedge L_n$ a goal, and $F_0 \subseteq IC$ a set of constraints to enforce, where G_0 and F_0 characterize a certain schema validation property in the terms defined in [2]. A *CQC_E-node* is a 5-tuple of the form $(G_i\ F_i\ D_i\ C_i\ K_i)$, where G_i is a goal to attain; F_i is a set of conditions to enforce; D_i is a set of ground EDB atoms, an EDB under construction; C_i is the whole set of conditions that must be maintained; and K_i is the set of constants appearing in DR, G_0, F_0 and D_i.

A *CQC_E-tree* is inductively defined as follows:

1. The tree consisting of the single CQC_E-node $(G_0\ F_0\ \varnothing\ F_0\ K)$ is a CQC_E-tree.
2. Let T be a CQC_E-tree, and $(G_n\ F_n\ D_n\ C_n\ K_n)$ a leaf CQC_E-node of T such that $G_n \neq []$ or $F_n \neq \varnothing$. Then the tree obtained from T by appending one or more descendant CQC_E-nodes according to a CQC_E-expansion rule applicable to $(G_n\ F_n\ D_n\ C_n\ K_n)$ is again a CQC_E-tree.

It may happen that the application of a CQC_E-expansion rule on a leaf CQC_E-node $(G_n\ F_n\ D_n\ C_n\ K_n)$ does not obtain any new descendant CQC_E-node to be appended to

```
ExpandNode(T: CQC_E-tree, N: CQC_E-node): Boolean
  if N is a solution node then T.solution := N; B := true
  else
    B := false
    Apply one CQC_E-expansion rule R.
    if children(N, T) = ∅ then HandleLeaf(T, N)
    else
      U := children(N, T)
      while ∃M ∈ U ∧ ¬B
        if ExpandNode(T, M) then B := true
        else if N ∉ M.repairs then N.repairs := M.repairs; N.explanation := M.explanation; U := ∅
        else
          if R is A1-rule or A2.1-rule then HandleDecisionalNode(T, N)
          else /*R is B3-rule*/ HandleSelectionOfConstrWithNegs(T, N)
          U := U - {M}
  return B

HandleLeaf(T: CQC_E-tree, N: CQC_E-node)
  if N.selectedLiteral is from N.goal then
    N.repairs := RepairsOfGoalComparison(N.selectedLiteral, T); N.explanation := ∅
  else /*N.selectedLiteral is from N.selectedCondition*/
    N.repairs := RepairsOfIc(N.selectedCondition, T, N)
    Let us assume N.selectedCondition defines predicate Ic_i.
    if there is a constraint Ic defining predicate Ic_i in root(T).conditionsToEnforce then
      N.explanation := {Ic}
    else /*N.selectedCondition appeared as a result of a negative literal in the goal*/
      N.explanation := ∅

HandleSelectionOfConstrWithNegs(T: CQC_E-tree, N: CQC_E-node)
  Let children(N, T) = {M}; Let us assume N.selectedCondition defines predicate Ic_i.
  N.repairs := M.repairs - {N}
  if there is a constraint Ic defining predicate Ic_i in root(T).conditionsToEnforce then
    N.explanation := M.explanation ∪ {Ic}
  else
    N.explanation := M.explanation

HandleDecisionalNode(T: CQC_E-tree, N: CQC_E-node)
  N.explanation := ∅; N.repairs := ∅
  for each node C ∈ children(N, T)
    N.explanation := N.explanation ∪ C.explanation; N.repairs := N.repairs ∪ (C.repairs - {N})
```

Fig. 4. Formalization of the CQC_E-tree exploration process.

the CQC_E-tree because some necessary constraint defined on the CQC_E-expansion rule is not satisfied. In such a case, we say that (G_n F_n D_n C_n K_n) is a *failed* CQC_E-node. Each branch in a CQC_E-tree is a *CQC_E-derivation* consisting of a (finite or infinite) sequence (G_0 F_0 D_0 C_0 K_0), (G_1 F_1 D_1 C_1 K_1), … of CQC_E-nodes. A CQC_E-derivation is *successful* if it is finite and its last (leaf) CQC_E-node has the form ([] ∅ D_n C_n K_n). A CQC_E-derivation is *failed* if it is finite and its last (leaf) CQC_E-node is failed. A CQC_E-tree is *successful* when at least one of its branches is a successful CQC_E-derivation. A CQC_E-tree is *finitely failed* when each one of its branches is a failed CQC_E-derivation.

Fig. 4 shows the formalization of the CQC_E-tree exploration process. ExpandNode(T, N) is the main algorithm, which generates and explores the subtree of T that is rooted at N. The CQC_E method starts with a call to ExpandNode(T, N_{root}) where T contains only the initial node N_{root} = (G_0 F_0 ∅ F_0 K). If the CQC_E method constructs a

A#-Rules:

(A1) The selected literal $d(\bar{X})$ is a positive atom of a derived predicate:

$$\frac{(G_i = d(\bar{X}) \wedge L_1 \wedge ... \wedge L_n, F_i, D_i, C_i, K_i)^{id}}{(G_{i+1,1}, F_i, D_i, C_i, K_i) \mid ... \mid (G_{i+1,m}, F_i, D_i, C_i, K_i)}$$

where $G_{i+1,j} = (T_1^{id} \wedge ... \wedge T_s^{id} \wedge L_1 \wedge ... \wedge L_n)\sigma_j$, and $d(\bar{Z}) \leftarrow T_1 \wedge ... \wedge T_s$ is one of the m deductive rules in DR that define predicate d, and substitution σ_j is the most general unifier of $d(\bar{X})$ and $d(\bar{Z})$.

(A2.1) The selected literal $b(\bar{X})$ is a positive non-ground EDB atom:

$$\frac{(G_i, F_i, D_i, C_i, K_i)^{id}}{(G_i\,\sigma_1, F_i, D_i, C_i, K_{i+1,1}) \mid ... \mid (G_i\,\sigma_m, F_i, D_i, C_i, K_{i+1,m})}$$

where Y is a variable from $\bar{X}$, and each ground substitution $\sigma_j = \{Y \mapsto k_j^{id}\}$ is one of the m instantiations for variable Y provided by the corresponding VIP.

(A2.2) The selected literal $b(\bar{X})$ is a positive ground EDB atom:

$$\frac{(b(\bar{X}) \wedge G_{i+1}, F_i, D_i, C_i, K_i)^{id}}{(G_{i+1}, F_{i+1,j}, D_{i+1,j}, C_i, K_i)}$$

where $F_{i+1,j} = F_i \cup C_i$ and $D_{i+1,j} = D_i \cup \{b(\bar{X})^{id}\}$ if $b(\bar{X}) \notin D_i$ (disregarding labels); otherwise $F_{i+1,j} = F_i$ and $D_{i+1,j} = D_i$.

(A3) The selected literal $\neg p(\bar{X})$ is a ground negated atom:

$$\frac{(\neg p(\bar{X}) \wedge G_{i+1}, F_i, D_i, C_i, K_i)^{id}}{(G_{i+1}, F_i \cup \{Ic^{id}\}, D_i, C_i \cup \{Ic^{id}\}, K_i)}$$

where $Ic = Ic_{new} \leftarrow \mathsf{Normalize}(p(\bar{X}))$, and Ic_{new} is a fresh predicate.

(A4) The selected literal C is a ground built-in literal:

$$\frac{(C \wedge G_{i+1}, F_i, D_i, C_i, K_i)}{(G_{i+1}, F_i, D_i, C_i, K_i)}$$

only if C is evaluated true (disregarding labels).

B#-Rules:

(B1) The selected literal $d(\bar{X})$ is a positive atom of a derived predicate:

$$\frac{(G_i, \{Ic_k \leftarrow [B \wedge]\ d(\bar{X}) \wedge P_1 \wedge ... \wedge P_n\} \cup F_i, D_i, C_i, K_i)}{(G_i, \{S_1, ..., S_m\} \cup F_i, D_i, C_i, K_i)}$$

where $S_j = Ic_k \leftarrow [B \wedge]\ \mathsf{Normalize}((T_1 \wedge ... \wedge T_u \wedge P_1 \wedge ... \wedge P_n)\sigma_j)$, and $d(\bar{Z}) \leftarrow T_1 \wedge ... \wedge T_u$ is one of the m deductive rules in DR that define predicate d, and σ_j is the most general unifier of $d(\bar{X})$ and $d(\bar{Z})$.

(B2) The selected literal $b(X_1, ..., X_p)$ is a positive EDB atom:

$$\frac{(G_i, \{Ic_k \leftarrow [B \wedge] b(X_1,...,X_p) \wedge P_1 \wedge ... \wedge P_n\} \cup F_i, D_i, C_i, K_i)}{(G_i, S = \{S_1, ..., S_m\} \cup F_i, D_i, C_i, K_i)}$$

only if $S = \varnothing$ or $n \geq 1$, where $S_j = Ic_k \leftarrow [B \wedge b(k_1, ..., k_p)^{label} \wedge]\ (P_1 \wedge ... \wedge P_n)\sigma_j$, and $b(k_1, ..., k_p)^{label}$ is one out of the m facts about b in D_i, and $\sigma_j = \{X_1 \mapsto k_1, ..., X_p \mapsto k_p\}$ ($k_1, ..., k_p$ may be labeled).

(B3) The selected literal $\neg p(\bar{X})$ is a ground negated atom, and all positive literals in the condition have already been selected:

$$\frac{(G_i, \{Ic_k \leftarrow [B \wedge] \neg p(\bar{X}) \wedge \neg T_1 \wedge ... \wedge \neg T_n\} \cup F_i, D_i, C_i, K_i)^{id}}{(G_i \wedge Q_{new}^{id}, F_i, D_i, C_i, K_i)}$$

where Q_{new} is a fresh predicate of arity 0 defined by the following n deductive rules: $Q_{new} \leftarrow p(\bar{X})$, $Q_{new} \leftarrow T_1$, ..., $Q_{new} \leftarrow T_n$, which are added to DR.

(B4) The selected literal C is a ground built-in literal that is evaluated true (disregarding labels):

$$\frac{(G_i, \{Ic_k \leftarrow [B \wedge]\ C \wedge P_1 \wedge ... \wedge P_n\} \cup F_i, D_i, C_i, K_i)}{(G_i, \{Ic_k \leftarrow [B \wedge C \wedge]\ P_1 \wedge ... \wedge P_n\} \cup F_i, D_i, C_i, K_i)}$$

only if $n \geq 1$.

(B5) The selected literal C is a ground built-in literal that is evaluated false (disregarding labels):

$$\frac{(G_i, \{Ic_k \leftarrow [B \wedge]\ C \wedge P_1 \wedge ... \wedge P_n\} \cup F_i, D_i, C_i, K_i)}{(G_i, F_i, D_i, C_i, K_i)}$$

Fig. 5. Formalization of the CQC_E-expansion rules.

successful derivation, $\mathsf{ExpandNode}(T, N_{root})$ returns "true" and $T.solution$ pinpoints its leaf CQC_E-node. On the contrary, if the CQC_E-tree is *finitely failed*, $\mathsf{ExpandNode}(T, N_{root})$ returns "false" and $N_{root}.explanation \subseteq F_0$ is an explanation for the unsatisfiability of the tested schema validation property.

Regarding notation, we use $N.explanation$ and $N.repairs$ to denote the explanation and the set of repairs attached to CQC_E-node N. We assume that every CQC_E-node has a unique identifier. When it is necessary, we write $(G_i\ F_i\ D_i\ C_i\ K_i)^{id}$ to indicate that id is the identifier of the node. Similarly, constants, literals and constraints may have labels attached to them. We write I^{label} when we need to refer the label of I. The expansion rules attach these labels. Constants, literals and constraints in the initial CQC_E-node N_{root} are unlabeled.

```
RepairsOfGoalComparison(C: Built-in literal, T: CQC_E-tree): Set(CQC_E-node)
   R := AvoidLiteral(C, T)
   if the two constants in C are not labeled with the same label then
      R := R ∪ ChangeConstants(C, T)
   return R

RepairsOfIc(Ic: Constraint, T: CQC_E-tree, N: CQC_E-node): Set(CQC_E-node)
   if Ic has negated literals then R := {N} else R := ∅
   for each built-in literal C in Ic
      if the two constants in C are not labeled with the same label then
         R := R ∪ ChangeConstants(C, T)
   for each positive ordinary literal L in Ic
      Let id be the label of L; Let N^id be the node of T identified by id.
      R := R ∪ AvoidLiteral(N^id.selectedLiteral, T) /*expansion rule applied to N^id was A2.2*/
   R := R ∪ AvoidIc(Ic, T)
   return R

AvoidLiteral(L: Literal, T: CQC_E-tree): Set(CQC_E-node)
   if L is a labeled literal then
      Let id be the label of L; Let N^id be the node of T identified by id.
      if N^id.selectedLiteral is from N^id.goal then
         return {N^id}∪AvoidLiteral(N^id.selectedLiteral, T) /*expansion rule applied to N^id was A1*/
      else
         return RepairsOfIc(N^id.selectedCondition, T, N^id) /*expansion rule applied to N^id was B3*/
   else return ∅

AvoidIc(Ic: Constraint, T: CQC_E-tree): Set(CQC_E-node)
   if Ic is a labeled constraint then
      Let id be the label of Ic; Let N^id be the node of T identified by id.
      R := AvoidLiteral(N^id.selectedLiteral, T) /*expansion rule applied to N^id was A3*/
   else R := ∅
   return R

ChangeConstants(C: Ground built-in literal, T: CQC_E-tree): Set(CQC_E-node)
   R := ∅
   for each labeled constant K^id in C
      Let N^id be the node of T identified by id. /*expansion rule applied to N^id was A2.1*/
      R : = R ∪ {N^id}.
   return R
```

Fig. 6. Formalization of the violation analysis process.

Fig. 5 shows the *CQC_E-expansion rules* used by ExpandNode. The *Variable Instantiation Patterns* (*VIPs*) used by expansion rule A2.1 are those defined in [3]. We assume function Normalize returns the normalized version of the given conjunction of literals. We say that a conjunction of literals is *normalized* if it satisfies the following syntactic requirements: (1) there is no constant appearing in a positive ordinary literal, (2) there are no repeated variables in the positive ordinary literals, and (3) there is no variable appearing in more than one positive ordinary literal.

The application of a CQC_E-expansion rule to a given CQC_E-node (G_i F_i D_i C_i K_i) may result in none, one or several alternative (branching) descendant CQC_E-nodes depending on the selected literal L, which can be either from the goal G_i or from any of the conditions in F_i. Literal L is selected according to a safe computation rule, which selects negative and built-in literals only when they are fully grounded. If the selected literal is a ground negative literal from a condition, we assume all positive

literals in the body of the condition have already been selected along the CQC_E-derivation.

In each CQC_E-expansion rule, the part above the horizontal line presents the CQC_E-node to which the rule is applied. Below the horizontal line is the description of the resulting descendant CQC_E-nodes. Vertical bars separate alternatives corresponding to different descendants. Some rules such as A4, B2, and B4 include also an "only if" condition that constraints the circumstances under which the expansion is possible. If such a condition is evaluated false, the CQC_E-node to which the rule is applied becomes a failed CQC_E-node. In other words, the CQC_E-derivation fails because either a built-in literal in the goal or a constraint in the set of conditions to enforce is *violated*.

Fig. 6 shows the formalization of the violation analysis process, which is aimed to determine the set of *repairs* for a failed CQC_E-node. A *repair* denotes a CQC_E-node that is relevant for the violation. RepairsOfGoalComparison and RepairsOfIc return the corresponding set of repairs for the case in which the violation is in the goal and in a condition to enforce, respectively. AvoidLiteral (AvoidIc) returns the nodes that are responsible for the presence of the given literal (constraint) in the goal (set of conditions to maintain). Finally, ChangeConstants returns the nodes in which the labeled constants that appear in the given comparison were used to instantiate certain variables.

4 Experimental Evaluation

We have performed a set of experiments to compare the efficiency of the CQC_E method with regards to the original CQC method as implemented at the core of the SVT (Schema Validation Tool) [9]. We have executed the experiments on an Intel Core 2 Duo, 2.16 GHz machine with Windows XP (SP2) and 2 GB RAM. Each experiment was repeated three times and we report the average of these three trials.

Each set of experiments checks whether a given view (or query) of the schema is lively, i.e. it admits a non-empty extension. Both the original CQC method and the extended version are used to perform the corresponding test.

The first set of experiments reported in Fig. 7 (note the logarithmic scale) focus on the case in which liveliness does not hold, i.e. the contents of the view is always empty. The property tested by means of the chosen view allows stating also that a mapping among different database schemas loses certain information from one schema to another [5] and this is why we have chosen such a view. The mapped schemas are based on the relational schema of the *Mondial* database [6]. The database schema that results from the reformulation of the property is as follows. It consists of three copies of the Mondial schema, say S_1, S_2 and S_3, each one with its primary keys, foreign keys and unique constraints. Additionally, a set $M^k = \{Q^k_1, ..., Q^k_{14}\}$ of queries is defined over each S_k ($1 \le k \le 3$). These queries have the form $Q(\bar{X}) \leftarrow T_1(\bar{X}_1) \wedge ... \wedge T_n(\bar{X}_n)$, where n varies from 1 to 4, and T_1, ..., T_n are tables randomly selected from the schema. Finally, we also add a set of constraints that relates S_1, S_2 and S_3. These constraints have the form of $Ic_i \leftarrow Q^k_j(\bar{X}) \wedge \neg Q^m_j(\bar{X})$, where Q^k_j is a from M^k and Q^m_j is

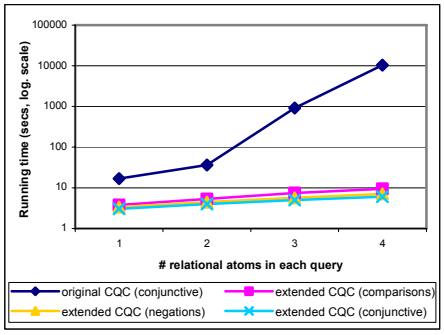

Fig. 7. Satisfiability tests with NO solution.

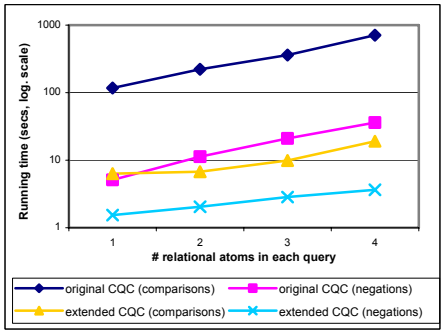

Fig. 8. Satisfiability tests that have solution.

Table 1. Running times (seconds) from Fig. 7 and Fig. 8.

	Fig. 7				Fig. 8			
# relational atoms in each query	**original CQC (conjunctive)**	**extended CQC (comparisons)**	**extended CQC (negations)**	**extended CQC (conjunctive)**	**original CQC (comparisons)**	**original CQC (negations)**	**extended CQC (comparisons)**	**extended CQC (negations)**
1	16.94	3.76	3.34	3.03	117.48	5.13	6.33	1.53
2	36.22	5.33	4.43	3.97	222.13	11.24	6.75	2.04
3	913.96	7.46	5.64	5	359.63	21.05	9.88	2.85
4	10369.94	9.43	7.09	6.03	711.34	36.03	19.1	3.65

Table 2. Characteristics of the database schemas.

	Fig. 7				Fig. 8			
	original CQC (conjunctive)	**extended CQC (comparisons)**	**extended CQC (negations)**	**extended CQC (conjunctive)**	**original CQC (comparisons)**	**original CQC (negations)**	**extended CQC (comparisons)**	**extended CQC (negations)**
# deductive rules	159	159	159	159	105	105	105	105
# constraints	341	341	341	341	218	218	218	218
# negated literals	171	171	303	171	104	188	104	188
# comparisons	0	126	0	0	74	0	74	0

from M^m ($1 \le k,m \le 3$; $k \ne m$; $1 \le j \le 14$). The goal of each liveliness test has the form of $G_0 = \leftarrow P(\bar{X}) \wedge \neg P'(\bar{X})$, where P is a query over S_1, and P' is its equivalent over S_2.

Fig. 7 shows that the use of the CQC_E method in this conjunctive setting results in a drastic reduction of running times. This is because its execution strategy helps to avoid the exploration of a high number of alternative CQC_E-(sub)derivations when exploring the CQC_E-tree. Moreover, Fig. 7 shows that the introduction of either comparisons or negations results also in lower times when using the extended method than when using the original one. These negations and comparisons are added to each query Q^k_i (and to query P). Therefore, Q^k_i has the form of either $Q^k_i(\bar{X}) \leftarrow T_1(\bar{X}_1) \wedge ... \wedge T_n(\bar{X}_n) \wedge \neg R_1(\bar{Y}_1) \wedge \neg R_2(\bar{Y}_2) \wedge \neg R_3(\bar{Y}_3)$ or $Q^k_i(\bar{X}) \leftarrow T_1(\bar{X}_1) \wedge ... \wedge T_n(\bar{X}_n) \wedge Z_1 > k_1 \wedge Z_2 > k_2 \wedge Z_3 > k_3$, where R_1, ..., R_3 are tables randomly selected from the schema, and k_1, k_2 and k_3 are fresh constants.

The second set of experiments reported in Fig. 8 focus on the case in which the liveliness tests have a solution, i.e. the view admits a non-empty instance. The used schemas are like those from the previous set of experiments, but now we have S_1 and

S_2 only. The goal of each satisfiability test has now the form of $G_0 = \leftarrow Q^1_1(\bar{X}_1) \wedge ... \wedge Q^1_{14}(\bar{X}_{14})$.

The graphics in Fig. 8 show that either when each query Q^k_i has 3 negations or when each query Q^k_i has 3 comparisons, the extended version of the method is faster than the original one. Although the computation of an explanation (as it is defined in Section 2) is not needed when the satisfiability test has a solution, Fig. 8 shows that we can still take advantage of the efficiency improvement that results from using the extended CQC method that we propose in this paper.

Table 1 shows the exact running times of both sets of experiments. Table 2 shows the main characteristics of the used schemas.

5 Conclusions

We have proposed the CQC_E method, an extension of the CQC method [3] for database schema validation, aimed at providing the database designer with an explanation for why a given database schema does not satisfy a certain desirable property.

The CQC_E method computes one explanation with a single execution, at the same time that it checks whether the tested property holds. This addresses an important drawback of previous research because none of the existing methods for schema validation [2,4,10] provides any kind of explanation when the tested property fails.

We have experimentally shown that using the CQC_E method results also in a significant efficiency improvement with respect to the original CQC method.

References

1. Bailey, J., Stuckey, P.J.: Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. PADL 2005, 174-186
2. Farré, C., Teniente, E., Urpí, T.: A New Approach for Checking Schema Validation Properties. DEXA 2004, 77-86
3. Farré, C., Teniente, E., Urpí, T.: Checking Query Containment with the CQC Method. Data Knowl. Eng. 53(2), 163-223 (2005)
4. Halevy, A.Y., Mumick, I.S., Sagiv, Y., Shmueli, O.: Static Analysis in Datalog Extensions. J. ACM 48(5), 971-1012 (2001)
5. Madhavan, J., Bernstein, P.A., Domingos, P., Halevy, A.Y.: Representing and Reasoning about Mappings between Domain Models. AAAI/IAAI 2002, 80-86
6. The Mondial Database. http://www.dbis.informatik.uni-goettingen.de/Mondial/
7. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. ER 2006, 497-512
8. Rull, G., Farré, C., Teniente, E., Urpí, T.: Computing Explanations for Unlively Queries in Databases. CIKM 2007, 955-958
9. Teniente, E., Farré, C., Urpí, T., Beltrán, C., Gañán, D.: SVT: Schema Validation Tool for Microsoft SQL-Server. VLDB 2004, 1349-1352
10. Zhang, X., Özsoyoglu, Z.M.: Implication and Referential Constraints: A New Formal Reasoning. IEEE Trans. Knowl. Data Eng. 9(6), 894-910 (1997)