Validation of Schema Mappings with Nested Queries

Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí

Departament d'Enginyeria de Serveis i Sistemes d'Informació
Universitat Politècnica de Catalunya (UPC)—BarcelonaTech
1-3 Jordi Girona, 08034 Barcelona, Spain
{grull, farre, teniente, urpi}@essi.upc.edu

Abstract. With the emergence of the Web and the wide use of XML for representing data, the ability to map not only flat relational but also nested data has become crucial. The design of schema mappings is a semi-automatic process. A human designer is needed to guide the process, choose among mapping candidates, and successively refine the mapping. The designer needs a way to figure out whether the mapping is what was intended. Our approach to mapping validation allows the designer to check whether the mapping satisfies certain desirable properties. In this paper, we focus on the validation of mappings between nested relational schemas, in which the mapping assertions are either inclusions or equalities of nested queries. We focus on the nested relational setting since most XML's Document Type Definitions (DTDs) can be represented in this model. We perform the validation by reasoning on the schemas and mapping definition. In particular, we encode the given mapping scenario into a single flat database schema, and reformulate each desirable property check as a query satisfiability problem.

Keywords: schema mapping, nested relational model, nested query, query equality, query inclusion, validation.

1. Introduction

Schema mappings are specifications that model a relationship between two data schemas. They are key elements in any system that requires the interaction of heterogeneous data and applications [16]. Such interaction usually involves databases that have been independently developed and that store the data of the common domain under different representations; that is, the involved databases have different schemas. In order to make the interaction possible, schema mappings are required to indicate how the data stored in each database relates to the data stored in the other databases. This problem, known as *information integration*, has been recognized as a challenge faced by all major organizations, including enterprises and governments [5].

Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí

With the emergence of the Web and the wide use of XML for representing data, the ability to map not only flat relational but also nested data has become crucial. A sign of this is the growing interest of the research community during the last years on the topics of XML mappings—see, for instance, [3, 4]—and mappings between nested relational schemas—e.g., [20, 15].

However, the mapping design process is not a fully automatic one. A human designer is needed to guide the process, choose among mapping candidates, and successively refine the mapping [20]. Intricate manual work may actually be required to refine a particular mapping. Since manual design is labor-intensive and error-prone, the designer needs a way to figure out whether the mapping is what was intended.

In order to address this need of validation, we propose an approach that allows the designer to ask questions about the mapping. In particular, it allows the designer to check whether the mapping satisfies certain desirable properties. In this paper, we focus on two properties that have been identified as important properties of mappings in the literature: *mapping satisfiability* [3] and *mapping inference* [19]. An additional property, *mapping losslessness* [22], is also addressed in the extended version of the paper [23].

Our approach is based on reasoning on the schemas and the mapping definition, and does not rely on specific schema instances, since that might not reveal all the potential pitfalls.

In this paper, we focus on the application of this validation approach to mapping scenarios in which nested data is involved. More specifically, we address the validation of mapping scenarios in which the source and the target schema are nested relational [20], and in which the mapping is a set of assertions. Mapping assertions are in the form of either query inclusions, i.e., $Q_S \subseteq Q_T$, or query equalities, i.e., $Q_S = Q_T$, where Q_S and Q_T are queries over the source and the target schema, respectively, and whose result is a nested relation (i.e., Q_S and Q_T are *nested queries*). Note that a query inclusion (equality) assertion holds for a given pair of mapped schema instances if and only if the answer to Q_S over the source instance is a subset of (equal to) the answer to Q_T over the target instance.

We focus on the nested relational setting since it covers the most common class of the well-known Document Type Definitions (DTDs) [3], and also because it is the model that is typically used in the data exchange context to represent semi-structured schemas [20].

The class of schemas and mappings that we consider is quite expressive. We consider schemas with integrity constraints, where these constraints are in the form of disjunctive embedded dependencies [10] (this class of dependencies is applied here to the nested relational setting instead of the traditional flat relational one in the same way as tuple-generating dependencies are applied to the nested relational setting in [20]). The integrity constraints of the schemas and the queries of the mapping may contain arithmetic comparisons and negations. Union of nested queries is also allowed. This class of mapping scenarios subsumes those considered by previous works on mapping validation [7, 6, 1], which also focus on the nested

relational setting but do not consider arithmetic comparisons nor negation. Moreover, these previous works deal with a class of constraints and mapping assertions—in the form of tuple-generating dependencies [13]—that is known to be a particular class of the disjunctive embedded dependencies that we consider [10].

To actually perform the validation, we propose a reformulation of each desirable property check in terms of the query satisfiability problem over a single flat relational database. Given a nested relational mapping scenario, we encode it into a flat database and define a query over this database such that the query is satisfiable if and only if the desirable property holds. This encoding takes into account the nested structure of the schemas, their integrity constraints, and the nested queries defined over them. Moreover, this encoding rewrites the mapping assertions as integrity constraints over the new flat relational database.

In this way, we extend our previous work on validating relational mappings [22] and make it applicable to the nested case.

We solve the query satisfiability problem by means of the Constructive Query Containment (CQC) method [14]. This method is able to deal with flat relational databases in which queries and integrity constraints have no recursion and may contain safe negation—on base and derived predicates—, equality and inequality ($\neq$) comparisons, and also order comparisons ($<$, $\leq$, $>$, $\geq$). To the best of our knowledge, the CQC method is the only query satisfiability method able to handle this class of schemas and queries. The use of the this method together with the encoding that we present in this paper is what allows us to address nested relational mapping scenarios that are more expressive than the ones addressed in the previous literature.

Reasoning on the class of mapping scenarios that we consider here is, unfortunately, undecidable. However, extending the approach proposed by [21], we studied in [24] a series of conditions that, if satisfied, guarantee the termination of the CQC method for the current query satisfiability check. A detailed performance evaluation of the CQC method has been done in [22, 24] for the case of flat relational mapping scenarios. This performance evaluation showed that, for those scenarios in which termination is guaranteed, the cost of the method is exponential with respect to the size of the mapping scenario, as expected given the complexity of reasoning on such an expressive language.

We would also like to remark that the reduction that we propose of each desirable property in terms of query satisfiability is linear with respect to the size of the given mapping scenario. Moreover, this reduction does not increase the complexity of the problem, that is, checking query satisfiability is not more complex than checking the desirable properties [22].

We have performed some experiments to show the feasibility of our approach, using mapping scenarios from the *STBenchmark* [2]. The results are reported in the extended version of the paper [23].

Summarizing, the main contributions of the paper are the following:

- We validate nested relational mappings by means of checking whether they satisfy certain desirable properties. We focus on two properties that have

been identified as important properties of mappings: mapping satisfiability and mapping inference.

- We consider a class of mapping scenarios that is significantly more expressive than those considered by previous works on nested relational mapping validation.
- We propose an encoding of the nested relational schemas in the mapping scenario into a single flat relational database.
- We propose a rewriting of the mapping assertions as integrity constraints over the new relational database.
- We extend our previous work on validating relational mappings [22] to the nested relational case. In particular, we propose a reformulation of each desirable property of nested relational mappings in terms of the query satisfiability problem over a flat relational database. Such a query satisfiability check can be solved by means of the CQC method.

To better motivate the kind of validation that we propose, the next subsection discusses detailed examples. The rest of the paper is structured as follows. Section 2 introduces base concepts. Section 3 outlines our approach for validating mappings with nested queries. Section 4 and Section 5 detail how to encode a given nested relational mapping scenario into a single flat database schema. Section 6 explains how to reformulate the check of each desirable property of mappings in terms of the query satisfiability problem. Section 7 reviews the related work. Section 8 concludes the paper.

1.1 Examples of Mapping Validation

Consider a mapping scenario in which an airline company wants to publish information about their flights and connecting flights into a certain flight-searching Web site. Fig. 1 shows the source and the target schema of this scenario, where dashed lines denote referential constraints and the underlined attribute denotes a key.

Example 1
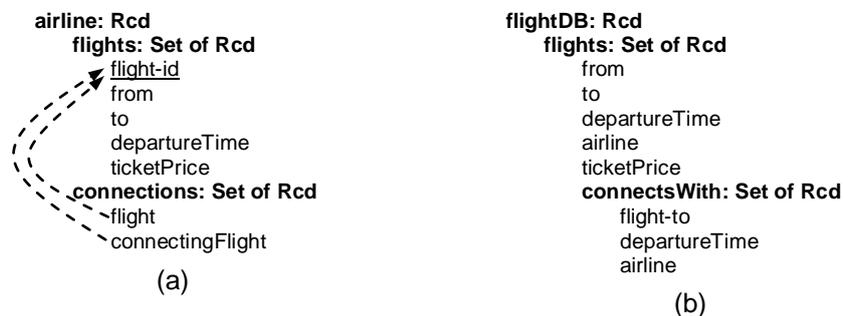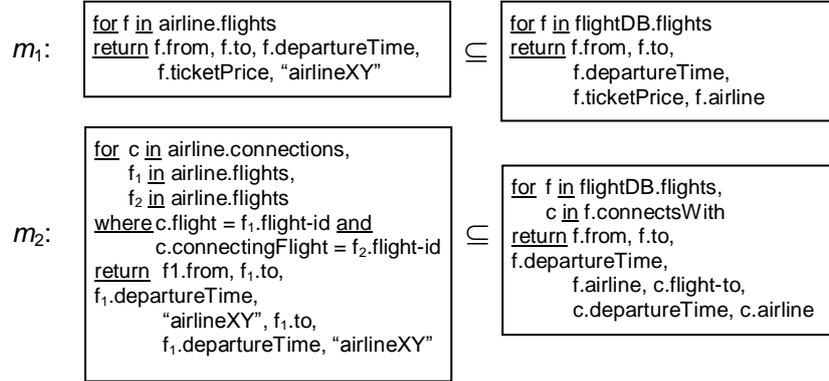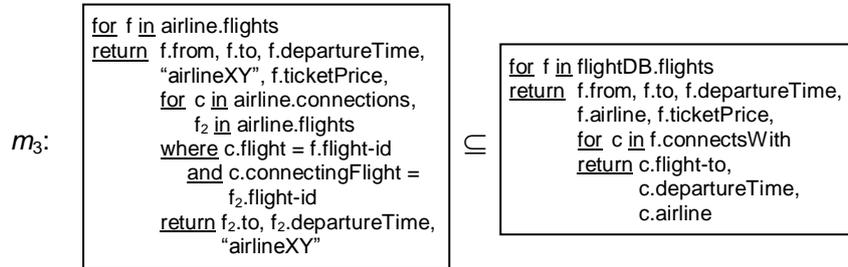Let us assume the mapping designer has come up with two mapping

airline: Rcd
  flights: Set of Rcd
    <u>flight-id</u>
    from
    to
    departureTime
    ticketPrice
  connections: Set of Rcd
    flight
    connectingFlight
      (a)

flightDB: Rcd
  flights: Set of Rcd
    from
    to
    departureTime
    airline
    ticketPrice
  connectsWith: Set of Rcd
    flight-to
    departureTime
    airline
      (b)

Fig. 1. *Example source (a) and target (b) nested relational schemas.*

candidates. The first candidate is a mapping with two assertions: $\{m_1, m_2\}$. Assertion m_1 maps the information of individual flights available in the source schema, independently of whether these flights have connecting flights or not. Assertion m_2, maps the information about the connecting flights.

m_1:

```
for f in airline.flights
return f.from, f.to, f.departureTime,
       f.ticketPrice, "airlineXY"
```
$\subseteq$
```
for f in flightDB.flights
return f.from, f.to,
       f.departureTime,
       f.ticketPrice, f.airline
```

m_2:

```
for c in airline.connections,
    f_1 in airline.flights,
    f_2 in airline.flights
where c.flight = f_1.flight-id and
      c.connectingFlight = f_2.flight-id
return  f1.from, f_1.to,
f_1.departureTime,
       "airlineXY", f_1.to,
       f_1.departureTime, "airlineXY"
```
$\subseteq$
```
for f in flightDB.flights,
    c in f.connectsWith
return f.from, f.to,
f.departureTime,
       f.airline, c.flight-to,
       c.departureTime, c.airline
```

The second candidate is a mapping with a single assertion: $\{m_3\}$. Assertion m_3 maps both the information of individual flights and of their connecting flights at the same time. It uses nested queries to ensure that flights without connecting flights are also mapped; that is, for each flight in the source, it creates a tuple that contains not only the flight's data but also a set with the corresponding connecting flights; a set that may be empty if the flight has no connecting flights.

m_3:

```
for f in airline.flights
return  f.from, f.to, f.departureTime,
       "airlineXY", f.ticketPrice,
       for c in airline.connections,
           f_2 in airline.flights
       where c.flight = f.flight-id
         and c.connectingFlight =
             f_2.flight-id
       return f_2.to, f_2.departureTime,
             "airlineXY"
```
$\subseteq$
```
for f in flightDB.flights
return  f.from, f.to, f.departureTime,
       f.airline, f.ticketPrice,
       for c in f.connectsWith
       return c.flight-to,
             c.departureTime,
             c.airline
```

The designer could think that both mapping candidates may be actually equivalent and that in that case he would feel more inclined to choose mapping $\{m_3\}$ since it seems more compact. Let us suppose that the designer wants to be sure before making the decision. He could then check whether m_3 is actually inferred from $\{m_1, m_2\}$, and whether m_1 and m_2 are both inferred from $\{m_3\}$.

The check of the *mapping inference* property [19] would reveal that while assertions m_1 and m_2 are indeed inferred from mapping $\{m_3\}$, assertion m_3 is not inferred from mapping $\{m_1, m_2\}$. Fig. 2 shows an instantiation of the mapping scenario that exemplifies the latter, i.e., it shows a source and a target instance that satisfy $\{m_1, m_2\}$ but not m_3. The example shows that mapping $\{m_1, m_2\}$ does not ensure the correlation between a flight's ticket price and the flight's connecting flights. Notice that there is one single flight

(a) Source instance:

flights				
flight-id	from	to	departureTime	ticketPrice
1	A	B	T_1	50
2	A	C	T_2	70
3	C	D	T_3	45
4	C	E	T_4	60
5	C	F	T_5	55

connections	
flight	connectingFlight
2	3
2	4
2	5

(b) Target instance:

flights							
from	to	departureTime	airline	ticketPrice	connectsWith		
A	B	T_1	airlineXY	50	$\varnothing$		
A	C	T_2	airlineXY	70	$\varnothing$		
C	D	T_3	airlineXY	45	$\varnothing$		
C	E	T_4	airlineXY	60	$\varnothing$		
C	F	T_5	airlineXY	55	$\varnothing$		
A	C	T_2	airlineXY	80	flight-to	departureTime	airline
					D	T_3	airlineXY
					E	T_4	airlineXY
A	C	T_2	airlineXY	90	flight-to	departureTime	airline
					F	T_5	airlineXY

Fig. 2. *Example (a) source and (b) target instances.*

with connecting flights on the source instance, and that the data of that flight is split in three tuples on the target instance: a first one with no connecting flights but with the right ticket price, a second one with a wrong ticket price and with only two of the three connecting flights, and a third one also with a wrong ticket price and with the remaining connecting flight.

The designer could thus conclude that mapping $\{m_3\}$ is preferable not only because is more compact but also because is more accurate than $\{m_1, m_2\}$.
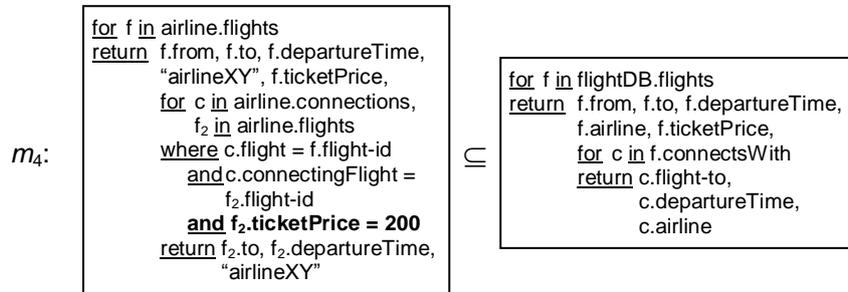
Example 2

Let us assume now that, according to a new business rule, only the most expensive connecting flights should be advertised by means of the flight-searching Web site. Let us also assume that the Web site has a constraint t_1 according to which, only flights with a ticket price no greater than 200 can be published.

t_1: for f in flightDB.flights then f.ticketPrice $\leq$ 200

Taking into account the business requirement and target schema's t_1 constraint, the designer could decide to adapt mapping $\{m_3\}$ and introduce and additional condition in the inner query block (shown in bold). The result is mapping $\{m_4\}$.

Let us also assume that another business rule was introduced, which the designer thinks has no effect on the mapping. The requirement is enforced by a new constraint s_1 on the source schema, which requires that the connecting flights must be cheaper than the initial flight.

s_1: for c in airline.connections, f_1 in airline.flights, f_2 in airline.flights
where c.flight = f_1.flight-id and c.connectingFlight = f_2.flight-id
then f_2.ticketPrice < f_1.ticketPrice

m_4:

```
for f in airline.flights
return f.from, f.to, f.departureTime,
       "airlineXY", f.ticketPrice,
       for c in airline.connections,
           f₂ in airline.flights
       where c.flight = f.flight-id
           and c.connectingFlight =
               f₂.flight-id
           and f₂.ticketPrice = 200
       return f₂.to, f₂.departureTime,
              "airlineXY"
```

$\subseteq$

```
for f in flightDB.flights
return f.from, f.to, f.departureTime,
       f.airline, f.ticketPrice,
       for c in f.connectsWith
       return c.flight-to,
              c.departureTime,
              c.airline
```

In order to be sure that no further modifications to the mapping should be made as a result of this new business requirement, the designer could check the non-trivial *satisfiability* of mapping $\{m_4\}$ at all its levels of nesting. By doing that, he would realize that m_4's inner level of nesting never maps any data, i.e., mapping $\{m_4\}$ is only mapping those flights with no connecting flights. The problem is that there is a contradiction between source constraint s_1 and the source query of m_4; in particular, since the source query of m_4 selects only connecting flights with ticket price equal to 200 in its inner query block, and s_1 requires these connecting flights to be cheaper than the initial flight selected by the outer query block, that implies the initial flight should have a ticket price greater than 200, which is not allowed by the target schema.

2. Preliminaries

In this section, we introduce the basic concepts of nested relational mapping scenarios and of flat relational databases. We also discuss the query satisfiability problem and its solution by means of the CQC method.

2.1 Nested Relational Mapping Scenarios

A *nested relation* $R(A1, ..., A_n)$ is a relation in which each attribute A_i can be defined either as a simple type (e.g., integer, real, string) or as another nested relation. For instance, the nested relation *flights* on Fig. 1b has five simple-type attributes: *from*, *to*, *departureTime*, *airline* and *ticketPrice*; and one attribute that is also a nested relation: *connectsWith*.

A *nested relational schema* consists of a root record whose elements are either simple types or nested relations. Nested relational model generalizes the relational one. A flat relational schema can be modeled as a nested relational schema in which the root record is a collection of flat relations, i.e., relations with all their attributes defined as simple types. Fig. 1a shows a flat relational schema and Fig. 1b a truly nested relational one.

We consider nested relational schemas with integrity constraints. An *integrity constraint* is a Boolean condition in the form (we adapt the XQuery-like notation of [20]):

<u>for</u> $variable_1$ <u>in</u> $relation_1$, ..., $variable_n$ <u>in</u> $relation_n$ <u>where</u> $condition_1$ <u>then</u> $condition_2$

The variables in the <u>for</u> clause are bound to tuples from the relation that follows the <u>in</u>. A $variable_i$ can be used in $relation_{i+1}$,...,$relation_n$, $condition_1$ and $condition_2$. The condition in the <u>where</u> and <u>then</u> clauses denotes a Boolean expression that may include arithmetic comparisons ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) and make use of conjunction, disjunction, and negation. As an example, see the constraints s_1 and t_1 on the Example 2 of Section 1.1.

An *instance* of a nested relational schema is *consistent* if it satisfies all the integrity constraints defined over the schema. Fig. 2 shows a consistent instance for each of the two schemas in Fig. 1.

A *nested query* is a query whose answer is a nested relation. That is, nested queries define derived nested relations. We use a notation similar to that of the integrity constraints (also adapted from [20]):

<u>for</u> $variable_1$ <u>in</u> $relation_1$, ..., $variable_n$ <u>in</u> $relation_n$
<u>where</u> $condition_1$ <u>return</u> $result_1$, ..., $result_n$

where each $result_i$ can be either a simple-type expression or another nested query. See, for example, the queries on assertion m_3 in the Example 1 of Section 1.1.

A *mapping scenario* is a triplet (S, T, M), where S is a source nested relational schema, T is a target nested relational schema, and M is a set of mapping assertions.

A mapping assertion m is a pair of nested queries related by a $\subseteq$ or $=$ operator; the query on the left-hand side being defined over the source schema, and the query on the right-hand side being defined over the target schema: $Q_{source} \subseteq/= Q_{target}$.

An instantiation of a mapping scenario (S, T, M) consists of an instance I_S of S and an instance I_T of T, such that I_S and I_T satisfy all the assertions in M.

A mapping assertion $Q_{source} \subseteq/= Q_{target}$ is satisfied by instances I_S, I_T iff the answer to Q_{source} on I_S is included/equal to the answer to Q_{target} on I_T.

We apply the definition of inclusion and equality of nested relations used in [18].

The *inclusion* of two nested structures R_1, R_2 of the same type T, i.e., $R_1 \subseteq R_2$, can be defined by induction on T as follows:

(1) If T is a simple type, $R_1 \subseteq R_2$ iff $R_1 = R_2$
(2) If T is a record type (i.e., a tuple), $R_1=[R_{1,1},...,R_{1,n}] \subseteq R_2=[R_{2,1},...,R_{2,n}]$ iff $R_{1,1} \subseteq R_{2,1} \wedge ... \wedge R_{1,n} \subseteq R_{2,n}$
(3) If T is a set type, $R_1=\{R_{1,1},...,R_{1,n}\} \subseteq R_2=\{R_{2,1},...,R_{2,n}\}$ iff $\forall i\ \exists j\ R_{1,i} \subseteq R_{2,j}$

Equality of nested structures, i.e., $R_1 = R_2$, can be defined similarly:

(1) If T is a simple type, $R_1 = R_2$
(2) If T is a record type, $[R_{1,1},...,R_{1,n}] = [R_{2,1},...,R_{2,n}]$ iff $R_{1,1} = R_{2,1} \wedge ... \wedge R_{1,n} = R_{2,n}$

(3) If T is a set type, $\{R_{1,1},\ldots,R_{1,n}\}=\{R_{2,1},\ldots,R_{2,n}\}$ iff $\forall i \; \exists j \; R_{1,i} = R_{2,j} \land \forall j \; \exists i \; R_{2,j} = R_{1,i}$

Note that, given the definitions above, $Q_1 = Q_2$ is not equivalent to $Q_1 \subseteq Q_2 \land Q_2 \subseteq Q_1$ [18].

2.2 Flat Relational Databases

A flat relational schema is a finite set of flat relations with integrity constraints. We use first-order logic notation and represent relations by means of predicates. Each predicate P has a *predicate definition* $P(A1,\ldots,A_n)$, where $A1,\ldots,A_n$ are the *attributes*. A predicate is said to be of *arity n* if it has n 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:

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

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. A *term* can be either a variable or a constant. Literals $r_i(\bar{Y}_i)$ and $\neg r_i(\bar{Z}_i)$ are positive and negated *ordinary literals*, respectively (note that in both cases r_i can be either a base predicate or a derived predicate). Literal $q(\bar{X})$ is the *head* of the deductive rule, and the other literals are the *body*. Symbols $\bar{X}$, $\bar{Y}_i$ and $\bar{Z}_i$ denote lists of terms. We assume deductive rules to be *safe*, which means that the variables in $\bar{Z}_i$, $\bar{X}$ and C_i are taken from $\bar{Y}_1, \ldots, \bar{Y}_n$, i.e., the variables in the negated literals, the head and the built-in literals must appear in the positive literals in the body. Literals about base predicates are often referred to as *base literals* and literals about derived predicates are referred to as *derived literals*.

We consider integrity constraints that are *disjunctive embedded dependencies* (DEDs) [10] extended with arithmetic comparisons and the possibility of being defined over views (i.e., they may have derived predicates in their definition). A *constraint* has one of the following two forms:

$$r_1(\bar{Y}_1) \land \ldots \land r_n(\bar{Y}_n) \to C_1 \lor \ldots \lor C_t$$
$$r_1(\bar{Y}_1) \land \ldots \land r_n(\bar{Y}_n) \land C_1 \land \ldots \land C_t \to \exists \bar{V}_1 \; r_{n+1}(\bar{U}_1) \lor \ldots \lor \exists \bar{V}_s \; r_{n+s}(\bar{U}_s)$$

Each $\bar{V}_i$ is a list of fresh variables (i.e., variables that have not been used anywhere else before), and the variables in $\bar{U}_i$ are taken from $\bar{V}_i$ and $\bar{Y}_1, \ldots, \bar{Y}_n$. Note that each predicate r_i (on both sides of the implication) can be either base or derived. We refer to the left-hand side of a constraint as the *premise*, and to the right-hand side as the *consequent*.

Formally, we write $S = (PD, DR, IC)$ to indicate that S is a database schema with predicate definitions PD, deductive rules DR, and integrity constraints IC. We omit the PD component when it is clear from the context.

An *instance D* of a schema *S* is a set of facts about the base predicates of *S*. A *fact* is a *ground literal*, i.e., a literal with all its terms constant. An instance *D* is *consistent* with schema *S* if it satisfies all the constraints in *IC*. The extension of the queries and views of *S* when evaluated on *D* is the *intensional database* (IDB) of *D*, denoted IDB(*D*). The answer to a query *Q* on an instance *D*, denoted $A_Q(D)$, is the set of all facts about predicate *q* in the IDB of *D*, i.e., $A_Q(D) = \{q(\bar{a}) \mid q(\bar{a}) \in IDB(D)\}$, where $\bar{a}$ denotes a list of constants.

2.3   Query Satisfiability and the CQC Method

A query *Q* is said to be *satisfiable* on a database schema *S* if there is some consistent instance *D* of *S* in which *Q* has a non-empty answer, i.e., $A_Q(D) \neq \varnothing$ [17].

The *CQC* (*Constructive Query Containment*) *method* [14], originally designed to check query containment, tries to build a consistent instance of a database schema in order to satisfy a given goal (a conjunction of literals). Clearly, using literal $q(\bar{X})$ as goal, where $\bar{X}$ is a list of distinct variables, results in the CQC method checking the satisfiability of query *Q*.

The CQC method starts by taking the empty instance and uses different *Variable Instantiation Patterns* (VIPs) based on the syntactic properties of the views/queries and constraints in the schema, attempting to generate only the relevant facts that are to be added to the instance under construction. If the method is able to build an instance that satisfies all the literals in the goal and does not violate any of the constraints, then that instance is a solution and proves the goal is satisfiable. The key point is that the VIPs guarantee that if the variables in the goal are instantiated using the constants they provide and the method does not find any solution, then no solution is possible.

The solution space that the CQC method explores is a tree, called the *CQC-tree*. Each branch of the CQC-tree is what is called a *CQC-derivation*. A CQC-derivation can be either *finite* or *infinite*. Finite CQC-derivations can be either *successful*, if they reach a solution, or *failed*, if they reach a violation that cannot be repaired. As proven in [14], the CQC method terminates when there is no solution, that is, when all CQC-derivations are finite and failed, or when there is some finite solution, i.e., when there is a finite, successful CQC-derivation.

A series of sufficient conditions for the termination of the CQC method has been studied in [24]. These conditions extend the ones proposed by [21].

A detailed performance evaluation of the CQC method has been done in [22, 24] for the case of flat relational mapping scenarios. It showed that, for those scenarios in which termination is guaranteed, the cost of the method is exponential with respect to the size of the mapping scenario. This is expected given the complexity of reasoning on such an expressive class of mapping scenarios.

3. Validation by Means of Checking Desirable Properties

We understand mapping validation as checking whether the mapping being designed meets the intended needs and requirements. To perform this validation, we propose to allow the designer to check whether the mapping has certain desirable properties. In this paper, we focus on two desirable properties of mappings (we will provide the formal definition of these properties in Section 6): satisfiability and inference.

As illustrated in the Example 2 of Section 1.1, mapping satisfiability allows detecting contradictions either between the mapping assertions or between the mapping assertions and the integrity constraints of the schemas. Mapping inference allows to detect redundancies in the mapping, i.e., redundant mapping assertions, and also to compare mapping candidates.

In order to actually check these desirable properties of mappings, we propose to translate the mapping scenario from the nested relational setting into the flat relational one. That implies flattening not only the nested relational schemas, but also the nested queries on the mappings. Then, we propose to take advantage of previous work on validating mappings in the relational setting [22] and reformulate the desirable property checking on this new flat relational mapping scenario in terms of the query satisfiability problem. To do so, we firstly combine the relational versions of the two mapped schemas into a single relational schema. Secondly, we rewrite the mapping assertions as integrity constraints over this single relational schema. Finally, for each mapping desirable property that we want to check on the original nested mapping scenario, we define a query on the single relational schema in such a way that this query will be satisfiable on this schema if and only if the mapping desirable property holds on the original mapping scenario.

In the next sections we discuss in detail how to translate the nested relational mapping scenario into the flat relational one (Section 4 and Section 5), and how to reformulate each desirable property check in terms of a query satisfiability problem over this flat relational translation (Section 6).

4. Flattening Nested Schemas and Queries

In this section, we detail how to encode the nested schemas and the nested queries of a given nested relational mapping scenario into a single flat database schema. Note that when we say nested queries we mean those in the mapping assertions. We will later rely on this encoding of the nested queries to rewrite the mapping assertions as integrity constraints.
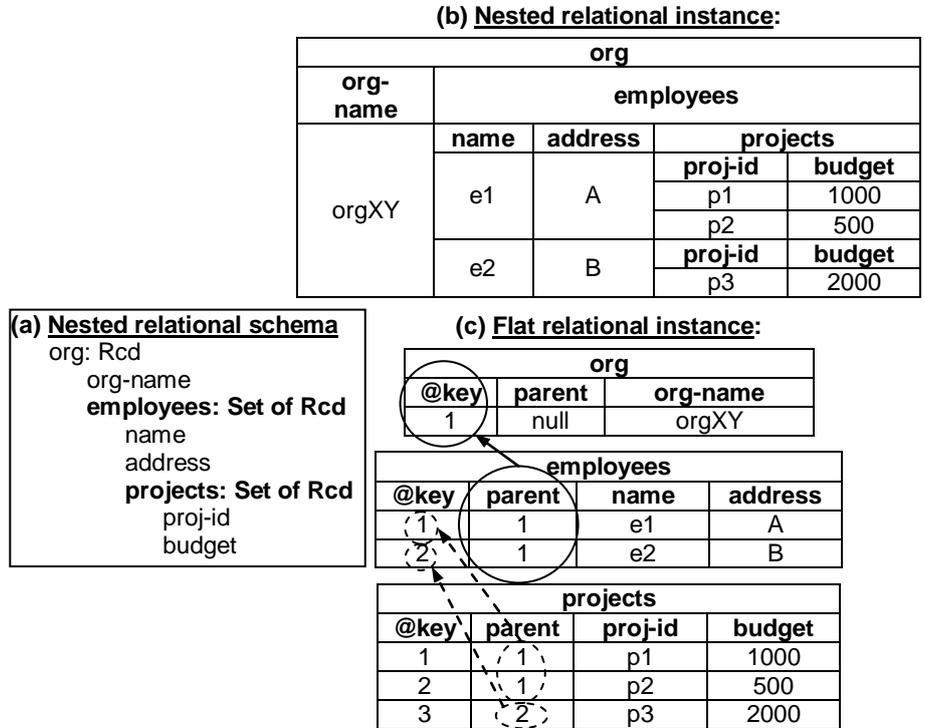
(b) Nested relational instance:

org				
org-name	**employees**			
	name	**address**	**projects**	
orgXY	e1	A	**proj-id**	**budget**
			p1	1000
			p2	500
	e2	B	**proj-id**	**budget**
			p3	2000

(a) Nested relational schema

org: Rcd
    org-name
    employees: Set of Rcd
        name
        address
        projects: Set of Rcd
            proj-id
            budget

(c) Flat relational instance:

org		
@key	**parent**	**org-name**
1	null	orgXY

employees			
@key	**parent**	**name**	**address**
1	1	e1	A
2	1	e2	B

projects			
@key	**parent**	**proj-id**	**budget**
1	1	p1	1000
2	1	p2	500
3	2	p3	2000

Fig. 3. *A (a) nested relational schema, an (b) instance of this schema, and (c) the translation of the instance into flat relations*

4.1 Nested Schemas

Our translation of nested relational schemas into flat relational ones is based on the *hierarchical representation* used by Yu and Jagadish in [25]. They address the problem of discovering functional dependencies on nested relational schemas. They translate the schemas into a flat representation, so algorithms for finding functional dependencies on relational schemas can be applied.

The hierarchical representation assigns a flat relation to each nested table. To illustrate that, consider the nested relational schema in Fig. 3a, which models data about an organization, its employees, and the projects each employee works on. The hierarchical representation, as defined in [25], of this nested relational schema would be the following set of flat relations:

{org(*@key*, *parent*, *org-name*), employees(*@key*, *parent*, *name*, *address*), projects(*@key*, *parent*, *proj-id*, *budget*)}

Note that each flat relation keeps the simple-type attributes of the nested relation, and has two additional attributes: the *@key* attribute, which models an implicit tuple id; and the *parent* attribute, which references the *@key* attribute of the parent table and models the parent-child relationship of the nested relations. Fig. 3b shows an instance of the previous nested relational schema, and Fig. 3c shows the corresponding instance of the flat relational schema into which the previous nested schema is translated.

For simplicity, we skip the flat relation of the root record when it has only set-type attributes and no simple-type ones. We also skip the *parent* attribute of those relations that do not have a parent relation, and we skip the *@key* attribute of those relations that do not have child relations. For example, we would translate the source and target schema of the mapping scenario in Fig. 1 into flat relations as follows:

$$source = \{flights_S(\textit{flight-id, from, to, departureTime, ticketPrice}),$$
$$connections(\textit{flight, connectingFlight})\}$$
$$target = \{flights_T(\textit{@key, from, to, departureTime, airline, ticketPrice}),$$
$$connectsWith(\textit{parent, flight-to, departureTime, airline})\}$$

The semantics of the *@key* and *parent* attributes are made explicit by means of adding the corresponding key and referential constraints to the flat relational schema that results from the flattening process. As an example, the flat version of the target schema in Fig. 1 (see above) would include the following key and referential constraint:

key: $flights_T(@key, f, t, dt, a, tp) \land flights_T(@key', f, t, dt, a, tp) \rightarrow @key = @key'$
ref: $connectsWith(parent, ft, dt, a) \rightarrow flights_T(parent, f', t', dt', a', tp')$

The integrity constraints that already exist on the original nested schemas can be straightforwardly translated into constraints over the flat relational version of the schema. For example, let us consider again the source and target constraint s_1 and t_1 of the Example 2 of Section 1.1; the constraints would be translated into the following:

s_1': $connections(f, cf) \land flights_S(f, frm, to, dt, tp) \land flights_S(cf, frm', to', dt', tp') \rightarrow tp' < tp$
t_1': $flights_T(@k, frm, to, dt, a, tp) \rightarrow tp \leq 200$

4.2   Nested Queries

Regarding the flattening of nested queries, we follow a variation of the approach used in [18]. In this approach, each nested query is translated into a collection of flat queries, one for each nested query block. For example, let us consider the source schema from Fig. 1, and let us suppose that we have the following nested query Q defined over this source schema, which selects the flights with a ticket price of at least 50 and, for each of these flights, selects the connecting flights that are cheaper than the original flight:

Q: <u>for</u> f <u>in</u> airline.flights <u>where</u> f.tp $\geq$ 50
    <u>return</u> f.from, f.to, f.departureTime, "airlineXY", f.ticketPrice,
        <u>for</u> c <u>in</u> airline.connections, f_2 <u>in</u> airline.flights
        <u>where</u> c.flight = f.flight-id <u>and</u> c.connectingFlight = f_2.flight-id
            <u>and</u> f_2.ticketPrice < f.ticketPrice
        <u>return</u> f_2.to, f_2.departureTime, "airlineXY"

The nested query Q has two query blocks: the outer block Q_{outer}

Q_{outer}: <u>for</u> f <u>in</u> airline.flights <u>where</u> f.tp $\geq$ 50
        <u>return</u> f.from, f.to, f.departureTime, "airlineXY", f.ticketPrice

and the inner block Q_{inner}.

Q_{inner}: <u>for</u> c <u>in</u> airline.connections, f_2 <u>in</u> airline.flights
        <u>where</u> c.flight = f.flight-id <u>and</u> c.connectingFlight = f_2.flight-id
            <u>and</u> f_2.ticketPrice < f.ticketPrice
        <u>return</u> f_2.to, f_2.departureTime, "airlineXY"

Since both query blocks are flat queries when considered independently, and assuming we have already flattened the corresponding schema (the source schema in this case), each of these blocks can be straightforwardly translated into a query over the flat version of the schema. The only technical detail, and the main difference with respect to [18], is the treatment of the *inherited variables*—called *indexes* in [18]—, which are the variables defined in the <u>for</u> clause of the outer block that are also used in the inner block. In [18], the translation of both the outer and the inner block would be extended to select the key attributes of the relations bound to the inherited variables; in the case of the inner block, since it uses the inherited variables but does not define them, that would require copying in the inner block's translation those literals from the outer block's translation that correspond to the definition of the inherited variables. In our example, the inherited variable "f" is defined in the translation of Q_{outer} by the literal "flights$_S$(fid, frm, to, dt, tp)", where "fid" corresponds to the key attribute and is selected by this translation. The translation of Q_{inner} should thus contain a copy of this literal (shown below in bold) and also select "fid":

Q_{outer}(fid, frm, to, dt, "airlineXY") $\leftarrow$ flights$_S$(fid, frm, to, dt, tp) $\wedge$ tp $\geq$ 50

Q_{inner}(fid, to′, dp′, "airlineXY") $\leftarrow$ connections(fid, cf) $\wedge$ flight$_S$(cf, frm′, to′, dp′, tp′)
$\wedge$ **flights$_S$(fid, frm, to, dt, tp)** $\wedge$ tp′ < tp

Notice that without the literal in bold, Q_{inner} would not have access to variable "tp" (i.e., f.ticketPrice) and could not make the required comparison.

The flat relational equivalent to answering the original nested query Q would be making a left outer join of the translations of Q_{outer} and Q_{inner} with "fid" as the join variable.

In order to simplify the translation, not only that of the nested queries themselves but specially the translation of the mapping assertions (see next section), we use access patterns [9]; in particular, we consider derived relations with "input-only" attributes in addition to the traditional "input-output" ones. We use $R{<}x_1, ..., x_n{>}(y_1, ..., y_n)$ to denote that $x_1,...,x_n$ are input-only

terms bound to derived relation R, and $y_1,...,y_n$ are input-output ones. As an example, we would translate Q_{inner} and Q_{outer} as follows:

$Q_{outer}($**fid**, **tp**, frm, to, dt, "airlineXY"$) \leftarrow$ flights$_S($**fid**, frm, to, dt, **tp**$) \wedge$ **tp** ≥ 50

$Q_{inner}<$**fid**, **tp**$>($to′, dp′, "airlineXY"$) \leftarrow$ connections(fid, cf) $\wedge$ flight$_S($cf, frm′, to′, dp′, tp′$)$
$\qquad\qquad\qquad\qquad\qquad\qquad \wedge$ tp′ < **tp**

Notice that we enforce the translation of Q_{outer} to select the variables "fid" and "tp", which are then to be inherited by Q_{inner} through its input-only attributes. Note also that there is no need now to repeat the ordinary literal of Q_{outer} in Q_{inner}.

In order for a deductive rule to be *safe*, the variables that appear as input-only terms of some literal in the body of the rule must either appear as input-output terms of some other positive ordinary literal in the same body, or appear in the head of the rule as input-only terms. Similarly, the variables that appear in a negated or built-in literal in the body of a rule must either appear as input-output terms of some other positive ordinary literal in the same body, or appear in the head of the rule as input-only terms. See for instance, variable "tp" in Q_{inner} above, which appears in the body of the rule in a built-in literal, and in the head of the rule as an input-only term.

5. Rewriting Mapping Assertions As Integrity Constraints

A nested relational mapping scenario consists of two nested relational schemas and a mapping with nested queries that relates them. We have already discussed how to translate each nested schema into the flat relational formalism. In order to complete the translation of the nested relational mapping scenario into the flat relational setting, we must see now how to translate the mapping assertions. We assume the queries in both sides of the mapping are part of each schema's definition and have already been translated along with them.

To translate a mapping assertion $Q_{source} \subseteq/= Q_{target}$, we will make use of the definition of inclusion/equality of nested structures from [18] (see Section 2.1), and we will rely on the flat queries that result from flattening Q_{source} and Q_{target}. As an example, consider the mapping assertion m_3 from Example 1 (see Section 1.1). Let us assume the source and target query—let us call them Q^S and Q^T—are translated into the flat queries Q^S_{outer}, Q^S_{inner} and Q^T_{outer}, Q^T_{inner}, respectively, as follows:

$Q^S_{outer}($fid, frm, to, dt, "airlineXY", tp$) \leftarrow$ flights$_S($fid, frm, to, dt, tp$)$
$Q^S_{inner}<$fid$>($to, dt, "airlineXY"$) \leftarrow$ connections(fid, cf) $\wedge$ flights$_S($cf, frm, to, dt, tp$)$

$Q^T_{outer}($@k, frm, to, dt, a, tp$) \leftarrow$ flights$_T($@k, frm, to, dt, a, tp$)$
$Q^T_{inner}<$@k$>($to, dt, a$) \leftarrow$ connectsWith(@k, to, dt, a$)$

According to the semantics of query inclusion, two schema instances I_S and I_T satisfy m_3 if and only if the answer to Q^S on I_S, i.e., $AQ^S(I_S)$, is included in the answer to Q^T on I_T, i.e., $AQ^T(I_T)$. Recall that, as defined in [18], a nested

structure such as $AQ^S(I_S)$ is included in another nested structure such as $AQ^T(I_T)$ if and only if each tuple a in $AQ^S(I_S)$ "matches" some tuple b of $AQ^T(I_T)$, where "matches" means that each simple-type attribute on b (e.g., the *from* attribute) must have the same value than the corresponding attribute of a, and that the value of each set-type attribute on b (e.g., the *connectsWith* attribute) must be a set that recursively includes the set bound to the corresponding set-type attribute of a. Notice that this is a recursive definition, where simple-type attributes are the base case and set-type ones are the recursive case.

We can express the above definition as a Boolean condition over the flat translations of the mapped schemas. The condition will be true if the given schema instances satisfy the mapping assertion, and false otherwise. The condition begins with the requirement that for all tuple a returned by the outer query block of Q^S there must be a matching b on the result of the outer query block of Q^T with the same value on the simple-type attributes:

$\forall$fid,frm,to,dt,a,tp $(Q^S_{outer}(fid, frm , to, dt, a, tp) \rightarrow \exists @k (Q^T_{outer}(@k, frm, to, dt, a, tp) ...$

The condition must also include the requirement that the set-type attributes of a must be included in the corresponding set-type attributes of b, i.e., the recursive case:

$... \wedge \forall to',dt',a' (Q^S_{inner}<fid>(to', dt', a') \rightarrow Q^T_{inner}<@k>(to', dt', a'))$))

By making the union of the flat mapped schemas and introducing this Boolean condition as an integrity constraints over this union, we will get that each consistent instance of the resulting flat database schema will correspond to a consistent instantiation of the mapping scenario (i.e., an instantiation in which the mapping assertions are true), and vice versa. The only problem is that the above condition does not fit the syntactic requirements of the class of constraints we consider, i.e., disjunctive embedded dependencies (DEDs), which are expressions of the form $\forall \overline{X} (\phi(\overline{X}) \rightarrow \exists \overline{Y}_1 \psi_1(\overline{X}, \overline{Y}_1) \vee ... \vee \exists \overline{Y}_n \psi_n(\overline{X}, \overline{Y}_n))$ in which $\forall$ quantifiers are not allowed inside $\psi_1,...,\psi_n$. Fortunately, we can take advantage of the fact that we are able to deal with negation and get rid of that inner $\forall$ quantifier. We can introduce a double negation $\neg\neg$ in front of the $\forall$ quantifier, and move one of the negations inwards:

$... \wedge \neg\exists to',dt',a' (Q^S_{inner}<fid>(to', dt', a') \wedge \neg Q^T_{inner}<@k>(to', dt', a'))$))

There are only two details remaining now. The first is that we only allow direct negation of single literals and not of conjunction of literals. However, we do allow negation of derived literals, so we can just fold the conjunction into a new derived relation:

$\forall$fid,frm,to,dt,a,tp $(Q^S_{outer}(fid, frm , to, dt, a, tp) \rightarrow \exists @k (Q^T_{outer}(@k, frm, to, dt, a, tp)$
$\wedge \neg\mathbf{Q^S_{inner}\text{-not-included-in-} Q^T_{inner}<fid, @k>()}$))

where

$Q^S_{inner}\text{-not-included-in-} Q^T_{inner}<fid, @k>() \leftarrow Q^S_{inner}<fid>(to', dt', a')$
$\wedge \neg Q^T_{inner}<@k>(to', dt', a')$

The second detail is that we do not allow the explicit use of negation in the integrity constraints, i.e., the literals in ϕ and in $\psi_1,...,\psi_n$ cannot be negated. We do however allow constraints in which the consequent is a contradiction, e.g., $1 = 0$. With that and the introduction of double negation in front of the remaining $\forall$ quantifier, we can rewrite the expression as follows. First, we introduce the double negation and move one of the negation inwards just as we did before:

$\neg\exists$fid,frm,to,dt,a,tp $(Q^S_{outer}($fid, frm, to, dt, a, tp$) \wedge \neg\exists@k (Q^T_{outer}(@k,$ frm, to, dt, a, tp$)$
$\wedge \neg Q^S_{inner}$**-not-included-in- Q^T_{inner}<fid, @k>( )**))

To get rid of the inner $\neg\exists$ quantifier, we fold the conjunction into a new derived relation:

$\neg\exists$fid,frm,to,dt,a,tp $(Q^S_{outer}($fid, frm, to, dt, a, tp$) \wedge$
$\neg$**aux- Q^S_{outer}-not-included-in-Q^T_{outer}<fid, frm, to, dt, a, tp>( )**

where

aux- Q^S_{outer}-not-included-in-Q^T_{outer}<fid, frm, to, dt, a, tp>( ) $\leftarrow$
$Q^T_{outer}(@k,$ frm, to, dt, a, tp$) \wedge \neg Q^S_{inner}$-not-included-in- Q^T_{inner}<fid, @k>( )

We still make an additional folding to get rid of the remaining $\neg\exists$ quantifier, and we get:

$\neg Q^S_{outer}$**-not-included-in-Q^T_{outer}( )**

where

Q^S_{outer}-not-included-in-Q^T_{outer}( ) $\leftarrow Q^S_{outer}($fid, frm, to, dt, a, tp$) \wedge$
$\neg$aux- Q^S_{outer}-not-included-in-Q^T_{outer}<fid, frm, to, dt, a, tp>( )

Finally, we can get rid of the $\neg$ by stating that the atom implies a contradiction:

Q^S_{outer}-not-included-in-Q^T_{outer}( ) $\rightarrow$ **1 = 0**

This constraint, together with the deductive rules that define the new derived relations that we just introduced, enforces the mapping assertion m_3.

In a more generic way, the rewriting of a query inclusion mapping assertion can be formalized as follows.
Let Q^A and Q^B be two generic (sub)queries with compatible answer:

Q^A:<u>for</u> var_1 <u>in</u> rel_1, ..., var_{na} <u>in</u> rel_{na} <u>where</u> $cond$ <u>return</u> A_1, ..., A_m, B_1, ..., B_k
Q^B:<u>for</u> var_1' <u>in</u> rel_1',..., var_{nb}' <u>in</u> rel_{nb}' <u>where</u> $cond'$ <u>return</u> A_1', ..., A_m', B_1',..., B_k'

where each A_i and A_i' are simple-type expressions, and each B_i and B_i' are subqueries. Let us assume the outer block of Q^A is translated into the derived relation $Q^A_{outer}\langle x_1, ..., x_{ka}\rangle(v_1, ..., v_{na}, r_1, ..., r_m)$, where $x_1,...,x_{ka}$ denote the variables inherited from the ancestor query blocks, $v_1,...,v_n$ denote the additional variables to be inherited by the inner query blocks of Q^A_{outer}, and $r_1,...,r_m$ denote the simple-type values returned by the block. Similarly, let us also assume the outer block of Q^B is translated into $Q^B_{outer}\langle x_1', ..., x_{kb}'\rangle(v_1', ..., v_{nb}', r_1', ..., r_m')$.

We use T-inclusion(Q^A, Q^B, $\{i_1, \ldots, i_h\}$) to denote the translation of $Q^A \subseteq Q^B$, where $\{i_1, \ldots, i_h\}$ is the union of the variables inherited by Q^A and Q^B from their respective parent blocks (if any):

$$\text{T-inclusion}(Q^A, Q^B, \{i_1,\ldots,i_h\}) = \neg Q^A\text{-not-included-in-}Q^B\langle i_1,\ldots,i_h\rangle$$

where

$$Q^A\text{-not-included-in-}Q^B\langle i_1,\ldots,i_h\rangle \leftarrow Q^A_{outer}\langle x_1,\ldots,x_{ka}\rangle(v_1,\ldots,v_{na}, r_1,\ldots,r_m) \wedge$$
$$\neg aux\text{-}Q^A\text{-not-included-in-}Q^B\langle i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m\rangle$$

$$aux\text{-}Q^A\text{-not-included-in-}Q^B\langle i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m\rangle \leftarrow$$
$$Q^B_{outer}\langle x_1',\ldots,x_{kb}'\rangle(v_1',\ldots,v_{nb}', r_1,\ldots,r_m) \wedge$$
$$\text{T-inclusion}(B_1, B_1', \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\}) \wedge \ldots \wedge$$
$$\text{T-inclusion}(B_k, B_k', \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\})$$

If Q^A and Q^B are not subqueries but full queries, then the following constraint is to be introduced:

$$\neg\text{T-inclusion}(Q^A, Q^B, \{i_1,\ldots,i_h\}) \rightarrow 1 = 0$$

Similarly, the rewriting of a generic query equality assertion $Q^A = Q^B$ as a set of integrity constraints can be formalized as follows:

$$\neg\text{T-equality}(Q^A, Q^B, \{i_1, \ldots, i_h\}) \rightarrow 1 = 0$$
$$\neg\text{T-equality}(Q^B, Q^A, \{i_1, \ldots, i_h\}) \rightarrow 1 = 0$$

where

$$\text{T-equality}(Q^A, Q^B, \{i_1,\ldots,i_h\}) = \neg Q^A\text{-not-eq-to-}Q^B\langle i_1,\ldots,i_h\rangle$$

and

$$Q^A\text{-not-eq-to-}Q^B\langle i_1,\ldots,i_h\rangle \leftarrow Q^A_{outer}\langle x_1,\ldots,x_{ka}\rangle(v_1,\ldots,v_{na}, r_1,\ldots,r_m) \wedge$$
$$\neg aux\text{-}Q^A\text{-not-eq-to-}Q^B\langle i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m\rangle$$

$$aux\text{-}Q^A\text{-not-eq-to-}Q^B\langle i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m\rangle \leftarrow$$
$$Q^B_{outer}\langle x_1',\ldots,x_{kb}'\rangle(v_1',\ldots,v_{nb}', r_1,\ldots,r_m) \wedge$$
$$\text{T-equality}(B_1, B_1', \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\}) \wedge$$
$$\text{T-equality}(B_1', B_1, \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\}) \wedge \ldots \wedge$$
$$\text{T-equality}(B_k, B_k', \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\}) \wedge$$
$$\text{T-equality}(B_k', B_k, \{i_1,\ldots,i_h, v_1,\ldots,v_{na}, r_1,\ldots,r_m, v_1',\ldots,v_{nb}'\})$$

The two constraints above, together with the deductive rules of the corresponding derived relations, enforce the definition of query equality as defined in [18] (see Section 2.1).

Intuitively, T-equality(Q^A, Q^B, $\{i_1,\ldots,i_h\}$) denotes the condition that, for each instantiation of the mapping scenario, each tuple in the answer to Q^A must have an equal tuple in the answer to Q^B. Notice that in order to fully express the definition of query equality, we need to enforce both T-equality(Q^A, Q^B, $\{i_1,\ldots,i_h\}$) and T-equality(Q^B, Q^A, $\{i_1,\ldots,i_h\}$).

6. Desirable Properties in Terms of Query Satisfiability

In this section, we show how two desirable properties of mappings—satisfiability and inference—can be reformulated as a query satisfiability check over the flat relational translation of mapping scenarios we have presented in Section 4 and Section 5.

6.1 Mapping Satisfiability

We say a mapping is *satisfiable* if there is a pair of schema instances that make all the mapping assertions true in a non-trivial way. An example of trivial satisfaction would be a pair of empty schema instances, which is not the case we are interested in here. We distinguish two kinds of satisfiability: *strong* and *weak*.

Intuitively, a mapping is strongly satisfiable if all its mapping assertions can be non-trivially satisfied at the same time at all their levels of nesting, e.g., the inner query block of mapping assertion m_4's source query from the Example 2 of Section 1.1 never maps any data (i.e., always provides an empty answer); therefore, although the outer query block does map some data, mapping $\{m_4\}$ is not strongly satisfiable.

Definition 1 (*Strong Satisfiability*). A mapping M is *strongly satisfiable* iff there exist I_S, I_T instances of the source and target schema, respectively, such that I_S and I_T satisfy the assertions in M, and for each assertion Q_{source} *op* Q_{target} in M, the answer to Q_{source} in I_S is a strong answer. We say R is a *strong answer* iff

(1) R is a simple type value,
(2) R is a record $[R_1, ..., R_n]$ and $R_1, ..., R_n$ are all strong answers, or
(3) R is a non-empty set $\{R_1, ..., R_n\}$ and $R_1, ..., R_n$ are all strong answers.

Intuitively, we say a mapping is weakly satisfiable if at least one mapping assertion can be satisfied at least at its outermost level of nesting. As an example, mapping $\{m_4\}$ is indeed weakly satisfiable.

Definition 2 (*Weak satisfiability*). A mapping M is *weakly satisfiable* iff there exist I_S, I_T instances of the source and target schema, respectively, and some mapping assertion m: $Q_{source} \subseteq/= Q_{target}$ in M, such that I_S, I_T make m true and the answer to Q_{source} on I_S is not empty, i.e., $A_{Q_{source}}(I_S) \neq \varnothing$.

Let us assume M is a mapping with assertions $\{Q^S_1 \, op \, Q^T_1, ..., Q^S_n \, op \, Q^T_n\}$. Let $S = (PD_S, DR_S, IC_S)$ be the flat translation of the source schema, and $T = (PD_T, DR_T, IC_T)$ be the flat translation of the target schema. Let us also assume that IC_M and DR_M are the constraints and deductive rules that result from the rewriting of the assertions in M. The flat database schema that encodes the mapping scenario is:

$$DB = (PD_S \cup PD_T, \ DR_S \cup DR_T \cup DR_M, \ IC_S \cup IC_T \cup IC_M)$$

Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí

The reformulation of strong satisfiability of M as a query satisfiability check over DB is the following:

$$Q_{strongSat} \leftarrow \text{StrongSat}(Q^S_1, \varnothing) \wedge ... \wedge \text{StrongSat}(Q^S_n, \varnothing)$$

where StrongSat is a function generically defined as follows. Let Q be a generic (sub)query:

Q: <u>for</u> var_1 <u>in</u> rel_1, ..., var_s <u>in</u> rel_s <u>where</u> $cond$ <u>return</u> A_1, ..., A_m, B_1, ..., B_k

where $A_1,...,A_m$ are simple-type expressions and $B_1,...,B_k$ are inner query blocks. Let predicate Q_{outer} be the translation of the outer query block of Q. Then,

$$\begin{aligned} \text{StrongSat}(Q, inheritedVars) = &\ Q_{outer}\langle x_1,...,x_r\rangle(v_1,...,v_s, r_1,...,r_m) \wedge \\ &\ \text{StrongSat}(B_1, inheritedVars \cup \{v_1,...,v_s, r_1,...,r_m\}) \wedge ... \wedge \\ &\ \text{StrongSat}(B_k, inheritedVars \cup \{v_1,...,v_s, r_1,...,r_m\}) \end{aligned}$$

where $\{x_1,...,x_r\} \subseteq inheritedVars$.

Boolean query $Q_{strongSat}$ is satisfiable over DB if and only if mapping M is strongly satisfiable.

Intuitively, if we can find an instance of DB that satisfies $Q_{strongSat}$, we can obtain from that database instance a source and a target instance for the mapping scenario. These two instances will be consistent with their respective schemas and with the mapping assertions because DB includes the corresponding integrity constraints. The strong satisfiability property will hold, because $Q_{strongSat}$ is encoding its definition.

As an example, let us assume the outer query block of mapping assertion m_4's source query in Example 2 is translated into derived relation Q^S_{outer}(flight-id, from, to, departureTime, airline, ticketPrice), and the inner query block into derived relation Q^S_{inner}<flight-id>(to, departureTime, airline). Then, strong satisfiability of $\{m_4\}$ would be reformulated as follows:

$$Q_{strongSat} \leftarrow Q^S_{outer}(fid, frm, to, dt, a, tp) \wedge Q^S_{inner}<fid>(to', dt', a')$$

The reformulation of weak satisfiability of M as a query satisfiability check over DB is the following:

$$Q_{weakSat} \leftarrow Q^S_{1,outer}(\overline{X}_1)$$
$$\cdots$$
$$Q_{weakSat} \leftarrow Q^S_{n,outer}(\overline{X}_n)$$

where $Q^S_{1,outer},...,Q^S_{n,outer}$ are the translations of the outermost query blocks of the source mapping's queries.

Boolean query $Q_{weakSat}$ is satisfiable over DB if and only if mapping M is weakly satisfiable.

The intuition is that $Q_{weakSat}$ can only be if some of the outermost blocks of the source mapping's queries is not empty. Therefore, if $Q_{weakSat}$ is true, we can extract from the corresponding instance of DB an instantiation of the mapping scenario that exemplifies the property.

As an example, weak satisfiability of mapping $\{m_4\}$ would be reformulated as follows:

$$Q_{weakSat} \leftarrow Q^S_{outer}(fid, frm, to, dt, a, tp)$$

Notice that there is only one deductive rule for $Q_{weakSat}$ because the mapping has only one assertion.

6.2   Mapping Inference

The *mapping inference* property [19] checks whether a given mapping assertion is inferred from a set of others assertions. It can be used, for instance, to detect redundant assertions in a mapping, or to test equivalence of candidate mappings. As an example, recall mapping $\{m1, m2\}$ from Example 1. Assertions $m1$, $m2$ are each one inferred from mapping $\{m3\}$, but assertion $m3$ is not inferred from $\{m1, m2\}$.

Definition 3 (*Mapping Inference*).   Let M be a mapping from schema S to schema T. Let F be an assertion from S to T. We say F is *inferred* from M iff $\forall I_S$, I_T instances of schema S and T, respectively, such that I_S and I_T satisfy the assertions in M, then I_S and I_T also satisfy assertion F.

As with the previous property, the flat database schema that encodes the mapping scenario is:

$$DB = (PD_S \cup PD_T,\ DR_S \cup DR_T \cup DR_M,\ IC_S \cup IC_T \cup IC_M)$$

In order to reformulate mapping inference in terms of query satisfiability, we must get rid of the universal quantifier that appears in the property's definition. The reason is that by means of query satisfiability we can check whether there exists an instance that satisfies the property encoded by the query, but not whether all instances satisfy that property. We can address this situation by checking the negation of the property instead of checking the property directly; that is, we will check whether there is a pair of schema instances that satisfy the mapping but not the given assertion.

If the assertion to be tested is a query inclusion, i.e., $Q_{source} \subseteq Q_{target}$, then the query to be tested satisfiable on DB is defined by a single deductive rule:

$$Q_{notInferred} \leftarrow \neg \text{T-inclusion}(Q_{source},\ Q_{target},\ \varnothing)$$

If the assertion to be tested is a query equality, i.e., $Q_{source} = Q_{target}$, then the query to be tested satisfiable on DB is defined by two deductive rules:

$$Q_{notInferred} \leftarrow \neg \text{T-equality}(Q_{source},\ Q_{target},\ \varnothing)$$
$$Q_{notInferred} \leftarrow \neg \text{T-equality}(Q_{target},\ Q_{source},\ \varnothing)$$

Boolean query $Q_{notInferred}$ is satisfiable over DB if and only if the given assertion F is not inferred from mapping M.

Fig. 2 shows an instantiation of the example mapping scenario in Example 1 which satisfies mapping $\{m_1, m_2\}$ but not assertion m_3, i.e., the instantiation is an example that illustrates m_3 is not inferred from $\{m_1, m_2\}$.

7. Related Work

In this section, we compare our approach with the previous works on nested relational mapping validation and on translating nested queries into the flat relational setting.

7.1 Mapping Validation on Nested Scenarios

Previous work on mapping validation on the nested relational setting has mainly focused on instance-based approaches: the Routes approach [7], the Spicy system [6], and the Muse system [1]. These approaches rely on specific source and target instances in order to debug, refine and guide the user through the process of designing a schema mapping, which do not necessarily reflect all potential pitfalls.

The Routes approach requires both a source and a target instance in order to compute the routes. The Spicy system requires a source instance to be used to execute the mappings, and a target instance to compare the mapping results with. The Muse system can generate its own synthetic examples to illustrate the different design alternatives, but even in this case the detection of semantic errors is left to the user, who may miss to detect them.

Routes, Spicy and Muse allow both relational and nested relational schemas with key and foreign key-like constraints—typically formalized by means of tuple-generating dependencies (TGDs) and equality-generating dependencies (EGDs)—, and mappings expressed as source-to-target TGDs [20]. Muse is also able to deal with the nested mapping formalism [15], which allows the nesting of TGDs. Comparing with our setting, the class of disjunctive embedded dependencies (DEDs) with derived relation symbols and arithmetic comparisons that we consider includes that of TGDs and EGDs. That is easy to see since it is well-known that traditional DEDs already subsume both TGDs and EGDs [11]. Similarly, our mapping assertions go beyond TGDs in two ways: (1) they may contain negations and arithmetic comparisons, while TGDs are conjunctive; and (2) they may be bidirectional, i.e., assertions in the form of $Q_A = Q_B$ (which state the equivalence of two queries), while TGDs are known to be equivalent to global-and-local-as-view (GLAV) assertions in the form of $Q_A \subseteq Q_B$ [13].

Outside the nested relational setting, other works have proposed and studied desirable properties for different classes of XML mappings.

In [3], the authors study the *consistency* checking problem for XML mappings that consist of source-to-target implications of tree patterns between DTDs. Such a mapping is consistent if at least one tree that conforms to the source DTD is mapped into a tree that conforms to the target DTD. This work extends the previous work of [4], where mapping consistency is addressed for a simpler class of XML mappings.

The mapping consistency property of [3] is very similar to our notion of mapping satisfiability; the main difference is that we introduce the requirement

that mapping assertions have to be satisfied in a non-trivial way, that is, a source instance should not be mapped into the empty target instance. We introduce this requirement because the class of mapping scenarios we consider—with integrity constraints, negations and arithmetic comparisons—makes likely the existence of contradictions either in the mapping assertions, or between the mapping assertions and the schema constraints, or between the mapping assertions themselves; which may result in mapping assertions that can only be satisfied in a trivial way.

7.2   Translation of Assertions with Nested Queries into Flat Relational

Since our mapping assertions are in the form of query inclusions and query equalities, the problem of translating these assertions into the flat relational setting matches the problem of reducing the containment and equivalence check of nested queries to some other property check over flat relational queries. The works in this latter area that are closer to ours are [18, 12, 8].

In [18], Levy and Suciu address the containment and equivalence of *COQL* queries (*Conjunctive OQL queries*), which are queries that return a nested relation. They encode each COQL query as a set of flat conjunctive queries using indexes. An *indexed query* Q is a query hose head is in the form of $Q(\overline{I}_1;$ ...; $\overline{I}_d;$ $V_1, ..., V_n)$, where $\overline{I}_1, ..., \overline{I}_d$ denote sets of *index variables*, and variables $V_1, ..., V_n$ denote the resulting tuple. Relying on the concept of indexed query, Levy and Suciu define in [18] the property of query simulation, and reduce containment of COQL queries to an exponential number of query simulation conditions between the indexed queries that encode them. Levy and Suciu also define the property of strong simulation [18], and reduce equivalence of COQL queries which cannot construct empty sets to a pair of strong simulation conditions (equivalence of general COQL queries is left open).

In [12], Dong et al. adapt the technique proposed by Levy and Suciu [18] to the problem of checking the containment of conjunctive XQueries. They also encode the nested queries into a set of indexed queries, and also reduce the containment checking to a set of query simulation tests between the indexed queries. Dong et al. also propose some extensions to the query language, such as the use of negation and the use of arithmetic comparisons. They however do not consider both extensions together as we do, and they do not consider the presence of integrity constraints in the schemas.

In [8], DeHaan addresses the problem of checking the equivalence of nested queries under *mixed semantics* (i.e., each collection can be either set, bag or normalized bag). DeHaan proposes a new encoding for the nested queries into flat queries that captures the mixed semantics, and proposes a new property: *encoding equivalence*, to which nested query equivalence under mixed semantics can be reduced to. Notice that this approach is different with respect to ours in the sense that it focus on mixed semantics while we focus on set semantics ([18, 12] focus on set semantics too). We consider set semantics since it makes easier the generalization of our

previous results from the relational setting. DeHaan also proposes some extensions to the query language, but he does not consider the use of negation or arithmetic comparisons.

The main difference of the approach followed by these three works with respect to ours is that we do not intend to translate the mapping assertions into some condition over conjunctive queries. Instead, we propose a translation that takes into account the class of queries and constraints the CQC method is able to deal with, especially the fact that the CQC method allows for the use of negation on derived atoms. We take advantage of this feature and propose a translation that expresses the definition of query inclusion and query equality into first-order logic, and then rewrites it into the syntax required by the CQC method by means of algebraic manipulation. We finally obtain a set of integrity constraints (DEDs) that model the semantics of the mapping assertions and that allows us to encode the mapping when we reformulate mapping validation in terms of query satisfiability.

8. Conclusion

We follow and approach to mapping validation that allows the designer to check whether the mapping satisfies certain desirable properties. We focus in this paper on how to apply this approach to the validation of nested relational mapping scenarios in which mapping assertions are either inclusions or equalities of nested queries. We encode the given nested relational mapping scenario into a single flat database schema. That includes the flattening of the mapped schemas and the mapping's queries, and the encoding of the mapping assertions as integrity constraints. Then, we take advantage from our previous work on validating flat relational mappings [22] and reformulate each desirable property check in terms of a query satisfiability problem over the flat database schema. The idea is that the nested relational mapping will satisfy a certain desirable property if and only if the query that results from the reformulation is satisfiable on the flat database schema. To solve the query satisfiability problem, we apply the CQC method [14], which, to the best of our knowledge, is the only method able to deal with the class of scenarios that we consider here.

References

1. Alexe, B., Chiticariu, L., Miller, R. J., Tan, W. C.: Muse: Mapping Understanding and deSign by Example. In: Proc. ICDE, 10-19. (2008)
2. Alexe, B., Tan, W. C., Velegrakis, Y.: STBenchmark: towards a benchmark for mapping systems. PVLDB 1(1), 230-244. (2008)

3. Amano, S., Libkin, L., Murlak, F.: XML schema mappings. In: Proc. PODS, 33-42. (2009)
4. Arenas, M., Libkin, L.: XML data exchange: Consistency and query answering. J. ACM 55(2). (2008)
5. Bernstein, P. A., Haas, L. M.: Information integration in the enterprise. Commun. ACM 51(9), 72-79. (2008)
6. Bonifati, A., Mecca, G., Pappalardo, A., Raunich, S., Summa, G.: Schema mapping verification: the spicy way. In: Proc. EDBT, 85-96. (2008)
7. Chiticariu, L., Tan, W. C.: Debugging Schema Mappings with Routes. In: Proc. VLDB, 79-90. (2006)
8. DeHaan, D.: Equivalence of nested queries with mixed semantics. In: Proc. PODS, 207-216. (2009)
9. Deutsch, A., Ludäscher, B., Nash, A.: Rewriting queries using views with access patterns under integrity constraints. Theor. Comput. Sci. 371(3), 200-226. (2007)
10. Deutsch, A., Tannen, V.: Optimization Properties for Classes of Conjunctive Regular Path Queries. In: Proc. DBPL, 21-39. (2001)
11. Deutsch, A., Tannen, V.: XML queries and constraints, containment and reformulation. Theor. Comput. Sci. 336(1), 57-87. (2005)
12. Dong, X., Halevy, A. Y., Tatarinov, I.: Containment of Nested XML Queries. In: Proc. VLDB, 132-143. (2004)
13. Fagin, R., Kolaitis, P. G., Miller, R. J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. 336(1), 89-124. (2005)
14. Farré, C., Teniente, E., Urpí, T.: Checking query containment with the CQC method. Data Knowl. Eng. 53(2), 163-223. (2005)
15. Fuxman, A., Hernández, M. A., Ho, H., Miller, R. J., Papotti, P., Popa, L.: Nested Mappings: Schema Mapping Reloaded. In: Proc. VLDB, 67-78. (2006)
16. Halevy, A. Y.: Technical perspective - Schema mappings: rules for mixing data. Commun. ACM 53(1), 100. (2010)
17. Halevy, A. Y., Mumick, I. S., Sagiv, Y., Shmueli, O.: Static analysis in datalog extensions. J. ACM 48(5), 971-1012. (2001)
18. Levy, A. Y., Suciu, D.: Deciding Containment for Queries with Complex Objects. In: Proc. PODS, 20-31. (1997)
19. Madhavan, J., Bernstein, P. A., Domingos, P., Halevy, A. Y.: Representing and Reasoning about Mappings between Domain Models. In: Proc. AAAI/IAAI, 80-86. (2002)
20. Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., Fagin, R.: Translating Web Data. In Proc. VLDB, 598-609. (2002)
21. Queralt, A., Teniente, E.: Decidable Reasoning in UML Schemas with Constraints. In: Proc. CAiSE, 281-295. (2008)
22. Rull, G., Farré, C., Teniente, E., Urpí, T.: Validation of mappings between schemas. Data Knowl. Eng. 66(3), 414-437. (2008)
23. Rull, G., Farré, C., Teniente, E., Urpí, T.: Validation of schema mappings with nested queries. Technical Report ESSI-TR-12-5 http://hdl.handle.net/2117/16746 (2012)
24. Rull, G.: Validation of Mappings between Data Schemas. Ph.D. Thesis. Universitat Politècnica de Catalunya. http://hdl.handle.net/10803/22679. (2011)
25. Yu, C., Jagadish, H. V.: XML schema refinement through redundancy detection and normalization. VLDB J. 17(2), 203-223. (2008)

Guillem Rull, Carles Farré, Ernest Teniente, and Toni Urpí

Guillem Rull is currently postdoc researcher at the Department of Service and Information System Engineering (ESSI) at the Universitat Politècnica de Catalunya – BarcelonaTech. He received his Ph.D. degree from the Technical University of Catalonia in 2011. His current research interests are involved with schema and mapping validation.

Carles Farré is currently associate professor at the Department of Service and Information System Engineering (ESSI) at the Universitat Politècnica de Catalunya – BarcelonaTech. He received his Ph.D. degree from the Technical University of Catalonia in 2003. He worked on deductive databases, query containment and schema validation. His research interests are involved with conceptual modeling and data and service integration.

Ernest Teniente is a full professor at the Department of Service and Information System Engineering at the Universitat Politècnica de Catalunya – BarcelonaTech, where he teaches graduate and undergraduate courses on software engineering and databases. He got his PhD in Computer Science from the same university. He was a visiting researcher at the Politecnico di Milano and at the Universita' di Roma Tre, in Italy. His current research interests are focused on conceptual modeling, automated reasoning on conceptual schemas and data integration. He is author of more than 50 publications in international conferences and journals in the areas of databases and software engineering, and he is regularly invited to serve on the Program Committees of international conferences in these areas.

Toni Urpí is currently associate professor at the Department of Service and Information System Engineering (ESSI) at the Universitat Politècnica de Catalunya – BarcelonaTech. He received his Ph.D. degree from the Technical University of Catalonia in 1993. He worked on deductive databases, database updates, integrity constraint maintenance, query containment, and schema validation. Current research interests are involved with conceptual modelling and data integration.