Testing Termination of Query Satisfiability Checking on Expressive Database Schemas

(Preliminary Version)

Guillem Rull	grull@essi.upc.edu
Carles Farré	farre@essi.upc.edu
Ernest Teniente	teniente@essi.upc.edu
Toni Urpí	urpi@essi.upc.edu

March 2010

Report ESSI-TR-10-4

Departament d'Enginyeria de Serveis i Sistemes d'Informació

Abstract

A query is satisfiable if there is at least one consistent instance of the database in which it has a non-empty answer. Defining queries on a database schema and checking their satisfiability can help the database designer to be sure whether the produced database schema is what was intended. The formulation of such queries may easily require the use of some arithmetic comparisons or negated expressions. Unfortunately, checking the satisfiability of this class of queries on a database schema that most likely have some integrity constraints (e.g., keys, foreign keys, Boolean checks) is, in general, undecidable. However, although the problem is undecidable for such a class of schemas and queries, it may not be so for a particular query satisfiability check. In this paper, we propose to perform a termination test as a previous step to query satisfiability checking. If positive, the termination test guarantees that the corresponding query satisfiability check will terminate. We assume the CQC method is the underlying query satisfiability checking method; to the best of our knowledge, it is the only method of this kind able to deal with schemas and queries as expressive as the ones we consider.

1. Introduction

A query is satisfiable on a database schema if there is at least one consistent instance of the database in which the query has a non-empty answer [3, 5, 8].

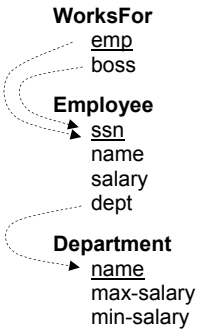
The motivation of this work is the application of query satisfiability checking to the validation of database schemas [3]. As an example, consider the database schema in Figure 1. It models data about employees, bosses and departments. All attributes are assumed to be not null. Underlined attributes denote keys, and dashed arrows denote foreign keys. The designer wanted to be sure that the database schema was what he intended, so he defined a set of views on the schema that will help him to answer the following questions:

- Can an employee be boss of himself?
- Is it possible to have an employee that is not assigned to any department?
- Can an employee have a salary higher than his boss?
- May two employees from different departments work for the same boss?

The key point is that these questions do not refer to a particular database instance, but to all possible instances that are consistent with the schema. Therefore, in order to answer them one needs to reason on the definition of the database schema. More specifically, these questions can be answer by means of checking whether the views on the schema are or not satisfiable:

- *BossOfSelf* is satisfiable if and only if an employee can be boss of himself.
- *EmpWithoutDept* is satisfiable if and only if it is possible to have an employee that is not assigned to any department.
- *RareEmp* is satisfiable if and only if an employee can have a salary higher than his boss.
- *MultiDeptBoss* is satisfiable if and only if two employees from different departments may work for the same boss.

These satisfiability checks can be performed by applying the CQC method [4], which, to the best of our knowledge, is the only query satisfiability method that is able to deal with the class of database schemas we consider. The checks will reveal that all views but



Additional Integrity Constraints:

ic_1: $Department(n, max, min) \rightarrow max \leq 5000$
ic_2: $Department(n, max, min) \rightarrow min \geq 700$
ic_3: $Employee(e, n, s, d) \wedge Department(d, max, min) \rightarrow s \geq min$
ic_4: $Employee(e, n, s, d) \wedge Department(d, max, min) \rightarrow s \leq max$

Views:

$BossOfSelf(e) \leftarrow WorksFor(e, e)$
$EmpWithoutDept(e) \leftarrow Employee(e, n, s, d) \wedge \neg isDept(d)$
$isDept(d) \leftarrow Department(d, max, min)$
$RareEmp(e) \leftarrow WorksFor(e, b) \wedge Employee(e, ne, se, de) \wedge Employee(b, nb, sb, db) \wedge se > sb$
$MultiDeptBoss(b) \leftarrow Employee(e1, n1, s1, d1) \wedge Employee(e2, n2, s2, d2) \wedge d1 \neq d2 \wedge WorksFor(e1, b) \wedge WorksFor(e2, b)$

Figure 1: Example database schema.

EmpWithoutDept are satisfiable. Then, it is up to the designer to decide whether the result of each satisfiability check points to some semantic flaw or not. For instance, view *EmpWithoutDept* being unsatisfiable is most likely the answer the designer was expecting; however, the satisfiability of views *BossOfSelf*, *RareEmps* and *MultiDeptBoss* is probably indicating that some integrity constraints are missing.

Unfortunately, checking query satisfiability on a class of database schemas that is as expressive as the one of the schema shown in Figure 1 is undecidable. Nevertheless, although the problem is undecidable with respect to the whole class of schemas, it may not be so for a particular schema. For instance, the process of checking the satisfiability of the views in Figure 1 does indeed terminate.

In this paper, we propose a termination test to be performed as a previous step to the application of the CQC method to check the query satisfiability. If the termination test is positive, then it guarantees that checking the query satisfiability with the CQC method will terminate. Otherwise, the satisfiability checking may or may not terminate. Note that the termination test is a sufficient but not necessary condition for the termination of the query satisfiability checking, as expected due to the undecidability of the termination problem.

We adapt the termination test proposed in [6] for the context of reasoning on UML conceptual schemas. Our main **contributions** are the following:

- We extend the class of schemas the termination test is able to deal with to that handled by the CQC method. In particular, we allow for multiple levels of negation, i.e., negated atoms that correspond to views that may also have negated atoms in their definition, which in turn may also correspond to other views with negated atoms, and so on.
- We provide formal proofs, which were not provided in [6].

The paper is structured as follows. Section 2 reviews basic concepts about database schemas and gives a brief overview of the CQC method. Section 3 explains the details of the termination test, and Section 4 summarizes the conclusions.

2. Preliminaries

2.1 Database Schemas

A *database schema* is a set of relations with integrity constraints. We use first-order logic notation and represent relations by means of predicates. Each predicate P has a *predicate definition* $P(A_1, ..., A_n)$, where $A_1, ..., A_n$ are the *attributes*. Predicates may be either *base predicates*, i.e., the tables in the database, or *derived predicates*, i.e., queries and views. Each derived predicate Q has attached a set of non-recursive deductive rules that describe how Q is computed from the other predicates. A *deductive rule* has the following form:

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

where each C_i is a *built-in literal*, that is, a literal in the form of t_1 *op* t_2, where *op* $\in \{<, \leq, >, \geq, =, \neq\}$ and t_1 and t_2 are *terms*, which can be either variables or constants; and

$r_i(\overline{Y}_i)$, $\neg r_i(\overline{Z}_i)$ are positive and negated *ordinary literals* (those not built-in), respectively. Literal $q(\overline{X})$ is the head of the rule, the other literals are the body. Symbols $\overline{X}$, $\overline{Y}_i$, and $\overline{Z}_i$ denote lists of terms. Variables in $\overline{Z}_i$, $\overline{X}$ and C_i are taken from $\overline{Y}_1$, ..., $\overline{Y}_n$ in order to make the deductive rule *safe* [7].

An *integrity constraint* is a *disjunctive embedded dependency* [1] extended with atoms of derived relations and arithmetic comparisons. It takes one of the following two forms:

$$r_1(\overline{Y}_1) \wedge ... \wedge r_n(\overline{Y}_n) \rightarrow C_1 \vee ... \vee C_t$$
$$r_1(\overline{Y}_1) \wedge ... \wedge r_n(\overline{Y}_n) \wedge C_1 \wedge ... \wedge C_t \rightarrow \exists \overline{V}_1\, r_{n+1}(\overline{U}_1) \vee ... \vee \exists \overline{V}_s\, r_{n+s}(\overline{U}_s),$$

where $\overline{V}_i$ is a set of fresh variables, and variables in $\overline{U}_i$ are taken from $\overline{V}_i$ and $\overline{Y}_1$, ..., $\overline{Y}_n$. Note that predicates r_1, ..., r_n and r_{n+1}, ..., r_{n+s} in the integrity constraints may be either base or derived.

Formally, we write $DS = \langle PD, DR, IC \rangle$ to indicate that DS is a database schema with predicate definitions PD, deductive rules DR, and integrity constraints IC.

A *database instance* is a set of facts about the base predicates. A *fact* is a ground literal, i.e., a literal with all its terms constant. An instance is *consistent* if it satisfies all integrity constraints.

2.2 The CQC Method

To the best of our knowledge, the CQC method [4] is the only query satisfiability method able to deal with the class of database schemas defined in Section 2.1. It is a constructive method, that is, it tries to build a database instance in which the query's answer contains at least one tuple, and in such a way that the database instance satisfies the integrity constraints of the schema.

The CQC method consists of two phases. The first phase is **query satisfaction**. Starting from an empty instance, it adds the necessary tuples so the body of one of the deductive rules of the query becomes true. The second phase is **integrity maintenance**. Since the database instance produced by the previous phase does not necessarily satisfy the integrity constraints of the schema, it is required to check them. If a constraint is violated, there are two possibilities: (1) the constraint is of the form $r_1(\overline{Y}_1) \wedge ... \wedge r_n(\overline{Y}_n) \rightarrow C_1 \vee ... \vee C_t$, in which case the current database instance is not only inconsistent but also irreparable, and previous decisions made during the application of the method should be reconsidered; or (2) the constraint is of the form $r_1(\overline{Y}_1) \wedge ... \wedge r_n(\overline{Y}_n) \wedge C_1 \wedge ... \wedge C_t \rightarrow \exists \overline{V}_1\, r_{n+1}(\overline{U}_1) \vee ... \vee \exists \overline{V}_s\, r_{n+s}(\overline{U}_s)$, in which case the violation of the constraint may still be repaired by the addition of a new tuple. In the latter case, and since the new tuple may be of a derived predicate, a recursive application of the query satisfaction and integrity maintenance phases is required. The process ends when either (1) a constraint violation is found and no previous decision is left to reconsideration (i.e., all alternatives have been tried), in which case the conclusion is that the query is not satisfiable; or (2) a database instance is found such that satisfies the query and the integrity constraints, in which case the conclusion is that the query is satisfiable, and the constructed database instance is an example of that.

In order to instantiate the tuples to be added to the instance under construction, the CQC method uses a set of *Variable Instantiation Patterns* (*VIPs*). Each application of a VIP results in a finite set of constants to be tried. The VIPs guarantee that if no solution

can be constructed using the constants they provide, then no one exists. There are four VIPs: *Simple VIP*, *Negation VIP*, *Dense Order VIP* and *Discrete Order VIP*. Its application depends on the syntactic characteristics of the database schema. We only sketch them here, the details can be found in [4]:

- *Simple VIP* assigns a fresh constant to each distinct variable. It is typically used on schemas with no constraints, negations or arithmetic comparisons.
- *Negation VIP* states that in order to instantiate a variable we must consider: (1) the used constants, that is, the constants that appear in the schema and those that have been used to instantiate previous variables, and (2) a fresh constant. It is used when there are constraints and/or negations, but no order comparisons.
- *Order VIPs* consider: (1) the used constants, (2) a fresh constant lower than all used constants, (3) a fresh constant greater than all used constants, and (4) for each pair of consecutive used constants in the domain's order, a fresh constant placed between them (in the discrete case, only if there is room for it). These VIPs are used on schemas with order comparisons.

3. Testing Termination

The termination test proposed in [6] in the context of reasoning on UML conceptual schemas relies on a simplified version of the CQC method, which allows the use of negation only on predicates that either are base or their deductive rules do not contain negated literals. In this paper, we extend the termination test so it can handle the full class of schemas the CQC method is able to deal with; in particular, we extend the termination test so it can deal with multiple levels of negation. We also provide formal proofs, which were not provided in [6].

In order to deal with the multiple levels of negations, we propose to perform a preprocessing of the database schema in which all derived predicates are materialized, i.e., converted into base predicates. Then, a variation of the termination test from [6] is applied on the resulting database schema.

3.1 Preprocessing the Schema

The result of the preprocessing is a new database schema that we refer to as *b-schema*. Intuitively, the b-schema is a database schema that materializes the derived predicates of the original schema, turning them into base predicates and introducing constraints to keep them updated.

DEFINITION 1 (*B-Schema*). Let $S = \langle PD_S, DR_S, IC_S \rangle$ be a database schema. The *b-schema* of S is $BS = \langle PD_{BS}, \varnothing, IC_{BS} \rangle$, where $PD_{BS} = PD_S \cup PD_{DR}$, $IC_{BS} = IC_S \cup IC_{DR} \cup IC_P \cup IC_N$, and, for each deductive rule $q_i = (q(\overline{X}_i) \leftarrow L_1 \wedge ... \wedge L_k) \in DR_S$, the following is true:

- PD_{DR} contains the predicate definition $q_i(A_1, ..., A_t, B_1, ..., B_n)$, where t is the number of terms in the head of q_i, i.e., $t = |\overline{X}_i|$, and n is the number of distinct variables in $L_1 \wedge ... \wedge L_k$.
- IC_{DR} contains the integrity constraints:

$q_i(\bar{A}_i, \bar{B}_i) \rightarrow q(\bar{A}_i)$,
$q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{j1} = k_1$,
..., $q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{ju} = k_u$,
$q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{g1} = B_{h1}$,
..., $q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{gv} = B_{hv}$,
$q(\bar{Z}) \rightarrow \exists \bar{B}_1\, q_1(\bar{Z}, \bar{B}_1) \vee ... \vee \exists \bar{B}_i\, q_i(\bar{Z}, \bar{B}_i) \vee ... \vee \exists \bar{B}_m\, q_m(\bar{Z}, \bar{B}_m)$,

where k_1, ..., k_u are the constants in $\bar{X}_i$; they appear in the positions j_1, ..., j_u; B_{h1}, ..., B_{hv} are the variables in $\bar{B}_i$ that correspond to variables in $L_1 \wedge ... \wedge L_k$ that appear in X_i with positions g_1, ..., g_v; A_{j1}, ..., A_{ju} and A_{g1}, ..., A_{gv} are the variables in $\bar{A}_i$ in the positions j_1, ..., j_u and g_1, ..., g_v, respectively; $\bar{Z}$ denotes a list of t distinct variables; and q_1, ..., q_m are the base predicates in PD_{DR} that correspond to those deductive rules in DR_S with the derived predicate q in their head.

- If L_1, ..., L_k are positive literals, IC_P contains the constraints:

  $L_1 \wedge ... \wedge L_k \rightarrow q_i(\bar{A}_i, \bar{B}_i)$,
  $q_i(\bar{A}_i, \bar{B}_i) \rightarrow L_1$,
  ..., $q_i(\bar{A}_i, \bar{B}_i) \rightarrow L_k$.

- If $L_1 \wedge ... \wedge L_k = P_1 \wedge ... \wedge P_r \wedge \neg N_1(\bar{Z}_1) \wedge ... \wedge \neg N_s(\bar{Z}_s)$ and $s > 1$, IC_N contains the constraints:

  $P_1 \wedge ... \wedge P_r \rightarrow N_1(\bar{Z}_1) \vee ... \vee N_s(\bar{Z}_s) \vee q_i(\bar{A}_i, \bar{B}_i)$,
  $q_i(\bar{A}_i, \bar{B}_i) \rightarrow P_1$,
  ..., $q_i(\bar{A}_i, \bar{B}_i) \rightarrow P_r$,
  $q_i(\bar{A}_i, \bar{B}_i) \wedge N_1(\bar{Z}_1) \rightarrow 1 = 0$,
  ..., $q_i(\bar{A}_i, \bar{B}_i) \wedge N_s(\bar{Z}_s) \rightarrow 1 = 0$.

As an example, consider the tables and integrity constraints in the schema of Figure 1. Consider also the following views:

EmpBoss(e, n, s, b) ← *WorksFor*(e, b) ∧ *Employee*(e, n, s, d) ∧ *Department*(d, *max*, *min*)
EmpBoss(e, n, s, null) ← *Employee*(e, n, s, d) ∧ *Department*(d, *max*, *min*) ∧ ¬*hasBoss*(e)
hassBoss(e) ← *WorksFor*(e, b)

The b-schema of this example contains the following (base) predicates: *WorksFor*, *Employee*, *Department*, *EmpBoss* and *hasBoss*. The integrity constraints that keep *EmpBoss* updated in the b-schema according to the deductive rules above are shown in Figure 2. Note that predicate names are abbreviated in Figure 2; *EB* stands for *EmpBos*, *hB* for *hasBoss*, *E* for *Employee*, *D* for *Department*, *WF* for *WorksFor*, and EB_1, EB_2 are the predicates in PD_{DR} that correspond to the two deductive rules of *EmpBoss*.

DEFINITION 2 (*B-Instance*). Let I_S be a database instance. The *b-instance* of I_S is

$$I_{BS} = I_S \cup Facts(DR_S, I_S),$$

where $Facts(DR, I) = \{q(\bar{X})\sigma, q_i(\bar{X}, \bar{Y})\sigma \mid q_i = (q(\bar{X}) \leftarrow L_1 \wedge ... \wedge L_k) \in DR,\ \bar{Y}$ denotes the variables in $L_1 \wedge ... \wedge L_k$, and σ is a ground substitution such that $I \vDash (L_1 \wedge ... \wedge L_k)\sigma\}$.

c_1: $WF(e, b) \wedge E(e, n, s, d) \wedge D(d, a, i) \rightarrow \exists(e', n', s', b')\ EB_1(e', n', s', b', e, b, n, s, d, a, i)$
c_2: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow WF(e, b)$
c_3: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow E(e, n, s, d)$
c_4: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow D_S(d, a, i)$
c_5: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow e' = e$
c_6: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow n' = n$
c_7: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow s' = s$
c_8: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow b' = b$
c_9: $EB_1(e', n', s', b', e, b, n, s, d, a, i) \rightarrow EB(e', n', s', b')$
c_{10}: $E(e, n, s, d) \wedge D(d, a, i) \rightarrow \exists b\ hB(e, b) \vee \exists(e', n', s', b')\ EB_2(e', n', s', b', e, n, s, d, a, i)$
c_{11}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow E(e, n, s, d)$
c_{12}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow D(d, a, i)$
c_{13}: $EB_2(e', n', s', b', e, n, s, d, a, i) \wedge hB(e, b) \rightarrow 1 = 0$
c_{14}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow e' = e$
c_{15}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow n' = n$
c_{16}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow s' = s$
c_{17}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow b' = \text{null}$
c_{18}: $EB_2(e', n', s', b', e, n, s, d, a, i) \rightarrow EB(e', n', s', b')$
c_{19}: $EB(e', n', s', b') \rightarrow \exists(e, b, n, s, d, a, i)\ EB_1(e', n', s', b', e, b, n, s, d, a, i) \vee \exists(e, n, s, d, a, i)\ EB_2(e', n', s', b', e, n, s, d, a, i)$

Figure 2: Portion of the integrity constraints of a b-schema.

LEMMA 1. *Let* S *be a database schema, and let* BS *be its b-schema. The following is true:*

- *Let* I_S *be an instance of* S*, then the b-instance* I_{BS} *of* I_S *is an instance of* BS.
- *Let* I_{BS} *be an instance of* BS*, then* I_{BS} *is the b-instance of an instance* I_S *of* S.

PROOF. It follows from (1) the fact that the set of predicate definitions of the b-schema is $PD_{BS} = PD_S \cup PD_{DR}$ and (2) the fact that a b-instance is built from the original instance I_S by materializing the derived predicates in S and populating the new predicates defined in PD_{DR}. □

LEMMA 2. *Let* I_S *be an instance of database schema* S. *Instance* I_S *is consistent if and only if the b-instance of* I_S *is a consistent instance of the b-schema of* S.

PROOF. Let us assume that I_S is a consistent instance of $S = \langle PD_S, DR_S, IC_S \rangle$. Let I_{BS} be the b-instance of I_S. By Lemma 1, I_{BS} is an instance of the b-schema $BS = \langle PD_{BS}, \emptyset, IC_{BS} \rangle$ of S. We know that $IC_{BS} = IC_S \cup IC_{DR} \cup IC_P \cup IC_N$, and that those facts in I_{BS} that are also facts of I_S do satisfy the constraints IC_S. The key point is to show that the facts in I_{BS} that are not facts of I_S do satisfy the constraints $IC_{DR} \cup IC_P \cup IC_N$.

Let us start with IC_{DR}. The constraints in the form of $q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{ju} = k_u$ and $q_i(\bar{A}_i, \bar{B}_i) \rightarrow A_{gv} = B_{hv}$ state that, in the materialized relation $q_i(\bar{X}, \bar{Y})$, $\bar{X}$ contains one variable for each term in the head of the deductive rule $q_i \in DR_S$, which can be either a constant (i.e., consequent is $A_{ju} = k_u$) or a variable from the body of q_i (i.e., consequent is $A_{gv} = B_{hv}$). We can be sure that these constraints hold on I_{BS} because of the definition of $Facts(DR_S, I_S)$, which adds $q(\bar{X})\sigma$ and $q_i(\bar{X}, \bar{Y})\sigma$ to I_{BS} for each ground substitution σ that makes the body of q_i true on I_S.

Still in IC_{DR}, the constraints in the form of $q_i(\bar{A}_i, \bar{B}_i) \rightarrow q(\bar{A}_i)$ and $q(\bar{Z}) \rightarrow \exists \bar{B}_1\ q_1(\bar{Z}, \bar{B}_1) \vee ... \vee \exists \bar{B}_i\ q_i(\bar{Z}, \bar{B}_i) \vee ... \vee \exists \bar{B}_m\ q_m(\bar{Z}, \bar{B}_m)$ state that there must be one fact $q(\bar{X})\sigma$ about the derived predicate q for each instantiation $q_i(\bar{X}, \bar{Y})\sigma$ of the deductive rule q_i, and vice versa, i.e., if there is a fact about q, it has to come from some of the deductive rules of q.

Again, this clearly holds in I_{BS} since *Facts* includes both a fact $q(\bar{X})\sigma$ about q and a fact $q_i(\bar{X}, \bar{Y})\sigma$ about q_i for each instantiation σ of each deductive rule q_i.

The constraints in IC_P keep $q_i(\bar{X}, \bar{Y})$ updated according to the body of the deductive rule when this has no negated literals. If the body holds, then the corresponding tuple must exists, and if the tuple exists, the literals in the body must all be true. This obviously holds in I_{BS} given the definition of *Facts*.

Finally, IC_N addresses the case in which the body of q_i has negated literals. It is like IC_P with some additional algebraic manipulations: in $P_1 \wedge ... \wedge P_r \rightarrow N_1(\bar{Z}_1) \vee ... \vee N_s(\bar{Z}_s) \vee q_i(\bar{A}_i, \bar{B}_i)$, the negated literals have been moved into the consequent, and in $q_i(\bar{A}_i, \bar{B}_i) \wedge N_s(\bar{Z}_s) \rightarrow 1=0$ the negated literal has been moved into the premise. As above, it follows immediately from the definition of *Facts*.

We can therefore conclude that I_{BS} satisfies all constraints in IC_{BS}, that is, I_{BS} is a consistent instance of the b-schema.

On the other direction, let us assume I_S is an instance of S whose b-instance I_{BS} satisfies the integrity constraints of the b-schema. Since the facts in I_S are also facts of I_{BS}, i.e., $I_S \subseteq I_{BS}$, and the b-schema includes the constraints of S, i.e., $IC_S \subseteq IC_{BS}$, then I_S will be consistent. □

THEOREM 1. *Derived predicate* Q *of database schema* S *is satisfiable if and only if the base predicate* Q *in the b-schema of* S *is satisfiable*.

PROOF. Let us assume derived predicate Q of schema $S = \langle PD_S, DR_S, IC_S \rangle$ is satisfiable. There is a consistent instance I_S of S that contains at least one fact about Q. By Lemma 2, the b-instance I_{BS} of I_S is a consistent instance of the b-schema. By construction of I_{BS}, we know that I_{BS} contains a fact $q(\bar{X})\sigma$ for each instantiation σ that makes true the body of a deductive rule $(q(\bar{X}) \leftarrow L_1 \wedge ... \wedge L_k) \in DR_S$ on I_S. Since Q is satisfiable on I_S, there is at least one of such instantiations, i.e., I_{BS} contains at least one fact about Q. Therefore, instance I_{BS} exemplifies that base predicate Q of the b-schema is satisfiable.

On the other direction, let us assume base predicate Q in the b-schema of S is satisfiable. There must be a consistent instance I_{BS} of the b-schema with at least one fact about Q. By Lemmas 1 and 2, there is a consistent instance I_S of S such that I_{BS} is its b-instance. Since I_{BS} contains a fact $q(\bar{X})\sigma$ about Q, and by definition of b-instance, there must be an instantiation σ and a deductive rule $(q(\bar{X}) \leftarrow L_1 \wedge ... \wedge L_k) \in DR_S$ such that $(L_1 \wedge ... \wedge L_k)\sigma$ is true on I_S. Therefore, Q has a non-empty answer on I_S, i.e., I_S exemplifies that Q is satisfiable on S. □

3.2 The Termination Test

Once derived predicates have been materialized, we can already apply the termination test. The test is aimed to determine whether the CQC method is guaranteed to terminate for a particular query satisfiability checking. We adapt to the query satisfiability context the test proposed in [6] in the context of reasoning on UML conceptual schemas. The test consists of two activities: the construction of a dependency graph and the analysis of the cycles in that graph.

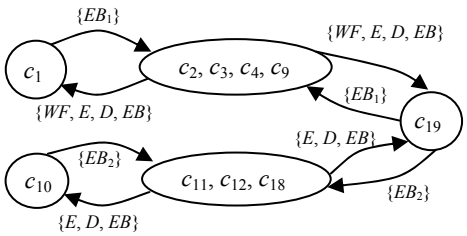


Figure 3: Portion of a dependency graph.

3.2.1 Dependency Graph

The dependency graph is intended to show the dependencies that exist between the integrity constraints of a given schema, in our case, the b-schema of the database.

DEFINITION 3 (*Potential Violation* and *Repair*). A literal $p(\bar{X})$ is a *potential violation* of an integrity constraint $ic \in IC_{ST}$ if it appears in the premise of the constraint. We denote by $\mathsf{PV}(ic)$ the set of potential violations of ic. There is a *repair* $\mathsf{RE}_i(ic) = \{L_i\}$ for each ordinary literal L_i in the consequent of ic. A variable X is *free* in a repair $\mathsf{RE}_i(ic)$ if $X \in \text{vars}(\mathsf{RE}_i(ic))$ and $X \notin \text{vars}(\mathsf{PV}(ic))$.

DEFINITION 4 (*Dependency Graph*). A *dependency graph* is a graph such that each vertex corresponds to an integrity constraint $ic_i \in IC_{ST}$. There is an arc labeled $\mathsf{RE}_k(ic_i)$ from ic_i to ic_j if there exists $p(\bar{X})$, $p(\bar{Y})$ such that $p(\bar{X}) \in \mathsf{RE}_k(ic_i)$ and $p(\bar{Y}) \in \mathsf{PV}(ic_j)$. Note that there may be more than one arc from ic_i to ic_j, since two different repairs of ic_i may lead to the violation of ic_j. Also, note that only the integrity constraints that have ordinary literals in its consequent are considered in the dependency graph.

A maximal set of constraints $SP = \{ic_1, ..., ic_s\}$ such that $ic_1, ..., ic_s$ have the same premise (modulo renaming of variables) is considered as a single constraint ic' from the point of view of the graph; thus, it corresponds to a single vertex. Let $L_{1,1} \vee ... \vee L_{1,r1}, ..., L_{s,1} \vee ... \vee L_{s,rs}$ be the consequents of the constraints in SP; there is a repair $\mathsf{RE}_k(ic') = \{L_{1,j1}, ..., L_{s,js}\}$ for each combination $j_1, ..., j_s$ with $1 \le j_1 \le r_1, ..., 1 \le j_s \le r_s$. The incoming and outgoing arcs of ic' in the graph are computed as defined above.

Figure 3 shows a portion of the dependency graph for the example in Section 3.1. It shows the dependencies that exist between the constraints c_1 to c_{19} from Figure 2. The literals in the repairs that label the arcs of the graph appear abbreviated.

3.2.2 Analysis of Cycles

Once constructed the dependency graph, each cycle in the graph must be checked to be finite. Intuitively, a cycle C in the dependency graph is finite if for all finite instance I of the b-schema, integrity maintenance of I with cycle C results in a finite instance I', that is, C cannot cause an infinite sequence of violations and repairs. When we say integrity maintenance with cycle C, we mean that only the constraints in C are considered, that they are checked in the order they appear in C, and that if a point is reached where the constraint under consideration is not violated, then the process ends.

DEFINITION 5 (*Cycle* and *Nested Cycle*). A *cycle* is a sequence in the form of $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$, where r_i denotes the label of the arc in the dependency graph from ic_i to ic_{i+1}. We say C is a *nested cycle* of cycle C' if C' is of the form $C' = (ic_1', r_1', ..., ic_1, r_1, ..., ic_2, r_2, ..., ic_n, r_n, ..., ic_m', r_m', ic_{m+1}' = ic_1')$.

DEFINITION 6 (*Finite Cycle*). We say cycle $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ is *finite* if

(1) *its nested cycles are finite*

(2) *for each possible starting* ic_i, $1 \le i \le n$, *and for all finite b-schema instance I, there exists a constant k such that* $Maint_k(I, ic_i) = Maint_{k+1}(I, ic_i)$ *(i.e., integrity maintenance reaches a fix-point)*, where $Maint_j$ denotes the result of the first j steps of the integrity maintenance process and is defined as follows:

$$Maint_0(I, ic_i) = I$$
$$Maint_j(I, ic_i) = Maint_{j-1}(I, ic_i) \cup r_{i+j-1}\theta_1 \cup ... \cup r_{i+j-1}\theta_m, \quad j > 0$$

where each $\theta_t = \delta_t \cup \delta_t'$ is one of the m possible instantiations such that $Maint_{j-1}(I, ic_i) \models \mathsf{PV}(ic_{i+j-1})\delta_t$, $\nexists\rho$ $(Maint_{j-1}(I, ic_i) \models (r_{i+j-1})(\delta_t \cup \rho))$, δ_t' instantiates each free variable in r_{i+j-1} with a fresh constant, and, if $j > 1$, $(\mathsf{PV}(ic_{i+j-1})\delta_t \cap (Maint_{j-1}(I, ic_i) - Maint_{j-2}(I, ic_i))) \neq \emptyset$.

Note that *Maint* assumes the worst case scenario for the integrity maintenance process, that is, assumes the built-in literals (i.e., the arithmetic comparisons) in the premise of an integrity constraint are always evaluated true, i.e., they will not prevent the violation of the constraint (see conditions "$Maint_{j-1}(I, ic_i) \models \mathsf{PV}(ic_{i+j-1})\delta_t$" and "$\nexists\rho$ $(Maint_{j-1}(I, ic_i) \models (r_{i+j-1})(\delta_t \cup \rho))$" in Definition 6); and assumes the free variables in the constraints are always instantiated with fresh constants, i.e., the cycle will not be closed by the reuse of previously used constants. Extending the termination test beyond this worst case scenario, i.e., exploiting the presence of comparisons and the reuse of variables in order to detect more finite cases, would be an interesting topic for further research.

The termination test consists in evaluating, on each cycle C, three sufficient conditions for C to be finite. It is important to note that each one of these conditions assumes that any nested cycle of C is tested separately.

THEOREM 2 (*Condition 1*). *A cycle* $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ *is a finite cycle if its nested cycles are finite and, for all constraint* ic_i *in* C *and for all pair of literals* $p(X_1, ..., X_m) \in r_i$ *and* $p(Y_1, ..., Y_m) \in \mathsf{PV}(ic_{i+1})$, *variable* X_k *being free in* r_i *implies* $Y_k \notin vars(r_{i+1})$, $1 \le k \le m$.

As an example, consider the following cycle:

$$(ic_1) \quad p(X) \rightarrow \exists Y\, q(X, Y)$$
$$(ic_2) \quad q(X, Y) \rightarrow p(X)$$

Since ic_2 does not propagate the value of the existentially quantified variable in ic_1, we obtain at the end of the cycle the same fact about p that violated ic_1 in the first place.

Another example is the cycles in Figure 3.

PROOF. Performing integrity maintenance of an instance I with cycle C when nested cycles are disregarded and *Condition 1* holds can be reduced to chasing I with a *weakly acyclic* set of tgds, which is a well-known finite process [2]. The reduction is quite straightforward; each pair ic_i r_i in C produces a tgd $\mathsf{PV}(ic_i) \rightarrow r_i$. The only technical detail to take care of is the existence of nested cycles, i.e., the fact that some r_i in C may contain a literal that is not only a potential violation of the next constraint but also of some other constraint in C. Since the nested cycles are to be tested finite separately, and in order to avoid the nested cycles to "interfere" with the chase of the current cycle, we modify the name of the predicates in C so a fact introduced by the repair of ic_i only matches the potential violations of ic_{i+1}, e.g., $\{r(x) \rightarrow \exists y\ t(x, y),\ \ t(x, y) \wedge v(y) \rightarrow s(x),\ \ s(x) \wedge t(x, y) \rightarrow r(y)\}$ would become $\{r(x) \rightarrow \exists y\ t_1(x, y),\ \ t_1(x, y) \wedge v(y) \rightarrow s(x),\ \ s(x) \wedge t_2(x, y) \rightarrow r(y)\}$. Now, since *Condition 1* forbids the propagation of the existentially quantified variables of any constraint by the next, there will be no cycles going through *special edges* (see [2]), which is the definition of weakly acyclic set of tgds. □

Consider now the following cycle, which would not be detected as finite by the previous condition:

$$(ic_1) \quad p(X, Y) \wedge q(X, Z) \rightarrow r(X)$$
$$(ic_2) \quad r(X) \rightarrow \exists Y \exists Z\, p(Y, Z)$$

We can see that since neither ic_1 nor ic_2 insert new tuples about q, given any finite instance I, the repair of ic_2 may only lead to new violations of ic_1 a finite number of times, one for each fact about q in I. Nevertheless, performing integrity maintenance through this cycle will certainly stop after a finite number of iterations. A new condition is thus necessary to identify this kind of finite cycles; see the following.

THEOREM 3 (*Condition 2*). *A cycle* $\mathrm{C} = (\mathrm{ic}_1, \mathrm{r}_1, ..., \mathrm{ic}_n, \mathrm{r}_n, \mathrm{ic}_{n+1} = \mathrm{ic}_1)$ *is a finite cycle if its nested cycles are finite and it contains a constraint* ic_i *that satisfies the following. Let* $\mathrm{UR} = \{\mathrm{p} \mid \mathrm{p}(\bar{\mathrm{X}}) \in \mathrm{r_j},\ 1 \le \mathrm{j} \le \mathrm{n}\}$ *be the union of repairs of the constraints in* C, *and let* $\mathrm{UV_i} = \{\mathrm{q} \mid \mathrm{q}(\bar{\mathrm{Y}}) \in \mathsf{PV}(\mathrm{ic_i})\}$ *be the union of potential violations of* $\mathrm{ic_i}$. *Then*,

(1) $(\mathrm{UR} \cap \mathrm{UV_i}) \subset \mathrm{UV_i}$, *where* $\{\mathrm{L}_1, ..., \mathrm{L_k}\}$ *are the literals in* $\mathsf{PV}(\mathrm{ic_i})$ *whose predicates belong to* $\mathrm{UV_i}$ *but not to* UR, *and*

(2) $\mathrm{vars}(\mathrm{r_i}) \subseteq \mathrm{vars}(\{\mathrm{L}_1, ..., \mathrm{L_k}\})$.

PROOF. Let us assume the nested cycles have been tested finite. Let I be any finite instance of the b-schema. Let $\sigma_1, ..., \sigma_m$ be the m possible ground substitutions such that $I \models (L_1 \wedge ... \wedge L_k)\sigma_j$, $1 \le j \le m$. Let us suppose performing integrity maintenance on I with cycle C produces an infinite instance. Let I' be I after m iterations of the integrity maintenance process. I' must violate some constraint in the cycle; let us assume it is $ic_i = L_1 \wedge ... \wedge L_k \wedge L_{k+1} \wedge ... \wedge L_n \rightarrow L_{n+1} \vee ... \vee L_{n+r}$ (otherwise, we keep following the sequence of violations and repairs until we reach ic_i). Let δ be such that $I' \models (L_1 \wedge ... \wedge L_k \wedge L_{k+1} \wedge ... \wedge L_n)\delta$ and $I' \not\models (L_{n+1} \vee ... \vee L_{n+r})\delta$. By point (1), $\exists j$, $1 \le j \le m$, $\delta = \sigma_j \cup \delta'$ and $I' \models (L_1 \wedge ... \wedge L_k)\sigma_j$. By point (2), $\mathrm{vars}(L_{n+1} \vee ... \vee L_{n+r}) \subseteq \mathrm{vars}(L_1 \wedge ... \wedge L_k)$. Therefore, $I' \not\models (L_{n+1} \vee ... \vee L_{n+r})\sigma_j$. However, since ic_i has already been violated and repaired m

times, $I' \supseteq \{L_{n+1}\sigma_j, ..., L_{n+r}\sigma_j \mid 1 \leq j \leq m\}$, which means $I' \models (L_{n+1} \vee ... \vee L_{n+r})\sigma_j$. So, we have reached a contradiction. □

The last condition (see below) simulates one iteration of integrity maintenance through the cycle, starting from a canonical database instance. The simulation is performed for each possible starting constraint in the cycle. The idea is to see whether all these simulations reach a fix-point.

THEOREM 4 (*Condition 3*). *A cycle* $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ *is a finite cycle if its nested cycles are finite and, for each possible starting* ic_i, $\exists k$, $1 \leq k \leq n$, *such that* $Sim_k(ic_i) = Sim_{k+1}(ic_i)$, *where* Sim *simulates an integrity maintenance process on a canonical instance as follows*:

$$\begin{aligned} Sim_0(ic_i) &= PV(ic_i)\sigma_0 \\ Sim_j(ic_i) &= Maint_1(Sim_{j-1}(ic_i), ic_{i+j-1}) \\ &\quad \cup PV(ic_{i+j})\sigma_1 \cup ... \cup PV(ic_{i+j})\sigma_r \qquad j > 0 \end{aligned}$$

and the following holds:

(*i*) *Substitution* σ_0 *assigns a fresh constant to each variable*.

(*ii*) *For each* $L \subseteq Maint_1(Sim_{j-1}(ic_i), ic_{i+j-1})$ *and* $M \subseteq PV(ic_{i+j})$ *such that* $(L \cap (Maint_1(Sim_{j-1}(ic_i), ic_{i+j-1}) - Sim_{j-1}(ic_i))) \neq \emptyset$ *and there is a most general unifier* δ *of* L *and* M, *then there exists* $\sigma_s = \delta \cup \sigma_s'$, $1 \leq s \leq r$, *where* σ_s' *assigns a fresh constant to each variable in* $PV(ic_{i+j})\delta - M\delta$.

Intuitively, the simulation begins with the construction of a canonical instance that "freezes" each variable in the premise of ic_i into a constant (point (i)). Then, *Sim* evaluates the premise of the constraint, disregarding the arithmetic comparisons, and, if the constraint is violated, adds the necessary facts to repair it (definition of $Maint_1$). Additionally, for each subset of existing facts that includes at least one of the previous repairs and that can be unified with some portion of the premise of the next constraint, it freezes the non-unified variables of this next premise into constants, and inserts the resulting facts (point (ii)); this is required since we want the satisfaction of a constraint to come from its repairs already holding and not from its potential violations being false. The process moves from one constraint in the cycle to the next, until it completes one iteration of the cycle or reaches a constraint that does not need to be repaired. As an example, consider the following cycle:

$$\begin{aligned} (ic_1) &\begin{cases} A(X) \rightarrow \exists Y\ B(X, Y) \\ A(X) \rightarrow \exists Y\ E(X, Y) \end{cases} \\ (ic_2) &\quad B(X, Y) \wedge C(X, Z) \rightarrow D(X, Y, Z) \\ (ic_3) &\begin{cases} D(X, Y, Z) \rightarrow A(X) \\ D(X, Y, Z) \rightarrow \exists V\ E(X, V) \end{cases} \end{aligned}$$

The successive violation and repair of constraints ic_1 and ic_2 leads to the satisfaction of the two consequents in ic_3, that is, $A(X)$ and $\exists V\ E(X, V)$ in ic_3 are guaranteed to hold because of the violation of ic_1 (remind that in order to violate a constraint, its premise must hold) and its repair, respectively. Similarly, in the case in which the integrity

maintenance process starts with ic_2, the violation and repair of ic_2, ic_3 leads to the satisfaction of ic_1. In the case in which it starts with ic_3, the violation and repair of ic_3, ic_1, ic_2 leads to the satisfaction of ic_3. Therefore, the simulation of one iteration of integrity maintenance always reaches a fix-point. The case in which it starts at ic_1 is shown below:

$$Sim_0(ic_1) = \{A(x)\}$$
$$Sim_1(ic_1) = Sim_0(ic_1) \cup \{B(x, y), E(x, y_2), C(x, z)\}$$
$$Sim_2(ic_1) = Sim_1(ic_1) \cup \{D(x, y, z)\}$$
$$Sim_3(ic_1) = Sim_2(ic_1) \cup \varnothing$$

Notice the insertion of $C(x, z)$ in Sim_1, which ensures the satisfaction of the premise of ic_2 in the next step of the simulation.

The conclusion is that the cycle in the example is finite, and that while *Condition 1* and *Condition 2* of the termination test do not hold, *Condition 3* does.

PROOF. Assume the nested cycles have been tested finite separately. Let us take as hypothesis that *Condition 3* holds, but assume there is a finite instance I such that integrity maintenance of I with cycle $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ is an infinite process. From *Condition 3*, we know that for each ic_h, $1 \le h \le n$, there is a certain constant $k \le n$, which we assume is the lowest possible, such that $Sim_k(ic_h) = Sim_{k+1}(ic_h)$. From I being infinite, we know there must be an ic_i, $1 \le i \le n$, such that $\forall j$, $j \ge 0$, $Maint_j(I, ic_i) \subset Maint_{j+1}(I, ic_i)$. Let us assume, without lost of generality, $i = 1$. That implies there exists an infinite sequence of constraint violations and repairs $ic_1\theta_1$, $r_1(\theta_1 \cup \theta_1')$, ..., $ic_{k+1}\theta_{k+1}$, $r_{k+1}(\theta_{k+1} \cup \theta_{k+1}')$, ... where, by definition of *Maint*, the following is true: $\forall j$, $j \ge 1$, $Maint_{j-1}(I, ic_1) \vDash \mathsf{PV}(ic_j)\theta_j$, $\nexists \rho$ $(Maint_{j-1}(I, ic_1) \vDash r_j(\theta_j \cup \rho))$, θ_j' is an instantiation for the free variables in r_j, and, if $j > 1$, $(\mathsf{PV}(ic_j)\theta_j \cap r_{j-1}(\theta_{j-1} \cup \theta_{j-1}')) \neq \varnothing$.

Our goal is to show that, since *Condition 3* holds, we can build a similar sequence $ic_1\delta_1$, $r_1(\delta_1 \cup \delta_1')$, ..., $ic_{k+1}\delta_{k+1}$, $r_{k+1}(\delta_{k+1} \cup \delta_{k+1}')$, where $Sim_k(ic_1) \vDash r_{k+1}(\delta_{k+1} \cup \delta_{k+1}')$ and the following is true: $\forall j$, $1 \le j \le k+1$, $Sim_{j-1}(ic_1) \vDash \mathsf{PV}(ic_j)\delta_j$, $\nexists \rho$ $(Sim_{j-1}(I, ic_i) \vDash r_j(\delta_j \cup \rho)$ and $j < k+1)$, δ_j' is an instantiation for the free variables in r_j, and, if $j > 1$, $(r_{j-1}\delta_{j-1}' \cap ic_j\delta_j) \neq \varnothing$. We will also show that this sequence represents a finite execution of an integrity maintenance process, and that it can be "unified" with the $2(k+1)$ first elements of the previous (infinite) sequence. That will lead us to a contradiction.

We know that $Sim_0(ic_1)$ is a canonical instance built by freezing the variables in $\mathsf{PV}(ic_1)$ into constants (point (i)), so let δ_1 be that instantiation. We also know that θ_1 unifies each literal in $\mathsf{PV}(ic_1)$ with a certain fact from I. Therefore, we can define (with a slight abuse of notation) a substitution σ_1 from the frozen variables in $ic_1\delta_1$ to the constants in $ic_1\theta_1$ such that $(ic_1\delta_1)\sigma_1 = ic_1\theta_1$. At this point, we have unified the first element of the two sequences.

Similarly, we know that $Sim_1(ic_1)$ instantiates the free variables in r_1 with fresh constants (definition of $Maint_1$), so let δ_1' be that instantiation. Since $(r_1\theta_1' \cap ic_2\theta_2) \neq \varnothing$, we use UR_1, UI_2 to denote the non-instantiated literals in r_1, ic_2, respectively, that are the source of the facts in $(r_1\theta_1' \cap ic_2\theta_2)$. We define σ_1' as the substitution that replaces the frozen variables in $UR_1(\delta_1 \cup \delta_1')$ with the constants in $UI_2\theta_2$ so $UR_1(\delta_1 \cup \delta_1')(\sigma_1 \cup \sigma_1') = UI_2\theta_2$; and we have unified the second element of the sequences.

Now, we apply induction and focus on an intermediate ic_j, $1 < j \le k+1$. Our hypothesis of induction is that what we just did in reference to ic_1 can be done in reference to all ic_i,

$1 \leq i < j$. By point (ii), we know there is δ_j such that $Sim_{j-1} \vDash \mathsf{PV}(ic_j)\delta_j$. Then, for each fact F_s in $\mathsf{PV}(ic_j)\theta_j$, there are two possibilities: either (1) $\exists t$, $1 \leq t < j$, such that F_s appears in $ic_t\theta_t$ or $r_t\theta_t'$, in which case, by hypothesis of induction, we already have defined a substitution σ_t that unifies a certain fact in $Sim_t(ic_1)$ with F_s, and we define $\gamma_s = \sigma_t$; or (2) F_s does not appear before in the sequence, in which case, we know from point (ii) that there is a fact F_s' in $\mathsf{PV}(ic_j)\delta_j$ that can be unified with F_s, and we define γ_s as that unifier. Finally, we define σ_j as the union of these γ_s's and get $(ic_j\delta_j)\sigma_j = ic_j\theta_j$. We have unified $ic_1\delta_1$, $r_1(\delta_1 \cup \delta_1')$, ..., $ic_j\delta_j$ with $ic_1\theta_1$, $r_1(\theta_1 \cup \theta_1')$, ... $ic_j\theta_j$.

Now, if $j \leq k$, we can proceed as we did with ic_1; otherwise, $j = k+1$, and we know that no new facts are added by Sim_{k+1}, which means $\exists\delta_{k+1}'$, $Sim_k(ic_1) \vDash (r_{k+1})(\delta_{k+1} \cup \delta_{k+1}')$. By hypothesis of induction, we have already defined a σ_t for each fact in $Sim_k(ic_1)$ that unifies it with some fact in $Maint_k(I, ic_1)$. Therefore, we can define σ_{k+1}' as the union of these σ_t's and get $r_{k+1}(\delta_{k+1} \cup \delta_{k+1}')(\sigma_k \cup \sigma_{k+1}') = r_{k+1}(\theta_{k+1} \cup \theta_{k+1}')$. We have finally unified the $2(k+1)$ first elements of the two sequences.

Since $r_{k+1}(\delta_{k+1} \cup \delta_{k+1}') \subseteq Sim_k(ic_1)$, we can conclude $r_{k+1}(\theta_{k+1} \cup \theta_{k+1}') \subseteq Maint_k(I, ic_1)$, which contradicts with our assumption that $\forall j, j \geq 1, \nexists\rho\ Maint_{j-1}(I, ic_1) \vDash r_j(\theta_j \cup \rho)$. In other words, the facts introduced by $Maint_{k+1}(I, ic_1)$ to repair ic_{k+1} do already exist in $Maint_k(I, ic_1)$, which means ic_{k+1} is not actually violated, and that contradicts with our assumption that the sequence of violations and repairs is infinite. □

COROLLARY 1. *If all cycles in the dependency graph of a given b-schema are finite, then checking the satisfiability of a predicate from the b-schema with the CQC method is a finite process*.

The corollary follows immediately from the definition of finite cycle and the fact that the CQC method is, after the first query satisfiability phase, an integrity maintenance process.

Note that the three conditions in the termination test are sufficient but not necessary, as expected due to the undecidability of the termination checking problem. Note also that the evaluation of the conditions is decidable.

4. Conclusion

We have proposed the performance of a termination test in order to determine whether a given query satisfiability check performed by means of the CQC method is guaranteed to terminate or not. We have adapted an existing termination test from the context of UML conceptual schemas to the query satisfiability context, and extended it so it can deal with the class of database schemas the CQC method works on. We have also provided formal proofs for the obtained results.

Acknowledgements

This work has been supported in part by Microsoft Research through the European PhD Scholarship Programme, and by the Ministerio de Ciencia e Innovación under project TIN2008-03863.

5. References

[1] Alin Deutsch, Val Tannen: Optimization Properties for Classes of Conjunctive Regular Path Queries. DBPL 2001: 21-39

[2] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, Lucian Popa: Data exchange: semantics and query answering. Theor. Comput. Sci. 336(1): 89-124 (2005)

[3] Carles Farré, Ernest Teniente, Toni Urpí: A New Approach for Checking Schema Validation Properties. DEXA 2004: 77-86

[4] Carles Farré, Ernest Teniente, Toni Urpí: Checking query containment with the CQC method. Data Knowl. Eng. 53(2): 163-223 (2005)

[5] Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, Oded Shmueli: Static analysis in datalog extensions. J. ACM 48(5): 971-1012 (2001)

[6] Anna Queralt, Ernest Teniente: Decidable Reasoning in UML Schemas with Constraints. CAiSE 2008: 281-295

[7] Jeffrey D. Ullman: Principles of Database and Knowledge-Base Systems, Volume II Computer Science Press 1989

[8] Xubo Zhang, Z. Meral Özsoyoglu: Implication and Referential Constraints: A New Formal Reasoning. IEEE Trans. Knowl. Data Eng. 9(6): 894-910 (1997)