VALIDATION OF MAPPINGS BETWEEN DATA SCHEMAS

PhD. THESIS
ADvised by DR. CARLES FARRÉ TOST
AND DR. ANTONI URPI TUBELLA

BARCELONA
2010
A thesis presented by Guillem Rull Fort
in partial fulfillment of the requirements for the degree of
Doctor en Informàtica per la Universitat Politècnica de Catalunya
Acknowledgements

This work was supported in part by Microsoft Research through the European PhD Scholarship Programme, by the Spanish Ministerio de Educación y Ciencia under projects TIN2005-05406 and TIN2008-03863, and by the Universitat Politècnica de Catalunya (UPC) under a FPI-UPC grant.
Abstract

In this thesis, we present a new approach to the validation of mappings between data schemas. It allows the designer to check whether the mapping satisfies certain desirable properties. The feedback that our approach provides to the designer is not only a Boolean answer, but either a (counter)example for the (un)satisfiability of the tested property, or the set of mapping assertions and schema constraints that are responsible for that (un)satisfiability.

One of the main characteristics of our approach is that it is able to deal with a very expressive class of relational mapping scenarios; in particular, it is able to deal with mapping assertions in the form of query inclusions and query equalities, and it allows the use of negation and arithmetic comparisons in both the mapping assertions and the views of the schemas; it also allows for integrity constraints, which can be defined not only over the base relations but also in terms of the views.

Since reasoning on the class of mapping scenarios that we consider is, unfortunately, undecidable, we propose to perform a termination test as a pre-validation step. If the answer of the test is positive, then checking the corresponding desirable property will terminate.

We also go beyond the relational setting and study the application of our approach to the context of mappings between XML schemas.
Introduction

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 [Hal10]. 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 [Haa07, BH08, CK10].

Two well-known approaches to information integration are data exchange [FKMP05] and data integration [Len02]. In the data exchange approach, data stored in multiple heterogeneous sources is extracted, restructured into a unified format, and finally materialized in a target schema [CK10]. In particular, the data exchange problem focuses on moving data from a source database into a target database, and a mapping is needed to specify how the source data is to be translated in terms of the target schema. In data integration, several local databases are to be queried by users through a single, integrated global schema; mappings are required to determine how the queries that users pose on the global schema are to be reformulated in terms of the local schemas.

Several formalisms are used to define mappings [CK10]. In data exchange, tuple-generating dependencies (TGDs) and equality-generating dependencies (EGDs) are widely used [FKMP05]. Source-to-target TGDs are logic formulas in the form of \( \forall X (\phi(X) \rightarrow \exists Y \psi(X, Y)) \), where \( \phi(X) \) is a conjunction of atomic formulas over the source schema and \( \psi(X, Y) \) is a conjunction of atomic formulas over the target schema. A target EGD is of the form \( \forall X (\phi(X) \rightarrow X_1 = X_2) \), where \( \phi(X) \) is a conjunction of atomic formulas over the target schema and \( X_1, X_2 \) are variables from \( X \).
In the context of data integration, global-as-view (GAV), local-as-view (LAV) and global-and-local-as-view (GLAV) [Len02, FLM99] are the most common approaches. A GAV mapping associates queries defined over the local schemas to tables in the global schema (e.g., a set of assertions in the form of $Q_{local} \subseteq T_{global}$). A LAV mapping associates queries over the global schema to tables in the local schemas (e.g., assertions in the form of $T_{local} \subseteq Q_{global}$). The GLAV mapping formalism is a combination of the other two; it associates queries defined over the local schemas with queries defined over the global schema (e.g., assertions in the form of $Q_{local} \subseteq Q_{global}$).

A further formalism that has recently emerged is that of nested mappings [FHH+06], which extends previous formalisms for relational and nested data by allowing the nesting of TGDs.

Model management [BHP00, Ber03, BM07, Qui09] is also a widely known approach which establishes a conceptual framework for handling schemas and mappings generically, and provides a set of generic operators such as the Merge operator [QKL07, PB08], which integrates two schemas given a mapping between them; the ModelGen operator [ACT+08], which translates a given schema from one model into another (e.g., from XML into the relational model); or the composition of mappings [NBM07, BGMN08, KQL+09].

The ModelGen operator is required for any model management system in order to be generic, since such a system must be able to deal with schemas represented in different models. An implementation for this operator is proposed in [ACT+08]. This implementation follows a metamodel approach in which each model is seen a set of constructs. A supermodel is then defined by considering all constructs in the supported models [AT96]; in this way, any schema of a supported model is also a schema of the supermodel, and the translation from the source model into the target model becomes a transformation inside the supermodel. This transformation consists of a sequence of elementary transformations that remove/add constructs as required for the given schema to fit the target model. The supermodel is implemented as a relational dictionary [ACB05, AGC09] in which models and schemas are represented in a uniform way. An extension of this approach is proposed in [ACB06], which translates both the schema and the data stored in the database. Another extension that does not translate the data directly but provides a mapping (a set of views) between the original schema and the resulting translation has recently been presented in [ABBG09a].

Another requirement for a model management system to be generic is the ability to provide a single implementation for each operator, which must be able to deal with schemas and mappings independently of their representation. An important work in this direction is the Generic Role-
based Metamodel (GeRoMe) [KQCJ07, KQLL07] and its generic schema mapping language [KQLJ07, KQL+09]. The use of GeRoMe provides a uniform representation of schemas defined in different modeling languages (in this sense GeRoMe is similar to the supermodel of [ACT+08]). In GeRoMe, schema elements are seen as objects that play different roles, and these roles act as interfaces to other schema elements. Model management operators are to be implemented to interact with the roles exposed by the elements of the manipulated schema; in this way, the operators become independent of the underlying model and also their implementation is simplified as it only needs to focus on those roles that are relevant to the operator. Another research effort in this same direction is the Model Independent Schema Management (MISM) platform [ABBG09b], which shows how to use the dictionary and the supermodel from [ACT+08] to implement different model management operators.

In the context of conceptual modeling, QVT (Query/View/Transformation) [OMG08] is a standard language defined by the OMG (Object Management Group) to specify transformations between conceptual schemas.

Languages like XSLT, XQuery and SQL are also used to specify mappings in several existing tools that help engineers to build them [Alt10, Sty10]. One example of a system for producing mappings is Clio [HHH+05], which can be used to semi-automatically generate a schema mapping from a set of correspondences between schema elements (e.g., attribute names). This set of inter-schema correspondences is usually called a matching [BMPQ04]. In fact, finding a matching between the schemas is the first step towards developing a mapping. A significant amount of work on schema-matching techniques can be found in the literature—see [RB01] for a survey.

Nevertheless, the process of designing a mapping always requires feedback from a human engineer. The designer guides the process by choosing among candidates and successively refining the mapping. The designer needs thus to check whether the mapping produced is in fact what was intended, that is, the developer must find a way to validate the mapping.

1.1 Our Approach to Mapping Validation

The goal of this thesis is to validate mappings by means of testing whether they meet certain desirable properties. Our approach is aimed at allowing the user to ask whether the desirable properties hold in the mapping being designed, and at providing the user with certain feedback that helps him to understand and fix the potential problems.
Consider, for example, the mapping scenario shown in Figure 1.1. Relational schemas $A$ and $B$ model data about employees, their salary and their bosses. Underlined attributes denote keys, and dashed arrows referential constraints. Attribute $Employee_{B}.boss$ is the only one that accepts null values. Solid arrows depict inter-schema correspondences, i.e., a matching of the schemas.

Let us assume that we have a mapping between schemas $A$ and $B$ which maps into database $B$ the employees in database $A$ that have a salary above a certain threshold. Let us also assume that the mapping consists of two assertions: $m_1$ and $m_2$, expressed in the GLAV formalism. Assertion $m_1$ maps information of employees that may or may not have a boss. Assertion $m_2$ takes care of specific information of employees that have a boss.

$$m_1: \quad \text{select name, salary from Employee}_A \quad \text{where salary} \geq 10000$$

$$m_2: \quad \text{select wf.emp, wf.boss from WorksFor}_A wf, Employee_A e \quad \text{where wf.emp} = e.name \quad \text{and e.salary} \geq 10000$$

The mapping is syntactically correct, and it may seem perfectly right at a first glance. However, it turns out that assertion $m_1$ can only be satisfied trivially, that is, only those instances of schema $A$ in which the left-hand-side query of $m_1$ gets an empty answer may satisfy the

Figure 1.1: Example mapping scenario.
mapping assertion. In this case, we say that the mapping is not strongly satisfiable. In general, we say that a mapping is strongly satisfiable if there is a source and target schema instance that satisfy all mapping assertions in a non-trivial way, where trivial satisfaction means that a mapping assertion \( Q_A \subseteq Q_B \) \((Q_A = Q_B)\) becomes \( \emptyset \subseteq \text{answer-of-}Q_B \) \((\emptyset = \emptyset)\) after the evaluation of its queries. We say that a mapping is weakly satisfiable if there is a source and target instance that satisfy at least one mapping assertion in a non-trivial way. Note that the previous mapping is weakly satisfiable since \( m_2 \) can be satisfied in a non-trivial way.

Most likely, the mapping scenario in Figure 1 is not what the designer intended. Therefore, being able to check the strong satisfiability desirable property and obtaining an explanation that highlights the mapping assertion \( m_1 \) and the constraint “\( \text{salary} \leq 2000 \)” of schema \( A \) as the source of the problem might help the designer to realize that \( m_1 \) was probably miswritten, and that it should be mapping those employees with a salary above one thousand, instead of ten thousand.

Assume now that we have come up with an alternative mapping that is more compact than the previous one. It consists of the single assertion \( m_3 \). The main difference with respect to \( m_1 \) and \( m_2 \) is that \( m_3 \) uses left outer join to map at the same time the information common to all employees and the information specific to the employees that have a boss.

We want to know if mapping \( \{m_3\} \) is equivalent to \( \{m_1, m_2\} \) (we assume the problem of \( m_1 \) not being strongly satisfiable has been fixed). We can achieve that by means of the mapping inference property [MBDH02], that is, by testing whether \( m_1 \) and \( m_2 \) are inferred from \( \{m_3\} \), and whether \( m_3 \) is inferred from \( \{m_1 \) and \( m_2\} \). The result of the test will be that \( m_1 \) and \( m_2 \) are indeed inferred from \( \{m_3\} \) (as expected), but not vice versa. To exemplify the latter, consider the following pair of schema instances:

**Instance of A:**
- EmployeeA('e1', 'addr1', 1000)
- EmployeeA('e2', 'addr2', 1000)
- WorksForA('e1', 'e2')

**Instance of B:**
- EmployeeB(0, 'e1', null, 'cat1')
- EmployeeB(1, 'e1', 2, 'cat2')
- EmployeeB(2, 'e2', null, 'cat1')
- CategoryB('cat1', 1000)
- CategoryB('cat2', 2000)
The instances are consistent with respect to the integrity constraints of the schemas. They also satisfy mapping assertions \( m_1 \) and \( m_2 \), but do not satisfy assertion \( m_3 \). The question is that \( m_1 \) and \( m_2 \) do not guarantee the correlation between the salary of an employee and the information about who is his boss. That is shown in the counterexample by the employee ‘e1’ from \( A \), who is mapped into \( B \) as two different employees (same name, but different ids), one with the right salary and without boss, and the other with the right boss and a wrong salary. Therefore, this counterexample shows that \( \{ m_3 \} \) is not only more compact than \( \{ m_1, m_2 \} \), but also more accurate. It is also clear that, for this property, being able to feedback the user with a counterexample like the previous one would certainly help him to understand and fix the problem.

To illustrate one last desirable property, suppose that we wonder whether mapping \( \{ m_3 \} \) maps into database \( B \) not only the salary and the boss’s name of the employees selected from database \( A \), but also the personal information of each boss, i.e., their salary. To answer this question, we can define the following query, which selects, for each employee with a salary \( \geq 1000 \), the corresponding boss and the salary of the boss:

\[
\begin{align*}
\text{select} & \text{ wf.boss, e1.salary} \\
\text{from} & \text{ WorksForA wf, EmployeeA e1, EmployeeA e2} \\
\text{where} & \text{ wf.boss = e1.name and wf.emp = e2.name} \\
\text{and} & \text{ e2.salary >= 1000}
\end{align*}
\]

Then, we can check whether mapping \( \{ m_3 \} \) is lossless with respect to the query. If we do so, we will realize that \( \{ m_3 \} \) is actually lossy with respect to the query, that is, not all the salaries of the bosses of employees with a salary \( \geq 1000 \) in database \( A \) are mapped into database \( B \). As a counterexample, consider the following two instances of schema \( A \):

**Instance1 of A:**
- EmployeeA(‘e1’, ‘addr1’, 1000)
- EmployeeA(‘e2’, ‘addr2’, 1000)
- WorksForA(‘e1’, ‘e2’)

**Instance2 of A:**
- EmployeeA(‘e1’, ‘addr1’, 1000)
- EmployeeA(‘e2’, ‘addr2’, 700)
- WorksForA(‘e1’, ‘e2’)

They get a different answer for the query, since employee ‘e2’ (the boss) has a different salary on each instance. However, the mapping allows these two instances to be mapped into a same instance of schema \( B \), e.g., the one shown below:

**Instance of B:**
- EmployeeB(0, ‘e1’, 1, ‘cat1’)
- EmployeeB(1, ‘e2’, null, ‘cat1’)
- CategoryB(‘cat1’, 1000)
The counterexample shows that the problem is that in order for the salary of a boss to be mapped, it has to be \( \geq 1000 \), just like any other employee. However, since we are asking about the bosses of those employees that have a salary \( \geq 1000 \), if the salary of a boss was \( < 1000 \), it would mean that there is an employee with a salary higher than his boss’s, which most likely is not an intended valid state for database \( A \). Therefore, being able to provide such a counterexample to the designer may help him to realize that schema \( A \) is probably underspecified, and that a situation in which an employee has a salary higher than his boss’s is unlikely to happen in practice, i.e., the designer could conclude that the mapping is actually enough to capture the information represented by the query.

1.2 Existing Approaches to Mapping Validation

In this section, we briefly review the main existing approaches to validate mappings. More details on these and other previous work are given in the related work chapter.

Our work is inspired by [MBDH02], where a generic framework for representing and reasoning about mappings is presented. [MBDH02] identifies three important properties of mappings: mapping inference, query answerability and mapping composition. The last property is however not very interesting from the point of view of validation, since existing techniques for mapping composition [FKPT05, NBM07, BGMN08] already produce mappings that satisfy the property. Regarding the other two properties: mapping inference and query satisfiability, they are addressed in [MBDH02] for the particular setting of relational schemas without integrity constraints and the class of mappings that consists of assertions in the form of \( Q_1 = Q_2 \), where \( Q_1 \) and \( Q_2 \) are conjunctive queries over the mapped schemas.

In [ACT06, CT06], a system for debugging mappings in the data exchange context is presented. The main feature of this system is the computation of routes [CT06]. Given a source and a target instance, the system allows the user to select a subset of the tuples in the target instance, and then it provides the routes that explain how these tuples have been obtained from the source, that is, it indicates which mapping assertions have been applied and which source tuples they have been applied to. Algorithms to compute one or all routes for a given user selection are provided. [ACT06] considers relational and nested relational schemas, and mappings formed by tuple-generating dependencies (TGDs).

The Spicy system [BMP+08] allows obtaining a ranking of the mapping candidates generated by a mapping-generation tool like Clio. The goal of this system is to help the designer to choose
among the different mapping candidates. Mappings are expressed as sets of TGDs, and schemas are either relational or nested relational. The Spicy system also requires that a source and a target instance are available. The idea is that each mapping candidate is applied to the source instance and the produced target instance is compared to the existing one. This comparison produces a similarity measure that is then used to make the ranking. The Spicy system has evolved into the +Spicy system [MPR09, MPRB09], which introduces the computation of cores into the mapping generation algorithms in order to further improve the quality of mappings. +Spicy deals with (nested) relational schemas with TGDs as schema constraints and mapping assertions. It rewrites the source-to-target TGDs in order to allow them to be “compiled” into executable SQL scripts that compute core solutions for the corresponding data exchange problem. Recently, algorithms that are able to generate such executable SQL scripts while taking into account the presence of EGDs in the schemas have been introduced into +Spicy [MMP10].

[YMHF01] proposes an approach to refine mappings between relational schemas by means of examples. This approach requires the availability of a source instance so the system can select a subset of tuples from this instance and build and example that shows the user how the target instance produced by the mapping would look like. The user can modify the mapping and see then how the modifications affect the produced target instance. Moreover, the examples are also intended to show the user the differences between the mapping candidates. The formalism of the produced mappings is the global-as-view (GAV), where assertions are in the form of $Q_{source} \subseteq T_{target}$, and $Q_{source}$ is a SQL query over the source, and $T_{target}$ is a target table.

The Muse system [ACMT08] extends the work of [YMHF01] to the context of nested mappings between nested relational schemas. It does not only help the user to choose among alternative representations of ambiguous mappings, but also guides the designer on the specification of the desired grouping semantics. Muse is also able to construct synthetic examples whenever meaningful ones cannot be drawn from the available source instance. Such synthetic examples are obtained from the mapping definition by freezing variables into constants.

TRAMP [GAMH10] is a system for understanding and debugging schema mappings and data transformations in the context of data exchange. It allows the user to trace errors caused either by the data, the mapping or the executable transformation that implements the mapping. TRAMP is based on provenance. The user can query which source data contributed to the existence of some target data (data provenance), or he can also query which parts of the executable transformations contribute to a target tuple (transformation provenance), or which parts of the transformation
correspond to which mapping assertions (mapping provenance). TRAMP assumes mappings to be sets of source-to-target TGDs. Schemas may contain key and referential constraints.

In [SVC+05], the authors propose a methodology for the integration of spatio-temporal databases. One of the steps of this methodology is the validation of the mapping. Since they represent both the database schemas and the mapping in a Description Logics (DL) language, the validation is done by means of the reasoning mechanisms DL provide. Specifically, the validation consists in checking whether some of the \textit{concepts} defined in the schemas become \textit{unsatisfiable} once the mapping is taking into account. The methodology simply proposes that the mapping and/or the schemas must be successively refined until all concepts are satisfiable.

A framework for the automatic verification of mappings with respect to a domain ontology is presented in [CBA10]. It considers mappings to be source-to-target TGDs, and it requires mappings to be semantically annotated. A semantic annotation assigns meaning to each variable in the source-to-target dependencies relative to the domain ontology, that is, it attaches a concept from the ontology to each variable. A same variable may have however different meaning when appears on the source side of the TGD than when appears on the target side of the TGD, i.e., a variable may get attached two concepts from the ontology: one that denotes its meaning in the context of the source schema, and another that denotes its meaning in the context of the target schema. Based on these semantic annotations, [CBA10] derives a set of verification statements from each source-to-target TGD whose compatibility is then check against the domain ontology by means of formal reasoning.

[ALM09] studies the complexity of the \textit{consistency} and \textit{absolute consistency} problems for a language of XML mappings between DTDs based on mapping assertions expressed as implications of tree patterns. Such a mapping is consistent if there is at least one document that conforms to the source DTD and is mapped into a document that conforms to the target DTD. A mapping is absolute consistent if the former happens for all documents that conform to the source DTD. Translated into our setting, the former consistency property would correspond to our mapping satisfiability property. The main difference is that consistency only requires the satisfaction of the mapping assertions, but does not distinguish between trivial and non-trivial satisfaction as we do in our relational setting.

In [BFFN05], the \textit{query preservation} property is studied for a class of XML mappings between DTDs. A mapping is said to be query preserving with respect to a certain query language if all the queries that can be defined in that language on the source schema can also be computed over the target schema. Note that this property is related to our mapping losslessness property, but
they are not the same property. [BFFN05] shows that query preservation is undecidable for XML mappings expressed in a certain fragment of the XQuery and XSLT languages, and propose the notion of XML schema embedding, which is a class of mappings guaranteed to be query preserving with respect to the regular XPath language [W3C99].

1.3 Contributions of this Thesis

We propose an approach to validate mappings by means of checking certain desirable properties. We consider an expressive class of relational mapping scenarios that allows the use of negation and arithmetic comparisons in both the mapping assertions and the views of the schemas. The class of schema constraints we consider is that of disjunctive embedded dependencies (DEDs) [DT01] extended with derived relation symbols (views) and arithmetic comparisons. A DED (without extensions) is a logic formula in the form of:

$$\forall \bar{X} \phi(\bar{X}) \rightarrow \exists \bar{Y_1} \psi_1(\bar{X}, \bar{Y_1}) \lor ... \lor \exists \bar{Y_m} \psi_m(\bar{X}, \bar{Y_m})$$

where $$\phi(\bar{X})$$ and $$\psi_m(\bar{X}, \bar{Y_m})$$ are conjunctions of relational atoms of the form $$R(w_1, ..., w_n)$$ and (dis)equalities of the form $$(w_1 \neq w_2) w_1 = w_2$$, where $$w_1, w_2, ..., w_n$$ are either variables or constants.

We consider a global-and-local-as-view mapping formalism, which allows for assertions in the form of $$Q_A \subseteq Q_B$$ or $$Q_A = Q_B$$, where $$Q_A$$ and $$Q_B$$ are queries over the mapped schemas.

We identify three properties of mappings that have been already considered important in the literature: mapping satisfiability [ALM09], mapping inference [MBDH02] and query answerability [MBDH02].

We consider two flavors of mapping satisfiability: strong and weak, which address the trivial satisfaction of the mapping by requiring that all or at least one mapping assertion, respectively, is non-trivially satisfied.

We show that the query answerability property is not useful when mapping assertions express inclusion of queries. To address this, we propose a new property that we call mapping losslessness. We show that when all mapping assertions are equalities of queries, then mapping losslessness and query answerability are equivalent.

We perform the validation by reasoning on the mapped schemas and the mapping definition, and we do not require any instance data to be provided. This is important since relying on specific schema instances may not reveal all potential pitfalls. Therefore, schema-based mapping validation approaches like the one we propose here are a necessary complement to existing
instance-based mapping debuggers and mapping refining tools [YMHF01, CT06, BMP+08, ACMT08].

Moreover, our approach does not only provide the user with a Boolean answer, but also with additional feedback to help him understand why the tested property holds or not. The feedback can be in the form of some instances of the mapped schemas that serve as an example or counterexample for the tested property, or in the form of highlighting the subset of schema constraints and mapping assertions that is responsible for the test result. We refer to the latter task as computing an explanation.

Since the problem of reasoning on the class of mapping scenarios we consider is semi-decidable, we propose to perform a termination test as a pre-validation step. If positive, the test guarantees that the check of the target desirable property is going to terminate. We adapt the test from the one proposed in [QT08] for the context of reasoning on UML conceptual schemas.
Finally, we study the extension of our approach to a specific context that has received a growing attention during the last years: XML mappings. In particular, we study how to translate XML mapping scenarios into a flat, logic formalism so we can take advantage of our previous results from the relational setting.

We have also implemented our results in a mapping validation tool called MVT [RFTU09], which was presented in the demo track of the EDBT 2009 conference.

Table 1.1 compares our work with the existing approaches to mapping validation. As can be seen in the table, the existing approaches can be classified in two groups: those that check some desirable properties of the mappings by reasoning on the schema and mapping definitions, and those that rely on schema instances to help the designer understand, refine and debug the mappings. Our approach clearly falls in the first group. Regarding the formalisms, the schema and mapping formalism we deal with subsumes those in [MBDH02, CBA10, CT06, BMP+08, ACMT08, GAMH10, SVC+05] and intersects with those in [YMHF01, ALM09, BFFN05]; we will see that in detail in the related work chapter.

Table 1.2 compares our work with the other desirable-property checking approaches in terms of the properties that are considered and the feedback that is provided to the user. The table shows that while previous approaches focus on the check of the property and disregard the feedback provided to the user, we provide an answer that is more explanatory than a simple Boolean value. Regarding the desirable properties, we do not consider composition [MBDH02] or invertibility [BFFN05] of mappings, since we understand the interest of these problems is currently on the actual computation of the composition [MH03, FKPT05, NBM07, BGMN08] and the inverse [Fag07, FKPT08, FKPT09, APRR09], respectively, which are research fields on their own, and
thus beyond the scope of this thesis. We do not yet address the absolute consistency property identified in [ALM09], but we plan to do it as further work. We do not address either the compatibility w.r.t. a domain ontology proposed by [CBA10], since it requires the availability of such an ontology and of semantic annotations which we do not consider. Regarding query preservation [BFFN05], we consider the related properties of query answerability [MBDH02] and mapping losslessness, but not the property itself; we however intend to study it as future research together with the absolute consistency property, since we think there may be some connection between the two. The remaining properties are either addressed by our approach or easily inferred from the ones we consider—see the related work chapter for a detailed comparison.

See also the related work chapter for a comparison of our work with existing approaches in the areas of computing explanations and translating XML mapping scenarios into logic.

In the next subsections, we give more details on each one of our contributions.

### 1.3.1 Checking Desirable Properties of Mappings

We propose to reformulate each desirable property test as a query satisfiability problem. We define a new database schema that includes the two schemas being mapped, and include the mapping assertions as integrity constraints of the new schema. We finally define a distinguished query that encodes the property to be tested, in such a way that the satisfiability of the distinguished query over the new schema determines whether the desirable property holds or not for the current mapping. We also show that this reduction to query satisfiability does not increase the complexity of the mapping validation problem by showing that one can also make a reduction from query satisfiability to each desirable-property checking problem.

To perform the query satisfiability tests, we rely on the CQC method [FTU05], which has been successfully used in the context of database schema validation [FTU04, TFU+04]. The method works on a first-order logic representation of the database schema and the distinguished query, but the translation into logic is quite straightforward in the relational case. To the best of our knowledge, the CQC method is the only query satisfiability method able to deal with the class of database schemas and queries that we consider.

The CQC method is a constructive method, that is, it tries to build a database instance in which the query has a non-empty answer. To instantiate the tuples to be added to this database instance, the method uses a set of Variable Instantiation Patterns (VIPs). Each application of the VIPs provides a finite number of constants to be tried, which results in a finite number of candidate database instances to be considered. If one of these instances satisfies the query and the
integrity constraints at the same time, then the instance is an example that shows the query is satisfiable. Otherwise, the VIPs guarantee that if no solution can be constructed with the constants they provide, then no one exists.

We have published this work in Data & Knowledge Engineering, Volume 66, Number 3, 2008 [RFTU08a].

1.3.2 Explaining Validation Test Results

The CQC method provides two types of feedback: a database instance that exemplifies the satisfiability of the tested query, or a simple negative Boolean answer that indicates the query is not satisfiable.

The database instance provided by the CQC method can be straightforwardly translated back into an example/counterexample for the mapping validation test. Whether it will be an example or a counterexample is going to depend on the specific property that we are testing. For instance, mapping satisfiability is suitable to be exemplified when the mapping is indeed satisfiable, while for mapping inference is best to provide a counterexample when the inference cannot be made.

The remaining question is the computation of an explanation for the case in which the CQC method provides a negative Boolean answer. To the best of our knowledge, none of the existing methods for query satisfiability checking [DTU96, ZO97, HMSS01] provides any kind of explanation in this case.

We propose to explain such a test result by means of highlighting on the mapping scenario the schema constraints and mapping assertions responsible for the impossibility of finding an example/counterexample. For instance, in the strong mapping satisfiability test of \{m_1, m_2\} in Section 1.1, the explanation for the unsatisfiability of the mapping would be the set \{constraint “salary ≤ 2000” of schema A, mapping assertion “m_1”\}.

Actually, there may be more than one explanation of this kind for a single test, so we firstly propose a black-box method to compute all minimal explanations. The approach is black-box because it makes successive calls to an underlying method, in our case, the CQC method; and the computed explanations are minimal in the sense that any proper subset of them is not an explanation.

The black-box method works at the level of the query satisfiability problem, that is, before translating the test result back into the mapping validation context. The computed explanations can be easily converted into explanations for the mapping validation test.
In a first stage, the black-box method provides one minimal explanation, which can be then extended during a second stage into a maximal set of disjoint minimal explanations. These two first stages have the advantage that the number of calls required to the underlying method is linear with respect to the number of constraints in the schema. The third and final stage extends the outcome of the second one into the set of all possible minimal explanations. It however requires an exponential number of calls to the underlying method. Notice that this cost cannot be avoided, since, in the worst case, the number of explanations for a certain test is indeed exponential with respect to the number of constraints.

The drawback of the black-box method is the fact that, for large schemas, the runtime of each call to the CQC method may be high. That means that even computing one single minimal explanation can be a time-consuming process. It would be therefore desirable that the CQC method could provide an approximation to one of the possible minimal explanations as a result of its execution; this way, the user could decide whether the approximation suffices or more accurate explanations are needed. To achieve that, we propose a glass-box approach, that is, a modification of the CQC method that returns an approximated explanation when the tested query is not satisfiable. By approximated explanation, we mean that the explanation is not necessarily minimal.

The main advantage of the glass-box approach is that it does not require any additional call to the CQC method. Moreover, it may dramatically improve the efficiency of the CQC method as a side effect. That is because the approach is based on the analysis of the constraint violations that occur during the search for a solution, and the information obtained from these analyses is used to prune the remaining search space.

Going one step further, we combine the glass-box and the black-box approaches in order to obtain the advantages from both of them. The idea is that the black-box approach can use the approximated explanation provided by the glass-box approach in order to significantly reduce the number of calls to be made to the CQC method. This way, the user gets an initial, approximated explanation from the glass-box approach, which then can choose to refine into a minimal one by applying the first stage of the black-box approach. If the user still wants more information, the second and the third stage can be applied, and these stages would also benefit from the approximations provided by the successive calls they make to the CQC method.

We have published our black-box approach in the CIKM 2007 conference [RFTU07], and our glass-box method in the DEXA 2008 conference [RFTU08b].
1.3.3 Testing Termination of Validation Tests

Reasoning on the general class of mapping scenarios we consider is, unfortunately, undecidable, which means the validation tests may never end. To deal with this, we propose to perform a termination test previous to the validation of a desirable property.

We adapt the termination test proposed in [QT08] for the context of reasoning on UML schemas to our mapping context. We apply the termination test after the reformulation of the validation problem into a query satisfiability problem, and before the application of the CQC method. This way, we have a single database schema to analyze.

The termination test builds a graph that represents the dependencies that exist between the integrity constraints of the schema. Then, it analyzes the cycles in the graph and checks whether they satisfy certain termination conditions. In particular, the termination test defines three conditions that are sufficient for the termination of the CQC method. Note that these conditions are sufficient but not necessary, as expected due to the undecidability of the termination checking problem.

We extend the termination test in two directions:

- We consider database schemas whose integrity constraints and deductive rules may have more than one level of negation, that is, negated derived literals whose deductive rules contain also negated literals are allowed. This feature was not required in [QT08] since their translation of UML/OCL schemas into first-order logic did not require more than one level of negation. In our context, however, the translation of the mapping scenario into logic may contain more than one level of negation.

- We study the application of the termination conditions to database schemas in which the dependency graph contains overlapping cycles (by overlapping we mean vertex-overlapping). The case in which cycles are disjoint (i.e., vertex-disjoint) was already addressed by [QT08].

We provide formal proofs for our results.

1.3.4 Validating XML Mappings

Since the emergence of the Web, the ability to map not only relational but also XML data has become crucial. A sign of this is the growing interest of the research community on this kind of mappings during the last years, e.g., [PVM+02, DT05, BFFN05, RHC+06, ALM09]. Most tools and approaches to aid the construction of mappings support some class of XML mappings, e.g.,
Clio-based approaches typically allow mappings between nested relational schemas—see, for instance, [PVM+02, FHH+06, BMP+08, ACMT08].

We generalize our previous results so we can deal with schemas defined in a subset of the XML Schema Definition (XSD) language [W3C04], and mappings whose queries are defined in a subset of the XQuery language [W3C07]. This way, we are able to check the mapping desirable properties on a class of mapping scenarios that includes the nested relational one. The key point of this generalization is the translation of the given XML mapping scenario into the first-order logic formalism used by the CQC method. We combine existing proposals for the translation of different parts of the XML schemas and XQueries [YJ08, DT05]. We also propose a new way of translating the inclusion and equality mapping assertions, which takes into account the class of schemas and queries the CQC method is able to deal with. Existing approaches to the translation of this kind of assertions are mainly in the area of query containment checking [LS97, DHT04] and query equivalence checking [LS97, DeH09], and do not consider integrity constraints, negation and arithmetic comparisons all together. They are based on the reformulation of the query containment and equivalence problems in terms of a certain property—query simulation [LS97, DHT04] and encoding equivalence [DeH09]—over flat conjunctive queries.

In addition to the interest of validating mappings between XML schemas, being able to reason on mapping assertions with nested queries is also interesting in our previous context of mappings between flat relational schemas. For instance, consider again the example from Figure 1 (Section 1.1). We could think that mapping \( \{m_3\} \) does not ensure that the relationship which states that certain employees work for a same boss in database \( A \) is mapped into database \( B \), i.e., they could work for different bosses once mapped into database \( B \). In order to check if our suspicions are true, we could ask whether the following XML mapping assertion \( m_4 \) is inferred from the mapping. The two queries in assertion \( m_4 \) are XQueries with the same return type (assume the mapped databases allow XQueries to be posed on them); they select the bosses’ names along with the set of their employees’ names:

\[
\begin{align*}
m_4: \quad & \text{for $b$ in } //\text{Employee}_A[./\text{salary/text() } >= 1000] \\
& \text{return } <\text{result}> \\
& \quad <\text{boss}>\{b/\text{name/text()}/</\text{boss}> \\
& \quad \{\text{for } s e \text{ in } //\text{Employee}_A[./\text{salary/text() } >= 1000], \\
& \quad \quad \text{Swf in } //\text{WorksFor}_A[./\text{boss/text()} = b/\text{name/text()}] \\
& \quad \quad \text{where } s w f/\text{emp/text()} = s e/\text{name/text()} \\
& \quad \quad \text{return } <\text{emp}>\{s e/\text{name/text()}/</\text{emp}> \\
& \quad \}</\text{emps}> \\
& \quad </\text{result}>
\end{align*}
\]

\[
\begin{align*}
\text{for $b$ in } //\text{Employee}_B \\
& \text{return } <\text{result}> \\
& \quad <\text{boss}>\{b/\text{name/text()}/</\text{boss}> \\
& \quad \{\text{for } s e \text{ in } //\text{Employee}_B[./\text{boss/text()} = b/\text{emp-id/text()}] \\
& \quad \quad \text{return } <\text{emp}>\{s e/\text{name/text()}/</\text{emp}> \\
& \quad \}</\text{emps}> \\
& \quad </\text{result}>
\end{align*}
\]
The answer to the inference question would be that assertion $m_4$ is not inferred from $m_3$, which is illustrated by the following counterexample that satisfies $m_3$ but not $m_4$:

**Instance of A:**
- EmployeeA('e1', 'addr1', 1000)
- EmployeeA('e2', 'addr2', 1000)
- EmployeeA('e3', 'addr3', 1000)
- WorksForA('e2', 'e1')
- WorksForA('e3', 'e1')

**Instance of B:**
- EmployeeB(0, 'e1', null, 'cat1')
- EmployeeB(1, 'e1', null, 'cat2')
- EmployeeB(2, 'e2', 0, 'cat1')
- EmployeeB(3, 'e3', 1, 'cat1')
- CategoryB('cat1', 1000)
- CategoryB('cat2', 2000)

The counterexample shows that $m_3$ does not necessarily preserve the relationship between a boss and the set of employees that work for him, as we suspected. We can see this in the fact that while employee ‘e2’ and ‘e3’ work for the same boss in the instance of A, they work for different bosses (with the same name) in the instance of B. The conclusion would be that assertion $m_4$ was probably missing from the mapping.

The example also illustrates that mapping assertions with nested queries allow for more accurate mappings than flat formalisms, just like the nested mappings formalism [FHH+06] (see the related work chapter for a comparison of the two formalisms).
In this chapter, we introduce the basic concepts and notation that will be used throughout the thesis.

2.1 Schemas

For the most part of the thesis, we focus on relational database schemas. A relational schema is a finite 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 $P(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ 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 (we use a Datalog-style notation [AHV95]):

$$q(X) \leftarrow r_1(Y_1) \land \cdots \land r_n(Y_n) \land \neg r_{n+1}(Z_1) \land \cdots \land \neg r_m(Z_s) \land C_1 \land \cdots \land C_t$$

Each $C_i$ is a built-in literal, that is, a literal in the form of $t_1 \text{ op } t_2$, where $\text{ op } \in \{<, \leq, >, \geq, =, \neq\}$ and $t_1$ and $t_2$ are terms. A term can be either a variable or a constant. Literals $r_i(Y_i)$ and $\neg r_i(Z_i)$ 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 $q(X)$ is the head of the deductive rule, and the other literals are the body. Symbols $X$, $Y_i$ and $Z_i$ denote lists of terms. We assume deductive rules to be safe [Ull89], which means that the variables in $Z_i$, $X$ and $C_i$ are taken from $Y_1, \ldots, Y_n$, 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) [DT01] 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:

\[ r_1(\bar{Y}_1) \land ... \land r_n(\bar{Y}_n) \rightarrow C_1 \lor ... \lor C_i \]

\[ r_1(\bar{Y}_1) \land ... \land r_n(\bar{Y}_n) \land C_1 \land ... \land C_i \rightarrow \exists \bar{V}_1 r_{n+1}(\bar{U}_1) \lor ... \lor \exists \bar{V}_s r_{n+s}(\bar{U}_s) \]

Each \( \bar{V}_i \) is a list of fresh variables, and the variables in \( \bar{U}_i \) are taken from \( \bar{V}_i \) and \( \bar{Y}_1, ..., \bar{Y}_n \). 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*. We use \( \text{vars}(ic) \) to denote the non-existentially quantified variables of constraint \( ic \).

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 sometimes 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. Instances are also known as *extensional databases* (EDBs). The set of facts about the derived predicates of \( S \) that corresponds to a given instance \( D \) (i.e., the extension of the queries and views of \( S \) when evaluated on \( D \)) is the *intensional database* (IDB) of \( D \), denoted IDB\((D)\). It is worth noting that we consider the derived predicates under the exact view assumption [Len02], i.e., the extension of a view/query is exactly the set of tuples that satisfies the definition of the view/query on the database instance. Sometimes base and derived predicates/literals are referred to as EDB and IDB predicates/literals, respectively.

The answer to a query \( Q \) on an instance \( D \), denoted \( \text{AQ}(D) \), is the set of all facts about predicate \( q \) in the IDB of \( D \), i.e., \( \text{AQ}(D) = \{ q(\bar{a}) \mid q(\bar{a}) \in \text{IDB}(D) \} \), where \( \bar{a} \) denotes a list of constants.

A substitution \( \theta \) is a set of the form \( \{ X_1 \mapsto t_1, ..., X_n \mapsto t_n \} \), where \( X_1, ..., X_n \) are distinct variables, and \( t_1, ..., t_n \) are terms. The result of the application of a substitution \( \theta \) to a first-order logic expression \( E \), denoted \( E\theta \), is the expression obtained from \( E \) by simultaneously replacing each occurrence of each variable \( X_i \) by the corresponding term \( t_i \). A *unifier* of two expressions \( E_1 \) and \( E_2 \) is a substitution \( \sigma \) such that \( E_1\sigma = E_2\sigma \). Substitution \( \sigma \) is a most general unifier for \( E_1 \) and \( E_2 \) if for all other unifier \( \sigma' \) there is a substitution \( \theta \) such that \( \sigma' = \sigma \theta \) (i.e., \( \sigma' \) is the composition of \( \sigma \) and \( \theta \)).
A constraint \( ic \) is satisfied by an instance \( D \) if there is a ground substitution \( \theta \) from the variables in \( ic \) (both the existentially and the non-existentially quantified variables) to the constants in \( D \) such as \( ic\theta \) is true on \( D \), i.e., \( D \models ic\theta \). A constraint \( ic \) is violated by an instance \( D \) if \( D \) does not satisfy \( ic \).

An instance \( D \) is consistent with schema \( S \) if it does not violate any of the constraints in \( IC \).

This formalization of schemas has been taken from [FTU05]. A similar formalization, but considering only referential and implication constraints, is used in [ZO97].

### 2.2 Mappings

We write \( M = (F, A, B) \) to denote that \( M \) is a mapping between schemas \( A = (PDA, DR_A, IC_A) \) and \( B = (PDB, DR_B, IC_B) \), where \( F \) is a finite set of assertions \( \{m_1, ..., m_n\} \). Each mapping assertion \( m_i \) either takes the form \( Q^i_A = Q^i_B \), or \( Q^i_A \subseteq Q^i_B \), where \( Q^i_A \) and \( Q^i_B \) are queries over the schemas \( A \) and \( B \), respectively. Obviously, the queries must be compatible, that is, the predicates must have the same arity. We will assume that the deductive rules for these predicates are in either \( DR_A \) or \( DR_B \).

We say that schema instances \( D_A \) and \( D_B \) are consistent under mapping \( M = (F, A, B) \) if all the assertions in \( F \) are true. We say that a mapping assertion \( Q^i_A = Q^i_B \) is true if the tuples in the answer to \( Q^i_A \) on \( D_A \) are the same as the ones in the answer to \( Q^i_B \) on \( D_B \). In more formal terms, such a mapping assertion is true when the following holds: \( q^i_A(\bar{a}) \in AQ^i_A(D_A) \) if and only if \( q^i_B(\bar{a}) \in AQ^i_B(D_B) \) for each tuple of constants \( \bar{a} \), where \( q^i_A \) and \( q^i_B \) are the predicates defined by the two queries in the assertion. Similarly, a mapping assertion \( Q^i_A \subseteq Q^i_B \) is true when the tuples in the answer to \( Q^i_A \) on \( D_A \) are a subset of those in the answer to \( Q^i_B \) on \( D_B \), i.e., \( q^i_A(\bar{a}) \in AQ^i_A(D_A) \) implies \( q^i_B(\bar{a}) \in AQ^i_B(D_B) \).

This way of defining mappings is inspired by the framework for representing mappings presented in [MBDH02]. In that general framework, mapping formulas have the form \( e_1 \, op \, e_2 \), where \( e_1 \) and \( e_2 \) are expressions over the mapped schemas, and the operator \( op \) is well defined with respect to the output types of \( e_1 \) and \( e_2 \). Other similar formalisms are the GLAV [FLM99] approach and source-to-target TGDs [FKMP05]. Recall that GLAV mappings consist in assertions that have the form \( Q_A \subseteq Q_B \), where \( Q_A \) and \( Q_B \) are conjunctive queries, and TGDs consist in logic formulas of the form \( \forall \bar{X} (\phi(\bar{X}) \rightarrow \exists \bar{Y} \psi(\bar{X}, \bar{Y})) \), where \( \phi(\bar{X}) \) and \( \psi(\bar{X}, \bar{Y}) \) are conjunctions of relational atoms. Note that, with respect to GLAV and TGDs, we allow the use of a more expressive class of queries.
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 of $S$ in which $Q$ has a non-empty answer.

The CQC (Constructive Query Containment) method [FTU05], 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 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 two major VIPs are the *Negation VIP*, which is applied when all built-in literals in the schema are $=$ or $\neq$ comparisons, and the *Order VIP*, which is applied when the schema contains order comparisons. The Negation VIP works as follows: a given variable $X$ can be instantiated with one of the constants already used (those in the schema definition and those provided by previous applications of the VIP) or with a fresh constant. The Order VIP also gives the choice of reusing a constant or using a fresh one. However, in the latter case, the fresh constant may be either greater or lower than all those previously used, or it may fall between two previously used constants. The Order VIP comes in two flavors: *Dense Order VIP* and *Discrete Order VIP*; the main difference is that the Discrete Order VIP must ensure that when a fresh constant is provided that falls between two previously used constants there has to be enough room in that range for an additional integer value.

As an example, let us assume that the CQC method must instantiate the relational atom $R(X, Y)$ using the Negation VIP, and that the set of used constants is empty. The possible instantiations would be $R(0, 0)$ and $R(0, 1)$. As variable $X$ is instantiated first, the only option is to use a fresh constant, e.g., the constant 0. Thus, there are two possibilities for instantiating variable $Y$: using the constant 0 again, or using a fresh constant, e.g., the constant 1.

Intuitively, the CQC method works in two phases. The first phase is *query satisfaction*. In this phase, the CQC method generates an initial instance that satisfies the definition of the tested
query (i.e., the goal), but that is not necessarily consistent with the schema $S$. The second phase is **integrity maintenance**. In this phase, the CQC method tries to repair the inconsistent instance constructed by the previous phase by means of inserting new tuples into the database. If the integrity maintenance phase reaches a point when some violation cannot be repaired by the insertion of new tuples, then the CQC method has to reconsider the previous decisions (e.g., try another instantiation for the tuples previously inserted from those provided by the VIPs).

The fact that at certain points the CQC method has to make decisions causes the solution space the CQC method explores to be a tree. This tree is 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 [FTU05], 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.

For a more detailed discussion on the CQC method see [FTU05].
Checking Desirable Properties of Mappings

Our approach to mapping validation consists in checking whether mappings meet certain desirable properties. We identify three important properties already proposed in the literature—mapping satisfiability [ALM09], mapping inference and query answerability [MBDH02]—and propose a new one—mapping losslessness. We show how to perform such validation by means of its reformulation as a query satisfiability problem over a database schema.

We show that the proposed reformulation in terms of query satisfiability does not increase the complexity of the problem.

We finally perform a series of experiments to show the behavior of our approach. The experiments are carried out using the CQC method [FTU05] as implemented in our Schema Validation Tool (SVT) [TFU+04].

3.1 Desirable Properties and Their Reformulation in Terms of Query Satisfiability

In this section, we firstly formalize the desirable properties and, secondly, explain how the fulfillment of each property should be expressed in terms of query satisfiability.

Recall that a query $Q$ is satisfiable over a schema $S$ if there is any consistent instance of $S$ in which the answer to $Q$ is not empty [DTU96, HMSS01, ZO97]. We define schema $S$ in such a way that mapped schemas $A$ and $B$ and mapping $M$ are considered together. We assume that the two original schemas have different relation names; otherwise, relations can simply be renamed.

In general, schema $S$ is built by grouping the deductive rules and integrity constraints of the two schemas, and then adding new constraints to make the relationship stated by the mapping explicit. Formally, this is defined as
\[ S = (DR_A \cup DR_B, IC_A \cup IC_B \cup IC_M), \]

where \( IC_M \) is the set of additional constraints that enforces the mapping assertions.

For each mapping assertion of the form \( Q^i_A = Q^i_B \), the following two constraints are needed in \( IC_M \):

\[
q^i_A(\bar{X}) \rightarrow q^i_B(\bar{X}), \\
q^i_B(\bar{X}) \rightarrow q^i_A(\bar{X}).
\]

These constraints state that the two queries in the assertion must give the same answer, that is, both \( Q^i_A \subseteq Q^i_B \) and \( Q^i_B \subseteq Q^i_A \) must be true.

For the assertions of the form \( Q^i_A \subseteq Q^i_B \), only the first constraint is required:

\[
q^i_A(\bar{X}) \rightarrow q^i_B(\bar{X}).
\]

Having defined schema \( S \), we define a query \( Q_{\text{prop}} \) for each desirable property, such that \( Q_{\text{prop}} \) will be satisfiable over \( S \) if and only if the property holds.

Below, we describe each desirable property in detail and its specific reformulation in terms of query satisifiability.

### 3.1.1 Mapping Satisfiability

As stated in [Len02], when constraints are considered in the global schema in a data integration context, it may be the case that the data retrieved from the sources cannot be reconciled in the global schema in such a way that both the constraints of the global schema and the mapping are satisfied.

In general, whenever we have a mapping between schemas that have constraints, there may be incompatibilities between the constraints and the mapping, or even between the mapping assertions. Therefore, checking whether there is at least one case in which the mapping and the constraints are satisfied simultaneously is clearly a validation task that should be performed, and this is precisely the aim of this property.

**Definition 3.1.** We consider a mapping \( M = (F, A, B) \) to be *satisfiable* if there are at least two non-empty instances \( D_A, D_B \) such that they are consistent with schemas \( A \) and \( B \), respectively, and are also consistent under \( M \).
Note that the above definition explicitly avoids the trivial case in which $DA$ and $DB$ are both empty sets. However, the assertions in $F$ can still be satisfied trivially. We say that an assertion $QA_i = QB_i$ is satisfied trivially when both $AQA_i(DA)$ and $AQB_i(DB)$ are empty sets. An assertion $QA_i \subseteq QB_i$ is satisfied trivially when $AQA_i(DA)$ is the empty set. Therefore, in order to really validate the satisfiability of the mapping, we should ask whether all its assertions can be satisfied non-trivially, or at least one of them.

**Definition 3.2.** A mapping $M = (F, A, B)$ is strongly satisfiable if all assertions in $F$ are satisfied non-trivially. The mapping is weakly satisfiable if at least one assertion in $F$ is satisfied non-trivially. □

**Example 3.1.** Consider the schemas and the mapping shown graphically in Figure 3.1. The formalization of the mapped schemas is the following:

Schema $A = (DR_A, IC_A)$, where constraints $IC_A = \{$
\begin{align*}
\text{category}(C, S) & \rightarrow S \leq 100, \\
\text{employee}(E, C, H) & \rightarrow \exists S \text{ category}(C, S)
\end{align*}
\}
and deductive rules $DR_A = \{$
\begin{align*}
q^1_A(E, H) & \leftarrow \text{employee}(E, C, H) \land H > 10, \\
q^3_A(E, S) & \leftarrow \text{employee}(E, C, H) \land \text{category}(C, S)
\end{align*}
\}
Schema $B = (DR_B, IC_B)$, where
\[ IC_B = \{ \]
\[ \text{happy-emp}(E, H) \land \text{emp}(E, S) \rightarrow S > 200, \]
\[ \text{happy-emp}(E, H) \rightarrow \exists S \text{emp}(E, S) \} \] and
deductive rules $DR_B = \{$
\[ q^B_1(E, H) \leftarrow \text{happy-emp}(E, H), \]
\[ q^B_2(E, S) \leftarrow \text{emp}(E, S) \} \]

The formalization of the mapping is as follows:
\[ M = (F, A, B), \text{ where } \]
mapping assertions $F = \{ Q^A_1 = Q^B_1, Q^A_2 = Q^B_2 \}$

The deductive rules for the queries in $F$ are those defined in the schemas.

Schema $A$ has two tables: $\text{employee}(\text{emp-id, category, happiness-degree})$ and $\text{category}(\text{cat-id, salary})$. The $\text{employee}$ table is related to the $\text{category}$ table through a referential constraint from $\text{employee.category}$ to $\text{category.cat-id}$. The $\text{category}$ table has a constraint on salaries, which must not exceed 100.

Schema $B$ has also two tables: $\text{emp}(\text{id, salary})$ and $\text{happy-emp}(\text{emp-id, happiness-degree})$. It has a referential constraint from $\text{happy-emp.emp-id}$ to $\text{emp.id}$, and a constraint that states all happy employees must have a salary of more than 200.

Mapping $M$ links those instances of $A$ and $B$, say $D_A$ and $D_B$, in which (1) the employees in $D_A$ with a happiness degree greater than 10 are the same as the happy-emp in $D_B$, and (2) the employees in $D_A$ are the same and have the same salary as the emps in $D_B$.

We can see that the first mapping assertion, i.e., $Q^A_1 = Q^B_1$ (where $q^A_1(E, H) \leftarrow \text{employee}(E, C, H) \land H > 10$ and $q^B_1(E, H) \leftarrow \text{happy-emp}(E, H)$), can only be satisfied trivially. Mapping $M$ is thus not strongly satisfiable. There are two reasons for that. The first reason is that all happy-emps in schema $B$ must have a salary of over 200 while all employees in schema $A$, regardless of their happiness degree, must have a maximum salary of 100. The second reason is that mapping assertion $Q^A_2 = Q^B_2$ (where $q^A_2(E, S) \leftarrow \text{employee}(E, C, H) \land \text{category}(C, S)$ and $q^B_2(E, S) \leftarrow \text{emp}(E, S)$) dictates that all employees should have the same salary in both sides of the mapping.

In contrast, the second mapping assertion is non-trivially satisfiable, which means that $M$ is weakly satisfiable. The reason is that there may be employees in $A$ with a happiness degree of 10 or lower and emps in $B$ that are not happy-emps.

28
It should also be noted that if we removed the second assertion from the mapping and just kept the first one, the resulting mapping would be strongly satisfiable. For the sake of an example, instances

\[ D_A = \{ \text{employee}(\text{joan}, \text{sales}, 20), \text{category}(\text{sales}, 30) \} \]

\[ D_B = \{ \text{emp}(\text{joan}, 300), \text{happy-emp}(\text{joan}, 20) \} \]

are consistent and satisfy the first mapping assertion.

The mapping would also become strongly satisfiable if we removed the first assertion and kept the second. That is an example of how the satisfiability of a mapping assertion may be affected by the rest of assertions in the mapping. □

The mapping satisfiability property for a given a mapping \( M = (F, A, B) \) can be reformulated in terms of query satisfiability as follows.

First, we build the schema that groups schemas \( A \) and \( B \) and mapping \( M \):

\[ S = (DR_A \cup DR_B, IC_A \cup IC_B \cup IC_M) \]

The intuition is that from a consistent instance of \( S \) we can get one consistent instance of \( A \) and one consistent instance of \( B \) such that they are also consistent under \( M \).

Then, we define the distinguished query in order to check whether it is satisfiable over \( S \). As there are two types of satisfiability—strong and weak—we need to define a query for either case.

Assuming that \( F = \{ f_1, ..., f_n \} \) and that we want to check strong satisfiability, we define the \textit{strongsat} Boolean query as follows:

\[ \text{strongsat} \leftarrow q_{A_1}(X_1) \land ... \land q_{A_n}(X_n) \]

where the terms in \( X_1, ..., X_n \) are distinct variables. Intuitively, each \( Q^A_i \) query in the mapping must have a non-empty answer in order to satisfy this query. The same applies to \( Q^B_i \) queries because of the constraints in \( IC_M \) that enforce the mapping assertions.

Similarly, in the case of weak satisfiability, we would define the \textit{weaksat} Boolean query as follows:

\[ \text{weaksat} \leftarrow q_{A_1}(X_1) \lor ... \lor q_{A_n}(X_n) \]

However, because the bodies of the rules must be conjunctions of literals, the query should be defined using the following deductive rules:
Proposition 3.1. Boolean query strong_sat/weak_sat is satisfiable over schema S if and only if mapping M is strongly/weakly satisfiable.

Figure 3.2 shows the schema we obtain when Example 3.1 is expressed in terms of query satisfiability. Note that the deductive rule that defines the distinguished query strong_sat has been added to the resulting schema S.

3.1.2 Mapping Inference

The mapping inference property was identified in [MBDH02] as an important property of mappings. It consists in checking whether a mapping entails a given mapping assertion, that is, whether or not the given assertion adds new mapping information. One application of the property would be that of checking whether an assertion of the mapping is redundant, that is, whether it is entailed by the other assertions. Another application would be that of checking the equivalence of two different mappings. We can say that two mappings are equivalent if the assertions in the first mapping entail the assertions in the second, and vice versa.

The results presented in [MBDH02] are in the context of mapping scenarios in which assertions are equalities of conjunctive queries and schemas do not have integrity constraints. They show that checking the property in this setting involves finding a maximally contained rewriting and checking two equivalences of conjunctive queries. Here, we consider a broader class of assertions and queries and the presence of constraints in the schemas (see Chapter 2).
**Definition 3.3.** (see [MBDH02]) Let a mapping assertion $f$ be defined between schemas $A$ and $B$. Assertion $f$ is inferred from a mapping $M = (F, A, B)$ if all pair of instances of $A$ and $B$ that is consistent under $M$ also satisfies assertion $f$. □

**Example 3.2.** Consider again the schemas from Example 3.1, but without the salary constraints:

Schema $A = (DRA, IC_A)$, where

constraints $IC_A = \{$

employee($E, C, H$) $\rightarrow \exists S \text{ category}(C, S)$ $\}$ and
deductive rules $DR_A = \{$

$q^1_1(E, H)$ $\leftarrow$ employee($E, C, H$) $\land H > 10$,
$q^2_2(E, S)$ $\leftarrow$ employee($E, C, H$) $\land \text{ category}(C, S)$ $\}$

Schema $B = (DR_B, IC_B)$, where

constraints $IC_B = \{$

happy-emp($E, H$) $\rightarrow \exists S \text{ emp}(E, S)$ $\}$ and
deductive rules $DR_B = \{$

$q^1_3(E, H)$ $\leftarrow$ happy-emp($E, H$),
$q^2_4(E, S)$ $\leftarrow$ emp($E, S$) $\}$

Consider a new mapping:

$M_2 = (F_2, B, A)$,
where $F_2$ contains just the mapping assertion $Q^B_2 = Q^A_2$

and queries $Q^B_2, Q^A_2$ are those already defined in the schemas.

Let $f_1$ be the mapping assertion $Q_1 \subseteq Q_2$, where $Q_1$ and $Q_2$ are queries defined over schemas $B$ and $A$, respectively:

$q_1(E) \leftarrow$ happy-emp($E, H$)
$q_2(E) \leftarrow$ employee($E, C, H$)

The referential constraint in schema $B$ guarantees that all happy-emps are also emps, and if the mapping assertion in $M_2$ holds, that means they are also employees in the corresponding instance of schema $A$. Thus, assertion $f_1$ is true, namely, the employees’ identifiers in the happy-emp table are a subset of those in the employee table. Therefore, mapping $M_2$ entails assertion $f_1$.

Now, let $f_2$ be the assertion $Q_3 \subseteq Q_4$, where $Q_3$ and $Q_4$ are defined as follows:

$q_3(E, H) \leftarrow$ happy-emp($E, H$)
$q_4(E, H) \leftarrow$ employee($E, C, H$)
We can see that mapping $M_2$ does not entail assertion $f_2$. The difference is that we are not just projecting the employee’s identifier as before, but also the happiness degree. In addition, given that the assertion from $M_2$ disregards the happiness degree, we can build a counterexample to show that the entailment of $f_2$ does not hold. This counterexample would consist of a pair of instances, say $D_A$ and $D_B$, that would satisfy the mapping assertion from $M_2$ but that would not satisfy $f_2$, like, for instance, the following ones:

$$D_A = \{ \text{employee}(0, 0, 5), \text{category}(0, 50) \}$$
$$D_B = \{ \text{emp}(0, 50), \text{happy-emp}(0, 10) \}$$

It is not difficult to prove that the assertion from mapping $M_2$ holds over $D_A$ and $D_B$:

$$AQ^A_{\text{B}}(D_B) = \{ q^A_{\text{B}}(0, 50) \}$$
$$AQ^B_{\text{A}}(D_A) = \{ q^B_{\text{A}}(0, 50) \}$$

However, $f_2$ does not hold:

$$AQ(\bar{D}_B) = \{ q_3(0, 10) \}$$
$$AQ(\bar{D}_A) = \{ q_4(0, 5) \}$$

Expressing the mapping inference property in terms of query satisfiability is best done by checking the negation of the property (i.e., the lack of inference) instead of checking the property directly. The negated property states that a certain assertion $f$ is not inferred from a mapping $M = (F, A, B)$ if there are two schema instances $D_A, D_B$ that are consistent under $M$ and do not satisfy $f$. Therefore, the distinguished query to be check for satisfiability must state the negation of $f$. When $f$ has the form $Q_a = Q_b$, we define the map_inf Boolean query by means of the following two deductive rules:

$$\text{map_inf} \leftarrow q_a(X) \land \neg q_b(X)$$
$$\text{map_inf} \leftarrow q_b(X) \land \neg q_a(X)$$

Otherwise, when $f$ has the form $Q_a \subseteq Q_b$, only the first deductive rule is needed:

$$\text{map_inf} \leftarrow q_a(X) \land \neg q_b(X)$$

We define the schema $S$ in the usual way: by putting the deductive rules and constraints from schemas $A$ and $B$ together, and by considering additional constraints to enforce the mapping assertions. Formally,

$$S = (DR_A \cup DR_B, IC_A \cup IC_B \cup IC_M)$$
Proposition 3.2. Boolean query $\text{map}_\text{inf}$ is satisfiable over schema $S$ if and only if mapping assertion $f$ is not inferred from mapping $M$.

Proof. Let us assume that $f$ takes the form $Q_a = Q_b$ and that $\text{map}_\text{inf}$ is satisfiable over $S$. Then, there is a consistent instance of $S$ for which $\text{map}_\text{inf}$ is true. It follows that there are two consistent instances $D_A$, $D_B$ of schemas $A$ and $B$, respectively, such that they are also consistent under mapping $M$. Given that $\text{map}_\text{inf}$ is true, we can infer that there is a tuple that either belongs to the answer to $Q_a$ but that does not belong to the answer to $Q_b$, or that belongs to the answer to $Q_b$ but not to the answer to $Q_a$. Therefore, this pair of instances does not satisfy $f$.

In contrast, let us assume that there are two consistent instances $D_A$, $D_B$ that are also consistent under mapping $M$, but that do not satisfy assertion $f$. It follows that there is a consistent instance of $S$ for which assertion $f$ does not hold. That means $Q_a \neq Q_b$, i.e., either $Q_a \not\subseteq Q_b$ or $Q_b \not\subseteq Q_a$. We can therefore conclude that $\text{map}_\text{inf}$ is true over this instance of $S$, and so, that $\text{map}_\text{inf}$ is satisfiable over $S$.

The proof for the case in which $f$ takes the form $Q_a \subseteq Q_b$ can be directly obtained from this.

Figure 3.3 shows the schema that results from the reformulation of Example 3.2 in terms of query satisfiability, for the case of testing whether assertion $f_1$ is inferred from mapping $M_2$. Note the presence of the deductive rules that define the queries of assertion $f_1$ ($Q_1$ and $Q_2$) and the rule that defines the distinguished query $\text{map}_\text{inf}$.

3.1.3 Query Answerability

We consider now the query answerability property, which was also described in [MBDH02] as an important property of mappings. The reasoning behind this property is that a mapping that is
partial or incomplete may nevertheless be successfully used for certain tasks. These tasks will be represented by means of certain queries. The property checks whether the mapping enables the answering of these queries over the schemas being mapped. While the previous two properties are intended to validate the mapping without considering its context, this property validates the mapping with regard to the use for which it has been designed.

As with mapping inference, the results presented in [MBDH02] are in the context of equalities between conjunctive queries without constraints on the schemas. They show that the property can be checked by means of the existence of an equivalent rewriting. As in the previous case, we consider a broader class of assertions and queries and the presence of constraints in the schemas.

The intuition behind the property is that, given a mapping \( M = (F, A, B) \) and a query \( Q \) defined over schema \( A \), it checks whether every consistent instance of \( B \) uniquely determines the answer to \( Q \) over \( A \). In other words, if the property holds for a query \( Q \), and \( D_A, D_B \) are two instances consistent under \( M \), we may compute the exact answer to \( Q \) over \( D_A \) using only the tuples in \( D_B \).

**Definition 3.4.** (see [MBDH02]) Let \( Q \) be a query over schema \( A \). Mapping \( M = (F, A, B) \) enables query answering of \( Q \) if for all consistent instance \( D_B \) of schema \( B \), \( A_Q(D_A) = A_Q(D_A') \) for every pair \( D_A, D_A' \) of consistent instances of schema \( A \) that are also consistent under \( M \), we may compute the exact answer to \( Q \) over \( D_A \) using only the tuples in \( D_B \).

**Example 3.3.** Consider again the schemas from the previous example:

**Schema A** = \( (DR_A, IC_A) \), where

constraints \( IC_A = \{ \)

\[ \text{employee}(E, C, H) \rightarrow \exists S \text{category}(C, S) \] \} and  
deductive rules \( DR_A = \{ \)

\[ q_A^1(E, H) \leftarrow \text{employee}(E, C, H) \land H > 10, \]

\[ q_A^2(E, S) \leftarrow \text{employee}(E, C, H) \land \text{category}(C, S) \] \} 

**Schema B** = \( (DR_B, IC_B) \), where

constraints \( IC_B = \{ \)

\[ \text{happy-emp}(E, H) \rightarrow \exists S \text{emp}(E, S) \] \} and  
deductive rules \( DR_B = \{ \)

\[ q_B^1(E, H) \leftarrow \text{happy-emp}(E, H), \]

\[ q_B^2(E, S) \leftarrow \text{emp}(E, S) \] \} 

Consider also the mapping \( M \) from Example 3.1:

\[ M = (F, A, B), \]

mapping assertions \( F = \{ Q_A^1 = Q_B^1, \ Q_A^2 = Q_B^2 \} \)
and the following query $Q$ defined over schema $A$:

$$q(E) \leftarrow employee(E, C, H) \land H > 5$$

We can see that mapping $M$ does not enable the answering of query $Q$. The mapping only deals with those employees in schema $A$ who have a happiness degree greater than 10, while the evaluation of query $Q$ must also have access to the employees with a happiness degree of between 5 and 10. Thus, we can build a counterexample that will consist of three consistent instances: one instance $D_B$ of schema $B$ and two instances $D_A$, $D_A'$ of schema $A$. Instances $D_A$, $D_A'$ will be consistent under mapping $M$ with $D_B$, but the answer to $Q$ will not be the same in both instances, i.e., $AQ(D_A) \neq AQ(D_A')$. These criteria are satisfied, for example, by the following instances:

$$D_B = \{ \text{emp}(0, 150), \text{emp}(1, 200), \text{happy-emp}(1, 15) \}$$
$$D_A = \{ \text{employee}(0, 0, 6), \text{employee}(1, 1, 15), \text{category}(0, 150), \text{category}(1, 200) \}$$
$$D_A' = \{ \text{employee}(0, 0, 4), \text{employee}(1, 1, 15), \text{category}(0, 150), \text{category}(1, 120) \}$$

We can easily state that this is indeed a counterexample because

$$AQ(D_A) = \{ q(0), q(1) \}$$

but

$$AQ(D_A') = \{ q(1) \}$$

The previous example illustrates that query answerability is also easier to check by means of its negation. Therefore, as two instances of schema $A$ must be found in order to build a counterexample, we must extend the definition of schema $S$ as follows:

$$S = (DR_A \cup DR_A' \cup DR_B, IC_A \cup IC_A' \cup IC_B \cup IC_M \cup IC_M')$$

where $A' = (DR_A', IC_A')$ is a copy of schema $A = (DR_A, IC_A)$ in which we rename all the predicates (e.g., $q$ is renamed $q'$) and, similarly, $M' = (F', A', B)$ is a copy of mapping $M = (F, A, B)$ in which we rename the predicates in the mapping assertions that are predicates from schema $A$. The intuition is that having two copies of schema $A$ allows us to get from one single instance of schema $S$ the two instances of $A$ that are necessary for building the counterexample.

We will then check the satisfiability of the Boolean query $q\_answer$, which we define using the following deductive rule:

$$q\_answer \leftarrow q(\bar{X}) \land q'(\bar{X})$$
where \( Q \) is the parameter query of the property (defined over schema \( A \)) and \( Q' \) is its renamed version over schema \( A'. \) Intuitively, query \( q_{\text{answer}} \) can only be satisfied by an instance of \( S \) from which we can get two instances of \( A \) that do not have the same answer for query \( Q, \) i.e., there is a tuple \( q(\bar{a}) \) in the answer to \( Q \) over one of the instances that is not present in the answer to \( Q \) over the other instance.

**Proposition 3.3.** Boolean query \( q_{\text{answer}} \) is satisfiable over schema \( S \) if and only if mapping \( M \) does not enable query answering of \( Q. \)

**Proof.** Let us assume that \( q_{\text{answer}} \) is satisfiable over \( S. \) That means there is a consistent instance of \( S \) in which \( q_{\text{answer}} \) is true. It follows that there is an instance \( D_B \) of \( B, \) an instance \( D_A \) of \( A, \) and an instance \( D_{A'} \) of \( A', \) such that they are all consistent and \( D_A \) and \( D_{A'} \) are also both consistent with \( D_B \) under mappings \( M \) and \( M', \) respectively. As \( q_{\text{answer}} \) is true, we can infer that there is a tuple \( q(\bar{a}) \), such that \( q(\bar{a}) \in A_{\varnothing}(D_A) \) and \( q'(\bar{a}) \not\in A_{\varnothing}(D_{A'}). \) Given that schema \( A' \) is in fact a copy of schema \( A, \) we can conclude that for each instance of schema \( A' \) there is an identical one that conforms to schema \( A. \) Thus, \( D_{A'} \) can be seen as an instance of \( A, \) in such a way that if it was previously consistent with \( D_B \) under \( M', \) it is now also consistent with \( D_B \) under \( M \) and, for all previous \( q'(\bar{a}) \) in the answer to \( Q', \) there is now a \( q(\bar{a}) \) in the answer to \( Q. \) Therefore, we have found two instances \( D_A, D_{A'} \) of schema \( A \) such that \( A_{\varnothing}(D_A) \not\subset A_{\varnothing}(D_{A'}) \) and are both consistent under mapping \( M \) with a given instance \( D_B \) of schema \( B. \) We can thus conclude that \( M \) does not enable query answering of \( Q. \)

The other direction can easily be proved by inverting the line of reasoning. ■
Figure 3.4 shows the schema that we get when we reformulate Example 3.3 in terms of query satisfiability. Note that the deductive rule that defines the distinguished query $q_{\text{answer}}$ has been added to the resulting schema $S$, and that the rules corresponding to queries $Q$, $Q'$ have been added to $DR_A$ and $DR_A'$, respectively.

### 3.1.4 Mapping Losslessness

As we have seen, query answerability determines whether two mapped schemas are equivalent with respect to a given query in that it would obtain the same answer in both cases. However, in certain contexts this may be too restrictive. Consider data integration [Len02], for instance, and assume that for security reasons it must be known whether any sensitive local data is exposed by the integrator system. Clearly, in such a situation, the query intended to retrieve such sensitive data from a source is not expected to obtain the exact answer that would be obtained if the query were directly executed over the global schema. Therefore, such a query is not answerable under the terms specified above. Nevertheless, sensitive local data are in fact exposed if the query can be computed over the global schema. Thus, a new property that is able to deal with this is needed.

In fact, when a mapping has assertions of the form $Q_i^A \subseteq Q^B$, checking query answerability does not always provide the designer with useful information. Let us illustrate this with an example.

**Example 3.4.** Consider the schemas used in the previous two examples:

**Schema A** = $(DR_A, IC_A)$, where

- **constraints** $IC_A = \{ \text{employee}(E, C, H) \rightarrow \exists S \text{ category}(C, S) \}$ and
- **deductive rules** $DR_A = \{
    q^1_A(E, H) \leftarrow \text{employee}(E, C, H) \land H > 10,
    q^2_A(S, E) \leftarrow \text{employee}(E, C, H) \land \text{category}(C, S)
  \}$

**Schema B** = $(DR_B, IC_B)$, where

- **constraints** $IC_B = \{ \text{happy-emp}(E, H) \rightarrow \exists S \text{ emp}(E, S) \}$ and
- **deductive rules** $DR_B = \{
    q^1_B(E, H) \leftarrow \text{happy-emp}(E, H),
    q^2_B(S, E) \leftarrow \text{emp}(E, S)
  \}$

Consider also the following query $Q$:

$q(E) \leftarrow \text{employee}(E, C, H)$

It is not difficult to see that the mapping $M$ from Example 3.1 enables answering of query $Q$. 
\[ M = (F, A, B), \text{ where mapping assertions } F = \{ Q_A^1 = Q_B^1, \ Q_A^2 = Q_B^2 \} \]

The parameter query \( Q \) selects all employees in schema \( A \); and mapping \( M \) states that the \textit{employees} in \( A \) are the same as the \textit{emps} in \( B \). Thus, when an instance of schema \( B \) is given, the extension of the \textit{employee} table in schema \( A \) becomes uniquely determined, as well as the answer to \( Q \).

Consider now the following mapping:

\[ M_3 = (F_3, A, B), \text{ where mapping assertions } F_3 = \{ Q_A^1 \subseteq Q_B^1, \ Q_A^2 \subseteq Q_B^2 \} \]

Note that mapping \( M_3 \) is similar to the previous mapping \( M \), but the = operator has been replaced with the \( \subseteq \) operator.

If we consider mapping \( M_3 \), the extension of the \textit{employee} table is not uniquely determined by a given instance of \( B \); in fact, it can be any subset of the tuples in table \textit{emp} in the instance of \( B \).

For example, let \( D_B \) be an instance of schema \( B \) such that

\[ D_B = \{ \text{emp}(0, 70), \text{emp}(1, 40) \} \]

and let \( D_A \) and \( D_A' \) be two instances of schema \( A \) such that

\[ D_A = \{ \text{employee}(0, 0, 5), \text{category}(0, 70) \} \]
\[ D_A' = \{ \text{employee}(1, 0, 5), \text{category}(0, 40) \} \]

We have come up with a counterexample and may thus conclude that mapping \( M_3 \) does not enable query answering of \( Q \). □

The above example shows that when a mapping has assertions of the form \( Q_A^i \subseteq Q_B^i \), an instance of schema \( B \) does not generally determine the answer to a query \( Q \) defined over schema \( A \). This is because, over a given instance of \( B \), there is just one possible answer for each query \( Q_B^i, \ldots, Q_B^n \) in the mapping. However, due to the \( \subseteq \) operator, there is more than one possible answer to the queries \( Q_A^i, \ldots, Q_A^n \). A similar result would be obtained if \( Q \) were defined over schema \( B \). Thus, query answerability does not generally hold for mappings of this kind. Intuitively, we can say that the reason is that query answerability holds only when we are able to compute the exact answer for \( Q \) over an instance \( D_A \) using only the tuples in the corresponding mapped instance \( D_B \). However, if any of the mapping assertions has the \( \subseteq \) operator, we cannot
know, just by looking at $D_B$, which tuples are also in $D_A$, and we are therefore unable to compute an exact answer for $Q$.

To deal with that, we defined the mapping losslessness property, which, informally speaking, checks whether all pieces of data that are needed to answer a given query $Q$ over schema $A$ are captured by mapping $M = (F, A, B)$, in such a way that they have a counterpart in schema $B$. In other words, if $D_A$ and $D_B$ are two instances consistent under mapping $M$ and the property holds for query $Q$, an answer to $Q$ may be computed using the tuples in $D_B$, although not necessarily the same answer we would obtain evaluating $Q$ directly over $D_A$.

**Definition 3.5.** Let $Q$ be a query defined over schema $A$. We say that mapping $M = (F, A, B)$ is lossless with respect to $Q$ if for all pair of consistent instances $D_A, D_A'$ of schema $A$, both the existence of an instance $D_B$ that is consistent under $M$ with both instances of $A$ and the fact that $A_Q'(D_A) = A_Q'(D_A')$ for each query $Q^i$ in the mapping imply that $A_Q(D_A) = A_Q(D_A')$. □

**Example 3.5.** Consider once again the schemas $A$ and $B$ used in the previous examples, and the mapping $M_3$ and the query $Q$ from Example 3.4:

- **Schema $A = (D_{RA}, IC_A)$, where**
  constraints $IC_A = \{$
  employee($E, C, H$) $\rightarrow \exists S$ category($C, S$) $\}$ and deductive rules $DR_A =$ $\{$
  $q_A^1(E, H) \leftarrow employee(E, C, H) \land H > 10$,
  $q_A^2(E, S) \leftarrow employee(E, C, H) \land category(C, S)$ $\}$

- **Schema $B = (D_{RB}, IC_B)$, where**
  constraints $IC_B = \{$
  happy-emp($E, H$) $\rightarrow \exists S$ emp($E, S$) $\}$ and deductive rules $DR_B =$ $\{$
  $q_B^1(E, H) \leftarrow happy-emp(E, H)$,$
  q_B^2(E, S) \leftarrow emp(E, S)$ $\}$

- **Mapping $M_3 = (F_3, A, B)$, where**
  mapping assertions $F_3 =$ $\{ Q^{i_1}_1 \subseteq Q^{i_2}_1, \ Q^{i_2}_2 \subseteq Q^{i_2}_2 \}$

- **Query $Q = \{ q(E) \leftarrow employee(E, C, H) \}$

We saw that the query answerability property does not hold for mapping $M_3$. Let us now check the mapping losslessness property.

Let us assume that we have two consistent instances $D_A, D_A'$ of schema $A$, and a consistent instance $D_B$ of schema $B$ that is consistent under $M$ with both instances of $A$. Let us also assume that the answers to $Q^{i_2}$ and $Q^{i_1}$ are exactly the same over $D_A$ and $D_A'$. Let us now suppose that $Q$
obtains $q(0)$ over $D_A$ but not over $D_A'$. According to the definition of $Q$, it follows that $D_A$ contains at least one employee tuple, say employee(0, 0, 12), which $D_A'$ does not contain. Since $D_A$ is consistent with the integrity constraints, it must also contain its corresponding category tuple, say category(0, 20). Therefore, according to the definition of $Q^A_2$, the answer $q^A_2(0, 20)$ would be obtained over $D_A$ but not over $D_A'$. This clearly contradicts our initial assumption. Mapping $M_3$ is thus lossless with respect to $Q$. □

To reformulate the mapping losslessness property in terms of query satisfiability, we define the schema $S$ in a similar way as we did for query answerability:

$$S = (D_{RA} \cup D_{RA}', I_{CA} \cup I_{CA}', I_{CB} \cup I_{CM} \cup I_{CL})$$

where schema $A' = (D_{RA}', I_{CA}')$ is a copy of schema $A = (D_{RA}, I_{CA})$ in which predicates are renamed, and $IC_M$ is the set of constraints that enforce the assertions in mapping $M = (F, A, B)$. We use $IC_l$ to denote the set of constraints that force $A$ and $A'$ to share the same answers for the $Q^i$ queries in the mapping:

$$IC = \{q^i, q^i'(X\bar{1}), q^i, q^i'(X\bar{i}) \}
\ldots
q^n, q^n'(X\bar{n}), q^n, q^n'(X\bar{n}) \}

Let $Q$ be the query over schema $A$ to be checked for satisfiability, and let $Q'$ be the copy of $Q$ over schema $A'$. We define the Boolean query $map\_loss$ as follows:

$$map\_loss \leftarrow q(X) \land \neg q'(\bar{X})$$

The intuition is that query $map\_loss$ can only be satisfied by an instance of $S$ from which we can get two instances of $A$ that have the same answers for the $Q^i$ queries (because of $IC_l$) but not for the query $Q$ (because of the deductive rule). We are checking thus for the existence of a counterexample.

**Proposition 3.4.** Boolean query $map\_loss$ is satisfiable over schema $S$ if and only if mapping $M$ is not lossless with respect to query $Q$.

**Proof.** Let us assume that $map\_loss$ is satisfiable over $S$. Hence, there is an instance of $S$ in which $map\_loss$ is true. This means that the answer to $Q$ has a tuple that is not in the answer to $Q'$. Based on the instance of $S$, we can thus build an instance $D_A$ for schema $A$, an instance $D_A'$ for schema $A'$, and an instance $D_B$ for schema $B$. Given that $A$ and $A'$ are in fact the same schema with
are consistent under mapping

instances of schema

is also an instance of schema

if $M$ enables query answering of $Q$.

$F = \{f_1, \ldots, f_n\}$ and $f_i$ is $Q_i$, is $Q_i$, for $1 \leq i \leq n$. Mapping $M$ is lossless with respect to $Q$ if and only if $M$ enables query answering of $Q$. 

**Proposition 3.5.** Let $Q$ be a query over schema $A$, and let $M = (F, A, B)$ be a mapping where $F = \{f_1, \ldots, f_n\}$ and $f_i$ is $Q_i$, is $Q_i$, for $1 \leq i \leq n$. Mapping $M$ is lossless with respect to $Q$ if and only
**Proof.** Let us assume that mapping $M$ is lossless with respect to query $Q$, and let us also assume that mapping $M$ does not enable answering of query $Q$. By the negation of query answerability, there is an instance $D_B$ of schema $B$ and two instances $D_A$, $D_A'$ of schema $A$ such that $D_A$ and $D_A'$ are both consistent under $M$ with $D_B$ but $AQ(D_A) \neq AQ(D_A')$. Given that all mapping assertions are like $Q^i_A = Q^i_B$, $AQ^i(D_A) = AQ^i(D_A')$ is true for $1 \leq i \leq n$. Hence, instances $D_A$, $D_A'$ and $D_B$ form a counterexample for mapping losslessness and a contradiction is thus reached.

Let us now however assume that mapping $M$ enables answering of query $Q$, and let us also assume that $M$ is not lossless with respect to $Q$. By the negation of losslessness, there are two instances $D_A$, $D_A'$ of $A$ such that $AQ^i(D_A) = AQ^i(D_A')$ for $1 \leq i \leq n$, and there is also an instance $D_B$ of $B$ that is consistent under $M$ with both instances of $A$. It must also be true that $AQ(D_A) \neq AQ(D_A')$. In this case, the three instances form a counterexample for query answerability. Thus, a contradiction is again reached. ■

### 3.2 Decidability and Complexity Issues

The high expressiveness of the queries and schemas considered in the paper makes the problem of query satisfiability undecidable in the general case (that can be shown by reduction from query containment [HMSS01]). Possible sources of undecidability are the presence of recursively-defined derived predicates and the presence of either “axioms of infinity” [BM86] or “embedded TGDs” [Sag88]. For this reason, if we are using the CQC method [FTU05] to check the desirable properties of mappings defined in Section 3.1, the method may not terminate. However, we propose a pragmatic solution that ensures the method’s termination, and makes the approach useful in practice.

Intuitively, the aim of the CQC method is to construct an example that proves that the query being checked is satisfiable. In [FTU05], it is proved that the CQC method is sound and complete in the following terms:

- **Failure soundness:** If the method terminates without building any example, then the tested query is not satisfiable.
- **Finite success soundness:** If the method builds a finite example when queries contain no recursively-defined derived predicates, then the tested query is satisfiable.
− Failure completeness: If the tested query is not satisfiable, then the method terminates reporting its failure to build an example, when queries contain no recursively-defined derived predicates.

− Finite success completeness: If there is a finite example, the method finds it and terminates either when recursively-defined derived predicates are not considered or recursion and negation occur together in a strict-stratified manner.

Therefore, the CQC method does not terminate when there are no finite examples but infinite ones. However, if there is a finite example, the CQC method finds it and terminates, and if the tested query is not satisfiable, the method fails finitely and terminates.

One form to assure always termination is to directly avoid the CQC method to construct infinite instances. This can be done by restricting the maximum number of tuples in the instance that is constructed during the method’s execution (see Chapter 2). Once reached that maximum, the current instance under construction is considered “failed”, since it probably does not lead to a finite example, and the next relevant instance is tried. At the end of the process, if no solution has finally been found, the method reports to the user that no example with less tuples than the maximum exists. Then, the designer can choose to repeat the test with a greater maximum, and, if the situation persists, that may be an indicator that there is no finite database instance in which the tested query is satisfiable.

Such a solution could be regarded as some kind of “trickery”; however, it is a pragmatic solution in the sense that no “real” database is supposed to store an infinite number of tuples.

In Chapter 5, we propose a test that is aimed at detecting whether the CQC method is guaranteed to terminate when applied to a given mapping scenario. Such termination test is not complete, as expected given the undecidability of the termination checking problem, but can be complemented with the pragmatic solution proposed above.

Another key point regarding complexity is showing that for the class of mapping scenarios we consider (see Chapter 2), expressing the desirable properties of mappings in terms of query satisfiability does not increase their complexity.

For instance, the problem of checking query satisfiability can be reduced to the one of checking mapping losslessness: Let us assume that we want to check whether $V$ is satisfiable over $A$. Let $A'$ be a copy of $A$, and let $V'$ be the copy of $V$ over $A'$. Let us define $Q$ over $A$ as a copy of $V$, but with a contradiction (e.g., the built-in literal $1 = 0$) added to the body of all its deductive rules ($Q$ is thus not satisfiable). Let $M$ be a mapping between $A$ and $A'$, with one single mapping
assertion: \( Q \subseteq V \). If \( V \) is satisfiable, then there is a consistent instance \( D \) of schema \( A \) in which the answer to \( V \) is non-empty. Instance \( D \) together with the empty instances of \( A \) and \( A' \) form a counterexample for losslessness of \( M \) with respect to query \( V \). If \( V \) is not satisfiable, then no counterexample for losslessness exists. If there were a counterexample, we would have two instances of \( A \) giving different answers for \( V \). Therefore, \( V \) should be satisfiable and we would reach a contradiction.

Similar reductions can be made from query satisfiability to query answerability, mapping inference, and mapping satisfiability.

### 3.3 Experimental Evaluation

We experimentally evaluated the behavior of our approach for validating mappings by means of a number of experiments. The experiments were performed using the implementation of the CQC method that is the core of our SVT tool [TFU+04]. We executed the experiments on an Intel Core 2 Duo, 2.16 GHz machine with Windows XP (SP2) and 2GB RAM. Each experiment was repeated three times and we report the average of these three trials.

The experiments were designed with the goal of measuring the influence of two parameters: (1) the number of assertions in the mapping, and (2) the number of relational atoms in the queries (positive ordinary literals with a base predicate). We focused on the setting in which the two mapped schemas are of similar difficulty (i.e., a similar number and size of tables with a similar number and class of constraints), as well as the queries on each side of the mapping.

We designed the scenario for the experiments using the relational schema of the Mondial database [Mon98], which models geographic information. The schema consists of 28 tables with 38 foreign keys. We consider each table with its corresponding primary key, unique and foreign key constraints. The scenario consists of two copies of the Mondial database schema that play the roles of schema \( A \) and schema \( B \), respectively. The fact that both schemas are indeed copies of a single schema has no real effect on the performance of the CQC method; what is actually relevant is the difficulty of the schemas. The mapping between the two schemas varies from experiment to experiment, but mapping assertions always take the form \( Q_i^A = Q_i^B \). It must be remembered that equality assertions are expressed by means of two constraints, while inclusion assertions only require one of them; thus, we are considering the most general setting.
Figure 3.6 shows a graphic that compares the performance of the properties: strong mapping satisfiability, mapping losslessness, mapping inference and weak mapping satisfiability, when the number of assertions in the mapping varies. The query answerability property (not shown in the graphic) would show the same behavior as mapping losslessness (this generally happens when the two mapped schemas are of similar difficulty). Figure 3.6 focus on the case in which the distinguished query that describes the fulfillment of the corresponding property (see Section 3.1) is satisfiable. It must be remembered that the fact that the tested query is satisfiable has a different meaning depending on which property we are considering. For the two flavors of the mapping satisfiability property, it means that they hold, while for the other three properties it means that they do not.

In this experiment, the queries in the mapping take the form: $q^A_i(\bar{X}) \leftarrow R^A(\bar{X})$ and $q^B_i(\bar{X}) \leftarrow R^B(\bar{X})$, where $R^A$ is a table randomly selected from schema $A$ and $R^B$ is its counterpart in schema $B$.

The two variants of mapping satisfiability—strong and weak—can be checked without any change in the mapping because both properties already hold for the mapping scenario as it is.

Instead, in order to ensure that mapping inference does not hold, we tested the property with respect to the following assertion: $Q_1 = Q_2$. Queries $Q_1$ and $Q_2$ are: $q_1(\bar{X}) \leftarrow R^A(\bar{X}) \land X_i \neq K_1$ and $q_2(\bar{X}) \leftarrow R^B(\bar{X}) \land X_i \neq K_2$, where $X_i \in \bar{X}$, $K_1$ and $K_2$ are different constants, and $R^A$ is one of the tables that appear in the definition of the $Q^A_i$ queries, and $R^B$ is the counterpart of $R^A$. We built this assertion by taking one of the assertions in the mapping and adding disequalities to make it non-
inferable. We added disequalities on both sides of the assertion to keep both sides of the same difficulty.

In the case of mapping losslessness, we used a parameter query $Q$ that selects all the tuples from a table $T$ randomly selected from schema $A$. To ensure that the mapping is not lossless with respect to $Q$, we modified the assertion that maps $T$ to its counterpart in $B$, in such a way that the two queries in the assertion projected all the columns but one.

We can see in Figure 3.6 that the strong version of mapping satisfiability is slower than the weak one. This is expected, since strong satisfiability requires all assertions to be checked in order to ensure that all of them can be satisfied non-trivially. Instead, weak satisfiability can stop checking after finding one assertion that cannot be satisfied in a non-trivial way. It can also be seen that strong satisfiability has clearly higher running times than mapping losslessness and mapping inference. This is because these two properties have an additional parameter: a query and an assertion, respectively, and in order to check the properties, the CQC method only has to deal with the fragment of the schemas and mapping that is “mentioned” by the parameter query/annotation. However, strong satisfiability has to deal with the whole part of the schema that participates in the mapping. Figure 3.6 also shows that mapping losslessness has higher times than mapping inference. This is expected, given the difference in the size of schema $S$ between the two cases.

In Figure 3.7, we can see the same experiment as in the previous figure but now for the case in which the distinguished query is not satisfiable.
To make the two mapping satisfiability properties fail in this second experiment, we added to each table in schema $A$ a check constraint that required that one of the columns was greater than a fresh constant. We added the same constraint to the counterpart of the table in schema $B$. We also modified each mapping assertion in such a way that the two queries in the assertion were forced to select those tuples that violated the check constraints.

In the case of mapping inference, we used one of the assertions already in the mapping as the parameter assertion, and in the case of mapping losslessness, one of the $Q^A_i$ queries in the mapping as the parameter query.

The first thing we can observe in Figure 3.7 is the overall increment of all running times. The reason is that the CQC method must try all the relevant instances that are provided by the VIPs before concluding that there is no solution. Instead, in the previous experiment, the search stopped when a solution was found. It is worth noting that strong mapping satisfiability and weak mapping satisfiability have exchanged roles. The weak version of the property is now slower than the strong one. Intuitively, this is because strong mapping satisfiability may stop as soon as it finds an assertion that cannot be satisfied non-trivially, while weak mapping satisfiability must continue to search until all the assertions have been considered.

Figure 3.8 and Figure 3.9 compare the performance of the properties when the number of relational atoms in each query varies. Figure 3.8 focus on the case in which query satisfiability tests have a solution, and Figure 3.9 focus on the case in which they do not. In both experiments, the mapping has 14 assertions and its $Q^A_i$ queries follow the pattern $q^A_i(X) \leftarrow R^A_1(X_1) \land ... \land R^A_n(X_n)$, where $R^A_1, ..., R^A_n$ are $n$ tables randomly selected from schema $A$, and each query $Q^B_i$ is the counterpart over schema $B$ of the corresponding query $Q^A_i$.

To ensure that in Figure 3.8 the query satisfiability tests had a solution and that in Figure 3.9 they did not, we proceeded in a similar way as in the previous two experiments. To ensure that mapping inference in Figure 3.8 did not hold, we used one of the assertions in the mapping as parameter assertion but added one inequality on each side. In the case of mapping losslessness, we modified one of the assertions in the mapping in such a way that its queries projected all the columns minus one. We then used the original query $Q^A_i$ from the assertion as parameter query. In Figure 3.9, to make the mapping unsatisfiable, we added a check constraint (column $X$ must be greater than a constant $K$) in each table, and modified the mapping assertions in such a way that their queries now selected those tuples that violated the check constraints. To ensure that mapping
inference and mapping losslessness held in Figure 3.9, we used one of the assertions/queries in the mapping as parameter assertion/query.

Figure 3.8 shows similar results to those in Figure 3.6 for all the properties. However, Figure 3.9 shows a rapid degradation of time for mapping losslessness and mapping inference, while weak and strong mapping satisfiability show similar behavior to that in Figure 3.7 (only weak mapping satisfiability is shown in the graphic).

As mentioned above, it is expected that running times will be higher when there is no solution, since all possible instantiations for the literals in the definition of the tested query must be checked. The reason why mapping inference and mapping losslessness are so sensitive and grow so quickly with the addition of new literals can be seen by looking at the bodies of the rules that
define their distinguished queries (see Section 3.1), which follow the pattern $q(\bar{X}) \land \neg p(\bar{X})$. Therefore, in order to determine whether or not query $map_{inf}/map_{loss}$ is satisfiable, the unfolding of $q(\bar{X})$ must first be fully instantiated using the constants provided by the VIPs, e.g., $q(\bar{a}) \leftarrow r_1(\bar{a}_1) \land \ldots \land r_n(\bar{a}_n)$, and it must then be checked whether or not $\neg p(\bar{a})$ is true. If it is false, we must then try another instantiation and so on until all possible ones have been checked. Thanks to the VIPs there is a finite number of possible instantiations for the unfolding of $q(\bar{X})$, but its number is exponential with respect to the number of literals in the definition of $q$. However, this is a very straightforward implementation of the CQC method. Many of these instantiations could be avoided by using the knowledge we could obtain from the violations that occur during the search, which would reduce these running times considerably (see Chapter 4).
In Figure 3.9, the mapping satisfiability properties are not greatly affected by the addition of new literals because not all the literals have to be instantiated before concluding that the distinguished query is not satisfiable. Once a contradiction is found, only the literals that have been instantiated so far must be reconsidered.

Figure 3.10 studies the effect on strong mapping satisfiability of adding negated literals to each query in the mapping, for the setting in which the distinguished query is satisfiable and the number of positive relational atoms in each query varies.

To perform the experiment shown in Figure 3.10, we added a conjunction of $s$ negated literals $\neg T^l_1(\bar{Y}_1) \land \ldots \land \neg T^l_s(\bar{Y}_s)$ to each mapping query, where $T^l_i$ are tables randomly selected from schema $A$ not already appearing in the definition of the corresponding query $Q^l_i$, and each variable in $\bar{Y}_i$ appears in a positive atom of $Q^l_i$.

As the CQC method turns the negations in the goal into integrity constraints, adding negated literals makes more probable that a violation occurs during the search; thus, the method has to perform more backtracking and that results in an augment of running time.

Figure 3.11 studies the effect on strong mapping satisfiability of adding built-in literals (instead of the negations added in the previous experiment) to each query in the mapping.

The built-in literals added to the queries are comparisons with the form $X > K$, where $X$ is a variable corresponding to a column of one of the tables in the query’s definition, and $K$ is a constant.

<table>
<thead>
<tr>
<th>Tested properties</th>
<th>Query satisfiability tests</th>
<th># mapping assertions</th>
<th># relational atoms per query</th>
<th># deductive rules</th>
<th># constraints</th>
<th># negated literals in the schema</th>
<th># order comparisons in the schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong sat., weak sat., map. inference, map. losslessness</td>
<td>have solution</td>
<td>varies from 1 to 28</td>
<td>between 133 and 229</td>
<td>between 192 and 397</td>
<td>0, 1 or 2 (depending on the property)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>strong sat., weak sat., map. inference, map. losslessness</td>
<td>have no solution</td>
<td>varies from 1 to 28</td>
<td>between 133 and 229</td>
<td>between 192 and 397</td>
<td>0, 1 or 2 (depending on the property)</td>
<td>between 0 and 112</td>
<td></td>
</tr>
<tr>
<td>strong sat., weak sat., map. inference, map. losslessness</td>
<td>have solution</td>
<td>14</td>
<td>varies from 1 to 4</td>
<td>between 105 and 159</td>
<td>0, 1 or 2 (depending on the property)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>weak sat., map. inference, map. losslessness</td>
<td>have no solution</td>
<td>14</td>
<td>varies from 1 to 4</td>
<td>between 108 and 341</td>
<td>0, 1 or 2 (depending on the property)</td>
<td>between 0 and 112</td>
<td></td>
</tr>
<tr>
<td>strong sat.</td>
<td>have solution</td>
<td>14</td>
<td>varies from 1 to 4</td>
<td>105</td>
<td>218</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>strong sat.</td>
<td>have solution</td>
<td>14</td>
<td>varies from 1 to 4</td>
<td>105</td>
<td>218</td>
<td>varies from 0 to 84</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Summary of experiments.
Figure 3.11 shows a clear increment of times when there are more than two comparisons per query. The reason for that increment is the augment in the number of constants provided by the VIPs. The VIPs use the constants in the schema to determine the relevant instantiations. Thus, adding comparisons with constants increases the number of possible instantiations and, consequently, the running time. Moreover, the more comparisons there are, the more pronounced the increment of time.

Table 3.1 summarizes the performed experiments. It indicates which properties have been checked in each case, whether or not the corresponding distinguished queries were satisfiable, and other relevant features of the different scenarios.
Our approach does not only provide the designer with a Boolean answer; it also provides additional feedback to help the designer understand why the tested property holds or not for the given mapping. The feedback can be in the form of instances of the mapped schemas that serve as an example/counterexample for the tested property, or in the form of highlighting the subsets of schema constraints and mapping assertions that are responsible for the test result. Since the first kind of feedback—schema instances—is already provided by the CQC method, we focus on providing the second kind of feedback—subsets of schema constraints and mapping assertions. We refer to this task as computing explanations.

An explanation is minimal if no proper subset of it is also an explanation.

It is important to note that there may be more than one minimal explanation for a given desirable property test, and that all of them must be addressed in order to change the result of the test. For example, consider the mapping scenario depicted in Figure 4.1. It is easy to see that mapping \{Q_{Emp_1} \subseteq Q_{Per}, Q_{Emp_2} \subseteq Q_{Work}\} does not meet the strong mapping satisfiability property (Section 3.1.1). The problem is that the two mapping assertions conflict with the age and the salary constraint of the Employee schema, respectively (perhaps the designer missed one ‘0’ in the maximum age to be selected by \(Q_{Emp_1}\) and put an additional ‘0’ in the minimum salary to be selected by \(Q_{Emp_2}\)). Since strong mapping satisfiability requires all mapping assertions to be non-trivially satisfiable at the same time, the property will continue to “fail” although we fix one of the assertions, i.e., we must fix the two in order to solve the problem. Therefore, there are two minimal explanations: one is

\{ mapping assertion “\(Q_{Emp_1} \subseteq Q_{Per}\)”, Employee’s constraint “age \(\geq 18\)” \},

and the other is
In this chapter, we firstly propose a black-box method that computes all possible minimal explanations for a given validation test (Section 4.1). This method works at the level of the query satisfiability problem, that is, after the application of the CQC method and before translating the test result back into the mapping validation context (such translation is straightforward). The method is black-box because it relies on an underlying query satisfiability method, which, in our case, is the CQC method.

The drawback of the black-box method is that, for large schemas, the runtime of each call to the CQC method may be high. That means that even computing one single minimal explanation can be a time-consuming process. To deal with that, we propose a glass-box approach (Section 4.2), i.e., an extension of the CQC method that does not only check whether the given query is satisfiable but also provides an approximated explanation when needed. We call this extension: CQCₐ method.

The explanation provided by the CQCₐ method is approximated in the sense that it may be not minimal. The designer can decide whether the approximation is sufficient or more accurate explanations are needed.
We show that not only the designer but also the black-box method can take advantage from the approximated explanation provided by the CQCE method (Section 4.1.3).

An experimental evaluation of both approaches—black-box and glass-box—and of its combination is also provided in the corresponding sections.

4.1 Computing All Minimal Explanations

We assume that we have a procedure \texttt{isSat} to perform query satisfiability tests on a given database schema. Therefore, a query satisfiability test is a call to \texttt{isSat}(Q, S), which will return \texttt{true} if query \textit{Q} is satisfiable on schema \textit{S} and \texttt{false} otherwise. We say that an explanation for a query satisfiability test is a subset of integrity constraints from the schema such that it prevents the test from returning \texttt{true}. In other words, the query we are testing would still be not satisfiable if we removed from the schema all integrity constraints that are not in the explanation.

\textbf{Definition 4.1.} An explanation \textit{E} for a query satisfiability test \texttt{isSat}(Q, S = (DR, IC)) that returns false is a minimal subset of constraints from \textit{S} such that considering only these constraints the tested query \textit{Q} is still not satisfiable, i.e., \texttt{isSat}(Q, S' = (DR, E)) returns false. □

Note that, because \textit{E} is minimal, \texttt{isSat}(Q, S'' = (DR, E')) will return true for any \textit{E'} \subset \textit{E}, i.e., the query \textit{Q} is satisfiable for any proper subset of \textit{E}.

We address the problem of finding all possible explanations in a way that is independent of the particular query satisfiability method. That is, we see \texttt{isSat} as a black-box, and we call it several times, modifying the (sub)set of integrity constraints that is considered in each call. We do this “backwards”, which means that we call \texttt{isSat} successively, decreasing the number of constraints that are considered each time.

We also propose a filter that can be used to reduce the number of calls to the underlying query satisfiability method (Section 4.1.2). The filter is based on discarding those constraints that are not relevant for the current query satisfiability test.

4.1.1 Our Black-Box Method—The Backward Approach

The backward approach is intended to find a first explanation quickly, and then to use the knowledge from that explanation to find the remaining ones. It provides three levels of search, each one giving more information than the previous. The first level is aimed at finding just one explanation. This is done by reducing the number of constraints in the schema until only the
constraints forming the explanation remain. It requires only a linear number of calls to \texttt{isSat}, with respect to the number of constraints in the schema. In the second level, the backward approach finds a maximal set of non-overlapping explanations that includes the one found in the previous phase. This is interesting because we can provide more than just one explanation, while keeping the number of calls linear. Moreover, given that all the remaining explanations must overlap with the ones already found, the designer has already a clue of where the rest of issues might be. Finally, in the third level, the backward approach computes all the remaining explanations, but also introduces an exponential number of calls to \texttt{isSat}.

4.1.1.1 Phase 1

Let us assume that a given query $Q$ is not satisfiable on a certain database schema $S$, so \texttt{isSat}(Q, S) returns false. Phase 1 starts by performing the query satisfiability test of $Q$ on a new schema that contains all the integrity constraints from the former schema except one: $c$. If \texttt{isSat}(Q, S-\{c\}) returns false, that means there is at least one explanation that does not contain $c$. Therefore, we can discard $c$ definitely and repeat the query satisfiability test, removing another constraint. Note that this does not mean that $c$ does not belong to any explanation, but only that $c$ will not be included in the single explanation that we will obtain at the end of this phase. In contrast, if \texttt{isSat}(Q, S-\{c\}) returns true, that means there are one or more explanations that include $c$. As a consequence, $c$ cannot be discarded and must be re-introduced in the schema. Then, we repeat the query satisfiability test, removing another constraint. We continue this process of removing a constraint, testing query satisfiability, discarding or reintroducing the

phase_1(Q: query, S = (DR, IC): schema): explanation

\[
\begin{align*}
U & := IC \quad \text{// set of “unchecked” constraints} \\
E & := IC \quad \text{// explanation} \\
\text{while} (\exists c \in U) & \quad \\
& \quad E := E - \{c\} \\
& \quad \text{if} (\texttt{isSat}(Q, S' = (DR, E))) \\
& \quad \quad E := E \cup \{c\} \\
& \quad \text{endif} \\
& \quad U := U - \{c\} \\
\text{endwhile} \\
\text{return} E
\end{align*}
\]

Figure 4.2: Phase 1 of the backward approach.
constraint, removing another constraint and so on until all the constraints in the schema have been considered.

If at the end of this process all constraints have been removed from the schema, we obtain an empty explanation, which means that query $Q$ is unsatisfiable even without integrity constraints. Otherwise, we have obtained one explanation; it consists of all those constraints that remain in the schema, that is, the ones that have been considered but not discarded during the process described above. The algorithm in Figure 4.2 formalizes such a process.

For the sake of an example, let us assume that $Q$ is a query defined as follows:

$$Q \leftarrow R(X, Y, Z) \land V(Z, A, B) \land T(Z, U, V) \land Y > 5 \land B < X \land V = 2$$

Let us also assume that $S$ is a database schema with no deductive rules but with the following constraints, labeled as $c1$, $c2$, $c3$ and $c4$, respectively:

- $(c1)$ $T(X, Y, Z) \rightarrow Z \neq 2$
- $(c2)$ $R(X, Y, Z) \rightarrow Y \leq X$
- $(c3)$ $R(X, Y, Z) \rightarrow X \leq 5$
- $(c4)$ $V(X, Y, Z) \rightarrow Z \geq 10$

In this case, query $Q$ is not satisfiable on $S$. Concretely, there exist three explanations:

- $E1 = \{c1\}$
- $E2 = \{c2, c3\}$
- $E3 = \{c3, c4\}$

Let us call $\text{phase}_1(Q, S)$, with $S = \{c1, c2, c3, c4\}$ to find one of these three explanations. If we assume the constraints are considered in the order they were listed above, $c1$ is considered first. Since $\text{isSat}(Q, \{c2, c3, c4\})$ returns false, $c1$ is discarded. Constraint $c2$ is considered next. Since $\text{isSat}(Q, \{c3, c4\})$ returns false, $c2$ is also discarded. Constraint $c3$ is considered next. In this case, $\text{isSat}(Q, \{c4\})$ returns true. Therefore, $c3$ is not discarded. Finally, constraint $c4$ is considered. Since $\text{isSat}(Q, \{c3\})$ returns true, $c4$ cannot be discarded either. As a result, $\text{phase}_1(Q, S)$ returns $\{c3, c4\}$, that is, explanation $E3$. Note that if the constraints had been considered in reverse order, for instance, the returned explanation would have been another: $\{c1\} = E1$.

### 4.1.1.2 Phase 2

The second phase of the backward approach assumes that we already found a non-empty explanation in the previous phase. The goal now is to obtain, at the end of the phase, a maximal
set of explanations such that all the explanations in the set are disjoint, i.e., there is no constraint belonging to more than one explanation. One of these explanations will be the one we already found in Phase 1.

The phase proceeds as follows. We take the original schema and remove all the constraints included in the first explanation we found. In this way, we “disable” the explanation and have a chance to discover other explanations (if any), which in Phase 1 were “hidden” by it. Next, we perform the query satisfiability test over the remaining constraints. If the test returns false, that means there is still, at least, another explanation not overlapping with the one we have. To find out such a new explanation, we apply Phase 1 over the remaining explanations. On the contrary, if after removing the constraints from the former explanation, the query satisfiability test returns true, that means all the remaining explanations (if any) overlap with the one we have.

We repeat the process, removing the constraints from all the explanations we have found (the one from the first phase and the new ones we have already found in this phase), until there are no more explanations that do not overlap with the ones we already have. The algorithm in Figure 4.3 formalizes such a process.

Continuing with the example that we introduced to illustrate Phase 1, recall that we found that \{c_3, c_4\} was an explanation for the fact that \textit{isSat}(Q, \{c_1, c_2, c_3, c_4\}) had returned false. According to Phase 2, we start now by calling \textit{isSat}(Q, \{c_1, c_2\}). Since this call returns false too, that means there is another explanation and its constraints are in \{c_1, c_2\}. Therefore, we call \texttt{phase_1}((Q, \{c_1, c_2\}), which returns \{c_1\} as the new explanation. Next, we call \textit{isSat}(Q, \{c_2\}), which returns true and, thus, Phase 2 ends. The final output for this phase is \{\{c_3, c_4\}, \{c_1\}\}, which is a set of disjoint explanations.

Figure 4.3: Phase 2 of the backward approach.

```
SE := \{EP1\} // set of explanations
R := IC - EP1 // set of “remaining” constraints
while (not isSat(Q, S' = (DR, R)))
  E := phase_1(Q, S' = (DR, R))
  SE := SE \cup \{E\}
  R := R - E
endwhile
return SE
```

58
4.1.1.3 Phase 3

The third phase assumes that we already obtained a set of disjoint explanations by performing the
previous phases. The goal now is to find all the remaining explanations, that is, those that overlap
with some of the explanations that we already have. To do this, we must remove one constraint
from each known explanation to “disable” them, and then apply the first and second phases over
the remaining constraints. The drawback here is that there could be many constraints in each
explanation and, thus, many constraints to be the one to be removed. Nevertheless, we should try
all combinations to ensure we find all the remaining explanations.

Once we have removed one constraint from each explanation and executed the previous two
phases over the remaining constraints, we get some new explanations that we will add to the set
of explanations we already have. Next, we should repeat this third phase, taking into account the
added explanations, until no new explanations are found. The algorithm in Figure 4.4 formalizes
such a process.

phase_3($Q$: query, $S = (DR, IC)$: schema,
$SE$: Set(explanation)): Set(explanation)
$AE := SE$
$Combo := combinations(AE)$

while ($\exists C \in Combo$)
$R := IC – C$
if (not isSat($Q$, $S' = (DR, R)$))
$E := phase_1(Q, S' = (DR, R))$
$NE := phase_2(Q, S' = (DR, R), E)$
$AE := AE \cup NE$
$Combo := combinations(AE)$
endif
$Combo := Combo – \{C\}$
endwhile
return $AE$

combinations($SE$: Set(explanation)): Set(Set(constraint))
// returns all possible sets of constraints that can be obtained by
selecting one constraint from each explanation in $SE$.

Figure 4.4: Phase 3 of the backward approach.

4.1.1.3 Phase 3

The third phase assumes that we already obtained a set of disjoint explanations by performing the
previous phases. The goal now is to find all the remaining explanations, that is, those that overlap
with some of the explanations that we already have. To do this, we must remove one constraint
from each known explanation to “disable” them, and then apply the first and second phases over
the remaining constraints. The drawback here is that there could be many constraints in each
explanation and, thus, many constraints to be the one to be removed. Nevertheless, we should try
tall combinations to ensure we find all the remaining explanations.

Once we have removed one constraint from each explanation and executed the previous two
phases over the remaining constraints, we get some new explanations that we will add to the set
of explanations we already have. Next, we should repeat this third phase, taking into account the
added explanations, until no new explanations are found. The algorithm in Figure 4.4 formalizes
such a process.
Following the example of the previous subsections, we already had found two explanations: \{c_3, c_4\} and \{c_1\}. Now, if there is still some other explanation, it will overlap with these. Thus, to avoid these explanations to hide the remaining ones, we must select one constraint from each explanation and remove them from the original schema. In this example, there are two possibilities:

1) remove \{c_1, c_3\}
2) remove \{c_1, c_4\}

Let us consider the first option. In this case, \texttt{isSat}(Q, \{c_2, c_4\}) returns true, so no further explanation can be found.

In contrast, if we consider the second option, we get that \texttt{isSat}(Q, \{c_3, c_2\}) returns false. Therefore, we can still find further explanations. Next, we call phase_1(Q, \{c_3, c_2\}), which returns a new explanation: \{c_3, c_2\}. Clearly, phase_2(Q, \{c_3, c_2\}, \{c_3, e_2\}) will return \{\{c_3, c_2\}\} as new set of explanations.

As we have found new explanations, we must repeat the process taking now into account all the explanations discovered so far. This time, there two possible ways of disabling the three explanations that we have found so far (recall the explanations are: \{c_3, c_4\}, \{c_1\} and \{c_3, c_2\}):

1) remove \{c_1, c_3\}
2) remove \{c_1, c_2, c_4\}

It is worth noting that, since constraint \(c_3\) is shared by the explanations \{c_3, c_4\} and \{c_3, c_2\}, it is not necessary to try and remove the combinations \{c_1, c_2, c_3\} and \{c_1, c_3, c_4\}; the removal of \(c_3\) already disables \{c_3, c_4\} and \{c_3, c_2\}, so there is no need to remove and additional constraint from neither of them.

After trying the two possibilities, we reach the conclusion that there are no further explanations. Therefore, Phase 3 ends. The outcome of this phase and of the entire approach is the set formed by the three explanations: \{\{c_3, c_4\}, \{c_1\}, \{c_3, c_2\}\}.

### 4.1.2 Filtering Non-Relevant Constraints

As we have seen, the backward approach requires performing several calls to \texttt{isSat}, mostly to check whether the constraint we just removed from the schema is part or not of the explanation we are looking for. The filter described in this section consists in detecting those constraints that we can ensure are not relevant for the current query satisfiability test. We can say that a constraint is not relevant for the test when in order to get a fact about the query’s predicate it is not required.
to have also a fact about all the positive ordinary predicates in the constraint. The idea is that when we remove a constraint from the schema during Phase 1, we can also remove all those constraints that are no longer relevant for the query satisfiability test. Recall that Phase 1 is also called from Phase 2 and 3, so all three phases benefit from this filter.

For example, let us assume that we have the following database schema:

\[ R(X, Y, Z) \rightarrow \exists Y S(Z, Y) \]
\[ R(X, Y, Z) \rightarrow Z \geq 5 \]
\[ S(X, Y) \rightarrow X < 5 \]
\[ T(X, Y, Z) \rightarrow Y \geq Z \]

Let us also assume that we are testing whether query \( Q \) is satisfiable, being \( Q \) defined by the following rule:

\[ Q(X, Y) \leftarrow R(X, Y, Z) \]
Since $Q$ is not satisfiable, let us suppose we apply the backward approach to compute the explanations. We will start by finding one minimal explanations. During the process, we will remove constraint $R(X, Y, Z) \rightarrow \exists Y S(Z, Y)$ to see if it is in the explanation. In doing so, we will be eliminating the necessity of having to insert a fact about predicate $S$ in order to satisfy $Q$. The consequence of that is that since $S$ will remain empty, its constraints will never be violated, and therefore, they are not relevant for the query satisfiability test. In this case, there is just one constraint over $S$: $S(X, Y) \rightarrow X < 5$, which can also be removed from the schema before calling \text{isSat}.

More formally, the steps to apply the filter during the backward approach are the following:

1. Before starting Phase 1, we could remove the constraints that are already non-relevant for the test over the original schema (as we did with the forward approach).

2. During Phase 1, after we remove one integrity constraint $IC_i$ from the schema, we could recompute what constraints are relevant for the test over the schema that contains only the remaining constraints.

3. If some of the remaining constraints are not relevant, we can remove them before performing the test.

4. If then the test says that the predicate is still unsatisfiable we will have removed more than just one constraint and thus reduced the number of test executions we will have to do.

5. Otherwise, if the test says that the predicate is now satisfiable, we will have to put back the constraint $IC_i$ and the constraints removed in step 3.

6. If all the constraints are relevant, we can do nothing but continue the normal execution of Phase 1.

Let us consider again the example from above. In step 1 we would detect that constraint $T(X, Y, Z) \rightarrow Y \geq Z$ is not relevant. We could thus eliminate it and perform Phase 1 over the remaining three constraints. Let us suppose that we follow the order in which the constraints were listed above. Then, we would first eliminate the inclusion dependency. That would leave us with two constraints in the schema: $R(X, Y, Z) \rightarrow Z \geq 5$ and $S(X, Y) \rightarrow X < 5$. As we said, the later constraint is no longer relevant for the query satisfiability test. Thus, we could remove it and perform the test with only one constraint: $R(X, Y, Z) \rightarrow Z \geq 5$. Since the query becomes satisfiable, we should put back the two removed constraints (the inclusion and the one about $S$).

62
Phase 1 would remove then the next constraint: \( R(X, Y, Z) \rightarrow Z < 5 \), and it would continue the execution in a similar way. Figure 4.5 shows an algorithm for this new version of Phase 1.

To characterize formally the constraints that are relevant for a certain query satisfiability test, we are going to assume that each constraint is reformulated as a rule defining a derived predicate \( I^C_i \) in such a way that the constraint is violated when its corresponding predicate \( I^C_i \) is true in the database. Recall that we also assume that deductive rules have no recursion.

Let \( Q \) be a generic derived predicate defined by the following rules:

\[
Q(X) \leftarrow P^l(X_1) \land \ldots \land P^{s_l}(X_{s_l}) \land \neg S^l(X_1) \land \ldots \land \neg S^l(X_{m_l}) \\
\ldots \\
Q(X) \leftarrow P^k(X_1) \land \ldots \land P^{s_k}(X_{s_k}) \land \neg S^k(X_1) \land \ldots \land \neg S^k(X_{m_k})
\]

Symbols \( P^l, \ldots, P^{s_l}, S^l, \ldots, S^l_{m_l}, \ldots, P^k, \ldots, P^{s_k}, S^k, \ldots, S^k_{m_k} \) are predicates and \( C^l, \ldots, C^l_{r_l}, \ldots, C^k, \ldots, C^k_{r_k} \) are built-in literals. We will define \( \text{neg_preds}(Q) \) as the predicates in those negative literals that appear in the definition of \( Q \), taking into account all possible unfoldings. Formally:

\[
\text{neg_preds}(Q) = \{ \{ S^l_i \mid 1 \leq i \leq m_l \} \mid 1 \leq j \leq k \} \cup \{ \{ \text{neg_preds}(P^l_j) \mid 1 \leq i \leq s_l \} \mid 1 \leq j \leq k \}
\]

\[
\text{neg_preds}(R) = \emptyset \quad \text{if } R \text{ is a base predicate}
\]

Now, we are going to define what predicates are relevant for the satisfiability test of a certain predicate \( P \). There will be two types of relevancy: \( p\text{-relevancy} \) and \( q\text{-relevancy} \). The \( p\text{-relevant} \) predicates will be those that in order to build a database where \( P \) is intended to be satisfiable, it may be required to insert some fact about them in that database. The \( q\text{-relevant} \) predicates will be those derived predicates such that although it is not explicitly required for them to have some fact in order to make \( P \) satisfiable, they may end up having some as a result of other predicate’s facts being inserted in the database.

**Definition 4.2.** Assuming that we are testing the satisfiability of a certain predicate \( P \), we can say the following:

- Predicate \( P \) is \( p\text{-relevant} \).
- If \( Q \) is a derived predicate and it is \( p\text{-relevant} \), then \( P^l_j \) with \( 1 \leq i \leq s_l \) and \( 1 \leq j \leq k \), are also \( p\text{-relevant} \) predicates.
- If \( Q \) is a derived predicate and \( P^l_j, \ldots, P^l_{s_l} \) are \( p\text{-relevant} \) or \( q\text{-relevant} \), for some \( 1 \leq j \leq k \), then \( Q \) is \( q\text{-relevant} \).
• If $Q$ is a derived predicate and there is a negated literal about $Q$ in the body of a rule of some $p$-relevant derived predicate, and $P_j^1, ..., P_j^s$ are $p$-relevant or $q$-relevant predicates, for some $1 \leq j \leq k$, then $S_{j_1}, ..., S_{j_m}$ and the predicates in $\text{neg_preds}(P_j^1) \cup ... \cup \text{neg_preds}(P_j^s)$ are $p$-relevant.

• If $IC_i \leftarrow P_j^1(X) \land \ldots \land P_j^s(X) \land C_1 \land \ldots \land C_r \land \neg S_{j_1}(X) \land \ldots \land \neg S_{j_m}(X_m)$ is an integrity constraint and $P_j^1, ..., P_j^s$ are $p$-relevant or $q$-relevant predicates, then $IC_i$ is $q$-relevant and the predicates in $\text{neg_preds}(IC_i)$ are $p$-relevant. □

It is worth noting that a predicate defined by an integrity constraint cannot be $p$-relevant, as it is not mentioned anywhere but in the head of the constraint, and thus, only the last point of the definition is applicable.

**Definition 4.3.** We will say that an integrity constraint $IC_i \leftarrow L_1 \land \ldots \land L_n$ is relevant for the satisfiability test of $P$ if and only if the derived predicate $IC_i$ is $q$-relevant for that test. □

As an example, let us assume that we have a database schema with the following deductive rules and constraints:

\[
\begin{align*}
V(X,Y) & \leftarrow R(X,A,B) \land S(B,C,Y) \land \neg W(A,C) \\
W(X,Y) & \leftarrow P(X,Y) \land Y > 100 \\
P(X,Y) & \leftarrow T(X,Y) \land \neg H(X) \\
Q(X) & \leftarrow S(X,Y,Z) \land Y \geq 5 \land Y \leq 10 \\
IC_1 & \leftarrow R(X,Y,Z) \land \neg T(Y,Z) \\
IC_2 & \leftarrow F(X,Y) \land X \leq 0
\end{align*}
\]

Derived predicates $IC_1$ and $IC_2$ correspond to two constraints. Let us also assume that we want to test if $V$ is satisfiable in this schema. Let us now compute the predicates that are relevant for this satisfiability test:

(1) We start with $p$-relevant = $\emptyset$ and $q$-relevant = $\emptyset$

(2) The first point in the definition of predicate’s relevancy says that, as we are testing the satisfiability of $V$, $V$ is a $p$-relevant predicate.

(3) Then, $p$-relevant = $\{V\}$ and $q$-relevant = $\emptyset$

(4) Now that we know $V$ is $p$-relevant, by the second point of the definition we can infer that $R$ and $S$ are also $p$-relevant.

(5) $p$-relevant = $\{V, R, S\}$ and $q$-relevant = $\emptyset$
As long as \( S \) is p-relevant, by the third point of the definition, we can say that \( Q \) is q-relevant.

\[ p\text{-relevant} = \{V, R, S\} \text{ and } q\text{-relevant} = \{Q\} \]

By the fifth point, as \( R \) is p-relevant, we can say that \( IC_1 \) is q-relevant and \( T \) is p-relevant.

\[ p\text{-relevant} = \{V, R, S, T\} \text{ and } q\text{-relevant} = \{Q, IC_1\} \]

Once we know that \( T \) is p-relevant, by the third point again, we can conclude that \( P \) is q-relevant.

\[ p\text{-relevant} = \{V, R, S, T\} \text{ and } q\text{-relevant} = \{Q, IC_1, P\} \]

We can apply now the fourth point of the definition. The derived predicate \( W \) appears negated in the rule of \( V \), and \( V \) is p-relevant. The predicates that appear positively in \( W \), i.e., \( \{P\} \), are also relevant. Thus, we can infer that the predicates that appear negated in \( W \) or some of its unfoldings are p-relevant. That means \( H \) is p-relevant.

\[ p\text{-relevant} = \{V, R, S, T, H\} \text{ and } q\text{-relevant} = \{Q, IC_1, P\} \]

We still can apply the third point and say that as \( P \) is q-relevant, then \( W \) is q-relevant too.

\[ p\text{-relevant} = \{V, R, S, T, H\} \text{ and } q\text{-relevant} = \{Q, IC_1, P, W\} \]

We cannot infer anything new. Thus, there are no other relevant predicates.

Finally, we can say that \( IC_1 \) is a relevant constraint for the satisfiability test of \( V \) and that \( IC_2 \) is not relevant. Intuitively, it is easy to see that \( IC_2 \) is not relevant because predicate \( F \) is not mentioned anywhere else (\( F \) is also non-relevant).

**Proposition 4.1.** Let \( P \) be an unsatisfiable predicate and let \( IC_i \) be a constraint from the database schema. If \( IC_i \) is not relevant for the satisfiability test of \( P \), then \( P \) is still unsatisfiable after removing \( IC_i \) from the schema.

**Proof.** Let us assume that after removing \( IC_i \) from the schema, \( P \) becomes satisfiable. It follows that exists some minimal database \( D \) such that \( D \) is consistent and some fact about \( P \) is true in \( D \). Database \( D \) is minimal in the sense that there is no database \( D' \) with less tuples than \( D \) such that \( D' \) is also consistent and contains some fact about \( P \).

As long as \( P \) becomes satisfiable after removing \( IC_i \), database \( D \) should violate \( IC_i \). Our goal now is to show that it follows that \( IC_i \) is q-relevant for the satisfiability test of \( P \). To reach that, we will do induction over the unfolding level of the predicates. A base predicate has an unfolding level of 0. A derived predicate such that the maximum unfolding level of the predicates that
appear positively in its rules is \( n \), has an unfolding level of \( n+1 \). The base case will be thus when the predicate is a base predicate. Let \( T \) be this predicate. We assume that there is at least one fact about \( T \) in \( D \). Given that \( D \) is minimal, there are only two possibilities. The first is that a fact about \( T \) may be required to satisfy the definition of \( P \), i.e., a positive literal about \( T \) appears in the definition of \( P \) (taking into account all possible unfoldings). The second possibility is that the satisfaction of \( P \) leads to the violation of some integrity constraint that can be repaired by means of the addition of a fact about \( T \), i.e., there is some constraint with a negative literal about \( T \) and such that all its positive literals are true in \( D \). In both cases, the conclusion is that predicate \( T \) is p-relevant for the satisfiability test of \( P \). The induction case will be that in which \( T \) is a derived predicate. As long as some fact about \( T \) is true in \( D \), some rule defining \( T \) should have all its literals true in \( D \). By induction, we can conclude that all the predicates from the positive literals in that rule are p-relevant or q-relevant and that \( T \) is thus q-relevant itself.

Finally, as \( \text{IC}_i \) is true in \( D \), we can conclude that \( \text{IC}_i \) is q-relevant, and we reach a contradiction. ■

4.1.3 Taking Advantage of an Approximated Explanation

Let us assume that \( \text{isSat}(Q, S) \) returns a pair \((B, \text{ApproxE})\), where \( B \) is the Boolean result of the query satisfiability test, and \( \text{ApproxE} \) is an approximated explanation for \( Q \) being unsatisfiable on \( S \) (meaningful only when \( B = \text{false} \)). The idea is to offer the approximated explanation to the user in the first place. If he wants a more accurate explanation, we can apply the Phase 1 of the backward approach to minimize the explanation; if he then wants additional explanations, we can apply Phase 2; and if he wants all the possible explanations, we can just apply Phase 3.

Moreover, the Phase 1 of the backward approach can be modified so it takes advantage of the approximated explanation returned by \( \text{isSat} \). We do that as follows.

This new version of Phase 1—let us call it \( \text{Phase 1}'' \)—assumes that we have already tested the satisfiability of \( Q \) on \( S \) and found that it is not satisfiable, that is, it assumes that we already have an approximated explanation \( \text{ApproxE} \) returned by the initial query satisfiability test.

\( \text{Phase 1}'' \) starts by removing from \( S \) all those integrity constraints not present in \( \text{ApproxE} \). After that, it removes one additional constraint, namely \( c \). Now, it computes the integrity constraints that are no longer relevant for the satisfiability of \( Q \) on \( S \) (see Section 4.1.2), namely \( \{c_1, ..., c_n\} \), and removes them all. Let us assume \( \text{isSat}(Q, \text{ApproxE}-\{c, c_1, ..., c_n\}) \) returns \((B', \text{ApproxE}')\). If \( B' \) is true, \( \{c, c_1, ..., c_n\} \) must all be re-introduced in the schema. If \( B' \) is false, we can discard all those constraints not in \( \text{ApproxE}' \)—that includes both \( c \) and \( \{c_1, ..., c_n\} \)—and also
all those other constraints that become not relevant after these last removals. The process continues until all constraints have been considered. As in the original version, the constraints that remain at the end of the process are the ones that form the explanation. Figure 4.6 formalizes Phase 1".

We will discuss how to compute an approximated explanation with a single execution of isSat in Section 4.2.

4.1.4 Experimental Evaluation

We have performed some experiments to compare the efficiency of the backward approach with respect to one of the best methods known for finding minimal unsatisfiable subsets of constraints: the hitting set dualization approach [BS05]. We have also evaluated the behavior of the backward approach when varying some parameters: the size of the explanations, the number of explanations for each test, and the number of constraints in the schema. We executed the experiments on an Intel Core 2 Duo, 2.16 GHz machine with Windows XP (SP2) and 2 GB RAM.

Figure 4.6: Version of Phase 1 that takes advantage of approximated explanations.

\[
\text{phase}_1''(Q:\text{ query}, S = (DR, ApproxE): \text{ schema}): \text{ explanation} \\
\text{ApproxE} := \text{ApproxE} - \text{nonRelevantConstrs}(Q, S) \\
U := \text{ApproxE} \quad \text{// set of “unchecked” constraints} \\
E := \text{ApproxE} \quad \text{// explanation} \\
\text{while } (\exists c \in U) \\
\quad E := E - \{c\} \\
\quad NRC := \text{nonRelevantConstrs}(Q, S' = (DR, E)) \\
\quad E := E - NRC \\
\quad (B', \text{ApproxE}') := \text{isSat}(Q, S'' = (DR, E)) \\
\quad \text{if } (B') \\
\quad \quad E := E \cup \{c\} \cup NRC \\
\quad \quad U := U - \{c\} \\
\quad \text{else} \\
\quad \quad E := \text{ApproxE}' - \text{nonRelevantConstrs}(Q, S''' = (DR, \text{ApproxE}')) \\
\quad \quad U := U \cap E \\
\text{endif} \\
\text{ endwhile} \\
\text{return } E
\]
To perform the query satisfiability tests in the experiments, we used the CQCE method, which will be described in Section 4.2. More precisely, we used the implementation of the CQCE method that is the core of our SVTE tool (Schema Validation Tool with Explanations) [FRTU08]. Remind that our approach is however independent of the method used. We have used the CQCE method here since it allows us to consider schemas with a high degree of expressiveness and evaluate the behavior of the backward approach in the case in which it can take advantage of approximated explanations.

The first experiment, shown in Figure 4.7, is aimed at comparing the approach for computing explanations that we proposed on Section 4.1.1, the backward approach, with the hitting set dualization approach proposed in [BS05]. We have used an implementation of the dualization approach that uses incremental hitting set calculation, as described in [BS05], but replacing the
calls to the satisfiability method by calls to the CQC_E method. We performed the experiment using a database schema formed by $K$ chains of tables, each one with length $N$:

$$
R^1(A^1,B^1), \ldots, R^N(A^N,B^N)
$$

$$
\ldots
$$

$$
R^K(A^K,B^K), \ldots, R^K(N,N).
$$

Each table has two columns and two constraints: a foreign key from its second column to the first column of the next table, i.e., $R_i'B_i$ references $R_{i+1}'A_{i+1}'$, and a Boolean check constraint requiring that the first column must be greater than the second, i.e., $A_i > B_i$. Additionally, the first table of each chain has a check constraint stating that its first column must be greater than 5, i.e., $R_1'A_1 \leq 5$. The last table of each chain has another check constraint stating that its second column must not be lower than 100, i.e., $R_N'B_N \geq 100$. This schema is designed to allow us to study the effect of varying the number and size of explanations. Note that the value of $N$ determines the size of the explanations and that the value of $K$ determines their number. When $N$ is set to 1, we find explanations of size 3, and each increment in the value of $N$ results in 2 additional constraints in each explanation. Regarding $K$, its value is exactly the number of explanations we will find.

Note also that in this experiment all the explanations are disjoint. Each chain of tables in the schema provides one explanation, and all the chains are disjoint. That means, when we execute the phase 3 of the backward approach, it will not provide any new explanation with respect to the first two phases.

In this experiment, we computed the explanations for the satisfiability test of the following Boolean query $P$:

$$
P \leftarrow R^1(X^1, X^2) \land \ldots \land R^K(X^K, X^K).
$$

The symbols $X^1$, $X^2$, ..., $X^K$, $X^K$ are fresh variables. Due to the previous database schema definition, the satisfiability test of $P$ does not reach any solution, i.e., $P$ is not satisfiable over the former schema.

Figure 4.7 shows the running times for different values of $N$, which range from 1 to 5. The value of $K$ was set to 2. We executed the backward approach without using the filter described in Section 4.1.2. All three phases of the backward approach were executed.

The graphic shows that the dualization approach is quite much slower than our backward approach. It is worth noting, however, that the dualization approach [BS05] was proposed for the
context of type error and circuit error diagnosis and that we are applying it now to a different context. One difference is that while in [BS05] the authors use an incremental satisfiability method for Herbrand equations, we are not aware of any incremental method to check query satisfiability in the class of schemas that we consider here. Another difference is that the dualization approach computes the explanations by means of the relationship that exists between the minimal unsatisfiable subsets of constraints (the explanations) and the maximal satisfiable subsets of constraints. Thus, it finds a maximal unsatisfiable subset first, then computes its complements, and finally computes the hitting sets for the set of complements. The resulting hitting sets are the candidates for being explanations. In a different way, the backward approach finds a maximal set of disjoint explanations first, which requires only a linear number of test executions, and then focuses on finding the remaining explanations, taking into account that they must overlap with the ones already found. In this way, it can significantly reduce the number of candidates to be considered. Figure 4.8 shows the number of calls to the CQC\textsubscript{i} method performed by each approach.

Figure 4.9: Effect of varying the size and number of explanations.
Figure 4.9 shows the behavior of the dualization and backward approaches when the number of explanations varies from 1 to 3, the explanations are disjoint, and the size of each explanation ranges from 3 to 11. We used the same database schema than in the previous experiment and the same target query $P$. Focusing on our backward approach, Figure 4.9 shows an increase of running time when the number of explanations grows, which is higher when going from 2 to 3 explanations. This is expected since although phases 1 and 2 imply a linear number of test executions, phase 3 still requires an exponential number of them. Regarding the dualization approach, it shows a similar behavior, although its running times are significantly higher than those of the backward approach under the same number of explanations. The same behavior can be observed on the number of calls the two approaches make to the CQC$_E$ method.

In Figure 4.10, we compare the backward approach with its three phases against the first two phases only and against the first phase only. This time, we used a database schema similar to the one we used in the previous experiments but formed now by the following two chains:

\[
\begin{align*}
R_1^i(A_1^i,B_1^i), & \ldots, R_N^i(A_N^i,B_N^i), R_N^i(A_N^i,B_N^i,C_N^i) \\
R_2^i(A_1^i,B_1^i), & \ldots, R_2^i(A_2^i,B_2^i)
\end{align*}
\]

The integrity constraints are also similar to those in the previous schema but with two additions: a check constraint in $R_N^i$ that states $A_N^i \geq C_N^i$, and another check, also in $R_N^i$, which states $C_N^i \geq 200$. The target query $P$ is now the following:

\[
P \leftarrow R_1^i(X,Y) \land R_2^i(U,V)
\]

In this schema, there will be three explanations for the satisfiability test of $P$. The first chain will provide two of them, which will overlap. These two explanations will share all their constraints except those in $R_N^i$; one explanation will have the constraints: $A_N^i \geq B_N^i$ and $B_N^i \geq 100$, and the other explanations the constraints: $A_N^i \geq C_N^i$ and $C_N^i \geq 200$. The second chain will provide the third explanation. Phase 1 will thus find one of these three explanations; phase 2 will find an explanation disjoint with the previous one; and, finally, the third phase will find the remaining one. This way, since each phase provides one explanation, we will be able to compare them.

The graphics in Figure 4.10 show a big increment of running time when we introduce the third phase. This is expected since the third phase requires to select one constraint from each explanation already found, trying all the possible combinations. It can also be seen that the graphics for the cases of phases 1 & 2 and phase 1 only have also an exponential shape although they require just a linear number of test executions. This result is clearly due to the cost of each
one of these test executions, as the exponential cost of the used method (in this case the CQC\textsubscript{E} method) cannot be avoided because of the complexity of the satisfiability problem. Note that, however, the linear shape can indeed be observed in Figure 4.11, which shows the number of calls to the CQC\textsubscript{E} method made by the backward approach in Figure 4.10.

In Figure 4.12, we study the effect of both the filter described in Section 4.1.2 and the use of approximated explanations described in Section 4.1.3 in reducing the number of calls the backward approach makes to the underlying query satisfiability method. This time we used a database schema similar to the one from the first experiment (with K = 2), but with some additions. First, we added a third attribute $R_{ji}^i.C$ to each table $R_{ji}^i$. Moreover, for each chain $R_{ji}(A_{j1},B_{j1},C_{j1}), \ldots, R_{ji}(A_{jN},B_{jN},C_{jN})$, we added L constraints in the form of: $R_{ji}(A_{j1},B_{j1},C_{j1}) \land \ldots \land R_{ji}(A_{jN},B_{jN},C_{jN}) \rightarrow C_{j1} \neq k_{j1} \lor \ldots \lor C_{jN} \neq k_{jN}$, where $k_{j1}, \ldots, k_{jN}$ are fresh constants. These new constraints will allow us to see the difference between using or not approximated explanations.
Note that the constraints are relevant for the query satisfiability test (thus, they will not be removed by the filter), but are not part of any explanation (which makes them candidates to be removed when using approximated explanations). The next modifications are aimed at making visible the difference between using the filter and using no optimization. To that end, we added, for each table \( R^i \), the following chain: \( R^i_1(A_1, B_1), \ldots, R^i_N(A_N, B_N) \). Each one of the tables in the chain has the following constraints: a check \( R^i_s.A_s > R^i_s.B_s \), and a referential constraint in which \( R^i_s.B_s \) references \( R^i_{s+1}.A_{s+1} \). We also added an additional referential constraint to each table \( R^i_j \) that references the first table of its corresponding new chain, i.e., \( R^i_j.A_j \) references \( R^i_1.A_1 \). Finally, we added \( M \) tables from the relational schema of the Mondial [Mon98] database (out of 28 tables), and connected them with the tables \( R^i_j \) by means of referential constraints in which an attribute of each table from the Mondial schema references some \( R^i_j.A_j \). We considered the Mondial database schema with its primary key, unique and foreign key constraints.

The graphics in Figure 4.12 show the behavior of the backward approach with/without filter and with/without taking advantage of approximated explanations, when increasing the number of constraints in the database schema. We used schemas with 40, 78, 125 and 189 constraints, respectively, which we got by changing the value of \( N, L \) and \( M \). It can be seen how using the filter reduces dramatically the number of calls to \( \text{isSat} \) with respect to the version of the backward approach without any optimization. It is also clear that the combination of the filter with the approximated explanations reduces even more the required number of calls.

Figure 4.12: Effect of the filter and the use of approximated explanations on the number of calls to the CQC_F method.
4.2 Computing an Approximated Explanation

In this section, we propose to explain the unsatisfiability of a given query be means of a glass-box approach. That is, we propose to extend the query satisfiability method in such a way that when the tested query is unsatisfiable, it returns not only a Boolean answer but also some sort of explanation. More specifically, we propose an extension of the CQC method, which we refer to as the CQC method.

Recall that the CQC method is a query satisfiability method. That means the explanation provided by this glass-box approach is to be translated back into the mapping validation context, as in the case of our black-box method.

In contraposition to the black-box method, this glass-box approach does not require multiple executions of the query satisfiability test but just one, which has a significant impact on running time, especially when schemas are large. The drawback is the fact that the explanation provided by the CQC method may be not minimal, and the fact that it provides just one explanation and not all the possible ones. However, as discussed in Section 4.1.3, our black-box method can be combined with this glass-box approach in such a way that we benefit from the advantages of both of them.

Before introducing the CQC method, we must discuss some formalism issues:

- For the sake of uniformity when dealing with deductive rules and constraints, we associate an inconsistency predicate $I_c$ to each integrity constraint (we did the same in Section 4.1.2). Then, a database instance violates a constraint $I_c \leftarrow L_1 \land ... \land L_k$ if predicate $I_c$ is true in that database, i.e., if there is some ground substitution $\sigma$ that makes $(L_1 \land ... \land L_k)\sigma$ true.

- We assume that the satisfiability test of the given query is expressed in terms of a goal to attain $G = L_1 \land ... \land L_m$ and a set of conditions to enforce $F \subseteq IC$ [FTU04]. In this way, we say that $(G, F)$ is satisfiable if there is at least one database instance that makes $G$ true and does not violate any integrity constraint in $F$.

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

4.2.1 Our Glass-Box Approach—The CQC Method

The main aim of our approach is to perform query satisfiability tests expressed in the formalism stated above, in such a way that: (1) if the property is satisfiable, we provide a concrete database
instance in which the query has a non-empty answer; and (2) if the query is not satisfiable, we provide an approximated explanation.

As defined in [FTU05], the CQC method does not provide any kind of explanation when a query satisfiability test “fails”. Roughly, the original CQC method performs query satisfiability tests by trying to construct a database instance in which the tested query has at least one tuple in its answer (see Chapter 2 for an overview). The method uses different Variable Instantiation Patterns (VIPs), according to the syntactic properties of the database schema considered in each test, to instantiate the ground EDB facts (i.e., tuples) to be added to the database. Adding a new fact to the database under construction may cause the violation of some constraints. When a violation is detected, some previous decisions must be reconsidered in order to explore alternative ways to reach a solution (e.g., reinstantiate a variable with another constant). In any case, the CQC method does not prescribe any particular execution strategy for the generation of the different alternatives.

The extension we propose in this section is to define an execution strategy that explores only those alternatives that are indeed relevant for reaching the solution. In order to do this, we need to modify the internal mechanisms of the CQC method to gather the additional information that is required for detecting which alternatives are relevant. If none of these alternatives leads to a solution, the gathered information will be used to build one explanation: the explanation of why this execution has failed. This explanation may however not be minimal in the context of explaining the unsatisfiability of the query.

In addition to allow us the computation of an approximated explanation, using the CQC method results in a significant efficiency improvement, as we will show in Section 4.2.3.

4.2.1.1 Example

Let us consider a database schema with two tables: Category(name, salary) and Employee(ssn, name, category). The salary is constraint to be $\geq 50$ and $\leq 30$; the category of an employee must be different from ‘ceo’; and there is a referential constraint from attribute Employee.category to Category.name. It is easy to see that this database cannot store any tuple. The constraint in the salary is impossible to satisfy, which means we cannot insert any tuple into the Category table. Since employees must always have a category, we cannot insert any tuple into the Employee table either (we assume null values are not allowed). The deductive rules and integrity constraints of this schema, expressed in the formalism required by our method, are as follows:

\[
\text{Deductive rules } DR = \{ \text{isCat}(X) \leftarrow \text{Cat}(X, S) \} 
\]
Integrity constraints \( IC = \{ \)
\[ Ic_1 \leftarrow \text{Emp}(X, Y) \land Y = \text{ceo}, \]
\[ Ic_2 \leftarrow \text{Emp}(X, Y) \land \neg \text{isCat}(Y), \]
\[ Ic_3 \leftarrow \text{Cat}(X, S) \land S > 30, \]
\[ Ic_4 \leftarrow \text{Cat}(X, S) \land S < 50 \} \]

Suppose that we want to check whether a query that selects all employees is satisfiable on this database schema, that is, whether \( (G = \text{Emp}(X, Y), IC) \) is satisfiable. Figure 4.13 shows a CQC\(_E\)-derivation that tries to construct an EDB to prove that this query is satisfiable. Each row in the figure corresponds to a CQC\(_E\)-node that contains the following information (columns):

1. The goal to attain: the literals that must be made true by the EDB under construction.
2. The conditions to be enforced: the set of conditions that the constructed EDB is required to satisfy.
3. The extensional database (EDB) under construction.
4. The conditions to be maintained: a set containing those conditions that must remain satisfied until the end of the CQC\(_E\)-derivation.
5. The set of constants used so far.

The transition between an ancestor CQC\(_E\)-node and its successor is performed by applying a CQC\(_E\)-expansion rule to a selected literal (underlined in Figure 4.13) of the ancestor CQC\(_E\)-node (see Section 4.2.2).

The first two steps shown in Figure 4.13 instantiate variables \( X \) and \( Y \) from literal \( \text{Emp}(X, Y) \) in order to obtain a ground fact to be added to the EDB. The constants used to instantiate the variables are determined according to the corresponding Variable Instantiation Patterns (VIPs) [FTU05] and their data type (int, real or string). A label is attached to the constant occurrences,

<table>
<thead>
<tr>
<th>Goal to attain</th>
<th>Conditions to enforce</th>
<th>EDB</th>
<th>Conditions to maintain</th>
<th>Used constants</th>
<th>Node ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leftarrow \text{Emp}(X, Y) )</td>
<td>( {Ic_1, Ic_2, Ic_3, Ic_4} = C_0 )</td>
<td>( \emptyset )</td>
<td>( C_0 )</td>
<td>{50, 30, ceo}</td>
<td>1</td>
</tr>
<tr>
<td>1:A2.1</td>
<td>( \leftarrow \text{Emp}(01, Y) )</td>
<td>( {Ic_1, Ic_2, Ic_3, Ic_4} )</td>
<td>( \emptyset )</td>
<td>( C_0 )</td>
<td>{50, 30, ceo, 0}</td>
</tr>
<tr>
<td>2:A2.1</td>
<td>( \leftarrow \text{Emp}(01, \text{ceo}) )</td>
<td>( {Ic_1, Ic_2, Ic_3, Ic_4} )</td>
<td>( \emptyset )</td>
<td>( C_0 )</td>
<td>{50, 30, ceo, 0}</td>
</tr>
<tr>
<td>3:A2.2</td>
<td>( \leftarrow \text{Emp}(01, \text{ceo}) )</td>
<td>( {Ic_1, Ic_2, Ic_3, Ic_4} )</td>
<td>( \emptyset )</td>
<td>( C_0 )</td>
<td>{50, 30, ceo, 0}</td>
</tr>
<tr>
<td>4:B2</td>
<td>( \leftarrow \text{Emp}(01, \text{ceo}) )</td>
<td>( {Ic_1, Ic_2, Ic_3, Ic_4} )</td>
<td>( \emptyset )</td>
<td>( C_0 )</td>
<td>{50, 30, ceo, 0}</td>
</tr>
</tbody>
</table>

Figure 4.13: Example of CQC\(_E\)-derivation.
indicating the node where they were introduced. Step 3 inserts the instantiated literal into the EDB under construction. Label 3 is attached to the new tuple to keep record of which node was responsible for its insertion. After this step, we get a node with an empty goal, i.e. []. However, the work is not done yet, since we must ensure that the four constraints are not violated by the current EDB. Steps 4 and 5 evaluate constraint \( Ic_1 \), which is violated.

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

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

Figure 4.14 shows an alternative CQC\(_E\)-derivation rooted at node 2. Steps 6, 7, 8 of this new derivation are similar to steps 2, 3 and 4, but step 6 uses a fresh constant ‘\( a' \)’ to instantiate variable \( Y \). Step 9 selects literal \( a'^2 = ceo \). Since such a comparison is false, \( Ic_1 \) is not violated now, and it is thus removed from the set of conditions to enforce.

Steps 10 and 11 deal with referential constraint \( Ic_2 \), which introduces a new (sub)goal: \( \text{isCat}(a') \). To achieve it, tuple \( \text{Cat}(a'^2, 50^{1/2}) \) is added to the EDB (step 14), but this addition violates constraint \( Ic_3 \) (step 16).
As before, the analysis of the violation is performed. In this case, the set of repairs, recorded in node 15, is \{12, 10\}. The intuition is that the violation was originated by the instantiation of variable $S$ in node 12, and that this instantiation was required to achieve the (sub)goal introduced by node 10.

The method will try to construct another alternative (sub)derivation rooted at CQC E-node 12. Any derivation starting from node 12 will fail because each possible instantiation for variable $S$ in $\text{Cat}(a^2, S)$ will lead to the violation of either $Ic_3$ or $Ic_4$, with \{12, 10\} as the set of repairs in any case. Therefore, the method marks CQC E-node 12 as failed. Its explanation is \{Ic_3, Ic_4\}, and the set of repairs is \{10\}. The method will visit now this node 10. This node enforces referential constraint $Ic_2$, and so, leads to the violation of constraints $Ic_3$ and $Ic_4$. Since there is not an alternative (sub)derivation rooted at node 10, the method marks this node as failed. The explanation for this failure is the explanation of its only (sub)derivation plus the referential constraint $Ic_2$, i.e., \{Ic_2, Ic_3, Ic_4\}. The set of repairs of node 10 is the empty set. Therefore, there is no point in reconsidering any previous decision, so the method ends without being able of constructing an EDB that satisfies the initial goal, and returns \{Ic_2, Ic_3, Ic_4\} as the set of integrity constraints that explains such a failure (the explanation indicated in the introduction). Note that

**Figure 4.14: An alternative CQC\textsubscript{E}-(sub)derivation.**
since node 2 does not belong to the set of repairs of node 10, the explanation for the failed derivation in Figure 4.13, recorded at node 2, is discarded and not included in the final explanation.

4.2.2 Formalization

Let $S = (DR, IC)$ be a database schema, $G_0 = L_1 \land \ldots \land L_n$ a goal, and $F_0 \subseteq IC$ a set of constraints to enforce, where $G_0$ and $F_0$ characterize a certain query satisfiability test. A CQCE-node is a 5-tuple of the form $(G_i, F_i, D_i, C_i, K_i)$, where $G_i$ is a goal to attain; $F_i$ is a set of conditions to enforce; $D_i$ is a set of ground EDB atoms, i.e., an EDB under construction; $C_i$ is the whole set of conditions that must be maintained; and $K_i$ is the set of constants appearing in $DR$, $G_0$, $F_0$ and $D_i$.

A CQCE-tree is inductively defined as follows:

1. The tree consisting of the single CQCE-node $(G_0, F_0, \emptyset, F_0, K)$ is a CQCE-tree.
A#-Rules:

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

\[(G, d(\bar{x}) \land L_1 \land \ldots \land L_n, F, D, C, K)\]  

where \(G_{1_{\alpha}} = \{T^{d} \land \ldots \land T^{d} \land L_1 \land \ldots \land L_n\} \sigma_2\) and \(d(\bar{x}) \leftarrow T_1 \land \ldots \land T_n\) is one of the \(m\) deductive rules in \(DR\) that define predicate \(d\) and substitution \(\sigma_2\) is the most general unifier of \(d(\bar{x})\) and \(d(\bar{z})\).

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

\[(G_b, F_b, D_b, C_b, K)\]  

where \(Y\) is a variable from \(\bar{x}\), and each ground substitution \(\sigma = \{Y \mapsto k_i\}^d\) is one of the \(m\) instantiations for variable \(Y\) provided by the corresponding VIP.

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

\[(G_b, F_b, D_b, C_b, K)\]  

where \(F_{1_{\alpha}} = F_b \cup C\) and \(D_{1_{\alpha}} = D_b \cup \{b(\bar{x})^d\}\) if \(b(\bar{x}) \notin D_b\) (disregarding labels); otherwise \(F_{1_{\alpha}} = F_b\) and \(D_{1_{\alpha}} = D_b\).

(A3) The selected literal \(-p(\bar{x})\) is a ground negated atom:

\[(-p(\bar{x}) \land G_{1_{\alpha}}, F_{1_{\alpha}}, D_{1_{\alpha}}, C_{1_{\alpha}}, K)\]  

where \(l_c = \text{lcnew} \leftarrow \text{Normalize}(p, \bar{x})\), and \(\text{lcnew}\) is a fresh predicate.

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

\[(C \land G_{1_{\alpha}}, F_{1_{\alpha}}, D_{1_{\alpha}}, C_{1_{\alpha}}, K)\]  

only if \(C\) is evaluated true (disregarding labels).

B#-Rules:

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

\[(G, d(\bar{x}) \land L_1 \land \ldots \land L_n, F, D, C, K)\]  

where \(S_j = \text{lc} \leftarrow \{B \mapsto \text{Evaluate}(T_1 \land \ldots \land T_n \land P_1 \land \ldots \land P_m)\} \sigma_1\) and \(d(\bar{x}) \leftarrow T_1 \land \ldots \land T_n\) is one of the \(m\) deductive rules in \(DR\) that define predicate \(d\) and \(\sigma_1\) is the most general unifier of \(d(\bar{x})\) and \(d(\bar{z})\).

(B2) The selected literal \(h(x_1, \ldots, x_n)\) is a positive EDB atom:

\[(G, h(x_1, \ldots, x_n), F, D, C, K)\]  

only if \(S = \emptyset\) or \(n \geq 1\), where \(S_j = \text{lc} \leftarrow \{B \mapsto h(k_1, \ldots, k_n)\text{new}\} \sigma_1\)

\((P_1 \land \ldots \land P_m) \sigma_1\) and \(h(k_1, \ldots, k_n)\text{new}\) is one of the \(m\) facts about \(h\) in \(D_b\) and \(\sigma_1 = \{x_1 \mapsto k_1, \ldots, x_n \mapsto k_n\}\) (\(k_1, \ldots, k_n\) may be labeled).

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

\[(G, h(x_1, \ldots, x_n) \land \text{Evaluate}(T_1 \land \ldots \land T_n) \land \text{Evaluate}(F, D, C, K))\]  

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

\[(G, C \land C, F, D, C, K)\]  

only if \(n \geq 1\).

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

\[(G, C \land C, F, D, C, K)\]  

Figure 4.16: Formalization of the CQC\(_{E}\)-expansion rules.

2. Let \(T\) be a CQC\(_{E}\)-tree, and \((G_n, F_n, D_n, C_n, K_n)\) a leaf CQC\(_{E}\)-node of \(T\) such that \(G_n \neq \emptyset\) or \(F_n \neq \emptyset\). Then the tree obtained from \(T\) by appending one or more descendant CQC\(_{E}\)-nodes according to a CQC\(_{E}\)-expansion rule applicable to \((G_n, F_n, D_n, C_n, K_n)\) is again a CQC\(_{E}\)-tree.

It may happen that the application of a CQC\(_{E}\)-expansion rule on a leaf CQC\(_{E}\)-node \((G_n, F_n, D_n, C_n, K_n)\) does not obtain any new descendant CQC\(_{E}\)-node to be appended to the CQC\(_{E}\)-tree because some necessary constraint defined on the CQC\(_{E}\)-expansion rule is not satisfied. In such a case, we say that \((G_n, F_n, D_n, C_n, K_n)\) is a failed CQC\(_{E}\)-node. Each branch in a CQC\(_{E}\)-tree is a CQC\(_{E}\)-derivation consisting of a (finite or infinite) sequence \((G_0, F_0, D_0, C_0, K_0), (G_1, F_1, D_1, C_1, K_1), \ldots\) of CQC\(_{E}\)-nodes. A CQC\(_{E}\)-derivation is successful if it is finite and its last (leaf) CQC\(_{E}\)-node has
the form ([], ∅, D_n, C_n, K_n). A CQC_E-derivation is failed if it is finite and its last (leaf) CQC_E-node is failed. A CQC_E-tree is successful when at least one of its branches is a successful CQC_E-derivation. A CQC_E-tree is finitely failed when each one of its branches is a failed CQC_E-derivation.

Figure 4.15 shows the formalization of the CQC_E-tree exploration process. ExpandNode(T, N) is the main algorithm, which generates and explores the subtree of T that is rooted at N. The CQC_E method starts with a call to ExpandNode(T, N_root) where T contains only the initial node N_root = (G_0, F_0, ∅, F_0, K). If the CQC_E method constructs a successful derivation, ExpandNode(T, N_root) returns “true” and T.solution pinpoints its leaf CQC_E-node. On the contrary, if the CQC_E-tree is finitely failed, ExpandNode(T, N_root) returns “false” and N_root.explanation ⊆ F_0 is an explanation for the unsatisfiability of the tested query.

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

We assume the bodies of the constraints in \(N_{\text{root}}\) are normalized. We say that a conjunction of literals is \textit{normalized} if it satisfies the following syntactic requirements: (1) there is no constant appearing in a positive ordinary literal, (2) there are no repeated variables in the positive ordinary literals, and (3) there is no variable appearing in more than one positive ordinary literal. We consider normalized constraints because that simplifies the violation analysis process.

Figure 4.16 shows the \textit{CQC}\(_C\)-\textit{expansion rules} used by \textit{ExpandNode}. The \textit{Variable Instantiation Patterns (VIPs)} used by expansion rule A2.1 are those defined in [FTU05]. The \textit{Normalize} function used by rules A3 and B1 returns the normalized version of the given conjunction of literals.

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

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

Figure 4.17 shows the formalization of the violation analysis process, which is aimed to determine the set of \textit{repairs} for a failed \textit{CQC}\(_C\)-node. A \textit{repair} denotes a \textit{CQC}\(_C\)-node that is relevant for the violation. \textit{RepairsOfGoalComparison} and \textit{RepairsOfIc} return the corresponding set of repairs for the case in which the violation is in the goal and in a condition to

82
enforce, respectively. AvoidLiteral (AvoidIc) returns the nodes that are responsible for the presence of the given literal (constraint) in the goal (set of conditions to maintain). Finally, ChangeConstants returns the nodes in which the labeled constants that appear in the given comparison were used to instantiate certain variables.

4.2.3 Experimental Evaluation

We have performed a set of experiments to compare the efficiency of the CQC_E method as implemented in [FRTU08] (SVT_E tool) with respect to the original CQC method as implemented
in [TFU+04] (SVT tool). We have executed the experiments on an Intel Core 2 Duo, 2.16 GHz machine with Windows XP (SP2) and 2 GB RAM. Each experiment was repeated three times and we report the average of these three trials.

Each set of experiments checks whether a given view (or query) of the schema is satisfiable. Both the original CQC method and the extended version (the CQC_E method) are used to perform the corresponding tests.

The first set of experiments reported in Figure 4.18 (note the logarithmic scale) focus on the case in which satisfiability does not hold, i.e., the contents of the view is always empty. The tested query is actually encoding a mapping validation property (see Chapter 3), in particular, the mapping losslessness property. The mapped schemas are based on the relational schema of the Mondial database [Mon98]. The database schema that results from the reformulation of the property is as follows. It consists of three copies of the Mondial schema, say $S_1$, $S_2$ and $S_3$, each one with its primary keys, foreign keys and unique constraints. Additionally, a set $M^f = \{Q^k_1, \ldots, Q^k_{14}\}$ of queries is defined over each $S_k$ ($1 \leq k \leq 3$). These queries have the form $Q(X) \leftarrow T_1(X) \land \ldots \land T_n(X)$, where $n$ varies from 1 to 4, and $T_1, \ldots, T_n$ are tables randomly selected from the schema. Finally, we also add a set of constraints that relates $S_1$, $S_2$ and $S_3$. These constraints have the form of $Ic \leftarrow Q^k_1(X) \land \neg Q^m_1(X)$, where $Q^k_1$ is from $M^f$ and $Q^m_1$ is from $M^o$ ($1 \leq k, m \leq 3; k \neq m; 1 \leq j \leq 14$). The goal of each satisfiability test has the form of $G_0 = P(X) \land \neg P'(X)$, where $P$ is a query over $S_1$, and $P'$ is its equivalent over $S_2$.

Figure 4.18 shows that the use of the CQC_E method in this conjunctive setting results in a drastic reduction of running times. This is because its execution strategy helps to avoid the exploration of a high number of alternative CQC_E-(sub)derivations when exploring the CQC_E-tree. Moreover, Figure 4.18 shows that the introduction of either comparisons or negations results also in lower times when using the extended method than when using the original one. These negations and comparisons are added to each query $Q^k_1$ (and to query $P$). Therefore, $Q^k_1$ has the form of either $Q^k_1(X) \leftarrow T_1(X) \land \ldots \land T_n(X) \land \neg R_1(\bar{Y}_1) \land \neg R_2(\bar{Y}_2) \land \neg R_3(\bar{Y}_3)$ or $Q^k_1(X) \leftarrow T_1(X) \land$
... \land T_n(\bar{X}_n) \land Z_1 > k_1 \land Z_2 > k_2 \land Z_3 > k_3$, where $R_1, ..., R_3$ are tables randomly selected from the schema, and $k_1$, $k_2$ and $k_3$ are fresh constants.

The second set of experiments reported in Figure 4.19 focus on the case in which the satisfiability tests have a solution, i.e., the tested query admits a non-empty instance. The used schemas are like those from the previous set of experiments, but now we have $S_1$ and $S_2$ only. The goal of each satisfiability test has now the form of $G_0 = Q_{1,1}(\bar{X}_1) \land ... \land Q_{1,14}(\bar{X}_{14})$.

The graphics in Figure 4.19 show that, either when each query $Q_{1,i}^k$ has 3 negations or when each query $Q_{1,1}^k$ has 3 comparisons, the extended version of the method is faster than the original one. Although the computation of an explanation is not needed when the satisfiability test has a solution, Figure 4.19 shows that we can still take advantage of the efficiency improvement that results from using the CQC$_E$ method.

Table 4.1 shows the detail of the running times of both sets of experiments. Table 4.2 shows the main characteristics of the schemas.
As we already discussed in Section 3.2, the problem of checking the desirable properties of mappings presented in Chapter 3 on the class of mapping scenarios described in Chapter 2 is, unfortunately, undecidable. In order to deal with that, we propose to perform a termination test as a previous step to the check of each desirable property. Such a test is intended to detect situations in which the check of the target desirable property on the given mapping scenario is guaranteed to terminate. Such a test is obviously not complete, given the undecidability of the termination checking problem itself.

The termination test is based on the assumption that the target desirable property is going to be checked by means of the approach presented in Chapter 3, that is, by means of its reformulation in terms of query satisfiability and the subsequent application of the CQC method (or its extended version—see Section 4.2). More specifically, the termination test is to be applied after the reformulation in terms of query satisfiability and before the application of the CQC method.

We adapt to our mapping validation context the termination test that was presented in [QT08] in the context of reasoning on UML/OCL conceptual schemas. The test consists of two main tasks: the construction of a dependency graph of the constraints in the schema, and the analysis of the cycles in this graph.

We also extend the termination test in two ways.

First, the termination test, as presented in [QT08], is able to deal with a class of deductive rules and constraints that is very close to the one the CQC method handles, but it that does not allow more than one level of negation, i.e., negated literals must be about base predicates or about derived predicates whose deductive rules contain no negation. That is enough for the setting of [QT08], but in our mapping validation context, the database schema that results from the
reformulation of the desirable property checking in terms of query satisfiability may have more than one level of negation. Therefore, we propose and additional task: the materialization of all derived predicates in the schema, that is, the rewriting of the given schema into an equivalent one in which all predicates are base predicates. We refer to such a rewriting on a schema $S$ as the $b$-schema of $S$.

Second, the termination test consists of three sufficient conditions. The idea is that if each cycle in the dependency graph satisfies at least one of the conditions, then the reasoning on the schema is guaranteed to terminate. [QT08] studies termination for the case in which the cycles of the dependency graph are disjoint (i.e., vertex-disjoint). We extend this work by considering the case in which the cycles are overlapping (i.e., vertex-overlapping).

In the next sections, we firstly introduce the different stages of the termination test (i.e., computation of the $b$-schema, building of the dependency graph, and analysis of cycles) in an intuitive way, and then we provide their formalization (Section 5.4).

### 5.1 Dealing with Multiple Levels of Negation—Computing the B-Schema

The computation of the $b$-schema is the first stage of the termination test. This stage will allow us to rewrite a database schema whose constraints and deductive rules have multiple levels of negation into an equivalent schema with only one level of negation.

The input to this stage is the database schema that results from reformulating the current desirable property in terms of query satisfiability. The goal is to materialize all the derived predicates in the schema. We have to do that before we can construct the dependency graph and analyze the cycles.

The key point in the task of materializing derived predicates is replacing the deductive rules with constraints that keep the materialized predicates updated. Consider, for example, the following deductive rule:

$$q(X, Z) \leftarrow p(X, Y) \land r(Y, Z)$$

According to the semantics of deductive rules [Cla77, Llo87], the meaning of such a rule is stated explicitly by the formula:

$$\forall X, Z \ (q(X, Z) \leftrightarrow \exists Y (p(X, Y) \land r(Y, Z)))$$
That means we need two disjunctive embedded dependencies (DEDs) in order to keep the materialization of $q$ correctly updated, one for each direction of the implication:

$$q(X, Z) \rightarrow \exists Y (p(X, Y) \land r(Y, Z))$$
$$p(X, Y) \land r(Y, Z) \rightarrow q(X, Z)$$

For the sake of simplicity and uniformity, we will however prefer to have DEDs whose consequent is a disjunction of atoms, instead of DEDs whose consequent is a disjunction of conjunctions of atoms. This requirement is already fulfilled by the constraints that are present in the original schema (see Chapter 2). In order to enforce it in the new constraints, we could think of splitting $q(X, Z) \rightarrow \exists Y (p(X, Y) \land r(Y, Z))$ in two, as follows:

$$q(X, Z) \rightarrow \exists Y p(X, Y)$$
$$q(X, Z) \rightarrow \exists Y r(Y, Z)$$

However, the splitting is not correct, since it loses the correlation of $p$ and $r$ on $Y$. A more accurate way of doing it is to introduce an intermediate predicate—let us call it $q'$—that has one attribute for each distinct variable in the body of the original deductive rule. That is, the deductive rule $q(X, Z) \leftarrow p(X, Y) \land r(Y, Z)$ is to be rewritten into the following equivalent set of two rules, before starting the materialization:

$$q(X, Z) \leftarrow q'(X, Y, Z)$$
$$q'(X, Y, Z) \leftarrow p(X, Y) \land r(Y, Z)$$

Now, we can replace the two rules with the corresponding DEDs, and do the straightforward splitting without losing correctness:

$$q(X, Z) \rightarrow \exists Y q'(X, Y, Z)$$
$$q(X, Y, Z) \rightarrow q(X, Z)$$
$$q'(X, Y, Z) \rightarrow p(X, Y)$$
$$q'(X, Y, Z) \rightarrow r(Y, Z)$$
$$p(X, Y) \land r(Y, Z) \rightarrow q'(X, Y, Z)$$

The rewriting above is fine for the presented example, but in the general case, other issues may arise and need to be addressed. The first one is that the terms in the head of the deductive rule may not be all distinct variables, i.e., some of them may be constants, and some of the variables may appear more than once. For instance, consider the following deductive rule:

$$q_2(X, 10, X, Z) \leftarrow p(X, Y) \land r(Y, Z)$$

If we just apply the previous rewriting, we firstly obtain the next two rules:
\[
q_2(X, 10, X, Z) \leftarrow q'_2(X, Y, Z)
\]
\[
q'_2(X, Y, Z) \leftarrow p(X, Y) \land r(Y, Z)
\]

and then, the following DEDs:

\[
q_2(X, 10, X, Z) \rightarrow \exists Y q'_2(X, Y, Z)
\]
\[
q'_2(X, Y, Z) \rightarrow q_2(X, 10, X, Z)
\]
\[
q'_2(X, Y, Z) \rightarrow p(X, Y)
\]
\[
q_2(X, Y, Z) \rightarrow r(Y, Z)
\]
\[
p(X, Y) \land r(Y, Z) \rightarrow q'_2(X, Y, Z)
\]

The set of DEDs may look just fine, and it indeed keeps predicate \(q_2\) updated with respect to
insertions into \(p\) and \(r\), but it does not prevent facts such as, for instance, \(q_2(1, 2, 3, 4)\) from
existing in an instance of the database. According to the semantics of deductive rules, all facts
about \(q_2\) should fit the pattern \(q_2(X, 10, X, Z)\), i.e., they should be unifiable with the head of the
rule. The problem is that this is not guaranteed by the DEDs above.

To address the situation, we propose to extend the definition of predicate \(q'_2\) in such a way that
it has not only one attribute for each distinct variable in the body of the rule but also one attribute
for each term in the head of the original rule. Then, we can add equalities to the rule of \(q'_2\) to
enforce that each variable that corresponds to one of these new attributes must either be equal to a
certain constant, or be equal to a certain variable from the body of the rule:

\[
q_2(A, B, C, D) \leftarrow q'_2(A, B, C, D, X, Y, Z)
\]
\[
q_2(A, B, C, D, X, Y, Z) \leftarrow p(X, Y) \land r(Y, Z) \land A = X \land B = 10 \land C = X \land D = Z
\]

The set of DEDs that corresponds to the two rules is:

\[
q_2(A, B, C, D) \rightarrow \exists X, Y, Z \ q'_2(A, B, C, D, X, Y, Z)
\]
\[
q'_2(A, B, C, D, X, Y, Z) \rightarrow q_2(A, B, C, D)
\]
\[
q_2(A, B, C, D, X, Y, Z) \rightarrow p(X, Y)
\]
\[
q'_2(A, B, C, D, X, Y, Z) \rightarrow r(Y, Z)
\]
\[
q_2(A, B, C, D, X, Y, Z) \rightarrow A = X
\]
\[
q'_2(A, B, C, D, X, Y, Z) \rightarrow B = 10
\]
\[
q_2(A, B, C, D, X, Y, Z) \rightarrow C = X
\]
\[
q'_2(A, B, C, D, X, Y, Z) \rightarrow D = Z
\]
\[
p(X, Y) \land r(Y, Z) \rightarrow q'_2(X, 10, X, Z, X, Y, Z)
\]

Notice how we can easily deal with the new equalities in the last DED.
Another issue that needs to be addressed is that the body of a deductive rule may have negated literals. For instance:

\[
q_3(X, Z) \leftarrow s(X, Y, Z) \land \neg u(X) \land \neg w(Y, Z)
\]

In that case, a straightforward translation of the deductive rule into constraints would result in:

\[
\begin{align*}
q_3(X, Z) & \rightarrow s(X, Y, Z) \\
q_3(X, Z) & \rightarrow \neg u(X) \\
q_3(X, Z) & \rightarrow \neg w(Y, Z) \\
s(X, Y, Z) \land \neg u(X) \land \neg w(Y, Z) & \rightarrow q_3(X, Z)
\end{align*}
\]

However, DEDs cannot have negated literals, neither in the consequent nor in the premise. The case in which the negation is in the consequent can be amended by moving the negated literal into the premise (the negation will be lost in the process) and leaving some contradiction in the consequent (e.g., the comparison 1 = 0). DEDs \(q_3(X, Z) \rightarrow \neg u(X)\) and \(q_3(X, Z) \rightarrow \neg w(Y, Z)\) would thus become:

\[
\begin{align*}
q_3(X, Z) & \land u(X) \rightarrow 1 = 0 \\
q_3(X, Z) & \land w(Y, Z) \rightarrow 1 = 0
\end{align*}
\]

In the case in which the negated literals are in the premise, we can move them into the consequent (the negation will be lost in the process), where they will remain in disjunction with the literals already there. The DED \(s(X, Y, Z) \land \neg u(X) \land \neg w(Y, Z) \rightarrow q_3(X, Z)\) could be thus rewritten as:

\[
s(X, Y, Z) \rightarrow u(X) \lor w(Y, Z) \lor q_3(X, Z)
\]

Finally, there is one last issue to address. It refers to the case in which a single derived predicate has more than one deductive rule. As an example, consider:

\[
\begin{align*}
q_4(X, Z) & \leftarrow p(X, Y) \land r(Y, Z) \\
q_4(X, X) & \leftarrow u(X)
\end{align*}
\]

In order to ease the application of the previous rewritings, it is better if we just introduce a new intermediate predicate for each rule:

\[
\begin{align*}
q_4(X, Y) & \leftarrow q_4'(X, Y) \\
q_4(X, Y) & \leftarrow q_4''(X, Y) \\
q_4'(X, Z) & \leftarrow p(X, Y) \land r(Y, Z) \\
q_4''(X, X) & \leftarrow u(X)
\end{align*}
\]
Now, we can combine the rewritings that we just discussed, and modify each intermediate predicate (i.e., \( q_4' \) and \( q_4'' \)) independently and according to its characteristics:

\[
q_4(A, B) \leftarrow q_4'(A, Y, B)
\]
\[
q_4(A, B) \leftarrow q_4'(A, B, X)
\]
\[
q_4'(X, Y, Z) \leftarrow p(X, Y) \land r(Y, Z)
\]
\[
q_4''(A, B, X) \leftarrow u(X) \land A = X \land B = X
\]

The major difference with respect to the previous examples is the translation of the first two rules. According to the semantics of deductive rules, a fact about a derived predicate is true on a database instance if and only if it is “produced” by at least one of the predicate’s rules. Therefore, it is easy to see that \( q_4(A, B) \leftarrow q_4'(A, Y, B) \) and \( q_4(A, B) \leftarrow q_4''(A, B, X) \) can be translated into the following DEDs:

\[
q_4(A, B) \rightarrow \exists Y \; q_4'(A, Y, B) \lor \exists X \; q_4''(A, B, X)
\]
\[
q_4'(A, Y, B) \rightarrow q_4(A, B)
\]
\[
q_4''(A, B, X) \rightarrow q_4(A, B)
\]

After we have translated all the deductive rules of the given schema \( S \) into DEDs, the resulting schema is the b-schema of \( S \).

### 5.2 Dependency Graph

Once we have computed the b-schema, the next stage is the construction of the dependency graph. The dependency graph is intended to show the dependencies that exist between the integrity constraints of the schema.

The vertexes in the graph denote constraints that have the same ordinary literals in their premises (modulo renaming of variables). As an example, consider a b-schema with the following constraints:

\[
(c_1) \quad r(X, Y) \rightarrow \exists Z \; s(Z, Y) \lor \exists Z \; q(X, Z)
\]
\[
(c_2) \quad r(X, Y) \land Y > 5 \rightarrow \exists Z \; t(Y, Z, X)
\]
\[
(c_3) \quad s(X, Y) \land p(X, Z, U) \rightarrow \exists V \; t(X, Z, V)
\]
\[
(c_4) \quad t(X, Y, Z) \rightarrow \exists V \; r(Z, V)
\]

The dependency graph of this schema has 3 vertexes: \( \{c_1, c_2\}, \{c_3\} \) and \( \{c_4\} \).

In general, there is an edge from a vertex \( v_1 \) to a vertex \( v_2 \) if the constraints in \( v_1 \) may lead to the insertion of a new tuple and the violation of the constraints in \( v_2 \). The edge is labeled with the
5.3 Analysis of Cycles—Sufficient Conditions for Termination

The analysis of the cycles of the dependency graph will allow us to detect whether the query satisfiability check is guaranteed to terminate. Recall that we assume the satisfiability of the given query will be checked by means of the CQC method. Recall also that after the initial satisfaction of the query, the CQC method becomes an integrity maintenance process (see Chapter 2), that is, it keeps adding new tuples to the instance under construction until all constraints are satisfied, in which case the CQC method ends, or a violation that cannot be repaired is found, in which case another branch of the CQC-tree (i.e., the tree-shaped solution space that the CQC method explores) has to be considered. The analysis of cycles is aimed at detecting whether such an integrity maintenance process is guaranteed to terminate and, in particular, it guarantees that all branches of the CQC-tree will be finite.

From the point of view of the analysis, a cycle $C$ is a sequence

$$C = (v_1, r_1, v_2, r_2, \ldots, v_n, r_n, v_{n+1} = v_1)$$

where each $v_i$ denotes a vertex from the dependency graph—in particular, it denotes the conjunction of ordinary literals that is common to the premises of all the constraints in that

![Figure 5.1: A dependency graph.](image-url)
vertex—, and each $r_i$ denotes the label of the edge that goes from vertex $v_i$ to $v_{i+1}$. We refer to the literals in $v_i$ as potential violators, and we refer to $r_i$ as the repair of $v_i$ on $C$. We say that a vertex $v_i$ is violated on $C$ with respect to a given instance $I$ if the potential violators of $v_i$ are true on $I$ and the repair $r_i$ of $v_i$ on $C$ is false on $I$. Note that a repair $r_i$ is false on an instance $I$ if at least one of the literals in $r_i$ is false on $I$.

The analysis consists of three conditions to be evaluated on each cycle. Termination will be guaranteed if each cycle in the dependency graph satisfies at least one of the conditions.

Note that the conditions are sufficient but not necessary, i.e., we cannot say anything about the termination of integrity maintenance when there is some cycle in the dependency graph that does not satisfy any of the conditions. This is expected given the undecidability of the termination problem. In particular, the incompleteness of the termination test comes from two fronts. First, all branches of the CQC-tree being finite does not imply each cycle of the dependency graph will satisfy one of the termination conditions. Second, the CQC method is known to terminate when there is at least one finitely successful branch in the CQC-tree (i.e., a finite solution), even if the CQC-tree also contains infinite branches [FTU05].

It is worth noting that checking the termination conditions is a decidable process.

Note also that if the dependency graph has no cycles, then integrity maintenance will surely terminate.

In the next sections, we firstly review the termination conditions applicable to a dependency graph whose cycles are disjoint, and then discuss how to extend this work in order to deal with overlapping cycles.

**5.3.1 Condition 1—The Cycle Does Not Propagate Existentially Quantified Variables**

To illustrate what we mean by propagation of an existentially quantified variable, consider the following constraints:

\[(ic_1) \quad p(X, Y) \rightarrow \exists Z q(X, Z)\]
\[(ic_2) \quad q(A, B) \rightarrow \exists C p(A, C)\]

These constraints form a cycle:

\[C = (v_1 = p(X, Y), \; r_1 = \{\exists Z q(X, Z)\}, \; v_2 = q(A, B), \; r_2 = \{\exists C p(A, C)\})\]
The cycle propagates neither $Z$ nor $C$. In particular, $v_2$ does not propagate $Z$, since $B$ does not appear in $r_2$; and, similarly, $v_1$ does not propagate $C$, since $Y$ does not appear in $r_1$.

In general, Condition 1 holds for a given cycle $C$ if and only if no vertex from $C$ propagates the existentially quantified variables of the previous vertex in the cycle.

In the example above, it is easy to see that the insertion of a new $p$ or $q$ triggers an integrity maintenance process that terminates after one iteration on the cycle. In the general case, the intuition is that the constants a vertex propagates to its successor are either from the initial database instance, or have been obtained from the previous vertex; and since the previous vertex is not allowed to propagate existentially quantified variables (i.e., it cannot propagate “invented” values), then it must have obtained these constants from its own predecessor, and so on. If we follow this reasoning, we conclude that the constants each vertex in the cycle propagates are from the database obtained after the initial insertion that triggered the integrity maintenance process. Since the number of constants in this initial database instance is assumed to be finite and since the existentially quantified variables can always be unified with the values already in the database during a vertex’s violation check, then we can conclude that only a finite number of new tuples are generated as a consequence of the integrity maintenance process.

We will also show in Section 5.4.3 that there is a relationship between this Condition 1 and the well-known weak acyclicity property of sets of tuple-generating dependencies that guarantees termination of the chase [FKMP05].

5.3.2 Condition 2—There Is a Potential Violator that Is Not a Repair of Any Vertex

Roughly speaking, the aim of Condition 2 is to look for a constraint in the current cycle that has a potential violator that is not a repair of any vertex. The idea is that this will prevent the constraint from being violated further once it has already been violated and repaired a certain finite number of times.

In particular, Condition 2 holds for a cycle $C = (v_1, r_1, \ldots, v_n, r_n, v_{n+1} = v_1)$ whenever there is a vertex $v_i \in C$ such as the potential violators of $v_i$ include a literal $L$ about some predicate $P$ that does not appear in any $r' \in C$—that is, no new tuple about $P$ is created during the repair of the vertexes of $C$—, and the non-existentially quantified variables of $r_i$ appear all in literal $L$—that is, the number of distinct ways of repairing the violation of $v_i$ by means of $r_i$ is bound to the number of tuples that already exist in the database before starting the integrity maintenance of $C$ and that can be unified with $L$. 

95
As an example, consider the following constraints:

\[(ic_1) \quad p(X, Y) \land q(Z) \rightarrow \exists V \ r(Z, V)\]
\[(ic_2) \quad r(X, Y) \rightarrow \exists Z \ p(Y, Z)\]

The corresponding dependency graph is the following:

\[r_1 = \{\exists V \ r(Z, V)\}\]
\[r_2 = \{\exists Z \ p(Y, Z)\}\]

Notice that neither \(r_1\) nor \(r_2\) insert new tuples about \(q\), and that the non-existentially quantified variables of \(r_1\)—i.e., variable \(Z\)—are bound to the potential violator \(q(Z)\) of \(v_1\).

Let us assume the initial instance for the integrity maintenance process contains a finite number \(k\) of facts about \(q\). Then, after \(k\) violations of \(ic_1\) and its corresponding repairs, we can be sure that any subsequent insertion into \(p\) will not cause a violation of the constraint because we will be able to unify the consequent \(\exists V \ r(Z, V)\) with one of the tuples inserted by the previous repairs of \(ic_1\). The conclusion is that constraint \(ic_1\) can only be violated a finite number of times and the integrity maintenance process of the cycle is guaranteed to terminate.

Regarding the example from Section 5.2, the cycle that involves the constraints \(c_1\), \(c_3\) and \(c_4\) in Figure 5.1 satisfies Condition 2.

5.3.3 Condition 3—The Canonical Simulation of the Cycle Terminates Within One Iteration

The idea of Condition 3 is to simulate on a canonical database instance the integrity maintenance of the current cycle, and see if this simulation stops within one iteration through the cycle.

To illustrate this, consider the following example:

\[r_1 = \{\exists Y \ r(X, Y)\}\]
\[r_2 = \{\exists Z \ s(X, Y, Z)\}\]
\[r_3 = \{p(X)\}\]
\[r_4 = \{s(X, Y, Z)\}\]
Starting at $v_1$, the canonical simulation for the cycle in the graph would be as follows:

- **Initial canonical instance**: $\text{Sim}_0(v_1) = \{p(x)\}$
- **Instance after 1 step of integrity maintenance**: $\text{Sim}_1(v_1) = \text{Sim}_0(v_1) \cup \{r(x, y)\}$
- **Instance after 2 steps of integrity maintenance**: $\text{Sim}_2(v_1) = \text{Sim}_1(v_1) \cup \{s(x, y, z)\}$
- **Instance after 3 steps of integrity maintenance**: $\text{Sim}_3(v_1) = \text{Sim}_2(v_1) \cup \emptyset$

where $x$, $y$ and $z$ denote variables that have been “frozen” into constants.

Note that $\text{Sim}_2(v_1) = \text{Sim}_3(v_1)$. That means the simulation reaches a fix-point and ends within one iteration of the cycle. Similar results are obtained when starting at vertexes $v_2$ and $v_3$.

In general, Condition 3 simulates, for each vertex in the cycles, all the distinct canonical sequences of violations and repairs that start at that vertex (there is a finite number of them), e.g., the one above. The intuition is that any real sequence of violation and repairs that results from the execution of the CQC method (its integrity maintenance phase) has a “canonical representative” in the simulation; therefore, if all the sequences in the simulation are finite, the real sequence generated by the CQC method should be finite too—see Section 5.4.3 for a more detailed definition and the formal proof.

Regarding the example from Section 5.2, the two cycles in Figure 5.1 that involve the constraints $c_1$, $c_2$ and $c_4$ do satisfy Condition 3.

### 5.3.4 Overlapping Cycles

In this section we study the application of the previous termination conditions to the case in which the cycles in the dependency graph are overlapping.

Recall that the idea of the test is that if each cycle in the dependency graph satisfies at least one of the termination conditions, then termination is guaranteed on the whole schema.

In order to be able to apply the test in our setting, we need termination to be preserved by the overlapping of cycles, where each cycle may satisfy a different termination condition. We will show that termination is already preserved by all the combinations of overlapping cycles except two: the overlapping of cycles that satisfy Condition 1 and Condition 3, respectively; and the overlapping of cycles that satisfy Condition 2.

#### 5.3.4.1 Condition 1 and Condition 3

In Figure 5.2, we provide a counterexample for the case in which the dependency graph contains overlapping cycles some of which satisfy Condition 1 but not Condition 3 and some others satisfy
Condition 3 but not Condition 1. The counterexample consists of three cycles: two cycles satisfy Condition 1 and the remaining cycle satisfies Condition 3. The integrity maintenance process does indeed terminate when applied to each cycle individually, but it may not terminate when applied to the whole schema.

The two cycles that meet Condition 1 are:

\[ C_1 = (v_1, r_1, v_2, r_2, v_1) \]
\[ C_2 = (v_3, r_3, v_5, r_5, v_3) \]

It is easy to see that neither \( C_1 \) nor \( C_2 \) propagate existentially quantified variables.

The cycle that satisfies Condition 3 is:

\[ C_3 = (v_1, r_1, v_3, r_4, v_4, r_3, v_1) \]

We can see that the canonical simulation from Section 5.3.3 does indeed terminate within one iteration of \( C_3 \):

\[ Sim_2(v_1) = Sim_3(v_1) = \{A(x, y, x), D(y, z, x), B(y, x, u), C(y, x, u), E(x, u, y)\} \]
\[ Sim_3(v_3) = Sim_4(v_3) = \{B(x, y, z), C(x, y, z), E(y, z, x), A(y, x, u), D(x, u, y)\} \]
\[ Sim_4(v_4) = Sim_4(v_4) = \{C(x, y, z), A(y, x, u), D(x, u, y), B(x, y, u_2), C(x, y, u_2), E(y, u_2, x)\} \]

Consider now the instance \( I = \{D(0, 1, 2)\} \). Performing integrity maintenance on \( I \) with the whole dependency graph from Figure 5.2 into account may produce an infinite instance; for example, the following one:

\[ v_1 = A(X, Y, Z) \]
\[ r_1 = \{D(Y, Z, X), \exists U B(Y, X, U)\} \]
\[ v_2 = D(X, Y, Z) \]
\[ r_2 = \{A(Y, Z, X)\} \]
\[ v_3 = B(X, Y, Z) \]
\[ r_3 = \{C(X, Y, Z), E(Y, Z, X)\} \]
\[ v_4 = C(X, Y, Z) \]
\[ r_4 = \{\exists U A(Y, X, U)\} \]
\[ v_5 = E(X, Y, Z) \]
\[ r_5 = \{B(Y, Z, X)\} \]
\{D(0, 1, 2), A(1, 2, 0), B(2, 1, 3), E(1, 3, 2), B(3, 2, 1), A(2, 3, 4), D(3, 4, 2),
A(4, 2, 3), B(2, 4, 5), E(4, 5, 2), B(5, 2, 4), C(5, 2, 4), A(2, 5, 6), D(5, 6, 2), A(6, 2, 5), \ldots\}

Note that the instance above corresponds to an infinite sequence of violations and repairs that
goes through the following path:

\[v_2, r_2, v_1, r_1, v_3, r_3, v_5, r_5, v_4, r_4, v_1, r_1, v_2, \ldots\]

In the light of this counterexample, we will exclude the combination of Condition 1 and
Condition 3 as a guarantee for termination in the case of overlapping cycles.

5.3.4.2 **Condition 2**

Similarly as we did in the previous section, we can also come up with a counterexample for the
overlapping of cycles that satisfy Condition 2. In this case, however, instead of excluding the
overlapping of cycles that satisfy Condition 2 from the termination test, we will provide an
alternative definition for Condition 2 that will be a sufficient condition for the termination of
integrity maintenance in the presence of overlapping cycles.

Let \(G\) be the following dependency graph:

There are two cycles in \(G\):

\[C_1 = (v_1, r_1, v_2, r_2, v_4, r_4, v_5, r_5, v_1)\]
\[C_2 = (v_1, r_1, v_3, r_3, v_4, r_4, v_5, r_5, v_1)\]

Both cycles satisfy Condition 2. Predicate \(E\) does not appear in any repair from \(C_1\), and vertex
\(v_4\), which is part of \(C_1\), has a potential violator with predicate \(E\), i.e., \(E(X, Y)\), where \(X\) and \(Y\) are
precisely the variables that appear in \(r_4\). Similarly, predicate \(D\) does not appear in any repair from
\(C_2\), and vertex \(v_4\), which is also part of \(C_2\), has potential violator \(D(X, Y)\).
It is true that performing integrity maintenance on either $C_1$ or $C_2$, individually, is a finite process. However, it can be seen that, when the whole schema is considered, the integrity maintenance process may not end. As an example, consider the instance $I = \{A(0, 1, 2)\}$. Performing integrity maintenance on $I$ with the constraints represented in $G$ may produce an infinite instance such as:

$$\{A(0, 1, 2), B(2, 0), C(2, 0), D(2, 0), E(2, 0), F(2, 0), A(2, 0, 3), \ldots\}$$

The problem is that Condition 2 requires the existence of a potential violator that is not a repair of any vertex of the current cycle, but it allows the potential violator to be part of the repair of a vertex from another cycle. In the example, cycle $C_1$ has a potential violator $E(X, Y)$ that does not appear in the repairs from $C_1$ but does appear in the repairs from $C_2$. Similarly, potential violator $D(X, Y)$ does not appear in the repairs from $C_2$ but it does appear in the repairs from $C_1$. Therefore, when these two cycles are considered together, the reason that prevented each cycle from looping forever—i.e., the fact that the corresponding potential violator receives no new insertion during the integrity maintenance process—is no longer true.

In order to address this problem, which arises when overlapping cycles are considered, we propose an alternative definition for Condition 2 in which the potential violator $L_j$ is required to be about a predicate that does not appear in the repair of any vertex in the whole dependency graph (instead of in any repair of the current cycle). Note that the restriction that we discussed in Section 5.3.2 regarding the variables that appear in $L_j$ applies also here.

Summarizing, the alternative definition that we propose for Condition 2 in the presence of overlapping cycles is as follows: Condition 2 holds for a cycle $C$ if and only if there is a vertex $v_i$ in $C$ that has a potential violator $L_j = p(X)$ such that a literal about predicate $p$ does not appear in the repair of any vertex in the dependency graph and all the non-existentially quantified variables in the repair $r_i$ of $v_i$ on $C$ appear in $p(X)$—see the formal proof for correctness in Section 5.4.3.

5.4 Formalization

In this section, we provide the formal definitions and proofs for the three stages of the termination test.
5.4.1 Schema Preprocess

**Definition 5.1 (B-Schema).** Let $S = (PD_S, DR_S, IC_S)$ be a database schema. The b-schema of $S$ is $BS = (PD_{BS}, \emptyset, IC_{BS})$, 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 $q_i = (q(X) \leftarrow L_1 \land \ldots \land L_k) \in DR_S$, the following is true:

- $PD_{DR}$ contains the predicate definition $q(A_1, \ldots, A_t, B_1, \ldots, B_n)$, where $t$ is the number of terms in the head of $q$, i.e., $t = |X|$, and $n$ is the number of distinct variables in $L_1 \land \ldots \land L_k$.

- $IC_{DR}$ contains the integrity constraints:
  
  $q(\bar{A}_i, \bar{B}_i) \rightarrow q(\bar{A}_i)$
  
  $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{j_1} = k_1$
  
  ..., $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{j_u} = k_u$
  
  $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{j_1} = B_{h_1}$
  
  ..., $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{j_v} = B_{h_v}$
  
  $q(\bar{Z}) \rightarrow \exists \bar{B}_1 q_1(\bar{Z}, \bar{B}_1) \lor \ldots \lor \exists \bar{B}_m q_m(\bar{Z}, \bar{B}_m)$
  
  where $k_1, \ldots, k_u$ are the constants in $\bar{X}$; they appear in the positions $j_1, \ldots, j_u$; $B_{h_1}, \ldots, B_{h_v}$ are the variables in $\bar{B}_i$ that correspond to variables in $L_1 \land \ldots \land L_k$ that appear in $X_i$ with positions $g_1, \ldots, g_v$, respectively; $\bar{Z}$ denotes a list of $t$ distinct variables; and $q_1, \ldots, q_m$ 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 $L_1, \ldots, L_k$ are positive literals, $IC_P$ contains the constraints:
  
  $L_1 \land \ldots \land L_k \rightarrow q(\bar{A}_i, \bar{B}_i)$
  
  $q(\bar{A}_i, \bar{B}_i) \rightarrow L_1$
  
  ..., $q(\bar{A}_i, \bar{B}_i) \rightarrow L_k$

- If $L_1 \land \ldots \land L_k = P_1 \land \ldots \land P_r \land \neg N_1(\bar{Z}_1) \land \ldots \land \neg N_s(\bar{Z}_s)$ and $s > 1$, $IC_N$ contains the constraints:
  
  $P_1 \land \ldots \land P_r \land N_1(\bar{Z}_1) \land \ldots \land N_s(\bar{Z}_s) \land q(\bar{A}_i, \bar{B}_i)$
  
  $q(\bar{A}_i, \bar{B}_i) \rightarrow P_1$
  
  ..., $q(\bar{A}_i, \bar{B}_i) \rightarrow P_r$
  
  $q(\bar{A}_i, \bar{B}_i) \land N_1(\bar{Z}_1) \rightarrow 1 = 0$
  
  ..., $q(\bar{A}_i, \bar{B}_i) \land N_s(\bar{Z}_s) \rightarrow 1 = 0$  \[\square\]
Definition 5.2 (b-Instance). Let $I_S$ be a database instance. The b-instance of $I_S$ is

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

where $\text{Facts}(DR, I) = \{ q(\bar{x}) \sigma, q(\bar{x}, \bar{y}) \sigma \mid q_i = (q(\bar{x}) \leftarrow L_1 \land \ldots \land L_k) \in DR, \ \bar{y} \ \text{denotes} \ \text{the} \ \text{variables} \ \text{in} \ L_1 \land \ldots \land L_k, \ \text{and} \ \sigma \ \text{is a ground substitution such that} \ I = (L_1 \land \ldots \land L_k)\sigma \}. \ \square$

Lemma 5.1. 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 5.2. 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 $S = (PD_S, DR_S, IC_S)$. Let $I_{BS}$ be the b-instance of $I_S$. By Lemma 1, $I_{BS}$ is an instance of the b-schema $BS = (PD_{BS}, \emptyset, IC_{BS})$ 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 $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{j_u} = k_u$ and $q(\bar{A}_i, \bar{B}_i) \rightarrow A_{q_v} = B_{b_v}$ state that, in the materialized relation $q(\bar{x}, \bar{y})$, $\bar{x}$ 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 $A_{j_u} = k_u$) or a variable from the body of $q_i$ (i.e., consequent is $A_{q_v} = B_{b_v}$). We can be sure that these constraints hold on $I_{BS}$ because of the definition of $\text{Facts}(DR_S, I_S)$, which adds $q(\bar{x})\sigma$ and $q(\bar{x}, \bar{y})\sigma$ to $I_{BS}$ for each ground substitution $\sigma$ that makes the body of $q$ true on $I_S$.

Still in $IC_{DR}$, the constraints in the form of $q(\bar{A}_i, \bar{B}_i) \rightarrow q(\bar{A}_i)$ and $q(\bar{Z}) \rightarrow \exists \bar{B}_1, q_1(\bar{Z}, \bar{B}_1) \ldots \exists \bar{B}_m, q_m(\bar{Z}, \bar{B}_m)$ state that there must be one fact $q(\bar{x})\sigma$ about the derived predicate $q$ for each instantiation $q(\bar{x}, \bar{y})\sigma$ 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 $\text{Facts}$ includes both a fact $q(\bar{x})\sigma$ about $q$ and a fact $q(\bar{x}, \bar{y})\sigma$ about $q_i$ for each instantiation $\sigma$ 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 exist, and if the tuple exists, the literals in the body must all be true. This obviously holds in $IBS$ 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 $P_1 \land \ldots \land P_r \rightarrow N_1(\bar{Z}_1) \lor \ldots \lor N_s(\bar{Z}_s) \lor q_i(\bar{A}_i, \bar{B}_i)$, the negated literals have been moved into the consequent, and in $q_i(\bar{A}_i, \bar{B}_i) \land N_s(\bar{Z}_s) \rightarrow 1=0$ the negated literal has been moved into the premise. As above, it follows immediately from the definition of $Facts$.

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

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

**Theorem 5.1.** 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 $IS$ of $S$ that contains at least one fact about $Q$. By Lemma 2, the b-instance $IBS$ of $IS$ is a consistent instance of the b-schema. By construction of $IBS$, we know that $IBS$ contains a fact $q(\bar{X}) \sigma$ for each instantiation $\sigma$ that makes true the body of a deductive rule $(q(\bar{X}) \leftarrow L_1 \land \ldots \land L_k) \in DR_S$ on $IS$. Since $Q$ is satisfiable on $IS$, there is at least one of such instantiations, i.e., $IBS$ contains at least one fact about $Q$. Therefore, instance $IBS$ 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 $IBS$ of the b-schema with at least one fact about $Q$. By Lemmas 1 and 2, there is a consistent instance $IS$ of $S$ such that $IBS$ is its b-instance. Since $IBS$ contains a fact $q(\bar{X}) \sigma$ about $Q$, and by definition of b-instance, there must be an instantiation $\sigma$ and a deductive rule $(q(\bar{X}) \leftarrow L_1 \land \ldots \land L_k) \in DR_S$ such that $(L_1 \land \ldots \land L_k) \sigma$ is true on $IS$. Therefore, $Q$ has a non-empty answer on $IS$, i.e., $IS$ exemplifies that $Q$ is satisfiable on $S$. ■
5.4.2 Dependency Graph

Definition 5.3 (Potential Violation and Repair). A literal $p(\overline{X})$ is a potential violation of an integrity constraint $ic \in IC_{ST}$ if it appears in the premise of the constraint. We assume $\overline{X}$ is a list of distinct variables; otherwise, a constant $k$ (a repeated variable $Y$) can be replaced by a fresh variable $Z$ plus the equality $Z = k$ ($Z = \overline{X})$. We denote by $PV(ic)$ the set of potential violations of $ic$. There is a repair $RE_{k}(ic) = \{L_i\}$ for each ordinary literal $L_i$ in the consequent of $ic$. □

Definition 5.4 (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 $RE_{k}(ic_i)$ from $ic_i$ to $ic_j$ if there exists $p(\overline{X}), p(\overline{Y})$ such that $p(\overline{X}) \in RE_{k}(ic_i)$ and $p(\overline{Y}) \in PV(ic_j)$. 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 ordinary literals in their premises (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 $L_{i,1} \lor ... \lor L_{i,r_i}$ be the consequents of the constraints in $SP$; there is a repair $RE_{k}(ic') = \{L_{i,j_1}, ..., L_{i,j_s}\}$ for each combination $j_1, ..., j_s$ with $1 \leq j_1 \leq r_1, ..., 1 \leq j_s \leq r_s$. The incoming and outgoing arcs of $ic'$ in the graph are computed as defined above. □

5.4.3 Analysis of Cycles

Definition 5.5 (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 $ic_1, ..., ic_n$ are vertexes (i.e., constraints) from the dependency graph and $r_i$ denotes the label of an arc from $ic_i$ to $ic_{i+1}$. □

Before starting with the termination conditions, let us address the case in which the b-schema has no cycles.

Proposition 5.1. Let $S$ be a b-schema with no cycles in its dependency graph. Then, checking the satisfiability of a query $Q$ on $S$ with the CQC method is a finite process.

Proof. Let $G$ be the dependency graph of $S$. Let us assume $G$ has no cycles. Let us suppose that checking the satisfiability of a certain query $Q$ on $S$ with the CQC method does not terminate. Recall that the CQC method is, after the initial query satisfaction phase, an integrity maintenance
process (see Chapter 2). Then, if the CQC method does not terminate, that means there exists an infinite sequence of violations and repairs \( \text{seq} = (\text{ici}_1, r_1(\text{θ}_1 \cup \text{θ}_1'), \text{ici}_2, \text{θ}_2(\text{θ}_2 \cup \text{θ}_2'), \ldots) \), where each \( \text{θ}_i \) is a ground substitution that causes the violation of \( \text{ici}_i \), and, if \( i > 1 \), \( \text{PV}(\text{ici}_i) \text{θ}_i \) contains at least one tuple from the previous repair, i.e., \( \text{PV}(\text{ici}_i) \text{θ}_i \cap r_1(\text{θ}_1 \cup \text{θ}_1') \neq \emptyset \). Substitution \( \text{θ}_i' \) assigns a constant to each existentially quantified variable in \( r_1 \), according to the Variable Instantiation Patterns (VIPs) (see Chapter 2). Since \( S \) is finite, it contains a finite number of constraints. Therefore, \( \text{seq} \) being infinite implies that there must be a constraint \( \text{ici}_i \) from \( S \) that occurs more than once in \( \text{seq} \). Let us consider a fragment of \( \text{seq} \) between two consecutive occurrences of \( \text{ici}_i \), namely \( \text{seq}' = (\text{ici}_i \text{θ}_i, r_1(\text{θ}_1 \cup \text{θ}_1'), \ldots, \text{ici}_i \text{θ}_i) \), and let us do induction on the length of \( \text{seq}' \). The base case is that in which there is no other constraint in \( \text{seq}' \) but \( \text{ici}_i \), that is, \( \text{seq}' = (\text{ici}_i \text{θ}_i, r_1(\text{θ}_1 \cup \text{θ}_1'), \text{ici}_i \text{θ}_i) \). In this case, \( C = (\text{ici}_i, r_1, \text{ici}_i) \) must be a cycle from \( G \), and we have reached a contradiction with \( G \) being acyclic. The inductive case is that in which there are other constraints in \( \text{seq}' \) besides \( \text{ici}_i \). There are two possibilities: either the constraints in \( \text{seq}' \) besides \( \text{ici}_i \) are all different or there is some constraint \( \text{ici}_j \) that appears more than once. If all constraints in \( \text{seq}' \) different from \( \text{ici}_i \) are distinct, then \( C = (\text{ici}_i, r_1, \ldots, \text{ici}_i) \) must be a cycle from \( G \) and we have again reached a contradiction. If there is some constraint \( \text{ici}_j \) different from \( \text{ici}_i \) that appears at least twice in \( \text{seq}' \), then \( \text{seq}' = (\text{ici}_i \text{θ}_i, \ldots, \text{ici}_i \text{θ}_i, r_1(\text{θ}_1 \cup \text{θ}_1'), \ldots, \text{ici}_i \text{θ}_i, \ldots, \text{ici}_i \text{θ}_i) \). By hypothesis of induction, the sequence between the two consecutive occurrences of \( \text{ici}_j \), i.e., \( \text{seq}'' = (\text{ici}_j \text{θ}_j, r_1(\text{θ}_1 \cup \text{θ}_1'), \ldots, \text{ici}_i \text{θ}_i) \), must go through some cycle \( C \), and we reach a contradiction. ■

Let us now formalize the termination conditions.

**Definition 5.6 (Condition 1).** We say a cycle \( C = (\text{ici}_1, r_1, \ldots, \text{ici}_n, \text{ici}_n+1 = \text{ici}_1) \) satisfies **Condition 1** if for all constraint \( \text{ici}_i \) in \( C \) and for all pair of literals \( p(X_1, ..., X_n) \in r_1 \) and \( p(Y_1, ..., Y_m) \in \text{PV}(\text{ici}_{i+1}) \), variable \( X_k \) being existentially quantified implies \( Y_k \not\in \text{vars}(r_{i+1}) \), \( 1 \leq k \leq m \). □

**Theorem 5.2.** Let \( S \) be a b-schema. If all the cycles in the dependency graph of \( S \) satisfy Condition 1, then checking the satisfiability of a query \( Q \) on \( S \) with the CQC method is a finite process.

**Proof.** Let \( G \) be the dependency graph of \( S \). Let us assume all cycles in \( G \) satisfy Condition 1. Let us suppose that checking the satisfiability of a certain query \( Q \) on \( S \) with the CQC method does not terminate. That means there must exists an infinite sequence \( \text{seq} \) of violations and repairs (see proof of Proposition 5.1), \( \text{seq} = (\text{ici}_1 \text{θ}_1, r_1(\text{θ}_1 \cup \text{θ}_1'), \text{ici}_2 \text{θ}_2, r_2(\text{θ}_2 \cup \text{θ}_2'), \ldots) \), where each \( \text{θ}_i \) is a ground substitution that causes the violation of \( \text{ici}_i \); if \( i > 1 \), \( \text{PV}(\text{ici}_i) \text{θ}_i \) contains at least one tuple
from the previous repair, i.e., $\text{PV}(ic_i)\theta_i \cap r_{\lambda,1}(\theta_{\lambda,1} \cup \theta_{\lambda,1}') \neq \emptyset$; and substitution $\theta_i'$ assigns a constant to each existentially quantified variable in $r_i$, according to the VIPs. Since the number of constraints in $S$ is finite, there must be some constraint $ic_i$ that is violated an infinite number of times in $\text{seq}$. Let $ic_i\theta_a$ be the first occurrence of $ic_i$ in $\text{seq}$, and let $\text{seq}' = (ic_i\theta_a, r_i(\theta_a \cup \theta_a'), \ldots)$ be the (infinite) suffix of $\text{seq}$ that begins with this first occurrence of $ic_i$. We know each constraint in $\text{seq}'$ must belong to some cycle from $G$; otherwise, $\text{seq}'$ could not go through $ic_i$ an infinite number of times. Given that all cycles in $G$ satisfy Condition 1, then we also know that no constraint from $\text{seq}'$ propagates the existentially quantified variables of the previous constraint. Let $I$ be the instance on which the first occurrence of $ic_i$ is evaluated. The fact that no constraint from $\text{seq}'$ propagates “invented” values means that the non-existentially quantified variables in the repair of each occurrence of $ic_i$ in $\text{seq}'$ are unified with constants from $I$. Since $I$ is finite, there is only a finite number of possible unifications for these non-existentially quantified variables in the context of the whole $\text{seq}'$. Given also that constraint $ic_i$ has only a finite number of distinct repairs (note that different occurrences of $ic_i$ in $\text{seq}'$ may have different repairs, e.g., $r_i, r_i', r_i'', \ldots$) (see Definition 5.3), we can conclude that after $ic_i$ has been violated and repaired a finite number of times, $ic_i$ will not be violated again. We have thus reached a contradiction with $ic_i$ being violated infinite times in $\text{seq}'$. ■

We show next that there is a connection between Condition 1 and the well-known property of weak acyclicity, which is a property of sets of tuple-generating dependencies that guarantees termination of the chase [FKMP05].

**Proposition 5.2.** Let $S$ be a b-schema with integrity constraints $\text{IC}$. If the constraints in $\text{IC}$ have neither arithmetic comparisons nor disjunctions and are in the form of tuple-generating dependencies, then all cycles in the dependency graph of $S$ satisfying Condition 1 implies that $\text{IC}$ is weakly acyclic and chasing any instance $I$ of $S$ with $\text{IC}$ is a finite process.

**Proof.** Let $G$ be the dependency graph of $S$ as defined in Definition 5.4. Let $G_{\text{chase}}$ be the dependency graph of $\text{IC}$ as defined in [FKMP05]. Recall that $G_{\text{chase}}$ has one vertex for each position $(R, A)$, where $R$ is a relation and $A$ an attribute from the schema. Given a tgd $ic \in \text{IC}$, $ic = \phi \rightarrow \psi$, there is an edge from position $\pi_1$ to position $\pi_2$ if there is a variable $X$ in $\phi$ with position $\pi_1$ that also appears in $\psi$ with position $\pi_2$; there is a special edge from position $\pi_1$ to position $\pi_2$ if there is a variable $X$ in $\phi$ with position $\pi_1$ that also appears in some position of $\psi$ and there is an existentially quantified variable $Y$ in $\psi$ with position $\pi_2$. The set $\text{IC}$ of tgds is weakly acyclic if its dependency graph $G_{\text{chase}}$ has no cycle going through a special edge. Now, let us suppose all cycles
in $G$ satisfy Condition 1 and $IC$ is not weakly acyclic. Then, $G_{\text{chase}}$ has a cycle $C_{\text{chase}}$ going through a special edge. Let $\pi_1, \pi_2, \ldots, \pi_n$ be the positions in $C_{\text{chase}}$. There is an edge from each position $\pi_i$ to the next position $\pi_{i+1}$. We know each edge $(\pi_i, \pi_{i+1})$ is caused by a constraint $ic_i \in IC$. Let $ic_1, ic_2, \ldots, ic_{m-1}$ the constraints from $IC$ responsible for the edges in $C_{\text{chase}}$. Let us assume without loss of generality that the special edge in $C_{\text{chase}}$ is the one that goes from $\pi_1$ to $\pi_2$. That means constraint $ic_2$ propagates at least one existentially quantified variable of $ic_1$. Constraints $ic_1$ and $ic_2$ must belong to a cycle $C \in G$; otherwise, $C_{\text{chase}}$ would not be a cycle. Since all cycles from $G$ satisfy Condition 1, $ic_2$ should not propagate the existentially quantified variables of $ic_1$, that is, we have reached a contradiction.  

**Definition 5.7 (Condition 2)**. We say a cycle $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ satisfies Condition 2 if $C$ contains a constraint $ic_i$ that satisfies the following. Let $UR_{\text{graph}} = \{ p | p \bar{X} \in \text{RE}(ic_i), ic_i \in \text{dependency graph} \}$ be the union of the repairs of the constraints in the dependency graph, and let $UV_i = \{ q | q(\bar{Y}) \in \text{PV}(ic_i) \}$ be the union of the potential violators of $ic_i$. Then,

1. $UV_i \not\subseteq UR_{\text{graph}}$, where the literals in $\text{PV}(ic_i)$ whose predicates belong to $UV_i$ but not to $\text{UR}_{\text{graph}}$ are $\{L_1, ..., L_k\}$, and
2. $\text{vars}(r_i) \subseteq \text{vars}(\{L_1, ..., L_k\})$, where $\text{vars}(r_i)$ denotes the non-existentially quantified variables of $r_i$. □

**Lemma 5.3.** Let $C$ be a cycle that satisfies Condition 2, and let $ic_i \in C$ be the distinguished constraint Definition 5.7 refers to. Then, integrity maintenance on a finite instance can only violate $ic_i$ a finite number of times.

**Proof.** Let us suppose that integrity maintenance on a certain finite instance $I$ violates $ic_i$ an infinite number of times. Let $\sigma_1, ..., \sigma_m$ be the $m$ possible ground substitutions such that $I \models (L_1 \land \ldots \land L_k)\sigma_j$, $1 \leq j \leq m$. Let $I'$ be instance $I$ after $m$ iterations of the integrity maintenance process. Let us assume without loss of generality that $I'$ violates $ic_i$ (otherwise, we keep doing integrity maintenance until we find the next violation of $ic_i$). We know $ic_i = L_1 \land \ldots \land L_k \land L_{k+1} \land \ldots \land L_n \rightarrow L_{n+1} \lor \ldots \lor L_{n+m}$, where $\{L_1, ..., L_k\}$ are the literals Definition 5.7 refers to. Let $\delta$ be a ground substitution for the non-existentially quantified variables of $ic_i$ such that $I' \models (L_1 \land \ldots \land L_k \land L_{k+1} \land \ldots \land L_n)\delta$ and $I' \not\models (L_{n+1} \lor \ldots \lor L_{n+m})\delta$. By point (1) of Definition 5.7, we know $\exists j$, $1 \leq j \leq m$, such that $\delta = \sigma_j \cup \delta'$ and $I' \models (L_1 \land \ldots \land L_k)\sigma_j$. By point (2), $\text{vars}(L_{n+1} \lor \ldots \lor L_{n+m}) \subseteq \text{vars}(L_1 \land \ldots \land L_k)$. Therefore, $I' \not\models (L_{n+1} \lor \ldots \lor L_{n+m})\sigma_j$. However, since $ic_i$ has already been violated and repaired
m times, $I' \equiv \{L_{n+1}\sigma_j, \ldots, L_{n+m}\sigma_j \mid 1 \leq j \leq m\}$, which means $I' \models (L_{n+1} \lor \ldots \lor L_{n+m})\sigma_j$. So, we have reached a contradiction. ■

**Theorem 5.3.** Let $S$ be a b-schema. If all the cycles in the dependency graph of $S$ satisfy Condition 2, then checking the satisfiability of a query $Q$ on $S$ with the CQC method is a finite process.

**Proof.** Let $G$ be the dependency graph of $S$. Let us assume all cycles in $G$ satisfy Condition 2. Then, each cycle $C$ from $G$ has a constraint $ic_i$ to which Lemma 5.3 can be applied. That is, each cycle $C$ from $G$ has a constraint $ic_i$ that can only be violated a finite number $k$ of times. After $k$ violations and repairs of $ic_i$, there is no point on keep checking it, so we can remove $ic_i$ and continue the integrity maintenance process with the remaining constraints. Since this applies to all cycles in $G$, that leaves us with an acyclic schema; and we know by Proposition 5.1 that the CQC method (which, after the initial query satisfaction phase, becomes an integrity maintenance process) is guaranteed to terminate on an acyclic b-schema. ■

**Definition 5.8 (Canonical Integrity Maintenance Step).** Given an instance $I$, a constraint $ic_i$, and a repair $r_i$ for $ic_i$, a canonical integrity maintenance step is defined as follows:

$$\text{Maint}(I, ic_i, r_i) = I \cup r_i \theta_j$$

where $\theta_j = \delta_j \cup \delta'_j$ is one of the $m$ possible instantiations, $m \geq 0$, such that $I \models \text{PV}(ic_i)\delta_j$, and $I \not\models r_i \delta_j$, and $\delta'_j$ instantiates each existentially quantified variable of $r_i$ with a fresh constant, and $\text{PV}(ic_i)\delta_j$ contains at least one fact inserted by the previous canonical integrity maintenance step (if the current is not the first step). ■

**Definition 5.9 (Canonical Simulation of a Cycle).** Given a cycle $C = (ic_1, r_1, \ldots, ic_n, r_n, ic_{n+1} = ic_1)$ we define a canonical simulation of $C$ that starts at constraint $ic_j$ as follows (note that, when $j > 0$, $ic_{j-1}$ denotes the current constraint and $ic_j$ denotes the next constraint):

$$\text{Sim}_0(ic_i) = \text{PV}(ic_i)\sigma_0$$

$$\text{Sim}_j(ic_i) = \text{Maint}(\text{Sim}_{j-1}(ic_i), ic_{j-1}, r_{j-1}) \cup \text{PV}(ic_{j+1})\sigma_u \quad j > 0$$

where

(i) Substitution $\sigma_0$ assigns a fresh constant to each variable.

(ii) There is a substitution $\sigma_j$ for each $L \subseteq \text{Maint}(\text{Sim}_{j-1}(ic_i), ic_{j-1}, r_{j-1})$ and $M \subseteq \text{PV}(ic_{j+1})$ such that $L$ contains at least one tuple inserted by the previous canonical integrity maintenance step and
there exists a most general unifier $\delta$, of $L$ and $M$. Substitution $\sigma = \delta \cup \sigma'$, where $\sigma'$ assigns a fresh constant to each variable in $PV(\text{ici})\delta - M\delta$. Substitution $\sigma_n$ is one out of the $\sigma_i$'s.

Definition 5.10 (Condition 3). We say a cycle $C = (ic_1, r_1, ..., ic_n, r_n, ic_{n+1} = ic_1)$ satisfies Condition 3 if for each constraint $ic_1 \in C$, there exists a constant $k$, $1 \leq k \leq n$, such that all canonical simulations that start at $ic_1$ reach a fix-point in at most $k$ steps, that is, $Sim_k(\text{ici}) = Sim_{k+1}(\text{ici})$.

The simulation begins with the construction of a canonical instance that “freezes” each variable from the premise of $ic_1$ into a constant (point (i)). Then, $Sim$ evaluates the premise of the constraint, disregarding the arithmetic comparisons, and, if the constraint is violated, $Sim$ adds the necessary facts to repair that premise (definition of $Maint$). Additionally, for each subset of existing facts that includes at least one of the last 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 constraint’s 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 violators 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 cycle formed by the following constraints:

$$(ic_1) \left\{ \begin{array}{l} A(X) \rightarrow \exists Y \ B(X, Y) \\ A(X) \rightarrow \exists Y \ E(X, Y) \end{array} \right.$$ 

$$(ic_2) \ B(X, Y) \land C(X, Z) \rightarrow D(X, Y, Z)$$

$$(ic_3) \left\{ \begin{array}{l} D(X, Y, Z) \rightarrow A(X) \\ D(X, Y, Z) \rightarrow \exists V \ E(X, V) \end{array} \right.$$ 

The 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 canonical simulation that starts at $ic_1$ is shown below (in this example, there is only one simulation for each starting $ici$):
\[ Sim_0(ic_1) = \{A(x)\} \]
\[ Sim_1(ic_1) = Sim_0(ic_1) \cup \{B(x, y), E(x, y_2), C(x, z)\} \]
\[ Sim_2(ic_1) = Sim_1(ic_1) \cup \{D(x, y, z)\} \]
\[ Sim_3(ic_1) = Sim_2(ic_1) \cup \emptyset \]

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.

**Theorem 5.4.** Let \( S \) be a b-schema. If all the cycles in the dependency graph of \( S \) satisfy Condition 3, then checking the satisfiability of a query \( Q \) on \( S \) with the CQC method is a finite process.

**Proof.** Let \( G \) be the dependency graph of \( S \). Let us assume all cycles in \( G \) satisfy Condition 3.

Let us suppose that checking the satisfiability of a certain query \( Q \) on \( S \) with the CQC method does not terminate. We know there must exists an infinite sequence \( seq \) of violations and repairs (see proof of Proposition 5.1), \( seq = (ic_1\theta_1, r_1(\theta_1 \cup \theta_1'), ic_2\theta_2, r_2(\theta_2 \cup \theta_2'), \ldots) \), where each \( \theta_i \) is a ground substitution that causes the violation of \( ic_i \); if \( i > 1 \), \( PV(ic_i)\theta_i \) contains at least one tuple from the previous repair, i.e., \( PV(ic_i)\theta_i \cap r_{i-1}(\theta_{i-1} \cup \theta_{i-1}') \neq \emptyset \); and substitution \( \theta_i' \) assigns a constant to each existentially quantified variable in \( r_i \), according to the VIPs.

We also know that \( seq \) has to go through some cycle \( C \) from \( G \). Let us assume without loss of generality that \( C = (ic_1, r_1, \ldots, ic_n, r_n, ic_{n+1} = ic_1) \). Let \( seq_C \) be the first fragment of \( seq \) that iterates on \( C \), i.e., \( seq_C = (ic_i\theta_i, r_i(\theta_i \cup \theta_i'), \ldots, ic_n\theta_n, r_n(\theta_n \cup \theta_n'), ic_{n+1}\theta_{n+1}, r_{n+1}(\theta_{n+1} \cup \theta_{n+1}')) \). Let \( I \) be the instance on which \( ic_i\theta_i \) is evaluated.

Since \( C \) satisfies Condition 3, we know there is a certain constant \( k \leq n \), which we assume is the lowest possible, such that \( Sim_k(ic_1) = Sim_{k+1}(ic_1) \).

Our goal is to show that, based on \( seq_C \), we can build a sequence \( seq_{sim} = (PV(ic_1)\delta_1, r_1(\delta_1 \cup \delta_1), \ldots, PV(ic_{k+1})\delta_{k+1}, r_{k+1}(\delta_{k+1} \cup \delta_{k+1}')) \), \( \forall j, 1 \leq j \leq k+1, Sim_j(ic_1) = PV(ic)\delta_j \), \( Sim_{j+1}(ic_1) \neq r_j(\delta_j) \), substitution \( \delta_j' \) instantiates the existentially quantified variables of \( r_j \) with fresh constants, and, if \( j > 1 \), \( (r_{j-1}\delta_{j-1}' \cap PV(ic)\delta_j) \neq \emptyset \). Since the existence of \( seq_{sim} \) implies there is a canonical simulation such that \( Sim_k(ic_1) \subset Sim_{k+1}(ic_1) \), that will lead us to a contradiction.

We know that \( Sim_k(ic_1) \) is a canonical instance built by freezing the variables of \( PV(ic_1) \) into constants (point (i) of Definition 5.9), so let \( \delta_1 \) be that instantiation. We also know that \( \theta_1 \) unifies each literal in \( PV(ic_1) \) with a certain fact from \( I \). Therefore, we can define (with a slight abuse of...
notation) a substitution \( \sigma_1 \) from the frozen variables in \( \text{ic}_j \delta_1 \) to the constants in \( \text{ic}_i \theta_1 \) such that

\[
(PV(\text{ic}_1)\delta_1)^\sigma_1 = PV(\text{ic}_i)\theta_1.
\]

Then, we set \( PV(\text{ic}_i)\delta_1 \) as the first element of \( \text{seq}_{\text{sim}} \).

We know that \( \text{Sim} \) \( \text{ic}_i \) extends \( \delta_1 \) with \( \delta_i' \) in order to instantiate the existentially quantified variables of \( r_1 \) with fresh constants (definition of \( \text{Maint} \)). Since \( (r_1\theta_i' \cap PV(\text{ic}_2)\theta_2) \neq \emptyset \), we use \( A \) and \( B \) to denote the literals in \( r_1 \) and \( PV(\text{ic}_2) \), respectively, that, once fully instantiated, become the facts in \( (r_1\theta_i' \cap PV(\text{ic}_2)\theta_2) \). We extend \( \sigma_1 \) with \( \sigma_1' \), where \( \sigma_1' \) is a substitution that replaces the frozen variables in \( A(\delta_1 \cup \delta_i') \) with the constants in \( B\theta_2 \) in such a way that \( A(\delta_1 \cup \delta_i')(\sigma_1 \cup \sigma_i') = B\theta_2 \). That means \( r_1(\delta_1 \cup \delta_i')(\sigma_1 \cup \sigma_i') = r_1(\theta_1 \cup \theta_i') \). We then set \( r_1(\delta_1 \cup \delta_i') \) as the second element of \( \text{seq}_{\text{sim}} \).

Now, we apply induction and focus on an intermediate \( \text{ic}_j, 1 < j \leq k+1 \). Our hypothesis of induction is that what we just did in reference to \( \text{ic}_j \) and \( r_1 \) can be done in reference to all \( \text{ic}_i \) and \( r_n, 1 \leq i < j \). Since \( PV(\text{ic}_j)\theta_j \cap PV(\text{ic}_i)\theta_i \neq \emptyset \), that means there is a fact \( F_i\theta_j \) in \( PV(\text{ic}_j)\theta_j \) that is also present in \( r_{j,i}(\theta_1 \cup \theta_i') \). By hypothesis of induction, we already have defined a substitution \( \sigma_a \) that unifies a certain fact \( F_{\text{sim}}\gamma_1 \in \text{seq}_{\text{sim}} \) with \( F_i\theta_j \), i.e., \( F_{\text{sim}}\sigma_a = F_i\theta_j \). Since we are assuming that in the potential violators there are neither constants nor variables that appear more than once in a single literal (Definition 5.3), then we can be sure that \( F_{\text{sim}} \) can be unified with the literal \( F_i \in PV(\text{ic}_j) \). Let us focus now on some fact \( F_i\theta_j \in PV(\text{ic}_j)\theta_j \) such that \( F_i\theta_j \neq F_i\theta_j \). There are two possibilities: either (1) there is some fact \( F_s\gamma_s \in \text{seq}_{\text{sim}} \) and some substitution \( \sigma_a \) such that \( (F_s\gamma_s)^\sigma_a = F_i\theta_j \) and \( F_{\text{sim}}\sigma_a \) can be unified with the literal \( F_s \in PV(\text{ic}_j) \), or (2) otherwise. In case (2), we apply the point (ii) from Definition 5.9 and define substitution \( \gamma_s \), which assigns a fresh constant to each variable in \( F_s \), and substitution \( \rho_a \), which replaces the frozen variables of \( F_s\gamma_s \) with the constants from \( F_i\theta_j \) in such a way that \( (F_s\gamma_s)^\rho_a = F_i\theta_j \). Finally, we define \( \delta_a \) as the union of \( \gamma_s' \)'s and set \( PV(\text{ic}_j)\delta_a \) as the new last element of \( \text{seq}_{\text{sim}} \), i.e., \( \text{seq}_{\text{sim}} = (PV(\text{ic}_i)\delta_1, r_1(\delta_1 \cup \delta_i'), ..., PV(\text{ic}_j)\delta_a) \).

Now, let us focus on the repair of \( \text{ic}_j \), i.e., \( r_j \). We must show that \( r_j\delta_j \) is not true on \( \text{seq}_{\text{sim}} \). To do so, let us assume that it is, and we will reach a contradiction. If \( r_j\delta_j \) is true on \( \text{seq}_{\text{sim}} \), then for each literal \( F_s\delta_j \in r_j\delta_j \) (note that \( F_s\delta_j \) may not be ground) there is a substitution \( \delta_j'' \) such that \( (F_s\delta_j)^{\delta_j''} \in \text{seq}_{\text{sim}} \), which means that, by hypothesis of induction, we have already defined a substitution \( \sigma_s \) such that \( ((F_s\delta_j)^{\delta_j''})^\sigma_s = F_i\theta_j \). The conclusion is that \( r_j\delta_j \) is true on \( \text{seq} \), that is, constraint \( \text{ic}_j \) is not actually violated by the CQC method, which means \( \text{seq} \) is not infinite, and we have reached a contradiction.

At this point, we know that \( r_j\delta_j \) is not true and we can proceed as we did with \( r_1 \).
When we reach $j = k+1$, we have finally built the sequence $seq_{sim} = (PV(ic_1)\delta_1, r_1(\delta_1 \cup \delta_1'), \ldots, PV(ic_{k+1})\delta_{k+1}, r_{k+1}(\delta_{k+1} \cup \delta_{k+1}'))$. From the reasoning above we can conclude that $r_{k+1}\delta_{k+1}$ is not true on $seq_{sim}$. Since $Sim_k(ic_1) = PV(ic_1)\delta_1 \cup r_1(\delta_1 \cup \delta_1') \cup \ldots \cup PV(ic_{k+1})\delta_{k+1}$ is the result of the first $k$ steps of one of the canonical simulations of $C$ that start at $ic_1$ (modulo renaming of frozen variables), then $Sim_k(ic_1) \not\models r_{k+1}\delta_{k+1}$. That means $r_{k+1}(\delta_{k+1} \cup \delta_{k+1}')$ is inserted by the $k+1$ step of at least one of these simulations, i.e., $r_{k+1}\delta_{k+1} \subseteq Sim_{k+1}(ic_1)$. The conclusion is that not all canonical simulations that start at $ic_1$ reach a fix-point within $k$ steps, i.e., $Sim_k(ic_1) \subset Sim_{k+1}(ic_1)$ for some simulation. Therefore, we have reached a contradiction. ■

**Corollary 5.1.** If the dependency graph of a given $b$-schema satisfies one of the following conditions:

- Each cycle satisfies Condition 1 or Condition 2.
- Each cycle satisfies Condition 2 or Condition 3.

then checking the satisfiability of a query on the $b$-schema with the CQC method is a finite process.

**Proof.** We know from the proof of Theorem 5.3 that after a finite number of violations and repairs, at least one constraint from each cycle that satisfies Condition 2 can be removed. That leaves us with a schema in which either all cycles satisfy Condition 1 or all cycles satisfy Condition 3. In both cases, we know that checking the satisfiability of a query on that kind of schemas with the CQC method is guaranteed to terminate (Theorem 5.2 and Theorem 5.4, respectively). ■

112
6 Validating XML Mappings

In this chapter, we generalize our previous results so we can deal with XML mapping scenarios. This kind of mappings has been object of a growing interest by the research community during the last years. Most mapping design tools and approaches also support some kind of XML mappings.

First, we describe the class of XML schemas and mappings that we consider. Then, we propose a translation of the mapping scenario into the first-order logic formalism required by the CQC method. Finally, we show how the previous translation allows us to reformulate the desirable properties discussed in Chapter 3—applied now to the XML context—in terms of query satisfiability.

6.1 XML Schemas and Mappings

We consider XML schemas defined by means of a subset of the XML Schema Definition language (XSD) [W3C04]. Basically, these schemas consist of a root element definition followed by a collection of type definitions. Formally, an XML schema $S$ has the form $S = (r, T, IC)$, where $r$ is the root element definition, $T$ is the set of type definitions, and $IC$ is the set of integrity constraints.

Using a production-based notation (an extension of that used in [BNV07]), a type definition is in the form of:

$$
type \rightarrow elem_1[type_1][n_1..m_1], \ldots, elem_k[type_k][n_k..m_k]$$

Each $elem_i[type_i][n_i..m_i]$ is an element definition, where $elem_i$ is the name of the element, $type_i$ is either a simple type (e.g., integer, real, string) or a complex type (defined by another
production), and \([n_i..m_i]\) denotes the value of the \text{minOccurs} and \text{maxOccurs} facets [W3C04]
(i.e., a node of type “\text{type}” must have at least (at most) \(n_i \) \(m_i \) \text{elem} \text{child} nodes).

As an example, the production

\[
purchasetype \rightarrow \text{customer}[\text{string}][1..1], \text{item}[\text{itemtype}][0..*]
\]

states that any node of \text{purchasetype} \text{type} must have exactly one \text{customer} \text{child} node and zero or more \text{item} \text{child} nodes.

We assume the element definition of the root (i.e., \(r \) in \(S = (r, T, IC)\)) has \text{minOccurs} = 0 and \text{maxOccurs} = 1; in particular, we assume that the presence of the root is not required in order to allow the empty instance to be a valid instance of the schema.

Complex types can be defined either as a \text{<sequence>} of element definitions (see the productions above), or as a \text{<choice>} among element definitions. Productions that denote a choice are in the form of

\[
type \rightarrow \text{elem}_1[\text{type}_1][n_1..m_1] + … + \text{elem}_k[\text{type}_k][n_k..m_k]
\]

Elements can also be defined as of simple type, optionally with a restriction on its range. For example,

\[
producttype \rightarrow \text{name}[\text{string}][1..1], \text{price}[\text{decimal between 700 and 5000}][1..1]
\]

indicates that the price of a product must be at least 700 and at most 5000.

We consider XML schemas in which neither element names nor complex type names appear in more than one element definition. If a given schema does not meet this requirement, it can always be rewritten, as long as its productions are not recursive. For example, consider an XML schema with the following productions:

\[
purchasetype \rightarrow \text{customer}[\text{person}_\text{type}][1..1], \text{salesperson}[\text{person}_\text{type}][1..1], \\
\text{item}[\text{itemtype}][0..*]
\]
\[
persontype \rightarrow \text{name}[\text{string}][1..1], \text{address}[\text{string}][1..1]
\]
\[
itemtype \rightarrow \text{product}[\text{product}_\text{type}][1..1], \text{quantity}[\text{integer}][1..1]
\]
\[
producttype \rightarrow \text{name}[\text{string}][1..1], \text{price}[\text{decimal}][1..1]
\]

The schema has two different element definitions with the same element name, namely \text{name}[\text{string}][1..1] in the production of \text{person}_\text{type} and \text{name}[\text{string}][1..1] in the production of \text{product}_\text{type}. It also has two different element definitions with the same complex type, namely
customer[persontype][1..1] and salesperson[persontype][1..1]. In order to fulfill the requirement above, duplicate element names can be renamed, and repeated complex types can be split:

\[
\begin{align*}
purchase\_type &\rightarrow customer[customer\_type][1..1], salesperson[salesperson\_type][1..1], \\
item[item\_type][0..*] &
\end{align*}
\]

\[
\begin{align*}
customer\_type &\rightarrow customer-name[string][1..1], customer-address[string][1..1] \\
salesperson\_type &\rightarrow salesperson-name[string][1..1], salesperson-address[string][1..1] \\
item\_type &\rightarrow product[product\_type][1..1], quantity[integer][1..1] \\
product\_type &\rightarrow product-name[string][1..1], price[decimal][1..1]
\end{align*}
\]

Note that we have first split persontype into customertype and salespersontype, and then we have renamed the duplicate element definitions: the name and address of both customertype and salespersontype, and also the name of producttype.

Henceforth, we omit the [type] component of element definitions when it is clear from the context.

An instance of an XML schema \( S = (r, T, IC) \) is an XML document with root \( r \) that conforms to \( T \). Such instance is consistent if it satisfies the integrity constraints \( IC \). It is important to note that we do not consider the order in which sibling nodes appear in an XML document; therefore an XML document such as

\[
<\text{product}>
\begin{align*}
  &<\text{productName}>P1</\text{productName}> \\
  &<\text{price}>700</\text{price>
\end{align*}
</\text{product}>
\]

is equivalent to

\[
<\text{product}>
\begin{align*}
  &<\text{price}>700</\text{price}>
  &<\text{productName}>P1</\text{productName}>
\end{align*}
</\text{product}>
\]

Regarding path expressions, we consider paths in the form of

\[
/\text{elem}_1[\text{cond}_1]/ \ldots /\text{elem}_n[\text{cond}_n]
\]

where each \( \text{cond}_i \) is a Boolean condition that conforms to the following grammar:

\[
\text{Cond} ::= \text{Path} | \text{Cond}_1 \text{ and } \text{Cond}_2 | \text{Cond}_1 \text{ or } \text{Cond}_2 | \text{not} \text{ Cond} | \text{‘ ( Cond ) ‘} | (\text{Path}_1/\text{text()}|\text{Const}_1) (\text{‘=’} | \text{‘\neq’} | \text{‘<’} | \text{‘\leq’} | \text{‘>’} | \text{‘\geq’}) (\text{Path}_2/\text{text()}|\text{Const}_2)
\]

115
Path expressions may also use the descendant axis ‘//’. These paths can be easily rewritten into paths with the child axis ‘/’ only. Note that since element names cannot be duplicated, there is only one possible unfolding for a path ‘//elem’ into ‘/elem/ … /elem’.

We consider a subclass of the integrity constraints key and keyref. In particular, we consider the class of keys and referential constraints used in the nested relational setting. Such constraints are in the form of

\[
\text{key: } \{ \text{field}_1, \ldots, \text{field}_n \} \text{ key of } \text{selector} \\
\text{keyref: } (\text{selector}, \{ \text{field}_1, \ldots, \text{field}_n \}) \text{ references } (\text{selector}', \{ \text{field}_1', \ldots, \text{field}_m' \})
\]

where selector is a path expression that returns a set of nodes and each field is a path expression relative to selector that returns a single simple-type node.
We consider an XML schema mapping to be defined as a set of assertions \( M = \{ m_1, \ldots, m_k \} \) that specify a relationship between two XML schemas. Each assertion \( m_i \) is of the form \( Q^{S_1} \circ \text{op} Q^{S_2} \), where \( Q^{S_1} \) and \( Q^{S_2} \) are queries expressed in a subset of the XQuery language [W3C07], \( S_1 \) and \( S_2 \) are the two mapped schemas, and \( \text{op} \) is \( \subseteq \) or \( = \).

The queries in a mapping are XQueries with the “for”, “where” and “return” clauses. Their general form is:

\[
<\text{tag}> \text{for } Var_1 \text{ in } Path_1, \ldots, Var_k \text{ in } Path_k \\
\text{where } \text{Cond} \\
\text{return } <\text{tag}_1> \text{Result}_1 <\text{tag}_1> \ldots <\text{tag}_n> \text{Result}_n <\text{tag}_n> \\
</\text{tag}>
\]

where \( tag_1, \ldots, tag_n \) are all different (i.e., no union is allowed), and each \( \text{Result}_i \) denotes the following:

\[
\text{Result} ::= \text{Path}'/\text{text}()' | \text{Const} | \text{Query} | '<\text{tag}>' \text{Result}+ '<\text{tag}>
\]

The two queries in a same mapping assertion must return an answer of the same type, i.e., the XML documents generated by the two queries must conform to a same schema. As an example, see the XML mapping scenario in Figure 6.1; the two queries in the mapping return an XML document that conforms to

- \( \text{orderType} \rightarrow \text{order}[0..*] \)
- \( \text{orderType} \rightarrow \text{shipTo}[\text{string}][1..1], \text{billTo}[\text{string}][1..1], \text{items}[1..1] \)
- \( \text{itemType} \rightarrow \text{item}[0..*] \)
- \( \text{itemType} \rightarrow \text{productName}[\text{string}][1..1], \text{quantity}[\text{integer}][1..1], \text{price}[\text{decimal}][1..1] \)

Since the actual tag’s names in the result of the queries are not relevant for our purposes, we omit them and use the following nested relational notation:

\[
\text{Query} ::= \text{for}' (\text{Var} \text{’in’ } \text{Path})+ (‘\text{where’ } \text{Cond} \text{’} ‘\text{return’ } \text{’}[‘ \text{Result}+ ‘]’
\]

\[
\text{Result} ::= \text{Path}'/\text{text}()' | \text{Const} | \text{Query} | ‘[‘ \text{Result}+ ‘]’
\]

We say that two instances of the XML schemas being mapped are consistent with the mapping if all the mapping assertions are true. A mapping assertion \( Q^{S_1} \subseteq (=) Q^{S_2} \) is true if the answer to \( Q^{S_1} \) is included in (equal to) the answer to \( Q^{S_2} \) when the queries are executed over the pair of mapped schema instances.
We consider inclusion and equality of nested structures under set semantics [LS97, DHT04]. The answer to a query will be thus a set of records \([R_{1,1}, \ldots, R_{1,m}], \ldots, [R_{n,1}, \ldots, R_{n,m}]\), where each \(R_{i,j}\) is either a simple type value, a record, or a set of records.

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 [LS97]:

1. If \(T\) is a simple type, \(R_1 \subseteq R_2\) iff \(R_1 = R_2\)
2. If \(T\) is a record type, \(R_1 = [R_{1,1}^1, \ldots, R_{1,m}^1] \subseteq R_2 = [R_{2,1}^2, \ldots, R_{2,n}^2]\) iff \(R_{1,1}^1 \subseteq R_{2,1}^2 \land \ldots \land R_{1,n}^1 \subseteq R_{2,n}^2\)
3. If \(T\) is a set type, \(R_1 = \{R_{1,1}, \ldots, R_{1,n}\} \subseteq R_2 = \{R_{2,1}, \ldots, R_{2,n}\}\) iff \(\forall i \exists j R_{1,i} \subseteq R_{2,j}\)

Equality can be defined similarly [LS97]:

1. If \(T\) is a simple type, \(R_1 = R_2\)
2. If \(T\) is a record type, \([R_{1,1}^1, \ldots, R_{1,n}^1] = [R_{2,1}^2, \ldots, R_{2,n}^2]\) iff \(R_{1,1}^1 = R_{2,1}^2 \land \ldots \land R_{1,n}^1 = R_{2,n}^2\)
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 \(Q_1 \subseteq Q_2 \land Q_2 \subseteq Q_1\) [LS97].

See the Related Work chapter for a detailed comparison with other XML schema and mapping formalisms.

### 6.2 Translation of XML Mapping Scenarios into Logic

To validate XML schema mappings, we translate the problem from the initial XML setting into the first-order logic formalism the CQC method works with. The main goal of this section is to define such a translation for the XML schemas and the mapping.

#### 6.2.1 Translating the Nested Structure of Mapped Schemas

Each element definition \(name[type][n..m]\) is translated into a base predicate along the lines of the hierarchical representation of XML schemas used in [YJ08]. If the element is the root, then it is translated into the predicate \(name(id)\). Otherwise, the predicate will be either \(name(id, parentId)\) if type is complex, or \(name(id, parentId, value)\) if type is simple. The attributes of the predicates denote: the id of an XML node, the id of the parent node, and the simple-type value, respectively.

As an example, consider an XML schema with the following type definitions:

\[
purchaseType \rightarrow customer[string][1..1], item[0..*]\\
itemType \rightarrow product[string][1..1], quantity[integer][1..1], price[decimal][1..1]
\]
Figure 6.2 shows an XML document that conforms to this schema and also shows the instance of the logic schema into which the XML schema is translated.

Note that the logic representation identifies XML nodes by the value of the id attribute plus the name of the predicate, e.g., item(0, ...) and item(1, ...) have the same predicate but different ids, while customer(0, ...) and item(0, ...) have the same id but different predicates. In order to make this semantics explicit to the CQC method, we must add, for each element definition different from the root, the following constraints:

\[
\text{elem}(\text{id}, \text{parentId}_1, \text{value}_1) \land \text{elem}(\text{id}, \text{parentId}_2, \text{value}_2) \rightarrow \text{parentId}_1 = \text{parentId}_2 \\
\text{elem}(\text{id}, \text{parentId}_1, \text{value}_1) \land \text{elem}(\text{id}, \text{parentId}_2, \text{value}_2) \rightarrow \text{value}_1 = \text{value}_2
\]

where \( \text{elem} \) denotes the name of the element. These constraints state that it is not possible to have two different nodes with the same id. Note that the last constraint is only applicable to simple-type elements. It is also worth noting that no constraint is required to enforce the equality of contents for complex-type elements; the reason is that child nodes are the ones that “point to” its parent by means the \( \text{parentId} \) attribute, and that, in first-order logic, two tuples with the same values in their attributes are considered as the same tuple.

In the case of the root element, since it can only have one single node, the constraint required is the following:

\[
\text{root}(\text{id}_1) \land \text{root}(\text{id}_2) \rightarrow \text{id}_1 = \text{id}_2
\]
where $\text{root}$ denotes the name of the root element.

We also need additional constraints to make explicit the parent-child relationship between elements. For each element definition

$$\text{parentId} \rightarrow \ldots \text{elem}[\text{type}][n..m] \ldots$$

a referential constraint from the $\text{parentId}$ attribute of the $\text{elem}$ predicate to the $\text{id}$ attribute of the $\text{parent}$ predicate is required:

$$\text{elem}(\text{id}, \text{parentId}[1], \text{value}) \rightarrow [\exists \text{parentId}[2],\text{value}2] \text{parent}(\text{parentId}[1], \text{parentId}[2], \text{value}2))$$

Recall that since we are assuming that element names do not appear in more than one element definition, then there cannot be more than one possible parent for a given element.

In order to make explicit the semantics of the $<\text{choice}>$ construct

$$\text{parentId} \rightarrow \text{elem}_1[\text{type}_1][n_1..m_1] + \ldots + \text{elem}_k[\text{type}_k][n_k..m_k]$$

we need a constraint for each pair of element definitions $\text{elem}_i$ and $\text{elem}_j$, $i \neq j$, in the choice; the constraint is to state that there cannot be an $\text{elem}_i$ node and an $\text{elem}_j$ node both with the same parent:

$$\text{elem}(\text{id}_i, \text{parentId}[1], \text{value}_i) \land \ldots \land \text{elem}(\text{id}_j, \text{parentId}[1], \text{value}_j) \rightarrow \text{parentId}_i \neq \text{parentId}_j$$

Regarding the $\text{minOccurs}$ and $\text{maxOccurs}$ facets of element definitions, they also have to be made explicit by means of integrity constraints. In particular, the $\text{maxOccurs}$ facet of $\text{elem}[n..m]$ can be modeled by the following constraint:

$$\text{elem}(\text{id}_1, \text{parentId}[1], \text{value}_1) \land \ldots \land \text{elem}(\text{id}_{m+1}, \text{parentId}[1], \text{value}_{m+1}) \rightarrow \text{id}_1 = \text{id}_2 \lor \text{id}_1 = \text{id}_3 \lor \ldots \lor \text{id}_1 = \text{id}_{m+1} \lor \text{id}_2 = \text{id}_3 \lor \ldots \lor \text{id}_2 = \text{id}_{m+1} \lor \ldots \lor \text{id}_m = \text{id}_{m+1}$$

which is required only when $m \neq *$ and $\text{elem}$ is not the root. The constraint states that the only way we can have $m+1$ $\text{elem}$ tuples with the same $\text{parentId}$ if at least one of them is a duplicate (i.e., it has the same $\text{id}$ than one of the other $m$ tuples).

The translation into logic of the $\text{minOccurs}$ facet depends on whether the type definition in which the element appears is a sequence or a choice. If $\text{elem}[n..m]$ appears in a sequence

$$\text{parentId} \rightarrow \text{elem}_1[1..m_1], \ldots, \text{elem}[n..m], \ldots, \text{elem}_k[n_k..m_k]$$

and $n > 0$, then we have to introduce a constraint to ensure that whenever a node of the parent type exists, it has at least $n$ $\text{elem}$ child nodes:
\[\text{parent}(\text{id}_1, \text{grandparentId}) \rightarrow \text{aux}(\text{id})\]

where \(\text{aux}\) is an auxiliary predicate defined by the following deductive rule:

\[\text{aux}(\text{id}) \leftarrow \text{elem}(\text{id}_1, \text{id}_1[, \text{value}_1]) \land \ldots \land \text{elem}(\text{id}_n, \text{id}_n[, \text{value}_n]) \land \]

\[\text{id}_1 \neq \text{id}_2 \land \ldots \land \text{id}_1 \neq \text{id}_n \land \text{id}_2 \neq \text{id}_3 \land \ldots \land \text{id}_2 \neq \text{id}_n \land \ldots \land \text{id}_{n-1} \neq \text{id}_n\]

In the case in which \text{elem}[n..m] appears in a choice

\[\text{parent}_{\text{type}} \rightarrow \text{elem}_1[n_1..m_1] + \ldots + \text{elem}[n..m] + \ldots + \text{elem}_k[n_k..m_k]\]

two cases must be considered. First, if \(n_1, \ldots, n, \ldots, n_k\) are all \(> 0\), then a constraint that ensures that each parent node has either an \text{elem}_1, \ldots, \text{elem}, \ldots or \text{elem}_k\ child node is needed:

\[\text{parent}(\text{parentId}[, \text{grandparentId}]) \rightarrow \exists \text{id}_1[, \text{value}_1] \text{elem}_1(\text{id}_1, \text{parentId}[, \text{value}_1]) \lor \ldots \\
\exists \text{id}_1[, \text{value}_1] \text{elem}_1(\text{id}_1, \text{parentId}[, \text{value}_1]) \lor \ldots \\
\exists \text{id}_1[, \text{value}_1] \text{elem}_1(\text{id}_1, \text{parentId}[, \text{value}_1])\]

Second, if \(n > 1\), the presence of an \text{elem} node \(E\) must imply the existence of at least \(n-1\) other \text{elem} nodes siblings of \(E\):

\[\text{elem}(\text{id}, \text{parentId}[, \text{value}]) \rightarrow \text{aux}(\text{id}, \text{parentId})\]

where

\[\text{aux}(\text{id}, \text{parentId}) \leftarrow \text{elem}(\text{id}, \text{parentId}[, \text{value}]) \land \text{elem}(\text{id}_2, \text{parentId}[, \text{value}_2]) \land \ldots \land \]

\[\text{elem}(\text{id}_n, \text{parentId}[, \text{value}_n]) \land \text{id} \neq \text{id}_2 \land \ldots \land \text{id} \neq \text{id}_n \land \ldots \land \text{id}_{n-1} \neq \text{id}_n\]

### 6.2.2 Translating Path Expressions

Recall that we consider path expressions that have the form

\[/\text{root}_1[\text{cond}_1]/\text{elem}_2[\text{cond}_2]/ \ldots /\text{elem}_n[\text{cond}_n]\]

where \(n \geq 1\) and each \text{cond}_i\ is a Boolean condition. Recall also that if the path expression has some ‘//’ (descendant) axis, it can be unfolded into an expression of the form above.

We translate each path expression into a derived predicate along the lines suggested in [DT05]. The main difference is that we allow conditions with negations and order comparisons, which are not handled in [DT05]. The translation of \text{path}, denoted by \text{T-path(path, id)}, corresponds to \text{Ppath(id_n)}, that is, \text{T-path(path, id)} = \text{Ppath(id_n)}.

\text{Ppath} is a derived predicate defined by the following deductive rule:
If the path ends with “/text()”, the literal about \textit{name}_{n} should be \textit{name}_{n}(id_{n}, id_{n-1}, value), and the term in the head of the rule should be \textit{value}_{n} instead of id_{n}. In the body of the rule, T-cond stands for the translation of the Boolean condition \textit{cond}. It is defined as follows:

- T-cond(\textit{cond}_{1} \textbf{and} \textit{cond}_{2}, pid) = T-cond(\textit{cond}_{1}, pid) \land T-cond(\textit{cond}_{2}, pid)
- T-cond(\textit{cond}_{1} \textbf{or} \textit{cond}_{2}, pid) = aux(pid), where
  
  aux(pid) \leftarrow T-cond(\textit{cond}_{1}, pid)
  aux(pid) \leftarrow T-cond(\textit{cond}_{2}, pid)
- T-cond(\textbf{not} \textit{cond}, pid) = \neg aux(\textit{cond}, pid), where
  
  aux(pid) \leftarrow T-cond(\textit{cond}, pid)
- T-cond(\textit{path}_{1}/text() \textbf{op} \textit{path}_{2}/text(), pid) = T-relpath(\textit{path}_{1}/text(), pid, value_{1}) \land T-relpath(\textit{path}_{2}/text(), pid, value_{2}) \land value_{1} \textbf{op} value_{2}

where value_{1} and value_{2} are the simple-type results of the relative path expressions.

- T-cond(\textit{path}, pid) = T-relpath(\textit{path}, pid, res)

Relative paths have the following translation:

- T-relpath(.//\textit{elem}_{1}[\textit{cond}_{1}]/ ... //\textit{elem}_{n}[\textit{cond}_{n}], pid, id_{n}) = elem_{n}(id_{n}, pid) \land T-cond(\textit{cond}_{1}, id_{1}) \land ... \land elem_{n}(id_{n}, id_{n-1}) \land T-cond(\textit{cond}_{n}, id_{n})
- T-relpath(.//\textit{elem}_{1}[\textit{cond}_{1}]/ ... //\textit{elem}_{n}[\textit{cond}_{n}]/text(), pid, value) = elem_{n}(id_{1}, pid) \land T-cond(\textit{cond}_{1}, id_{1}) \land ... \land elem_{n}(id_{n}, id_{n-1}, value) \land T-cond(\textit{cond}_{n}, id_{n})

As an example, the path expression:

/\textit{orderDoc}/\textit{purchaseOrder}[\textbf{not}(./\textit{item}[./\textit{price}/text()]< 1000)]/\textit{customer}

would be translated as:

\[ P_{/\textit{orderDoc}/\textit{purchaseOrder}[\textbf{not}(./\textit{item}[./\textit{price}/text()]< 1000)]/\textit{customer}}(id) \leftarrow \textit{orderDoc}(id_{1}) \land \textit{purchaseOrder}(id_{2}, id_{1}) \land \neg aux./\textit{item}[./\textit{price}/text()]< 1000](id_{2}) \land \textit{costumer}(id, id_{2}) \land aux./\textit{item}[./\textit{price}/text()]< 1000](id_{2}) \leftarrow \textit{item}(id_{3}, id_{2}) \land \textit{price}(id_{4}, id_{3}, val) \land val < 1000 \]
6.2.3 Translating Integrity Constraints

A key constraint in the form of

\[ \{field_1, \ldots, field_n\} \text{ key of selector} \]

where selector is a path expression that returns a set of complex-type nodes, and each field\(_i\) is a path expression that returns one single simple-type node, can be expressed in our logic formalism by means of the following constraint:

\[
T\text{-path}(\text{selector}, id_1) \land T\text{-path}(\text{selector}, id_2) \land \\
T\text{-relpath}(field_1, id_1, value_1) \land T\text{-relpath}(field_1, id_2, value_1) \land \ldots \land \\
T\text{-relpath}(field_n, id_1, value_n) \land T\text{-relpath}(field_n, id_2, value_n) \rightarrow id_1 = id_2
\]

The constraint states that there cannot be two nodes in the set returned by selector with the same values for field\(_1\), \ldots, field\(_n\).

As an example, the constraint

\[ \{./id/text()\} \text{ key of /orderDB/order} \]

from Figure 6.1 would be translated into logic as follows (for simplicity, we fold both path and relative path translations into derived predicates):

\[
P/\text{orderDB}/\text{order}(id_1) \land P/\text{orderDB}/\text{order}(id_2) \land \\
P ./\text{id/text}() (id_1, value) \land P ./\text{id/text}() (id_2, value) \rightarrow id_1 = id_2
\]

where

\[
P/\text{orderDB}/\text{order}(id) \leftarrow \text{orderDB}(id_1) \land \text{order}(id, id_1) \\
P ./\text{id/text}() (pid, value) \leftarrow \text{id}(id, pid, value)
\]

Similarly, a keyref constraint in the form of

\[ (\text{selector}, \{field_1, \ldots, field_n\}) \text{ references } (\text{selector}', \{field'_1, \ldots, field'_n\}) \]

is translated into logic as the following constraint:

\[
T\text{-path}(\text{selector}, id) \land T\text{-relpath}(field_1, id, value_1) \land \ldots \land T\text{-relpath}(field_n, id, value_n) \land aux(value_1, \ldots, value_n)
\]

where
Finally, a range restriction on a simple type such as /orderDoc/purchaseOrder/item/price being a decimal between 0 and 5000 in Figure 6.1 is translated into logic as follows:

\[ P_{\text{orderDoc/purchaseOrder/item/price/text}}(\text{value}) \rightarrow \text{value} \geq 0 \]
\[ P_{\text{orderDoc/purchaseOrder/item/price/text}}(\text{value}) \rightarrow \text{value} \leq 5000 \]

where

\[ P_{\text{orderDoc/purchaseOrder/item/price/text}}(\text{value}) \leftarrow \text{orderDoc(id1)} \land \text{purchaseOrder(id2, id1)} \land \text{item(id3, id2)} \land \text{price(id4, id3, value)} \]

### 6.2.4 Translating Nested Queries

The queries in an XML mapping are XQueries whose answer is an XML document that conforms to some nested relational schema. We translate each of these nested queries as a collection of “flat” queries; we follow a variation of the approach that was used in [LS97] (see the Related Work chapter for a detailed comparison).

There will be one flat query for each nested block. For example, consider the query \( Q^{S1} \) from Figure 6.1 (shown here in our compact notation).

\[
Q^{S1}: \text{for } \text{po in } //\text{purchaseOrder[.//twoAddresses]} \n\quad \text{return } [\text{po}//\text{shipTo/text()}, \text{po}//\text{billTo}/\text{shipTo},
\quad \text{for } \text{sit in po/item}
\quad \text{return } [\text{sit}/\text{productName/text()}, \text{sit}/\text{quantity/text()}, \text{sit}/\text{price/text()}]]
\]

It has two “for … return …” blocks, i.e., the outer one and the inner one. The outer block iterates through those purchase orders with two addresses, while the inner block iterates through the items of the purchase orders selected by the outer block.

We translate the outer block into the following derived predicate:

\[
Q_{\text{outer}}^{S1}(\text{po, st, bt}) \leftarrow \text{T-path}(/\text{purchaseOrder[.//twoAddresses]}, \text{po}) \land
\text{T-relpath}(/\text{shipTo/text()}, \text{po, st}) \land \text{T-relpath}(/\text{billTo/text()}, \text{po, bt})
\]

which projects the id of each purchaseOrder XML node (i.e., variable po) together with the simple-type value of its shipTo and billTo descendants. Note that the predicate ignores the inner block of the query, which is to be translated into a separate predicate.
The translation of an inner block requires dealing with the variables inherited from its parent block, e.g., variable $po$ in $Q_{S1}$. We use access patterns [DLN07] to deal with this kind of variables. In particular, we consider derived predicates with both “input-only” and “input-output” terms. We denote these predicates by $Q(X_1, \ldots, X_n)(Y_1, \ldots, Y_m)$, where $X_1, \ldots, X_n$ are the input-only terms and $Y_1, \ldots, Y_m$ are the usual input-output terms. In this way, we translate the inner block of $Q_{S1}$ into the following derived predicate:

$$Q_{S1}^{\text{inner}}(po)(it, pn, q, p) \leftarrow \text{T-relpath}(./item, po, it) \land \text{T-relpath}(./productName/text(), it, pn) \land \text{T-relpath}(./quantity(text(), it, q) \land \text{T-relpath}(./price/text(), it, p)$$

In order to allow the CQC method to deal with predicates with access patterns the same way it does with the usual derived predicates, a requirement must be fulfilled. The requirement is that input-only variables must be *safe*, that is, whenever a literal $Q(X_1, \ldots, X_n)(Y_1, \ldots, Y_m)$ appears in the body of some deductive rule or condition, the variables in $\{X_1, \ldots, X_n\}$ must either appear in some other positive literal in the same body in an input-output position, or, if the body is from a deductive rule, they may appear in the head of the rule as input-only variables but then the requirement must be inductively fulfilled by the derived predicate defined by the deductive rule.

The translation of the “where” clause of a query block is similar to the translation of a Boolean condition from a path expression. The difference is that a condition from a path expression involves at most one single variable which denotes the node to which the condition is applied, while a where clause potentially involves all the variables in the “for” clause (plus the variables inherited from the ancestor blocks). As an example, consider the outer query block of $Q_{S2}$ in Figure 6.1, which has a “where” clause:

$$Q_{S2}: \text{for } o \in /orderDB/order \text{ where not}(/orderDB/item[./order/text() = o/id/text() \text{ and } ./price/text() \leq 5000]) \text{ return } [o/shipTo/text(), o/billTo/text(), \ldots \text{ return } \ldots]$$

The “where” clause will be translated into an auxiliary derived predicate as follows:

$$Q_{S2}^{\text{outer}}(o, st, bt) \leftarrow \text{T-path}(/orderDB/order, o) \land \neg \text{aux}(o) \land \text{T-relpath}(/shipTo/text(), o, st) \land \text{T-relpath}(/billTo/text(), o, bt)$$

$$\text{aux}(o) \leftarrow \text{T-path}(/orderDB/item, it) \land \text{T-relpath}(/order/text(), it, value_1) \land \text{T-relpath}(/id/text(), o, value_2) \land value_1 = value_2 \land \text{T-relpath}(/price/text(), it, value_3) \land value_3 \leq 5000$$
Note the use of the input-only variable “o” in the auxiliary predicate that models the “where” clause. This input-only variable denotes the variable inherited from the “for” clause.

6.2.5 Translating Mapping Assertions

An XML mapping scenario consists of two XML schemas and an XML mapping that relates them. We have already discussed how to translate each XML schema into our logic formalism. Therefore, in order to complete the translation of the mapping scenario into logic, we must see now how to translate the mapping assertions.

A mapping assertion consists of two nested XML queries related by means of a $\subseteq$ or $=$ operator. An instantiation of a mapping scenario is consistent only if it makes all the assertions in the mapping true. The mapping assertions can thus be modeled as integrity constraints defined over the translations of the two mapped schemas.

To translate a mapping assertion $Q_1 \subseteq (=) Q_2$, we will make use of the definition of inclusion (equality) of nested structures from Section 6.1 and the flat queries that result from the translation of $Q_1$ and $Q_2$.

Let $Q_a$ and $Q_b$ be two generic (sub)queries with the same return type:

$Q_a$: for $v_1$ in $path_1$, ..., $v_{na}$ in $path_{na}$ where $cond$

return $[A_1, \ldots, A_m, B_1, \ldots, B_k]$

$Q_b$: for $v_1'$ in $path_1'$, ..., $v_{nb}'$ in $path_{nb}'$ where $cond'$

return $[A_1', \ldots, A_m', B_1', \ldots, B_k']$,

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 predicate $Q_a(x_1, \ldots, x_{ka})(v_1, \ldots, v_{na}, r_1, \ldots, r_m)$, where $x_1, \ldots, x_{ka}$ denote the variables inherited from the ancestor query blocks, $v_1, \ldots, v_{na}$ denote the variables in the “for” clause, and $r_1, \ldots, r_m$ denote the simple-type values returned by the block. Similarly, let us also assume the outer block of $Q_b$ is translated into $Q_b(x_1', \ldots, x_{kb}')(v_1', \ldots, v_{nb}', r_1', \ldots, r_m')$.

Let us assume the mapping assertion is $Q_a \subseteq Q_b$. The assertion states that the nested structure returned by the execution of $Q_a$ must be included in the nested structure returned by the execution of $Q_b$. The first step is to express this in first-order logic.

We use $\text{T-inclusion}(Q_a, Q_b, \{i_1, \ldots, i_h\})$ to denote the first-order translation of $Q_a \subseteq Q_b$, according to the definition of inclusion from Section 6.1, where $\{i_1, \ldots, i_h\}$ is the union of the
variables inherited by $Q_{A}$ and the variables inherited by $Q_{B}$ from their respective parent blocks (if any):

$$T\text{-inclusion}(Q_{A}, Q_{B}, \{i_{1}, ..., i_{h}\}) = \forall(v_{1}, ..., v_{na}, r_{1}, ..., r_{m}) (Q_{A0}(x_{1}, ..., x_{ka})(v_{1}, ..., v_{na}, r_{1}, ..., r_{m}) \implies

\exists(v'_{1}, ..., v'_{na}) (Q_{B0}(x'_{1}, ..., x_{kb})'(v'_{1}, ..., v'_{na}, r_{1}, ..., r_{m})

\land T\text{-inclusion}(B_{1}, B_{1}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})

\land ... \land T\text{-inclusion}(B_{k}, B_{k}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})))

$$

where $\{x_{1}, ..., x_{ka}\} \cup \{x'_{1}, ..., x'_{kb}\} \subseteq \{i_{1}, ..., i_{h}\}$.

The constraint formally states that for all tuple in the answer to $Q_{A}$, there must be a tuple in the answer to $Q_{B}$ such that both tuples have the same value for their simple-type attributes ($r_{1}, ..., r_{m}$) and each complex-type attribute in the tuple of $Q_{A}$ has a value which is a nested structure (produced by the execution of $B_{i}$) and is included in the value (also a nested structure) of the corresponding attribute ($B_{i}'$) of the tuple of $Q_{B}$.

However, the constraint above does not fit the syntactic requirements of the class of logic database schemas the CQC method deals with (see Chapter 2). To address that, we first need to get rid of the universal quantifiers. To do so, we perform a double negation on $T$-inclusion and move one of the negations inwards:

$$T\text{-inclusion}(Q_{A}, Q_{B}, \{i_{1}, ..., i_{h}\}) = \neg\exists(v_{1}, ..., v_{na}, r_{1}, ..., r_{m}) (Q_{A0}(x_{1}, ..., x_{ka})(v_{1}, ..., v_{na}, r_{1}, ..., r_{m})

\land \neg\exists(v'_{1}, ..., v'_{na}) (Q_{B0}(x'_{1}, ..., x_{kb})'(v'_{1}, ..., v'_{na}, r_{1}, ..., r_{m})

\land T\text{-inclusion}(B_{1}, B_{1}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})

\land ... \land T\text{-inclusion}(B_{k}, B_{k}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})))

$$

Next, we fold each existentially quantified (sub)expression and get the following:

$$T\text{-inclusion}(Q_{A}, Q_{B}, \{i_{1}, ..., i_{h}\}) = \neg Q_{A}\text{-not-included-in-}Q_{B}(i_{1}, ..., i_{h})

$$

where

$$Q_{A}\text{-not-included-in-}Q_{B}(i_{1}, ..., i_{h}) \leftarrow Q_{A0}(x_{1}, ..., x_{ka})(v_{1}, ..., v_{na}, r_{1}, ..., r_{m})

\land \neg\text{aux-}Q_{A}\text{-not-included-in-}Q_{B}(i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m})

$$

aux-$Q_{A}\text{-not-included-in-}Q_{B}(i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}) \leftarrow Q_{B0}(x_{1}', ..., x_{kb})'(v_{1}', ..., v_{na}', r_{1}, ..., r_{m})

\land T\text{-inclusion}(B_{1}, B_{1}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})

\land ... \land T\text{-inclusion}(B_{k}, B_{k}', \{i_{1}, ..., i_{h}, v_{1}, ..., v_{na}, r_{1}, ..., r_{m}, v'_{1}, ..., v'_{na}'\})

$$

Finally, we put $\neg Q_{A}\text{-not-included-in-}Q_{B}(i_{1}, ..., i_{h})$ in form of DED:
\( Q_{t, \text{not-included-in}-Q_B(i_1, \ldots, i_h)} \rightarrow 1 = 0 \)

which is the same as

\( \neg T\text{-inclusion}(Q_t, Q_B, \{i_1, \ldots, i_h\}) \rightarrow 1 = 0 \)

As an example, consider the mapping assertion \( Q_{S^1} \subseteq Q_{S^2} \), where \( Q_{S^1} \) and \( Q_{S^2} \) are the queries in Figure 6.1. Let us assume the outer block of each query is translated into the derived predicate \( Q_{S^1, 0} \) and \( Q_{S^2, 0} \), respectively, and the inner block is translated into the derived predicate \( Q_{S^1, 1} \) and \( Q_{S^2, 1} \), respectively. The mapping assertion is then translated into the constraint

\( Q_{S^1, \text{not-included-in}-Q_{S^2}} \rightarrow 1 = 0 \)

where

\[
\begin{align*}
Q_{S^1, \text{not-included-in}-Q_{S^2}} &\leftarrow Q_{S^1, 0}(po, st, bt) \land \\
&\neg \text{aux-Q}_{S^1, \text{not-included-in}-Q_{S^2}}(po, st, bt)
\end{align*}
\]

\[
\begin{align*}
\text{aux-Q}_{S^1, \text{not-included-in}-Q_{S^2}} &\leftarrow Q_{S^2, 0}(o, st, bt) \land \\
&\neg Q_{S^1, 1, \text{not-included-in}-Q_{S^2}}(po, st, bt, o)
\end{align*}
\]

\[
\begin{align*}
Q_{S^1, 1, \text{not-included-in}-Q_{S^2}}(po, st, bt, o) &\leftarrow Q_{S^1, 0}(it, pn, q, p) \land \\
&\neg \text{aux-Q}_{S^1, 1, \text{not-included-in}-Q_{S^2}}(po, st, bt, o, it, pn, q, p)
\end{align*}
\]

\[
\begin{align*}
\text{aux-Q}_{S^1, 1, \text{not-included-in}-Q_{S^2}} &\leftarrow Q_{S^2, 0}(o, it', pn, q, p)
\end{align*}
\]

Similarly, the translation of an equality assertion \( Q_t = Q_B \) results in two constraints; one states that there cannot be a tuple in \( Q_t \) that is not present in \( Q_B \), and the other states that there cannot be a tuple in \( Q_B \) that is not present in \( Q_t \):

\[
\begin{align*}
\neg T\text{-equality}(Q_t, Q_B, \{i_1, \ldots, i_h\}) &\rightarrow 1 = 0 \\
\neg T\text{-equality}(Q_B, Q_t, \{i_1, \ldots, i_h\}) &\rightarrow 1 = 0
\end{align*}
\]

where \( T\text{-equality} \) is generically defined as follows:

\[
T\text{-equality}(Q_t, Q_B, \{i_1, \ldots, i_h\}) = \neg Q_{t, \text{not-eq-to}-Q_B(i_1, \ldots, i_h)}
\]

and \( Q_{t, \text{not-eq-to}-Q_B} \) is a derived predicate defined by the following deductive rules:

\[
\begin{align*}
Q_{t, \text{not-eq-to}-Q_B(i_1, \ldots, i_h)} &\leftarrow Q_B(\{x_{i_1}, \ldots, x_{i_h}\}(v_{1, \ldots, v_{n_k}, r_{1, \ldots, r_m})) \\
&\land \neg aux-Q_{t, \text{not-eq-to}-Q_B(i_1, \ldots, i_h, v_{1, \ldots, v_{n_k}, r_{1, \ldots, r_m}})}
\end{align*}
\]

\[
\begin{align*}
aux-Q_{t, \text{not-eq-to}-Q_B(i_1, \ldots, i_h, v_{1, \ldots, v_{n_k}, r_{1, \ldots, r_m}}) &\leftarrow Q_B(\{x_{i_1}', \ldots, x_{i_h}'\}(v_{1}', \ldots, v_{n_k}', r_{1}', \ldots, r_m')) \\
&\land T\text{-equality}(B_1, B_1', \{i_1, \ldots, i_h, v_{1, \ldots, v_{n_k}, r_{1, \ldots, r_m}, v_{1}', \ldots, v_{n_k}'}\}) \\
&\land T\text{-equality}(B_1', B_1, \{i_1, \ldots, i_h, v_{1, \ldots, v_{n_k}, r_{1, \ldots, r_m}, v_{1}', \ldots, v_{n_k}'}\})
\end{align*}
\]
As an example, consider the mapping assertion $Q^S_1 = Q^S_2$ from Figure 6.1. It would be translated into

\[
\begin{align*}
Q^{S_1}\text{-not-eq-to-}Q^{S_2}\langle \rangle \rightarrow 1 &= 0 \\
Q^{S_2}\text{-not-eq-to-}Q^{S_1}\langle \rangle \rightarrow 1 &= 0
\end{align*}
\]

where

\[
\begin{align*}
Q^{S_1}\text{-not-eq-to-}Q^{S_2}\langle \rangle &\leftarrow Q^{S_1}_0(\text{po}, \text{st}, \text{bt}) \land \neg\text{aux-}Q^{S_1}\text{-not-eq-to-}Q^{S_2}\langle \text{po}, \text{st}, \text{bt}\rangle \\
\text{aux-}Q^{S_1}\text{-not-eq-to-}Q^{S_2}\langle \text{po}, \text{st}, \text{bt}\rangle &\leftarrow Q^{S_2}_0(\text{st}, \text{bt}) \land \\
&\quad \neg Q^{S_1}_1\text{-not-eq-to-}Q^{S_2}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}\rangle \land \\
&\quad \neg Q^{S_2}_1\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}, \text{bt}\rangle
\end{align*}
\]

\[
\begin{align*}
Q^{S_1}_1\text{-not-eq-to-}Q^{S_2}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}\rangle &\leftarrow Q^{S_1}_1(\text{it}, \text{pn}, \text{q}, \text{p}) \land \\
&\quad \neg\text{aux-}Q^{S_1}_1\text{-not-eq-to-}Q^{S_2}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}, \text{it}, \text{pn}, \text{q}, \text{p}\rangle \\
\text{aux-}Q^{S_1}_1\text{-not-eq-to-}Q^{S_2}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}, \text{it}, \text{pn}, \text{q}, \text{p}\rangle &\leftarrow Q^{S_2}_1(\it', \text{pn}, \text{q}, \text{p})
\end{align*}
\]

\[
\begin{align*}
Q^{S_2}_1\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}\rangle &\leftarrow Q^{S_2}_1(\text{it}', \text{pn}', \text{q}', \text{p}') \land \\
&\quad \neg\text{aux-}Q^{S_2}_1\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}, \text{it}', \text{pn}', \text{q}', \text{p}'\rangle \\
\text{aux-}Q^{S_2}_1\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}, \text{it}', \text{pn}', \text{q}', \text{p}'\rangle &\leftarrow Q^{S_1}_1(\text{po}, \text{it}', \text{q}', \text{p}')
\end{align*}
\]

\[
\begin{align*}
Q^{S_2}_2\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}', \text{bt}'\rangle &\leftarrow Q^{S_2}_2(\text{po}, \text{st}', \text{bt}') \land \\
&\quad \neg\text{aux-}Q^{S_2}_2\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}', \text{bt}'\rangle \\
\text{aux-}Q^{S_2}_2\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}', \text{bt}'\rangle &\leftarrow Q^{S_1}_2(\text{po}', \text{st}', \text{bt}') \land \\
&\quad \neg Q^{S_2}_1\text{-not-eq-to-}Q^{S_1}_1\langle \text{po}, \text{st}, \text{bt}, \text{o}\rangle \land \\
&\quad \neg Q^{S_1}_1\text{-not-eq-to-}Q^{S_2}_1\langle \text{po}, \text{st}', \text{bt}', \text{o}\rangle
\end{align*}
\]

### 6.3 Checking Desirable Properties of XML Mappings

Our approach to validating XML mappings is an extension of the one we proposed for relational mapping scenarios. It is aimed at providing the designer with a set of desirable properties that the mapping should satisfy. For each property to be checked, a query that formalizes the property is defined. Then, the CQC method [FTU05] is used to determine whether the property is satisfied, i.e., whether the query into which the property is reformulated is satisfiable. In addition to this query, the CQC method also requires the logic database schema on which the query is defined. This logic database schema is the one that results from putting together the translation of the two
mapped XML schemas and the translation of the XML mapping assertions; translations that we have discussed in the previous sections.

In this section, we show how the desirable properties of relational mappings we saw in Chapter 3 are also applicable to the XML context and how these properties can be reformulated in terms of a query satisfiability check over the logic translation of the XML mapping scenario. We focus here on the reformulation of the strong mapping satisfiability, mapping losslessness, and mapping inference properties (the reformulation of weak mapping satisfiability and query answerability can be straightforwardly deduced from these).

6.3.1 Strong Mapping Satisfiability

A mapping is strongly satisfiable if there is a pair of schema instances that make all mapping assertions true in a non-trivial way. In the relational setting (see Chapter 3), the trivial case is that in which the two queries in the assertion return an empty answer. In XML, however, queries may return a nested structure; therefore, testing this property must make sure that all levels of nesting can be satisfied non-trivially. As an example, consider the mapping in Figure 6.1. The mapping may seem correct because it relates orders in $S_1$ with orders in $S_2$. However, only those orders in $S_1$ that have no items can satisfy the assertion. There is a contradiction between the “where” clause of $Q^{S_2}$ and the range restriction on the price of $S_1$’s items.

Strong satisfiability of XML schema mappings can thus be formalized as follows:

**Definition 6.1.** An XML schema mapping $M$ between schemas $S_1$ and $S_2$ is strongly satisfiable if $\exists I_{S_1}, I_{S_2}$ instances of $S_1$ and $S_2$, respectively, such that $I_{S_1}$ and $I_{S_2}$ satisfy the assertion in $M$, and for each assertion $Q^{S_1}_{op} Q^{S_2}$ in $M$, the answer to $Q^{S_1}$ in $I_{S_1}$ is a strong answer. We say that $R$ is a strong answer if

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

The query into which the strong satisfiability of a mapping $M$ is reformulated is the following:

$$Q_{stronglySat} \leftarrow \text{StrongSat}(Q^{S_1}_{op} Q^{S_2}) \land \ldots \land \text{StrongSat}(Q^{S_i}_{op} Q^{S_j})$$

where $\text{StrongSat}$ is a function generically defined as follows.
Let $V$ be a generic (sub)query:

\[
V: \text{for } v_1 \text{ in } \text{path}_1, \ldots, v_s \text{ in } \text{path}_s \text{ where } \text{cond} \\
\text{return } [A_1, \ldots, A_m, B_1, \ldots, B_k],
\]

where $A_1, \ldots, A_m$ are simple-type expressions and $B_1, \ldots, B_k$ are query blocks; and let predicate $V_0$ be the translation of the outer block of $V$. Then,

\[
\text{StrongSat}(V, \text{inheritedVars}) = V_0(x_1, \ldots, x_r)(v_1, \ldots, v_s, r_1, \ldots, r_m) \\
\wedge \text{StrongSat}(B_1, \text{inheritedVars} \cup \{v_1, \ldots, v_s, r_1, \ldots, r_m\}) \\
\wedge \ldots \wedge \text{StrongSat}(B_k, \text{inheritedVars} \cup \{v_1, \ldots, v_s, r_1, \ldots, r_m\})
\]

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

The logic schema $DB$ over which we are to check the satisfiability of $Q_{\text{stronglySat}}$ is obtained as follows. Let $\text{DR}_M$ and $\text{IC}_M$ be the deductive rules and constraints that result from the translation of the assertions from mapping $M = \{Q^{S_1}_1 \text{ op } Q^{S_2}_1, \ldots, Q^{S_1}_n \text{ op } Q^{S_2}_n\}$. Let $\text{DR}_{S_1}, \text{IC}_{S_1}$ and $\text{DR}_{S_2}, \text{IC}_{S_2}$ be the rules and constraints from the translation of mapped schemas $S_1$ and $S_2$, respectively. Then, $DB = (\text{DR}_{S_1} \cup \text{DR}_{S_2} \cup \text{DR}_M, \text{IC}_{S_1} \cup \text{IC}_{S_2} \cup \text{IC}_M)$.

As an example, consider again the mapping $M$ in Figure 6.1. Strong satisfiability of this mapping is defined by the query:

\[
Q_{\text{stronglySat}} \leftarrow Q^{S_1}_0(\text{po, st, bt}) \land Q^{S_1}_1(\text{po})(\text{it, pn, q, p})
\]

Note that the second literal in the body of this query can never be satisfied, since every possible instantiation either violates the range restriction on the price element of $S_1$ or it violates the mapping assertion (more specifically, the “where” clause of $Q^{S_2}$). Such unsatisfiability is detected by applying the CQC method.

### 6.3.2 Mapping Losslessness

Recall that the mapping losslessness property allows the designer to provide a query on the source schema and check whether all the data needed to answer that query is mapped into the target; and that it can be used, for example, to know whether a mapping that may be partial or incomplete suffices for the intended task, or to be sure that certain private information is not made public by the mapping.

The definition from Chapter 3 can be easily applied to the context of XML mappings.
Definition 6.2. Let $Q$ be a query posed on schema $S_1$. Let $M$ be an XML mapping between schemas $S_1$ and $S_2$ with assertions: \{$Q^{S_1}_{i} \text{ op } Q^{S_2}_{i}$, ..., $Q^{S_1}_{n} \text{ op } Q^{S_2}_{n}$\}. We say that $M$ is \emph{lossless} with respect to $Q$ if \(\forall IS_{1,1}, IS_{1,2} \text{ instances of } S_1 \) both

1. \(\exists IS_{2} \text{ instance of } S_2 \) such that $IS_{1,j}$ and $IS_{1,j}$ are both mapped into $IS_{2}$, and
2. for each mapping assertion $Q^{S_1}_{i} \text{ op } Q^{S_2}_{i}$ from $M$, the answer of $Q^{S_1}_{i}$ over $IS_{1,1}$ is equal to the answer of $Q^{S_1}_{n}$ over $IS_{1,2}$,

imply that the answer of $Q$ over $IS_{1,1}$ is equal to the answer of $Q$ over $IS_{1,2}$. \(\square\)

In other words, mapping $M$ is lossless w.r.t. $Q$ if the answer to $Q$ is determined by the extension of the $Q^{S_1}_{i}$ queries in the mapping, where these extensions must be the result of executing the queries over an instance of $S_1$ that is mapped into some consistent instance of $S_2$.

As an example, consider the mapping $M$ in Figure 6.1 and suppose that we have changed “./price/text() <= 5000” by “./price/text() > 5000” in the definition of $Q^{S_2}$ in order to make $M$ strongly satisfiable. Consider also the following query $Q$:

\[
Q: \text{ for } $sa \text{ in } //\text{singleAddress return } [$sa/text()]
\]

Intuitively, mapping $M$ is not lossless w.r.t. $Q$ because it maps the purchase orders that have two addresses, but not the ones with a single address. More formally, we can find a counterexample that shows $M$ is lossy w.r.t. $Q$. This counterexample is depicted in Figure 6.3, and it consists of two instances of $S_1$ that have the same extension for $Q^{S_1}_{i}$, that are both mapped to a consistent instance of $S_2$, and that have different answers for $Q$.

Figure 6.3: Counterexample for mapping losslessness.
Let $M = \{ QS_1^{S_1} \circ p_1 QS_1^{S_2} \ldots, QS_n^{S_1} \circ p_n QS_n^{S_2} \}$ be a mapping between schemas $S_1$ and $S_2$, and let $Q$ be a query over $S_1$. The query into which losslessness of mapping $M$ with respect to query $Q$ is reformulated is as follows:

$$Q_{\text{lossy}} \leftarrow \neg \text{T-inclusion}(Q, Q', \emptyset)$$

where $Q'$ is a copy of $Q$ in which each element name $elem$ in the path expressions has been renamed $elem'$.

The logic schema $DB$ over which we are to check the satisfiability of $Q_{\text{lossy}}$ is defined as follows. Let $DR_{SI}, IC_{SI}$ and $DR_{S2}, IC_{S2}$ be the rules and constraints from the translation of $S_1$ and $S_2$, respectively; let $DR_{SI}'$, $IC_{SI}'$ be a copy of $DR_{SI}$, $IC_{SI}$ in which each predicate $p$ has been renamed $p'$; and let $DR_t$, $IC_L$ be the result of translating the assertions: $Q^{S_1}_1 = Q^{S_1}_1'$, ..., $Q^{S_1}_n = Q^{S_1}_n'$. Then, $DB = (DR_{SI} \cup DR_{S2} \cup DR_M \cup DR_{SI}' \cup DR_t, IC_{SI} \cup IC_{S2} \cup IC_M \cup IC_{SI}' \cup IC_L)$.

If the CQC method can build an instance of $DB$ in which $Q_{\text{lossy}}$ is true, this instance can be partitioned in three instances: one for $S_1$, one for $S_1'$, and one for $S_2$. Since $S_1$ and $S_1'$ are actually two copies of the same schema, we can say that we have two instances of $S_1$, which are both mapped to the instance of $S_2$ (because $IC_M$) and share the same answer for the $Q^{S_1}_i$ queries in mapping $M$ (because $IC_L$). Moreover, since $Q_{\text{lossy}}$ is true and its definition requires that $Q \nsubseteq Q'$, then the two instances of $S_1$ must have different answers for query $Q$. In conclusion, we have got a counterexample that shows $M$ is lossy w.r.t. query $Q$.

6.3.3 Mapping Inference

The mapping inference property [MBDH02] checks whether a given mapping assertion is inferred from a set of others assertions. It can be used, for instance, to detect redundant assertions or to test equivalence of mappings.

**Definition 6.3.** Let $M$ be an XML mapping between schemas $S_1$ and $S_2$. Let $F$ be a mapping assertion between $S_1$ and $S_2$. We say that $F$ is inferred from $M$ if $\forall f^{S_1}, f^{S_2}$ instances of schemas $S_1$ and $S_2$, respectively, such that $f^{S_1}$ and $f^{S_2}$ satisfy the assertions in $M$, then $f^{S_1}$ and $f^{S_2}$ also satisfy assertion $F$. □

The query into which inference of a mapping $M$ with respect to an assertion $F$ is reformulated is defined as follows:

- If $F$ is an inclusion assertion $Q_1 \subseteq Q_2$, query $Q_{\text{notInferred}}$ will be defined by a single rule:

  $$Q_{\text{notInferred}} \leftarrow \neg \text{T-inclusion}(Q_1, Q_2, \emptyset)$$
Otherwise, if $F$ is like $Q_1 = Q_2$, there will be two rules:

$$
Q_{\text{notInferred}} \leftarrow \neg T\text{-equality}(Q_1, Q_2, \emptyset)
$$
$$
Q_{\text{notInferred}} \leftarrow \neg T\text{-equality}(Q_2, Q_1, \emptyset)
$$

The logic schema $DB$ over which we are to check the satisifiability of query $Q_{\text{notInferred}}$ is $DB = (DR_{S1} \cup DR_{S2} \cup DR_{M}, IC_{S1} \cup IC_{S2} \cup IC_{M})$.

As an example, let $F$ be $Q_1 = Q_2$, and let $Q_1$ and $Q_2$ be the following queries defined over the schemas shown in Figure 6.1:

$Q_1$: for $po$ in //purchaseOrder
  return [for $sa$ in $po/shipAddress/singleAddress$ return [$sa/text()$],
  for $ta$ in $po/shipAddress/twoAddresses$
    return [$ta/shipAddress/text(), $ta/billTo/text()$]
  for $it$ in $po/item$
    return [$it/productName/text(), $it/quantity/text(), $it/price/text()$ ]

$Q_2$: for $o$ in /orderDB/order
  where not(/orderDB/item[./order/text() = $o/id/text()$ and ./price/text() > 5000])
  return [for $st$ in $o/shipTo$, $bt$ in $o/billTo$ where $st/text() = $bt/text()$ return [$st/text()$],
  for $st$ in $o/shipTo$, $bt$ in $o/billTo$ where $st/text() \neq $bt/text()$ return [$st/text(), $bt/text()$],
  for $it$ in //item[./order/text() = $o/id/text()$]
    return [$it/name/text(), $it/quantity/text(), $it/price/text()$ ]

Assertion $F$ maps both the purchase orders that have a $twoAddresses$ node, and also those with a $singleAddress$ node. It fixes thus the problem of mapping $M$ not being lossless w.r.t. the $singleAddress$ information (see Section 6.3.2). Let us suppose that now we want to see whether $F$ is inferred from $M$. We apply the CQC method over $Q_{\text{notInferred}}$ and we obtain a counterexample, which consists in a pair of schema instances that satisfy $M$ (because $IC_M$), i.e., they share the $twoAddresses$ nodes, but do not satisfy $F$ (because the definition of $Q_{\text{notInferred}}$), i.e., they do not have the same $singleAddress$ nodes. Therefore, $F$ is not inferred from $M$.  

134
6.4 Experimental Evaluation

To show the feasibility of our approach to validate XML mappings, we perform a series of experiments and report the results in this section. We perform the experiments on an Intel Core2 Duo machine with 2GB RAM and Windows XP SP3.

The mapping scenarios we use in the experiments are adapted from the STBenchmark [ATV08]. From the basic mapping scenarios proposed in this benchmark, we consider those that can be easily rewritten into the class of mapping scenarios described in Section 6.1 and that have at least one level of nesting. These scenarios are the ones called unnesting and nesting. We also consider one of the flat relational scenarios, namely the one called self joins, to show that our approach generalizes the relational case. These mapping scenarios are depicted in Figure 6.4.

For each one of these three mapping scenarios we validate the three properties discussed in Section 6.3, i.e., strong mapping satisfiability, mapping losslessness and mapping inference. In order to do this, we apply the translation presented in Section 6.2 and Section 6.3 to transform each mapping scenario into a logic database schema and the mapping validation test into a query satisfiability test over the logic schema. Note that although STBenchmark [ATV08] expresses the mappings in the global-as-view (GAV) formalism, these mappings can be easily rewritten into mapping assertions in the form of $Q_{source} \subseteq Q_{target}$. Since we have not yet implemented the automatic XML-to-logic translation, we performed the translation manually. The number of constraints and deductive rules in the resulting logic schemas are shown in Table 6.1.

To execute the corresponding query satisfiability tests, we used the implementation of the $CQC_E$ method that is the core of our existing relational schema validation tool $SVT_E$ [FRTU08].

We performed two series of experiments, one in which the three properties hold for each mapping scenario, and one in which they do not. The results of these series are shown in Figure 6.5(a) and Figure 6.5(b), respectively.

Since the mapping inference and mapping losslessness properties must be checked with respect to a user-provided parameter, and given that we want the mappings to satisfy these properties, we check in Figure 6.5(a) whether a “strengthened” version of one of the mapping assertions is inferred from the mapping in each case, and whether each mapping is lossless with respect to a strengthened version of one of its mapping queries. These strengthened queries and assertions are built by taking the original ones and adding an additional arithmetic comparison. Similarly, in Figure 6.5(b), we strengthen the assertions/queries in the mapping and use one of the original ones as the parameter for the mapping inference and mapping losslessness test,
respectively. Regarding strong mapping satisfiability, we introduce two contradictory range restriction, one in each mapped schema, in order to ensure the property will “fail”. We can see in Figure 6.5(a) that the three properties are checked fast in the unnesting and self joins scenarios, while mapping inference and mapping losslessness require much more time to be tested in the nesting scenario. This is not unexpected since the mapping queries of the nesting scenario have two levels of nesting, while those from the other two scenarios are flat. To understand why mapping inference and mapping losslessness are the most affected by the increment of the level of nesting, we must recall how the properties are reformulated in terms of query satisfiability. In particular, the query to be tested for satisfiability in both mapping losslessness and mapping inference encodes the negation of a query inclusion assertion that depends on the parameter query/assertion, as shown in Section 6.3. Therefore, an increment of the level of nesting of the mapping scenario is likely to cause an increment of the level of nesting of the tested query, which is what happens in the nesting scenario; and a higher level of nesting means a more complex translation into logic, involving multiple levels of negation, as shown in Section 6.2.5.

In Figure 6.5(b), we can see that all three properties run fast and that there is no much difference between the mapping scenarios. It is also remarkable the performance improvement of
the nesting scenario with respect to Figure 6.5(a). To understand these results we must remember that mapping inference and mapping losslessness are both checked by means of searching for a counterexample. That means the test can stop as soon as the counterexample is found, while, in Figure 6.5(a), all relevant counterexample candidates had to be evaluated. The behavior of strong mapping satisfiability is exactly the opposite. However, the results of this property in this series of experiments are very similar to those in Figure 6.5(a). The intuition to this is that strong satisfiability requires all mapping assertions to be non-trivially satisfied; thus, as soon as one of them cannot be so, the query satisfiability checking process can stop.
In this chapter, we present MVT, a prototype mapping validation tool that implements the results presented in Chapter 3 and Chapter 4.

MVT allows the designer to ask whether the mapping has certain desirable properties. The answers to these questions will provide information on whether the mapping adequately matches the intended needs and requirements (see Chapter 3). The tool does not only provide a Boolean answer as test result, but also provides additional feedback. Depending on the tested property and on the test result, the provided feedback is in the form of example schema instances (Chapter 3), or in the form of highlighting the mapping assertions and schema constraints responsible for getting such a result (i.e., explanations as defined in Chapter 4).

MVT is able to deal with a highly expressive class of mappings and database schemas defined by means of a subset of the SQL language. Namely, MVT supports:

- Primary key, foreign key, Boolean check constraints.
- SPJ views, negation, subselects (exists, in), union, outer joins (left, right, full).
- Data types: integer, real, string.
- Null values.
- Mapping assertions in the form of $Q_1 \text{ op } Q_2$, where $Q_1$ and $Q_2$ are queries over the mapped schemas, and $\text{op}$ is $=, \subseteq$ or $\supseteq$.

MVT is, to the best of our knowledge, the first implemented tool able to check the kind of properties discussed in Chapter 3 in the context of schema mappings. Implementing the CQC$_E$ method presented in Section 4.2 allows MVT to compute one approximated explanation in the case in which example schema instances are not a suitable feedback for the test result.
Implementing the black-box method from Section 4.1 allows MVT to offer the designer the possibility of refining the approximated explanation into an exact one and compute all the additional possible explanations. Moreover, the fact that MVT also incorporates the treatment of null values is significant, since a single validation test may have a certain result when nulls are not allowed and a different result when they are.

### 7.1 Architecture

MVT extends our database schema validation tool [TFU+04, FRTU08] to the context of mappings. The architecture of MVT is depicted in Figure 7.1.

The GUI component allows using MVT in an easy and intuitive way. To perform the different available tests, users go along the following interaction pattern:

1. Load the mapping and the mapped schemas.
2. Select one of the available desirable property tests.
3. Enter the test parameters (if required).
4. Execute the test.
5. Obtain the test result and its feedback, which can be in the form of example schema instances, or in the form of highlighting the schema constraints and mapping assertions responsible for the test result.

The Test Controller processes the commands and data provided by users through the GUI and transfers back the obtained results.

The Mapping and Mapped Schemas Extractor is responsible for translating the loaded mapping and mapped schemas into a format that is tractable by the CQC$_E$ Method Engine. In this way, it generates an in-memory representation where both the mapping and the schemas are integrated into a single logic database schema that is expressed in terms of deductive rules.

According to the approach presented in Chapter 3, the Test Controller and the Mapping and Mapped Schemas Extractor work together to reformulate the problem of validating the selected mapping property in terms of the problem of testing whether a query is satisfiable over a database schema. The resulting query satisfiability test is performed by the CQC$_E$ Method Engine.

The CQC$_E$ Method Engine implements the CQC$_E$ method, the extended version of the CQC method that we presented in Section 4.2. Recall that the original CQC method [FTU05] can be used to check whether a certain query is satisfiable over a given database schema. It provides an example database instance when the query is indeed satisfiable. However, it does not provide any kind of explanation for why the tested query is not satisfiable. Other validation methods do not provide an explanation for this case either. The CQC$_E$ method addresses this issue. It extends the CQC method so this is able to provide an approximated explanation for the unsatisfiability of the tested query. The provided explanation is the subset of constraints that prevented the method from finding a solution. The explanation is approximated in the sense that it may be not minimal, but it will be as accurate as possible with a single execution of the method.

The Text Controller may ask the Explanation Engine to check whether the explanation provided by the CQC$_E$ Method Engine is minimal or not, and to find the other possible minimal explanations (if any). In order to do that, the Explanation Engine implements the black-box method presented in Section 4.1.

The feedback is translated back to the original SQL representation by the Test Controller and the Mapping and Mapped Schemas Extractor, and shown to the user through the GUI. If the CQC$_E$ Method Engine provides a database instance, it provides an instance of the integrated schema that resulted from the problem reformulation; therefore such an instance has to be split and translated in order to conform to the original mapped schemas. Similarly, if the feedback is
an explanation, i.e., a set of constraints that belong to the integrated schema, these constraints have to be translated in terms of the original mapped schema constraints and mapping assertions.

The whole MVT has been implemented in the C# language, using Microsoft Visual Studio as a development tool. Our implementation can be executed in any system that features the .NET 2.0 framework.

7.2 Example of Mapping Validation with MVT

Let us illustrate the use of MVT by means of an example. Consider the following database schema S1:

```sql
CREATE TABLE Category (  
  name   char(20) PRIMARY KEY,  
  salary real     NOT NULL,  
  CHECK (salary >= 700),  
  CHECK (salary <= 2000) )

CREATE TABLE Employee (  
  name     char(30) PRIMARY KEY,  
  category char(20) NOT NULL,  
  address  char(50),  
  CHECK (category <> 'exec'),  
  KEY(category) REFERENCES Category(name)  
)

CREATE TABLE WorksFor (  
  emp  char(30) PRIMARY KEY,  
  boss char(30) NOT NULL,  
  CHECK(emp <> boss),  
  FOREIGN KEY (emp)  REFERENCES Employee(name),  
  FOREIGN KEY (boss) REFERENCES Employee(name)  
)
```

and the following database schema S2:

```sql
CREATE TABLE Persons (  
  id      int      PRIMARY KEY,  
  name    char(30) NOT NULL,  
  address char(50)  
)

CREATE TABLE Emps (  
  empId  int  PRIMARY KEY,  
  salary real NOT NULL,  
  boss   int,  
  CHECK (salary BETWEEN 1000 AND 5000),  
  CHECK (empId <> boss),  
  FOREIGN KEY (empId) REFERENCES Persons(id),  
  FOREIGN KEY (boss) REFERENCES Emps(empId)  
)
```

and the following mapping assertions between S1 and S2:
MAPPING ASSERTION m1
(SELECT e.name, e.salary
FROM employee e, category c
WHERE e.category = c.name and c.salary >= 10000)
SUBSET OF
(SELECT p.name, e.salary
FROM persons p, emps e
WHERE p.id = e.empId)

MAPPING ASSERTION m2
(SELECT wf.emp, wf.boss
FROM worksFor wf, employee e, category c
WHERE wf.emp = e.name and e.category = c.name
and c.salary >= 1000)
SUBSET OF
(SELECT pEmp.name, pBoss.name
FROM emps e, persons pEmp, persons pBoss
WHERE e.empId = pEmp.id and e.boss = pBoss.id)

The mapping defined by these two assertions states that the employees of S1 that have a salary above a certain threshold are a subset of the emps of S2. Assertion m1 captures information of employees that may or may not have a boss, while assertion m2 takes care of specific information of employees that have a boss. Figure 7.2 shows these schemas and mapping loaded into MVT.

Testing mapping satisfiability. Mapped schemas S1 and S2 are themselves correct in the sense that their constraints are not contradictory. However, when the mapping is considered, it
It turns out that assertion $m_1$ can only be satisfied trivially. That is, if we want to satisfy $m_1$ without violating the constraints in $S_1$, the first query of $m_1$ must get an empty answer (recall that the empty set is a subset of any set).

MVT allows the designer to detect that problem by means of running a mapping satisfiability test (see Figure 7.3). Moreover, it highlights the schema constraints and mapping assertions that are responsible for the problem (see Figure 7.4). In this example, the problem is in the interaction between $m_1$ and the constraint $\text{CHECK (salary <= 2000)}$ from $S_1$. That explanation may help the designer to realize that $m_1$ was probably miswritten, and that it should be mapping those employees with a salary above one thousand, instead of ten thousand.

Figure 7.3: MVT shows a test result.

Figure 7.4: MVT explains why the tested mapping is not strongly satisfiable.
Let us assume that we decide to fix $m_1$ as indicated above. We can load the updated mapping into MVT, perform the satisfiability test again, and see that $m_1$ is now satisfiable. This time, the feedback the tool provides is a pair of instances, one for each mapped schema, that indeed satisfy both $m_1$ and $m_2$ non-trivially (we omit it here).

**Testing mapping assertion redundancy.** The next test uses the mapping inference property [MBDH02] to detect redundant assertions in the mapping. Recall that an assertion is inferred from a mapping if all pairs of schema instances that satisfy the mapping also satisfy the assertion. Based on that, a mapping assertion is redundant if it can be inferred from the other assertions in the mapping (taking into account the mapped schema constraints). Therefore, the expected
feedback for a mapping assertion that is redundant is the set of schema constraints and other mapping assertions the tested assertion is inferred from. If the tested assertion is not redundant, it is better to illustrate that by means of providing a pair of mapped schema instances that satisfy all mapping assertions except the tested one.

To illustrate this test, let us assume that we have come up with an alternative mapping, more compact that the one we already had. It consists of the following single assertion:

MAPPING ASSERTION m3
(SELECT e.name, c.salary, wf.boss
 FROM employee e LEFT OUTER JOIN worksFor wf
  ON wf.emp = e.name, category c
 WHERE e.category = c.name and c.salary >= 1000)
SUBSET OF
(SELECT pEmp.name, e.salary, pBoss.name
 FROM emps e LEFT OUTER JOIN persons pBoss
  ON e.boss = pBoss.id, persons pEmp
 WHERE e.empId = pEmp.id)

The main difference with respect to m1 and m2 is that m3 uses left outer join to capture both the employees with and without boss at the same time. Now, we may want to know how this assertion relates with the other two. Therefore, we load the schemas and the three assertions into MVT, and run the assertion redundancy test. We get the following results (see Figure 7.5).
Assertions \( m1 \) and \( m2 \) are both redundant. The explanation for \( m1 \) is \{\( m3 \)\}. The one for \( m2 \) is \{\( m3, \text{WorksFor.boss NOT NULL} \)\} (see Figure 7.6). However, \( m3 \) is not redundant, and the feedback provided by MVT is the following pair of schema instances (see Figure 7.7):

**Instance of S1:**

- Category('execA', 1000)
- Employee('A', 'execA', null)
- Employee('AA','execA', null)
- WorksFor('A', 'AA')

**Instance of S2:**

- Persons(0, 'A', null)
- Persons(1, 'A', null)
- Persons(2, 'AA', null)
- Emps(0, 1000, null)
- Emps(1, 2000, 2)
- Emps(2, 1000, null)

These schema instances show that \( m3 \) is not only more compact but also more accurate. Assertions \( m1 \) and \( m2 \) allow a single employee from S1 to be mapped to two persons with different ids. Assertion \( m3 \) prevents that by means of the outer join (other formalisms allow expressing this kind of correlations by means of Skolem functions [PVM+02]).

**Testing mapping losslessness.** Recall that we say a mapping is lossless with respect to a given query if the information needed to answer that query is captured by the mapping (see Chapter 3). More formally, mapping \{\( V_1 \text{ op } W_1, \ldots, V_n \text{ op } W_n \)\} is lossless w.r.t. query \( Q \) defined over S1 (S2) if \( Q \) is determined by the extension of the \( V_i (W_i) \) queries (these query extensions must satisfy the mapping assertions). The purpose of this property is to allow the designer to test whether a mapping that may be partial or incomplete is enough for the intended purpose.

When a mapping turns out to be lossy, MVT provides a counterexample as feedback. When the mapping is indeed lossless, the provided feedback is the explanation (schema constraints and mapping assertions) that prevents a counterexample from being constructed.

We illustrate the property with the following example. Let us assume that after replacing the mapping \{\( m1, m2 \)\} with \{\( m3 \)\} we want to know whether the names and addresses of all employees with a salary of at least 1000 are mapped. We perform a mapping losslessness test with the following query as parameter (see Figure 7.8):

```sql
SELECT e.name, e.address
FROM employee e, category c
WHERE e.category = c.name and c.salary >= 1000
```

The result of the test indicates that the mapping is not lossless with respect to the query, and provides the following schema instances as feedback (see Figure 7.9):
Instance 1 of S1:
Category('execA', 1000)
Employee('A', 'execA', null)

Instance 2 of S1:
Category('execA', 1000)
Employee('A', 'execA', 'A')

Instance of S2:
Persons(0, 'A', null)
Emps(0, 1000, null)

The above counterexample shows two instances of S1 that differ in the address of the employee, but are mapped to the same instance of S2 and have the same extension for the queries in the mapping. Seeing this, the designer can realize that the address of the employees is not
captured by the mapping. This result does not mean necessarily that the current mapping is wrong. That depends on the intended semantics. For example, if the address of the employees in S1 was considered classified for some reason, then a lossy mapping would be what the designer wanted. Let us assume that this is not the case, and that the designer decides to modify m3 in order to capture the addresses. Then, it suffices to add e.address and pEmp.address to the “select” clauses of the queries of m3, respectively.

MVT also allows testing the property of query answerability [MBDH02]. We omit its discussion here since its use case is similar to that of the mapping losslessness test (recall that mapping losslessness is a generalization of query answerability).
In this chapter, we review the previous work on the thesis topics and detail how it relates with our contributions. More specifically, we detail how the existing approaches to mapping validation in both the relational and the XML settings relate with ours; we compare our method for computing explanations with previous work on the SAT solver and Description Logics fields; and finally, we discuss existing work related with the topic of translating XML mapping scenarios into a first-order logic formalism.

8.1 Mapping Validation

In this section, we review existing instance-based and schema-based approaches to mapping validation. For each of them, we compare their validation approach with ours, and also the schema and mapping formalism they address with the one we consider.

8.1.1 Instance-Based Approaches

Instance-based approaches rely on source and target instances in order to debug, refine and guide the user through the process of designing a schema mapping. Several instance-based approaches have been proposed during the last years: the Routes approach [CT06], the Spicy system [BMP+08], the approach of Yan et al. [YMHF01], the Muse system [ACMT08], and the TRAMP [GAMH10] suite. They all rely on specific source and target schema instances, which do not necessarily reflect all potential pitfalls.

The Routes approach [CT06] requires both a source and a target instance in order to compute the routes. The Spicy system [BMP+08] requires a source instance to be used to execute the mappings, and a target instance to compare the mapping results with. The system proposed by
Yan et al. [YMHF01] requires a source instance to be available so it can extract from it the examples that it will show to the user. The Muse system [ACMT08] 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. The TRAMP suite [GAMH10] allows querying different kinds of provenance, in particular: data, transformation and mapping provenance. Data and transformation provenance depend on the available instances to indicate which source data and parts of the transformation contribute to a given target tuple. Mapping provenance relies on transformation provenance to determine which mapping assertions are responsible for a given target tuple and also to identify which parts of the transformation correspond to which mapping assertions. As with the previous approaches, detection of errors is entirely left to the user.

All these approaches can therefore benefit from the possibility of checking whether the mapping being designed satisfies certain desirable properties. For instance, such a checking can complement the similarity measure used to rank the mapping candidates in the Spicy system [BMP+08]; for the sake of an example, the designer might be interested on the mapping candidates with a better score in the ranking that preserve some information that is relevant for the intended use of the mapping. Similarly, in the approach of Yan et al. [YMHF01] and the Muse system [ACMT08], the check of desirable properties may be a complement to the examples provided by these systems in order to help choosing the mapping candidate that is closest to the designer’s intentions. The user of the TRAMP suite [GAMH10] could also benefit from the ability to check automatic properties such as the satisfiability of the mapping or the redundancy of mapping assertions. The use of mapping inference to compare alternative mappings could be another useful complement. Desirable properties that are to be parameterized by the user such as query answerability or mapping losslessness could be of help in order to uncover potential flaws that could then be examined in detail with the provenance query capabilities of TRAMP.

Regarding the Routes approach [CT06], the computation of such routes is not only interesting as a complementary tool to the validation of the mapping, but also as a tool to help the designer understand the feedback provided by our approach when this is in the form of a (counter)example. For instance, consider again the counterexample from Section 1.1, which illustrates that mapping assertion $m_3$ is not inferred from mapping assertions $m_1$ and $m_2$. Assume the designer obtains the following counterexample as feedback from the validation test:
It might be difficult to extract from this counterexample the knowledge of what the exact problem is, especially if the schemas were large. To address that, the designer could select the tuples in the instance of $B$ and compute their routes:

- $a_1 \xrightarrow{m_1} b_1, b_4$
- $a_2 \xrightarrow{m_2} b_3, b_4$
- $a_3, a_1 \xrightarrow{m_1} b_2, b_3 \xrightarrow{\text{foreign key Emp-Cat}} b_2, b_3, b_5 \xrightarrow{\text{foreign key Emp-Cat}} b_2, b_3, b_5, b_4$

This way, he could easily see that tuple $b_1$ is produced only by $m_1$ (first route), tuple $b_2$ is produced only by $m_2$ (last route), but tuple $b_3$ is produced by both $m_1$ and $m_2$ (last two routes). Most likely, the designer was expecting all $Employee_B$’s tuples to be produced by both mapping assertions. Moreover, since $b_1$ and $b_2$ both refer to the employee ‘e1’ which is unique in the instance of $A$, this might help the designer to realize the problem of correlation that exists between mapping assertions $m_1$ and $m_2$.

Regarding the schema and mapping formalism, the class of relational mapping scenarios we firstly consider includes the one allowed by Yan et al. system. Yan et al. consider relational schemas with no integrity constraints, and mappings expressed as SQL queries, which may be defined over views and contain arithmetic comparisons and functions. Disregarding functions, which we do not handle, we extend the rest of their schema formalism by allowing integrity constraints. We consider integrity constraints in the form of disjunctive embedded dependencies (DEDs) extended with derived relation symbols and arithmetic comparisons. We also extend the class of mappings they consider by allowing the use of negation (e.g., “not exists” and “not in” SQL expressions). Moreover, our mapping assertions do not consist of a single query but of a pair of queries related by a $\subseteq$ or $=$ operator, that is, we consider an extended GLAV mapping formalism while Yan et al. consider a GAV one.

Routes, Spicy and Muse allow both relational and nested relational schemas with key and foreign key-like constraints—typically formalized by means of TGDs and EGDs—, and mappings expressed as source-to-target TGDs. TRAMP considers a similar setting, but it focuses on flat relational schemas. Comparing with our contributions in the relational 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 [DT05]. 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 GLAV assertions in the form of \( Q_a \subseteq Q_b \) [FKMP05].

Comparing with our contributions in the XML setting, the nested relational formalism considered by Routes, Spicy and Muse is a subclass of the XML schemas we consider. More specifically, we consider XML schemas defined by means of a subset of the XML Schema Definition (XSD) language [W3C04]. We consider the choice construct and the possibility to restrict the range of simple types; features that are not typically allowed in the nested relational formalism. We also consider the XSD’s key and keyref constraints, which subsume the typical constraints in the nested relational setting. Regarding the mapping formalism, the nested mappings [FHH+06] considered by Muse allow the nesting of TGDs, which results in more expressive and compact mappings. The next example illustrates that nested mapping scenarios can be reformulated into the class of mapping scenarios we consider, in particular, into scenarios with mapping assertions in the form of \( Q_{source} \subseteq Q_{target} \), where \( Q_{source} \) and \( Q_{target} \) are expressed in a subset of the XQuery language [W3C07]. Consider the mapping scenario depicted in Figure 8.1(a) (taken from [FHH+06]).

![Diagram of mapping scenarios](image)

**Figure 8.1:** (a) nested mapping scenario and (b) its reformulated version.

The nested mapping in this scenario is the following:
for $p$ in $\text{proj}$ exists $d'$ in $\text{depts}$
where $d'.\text{name} = p.\text{dname} \land d'.\text{emps} = E[p.\text{dname}]$
\land (\text{for } e \text{ in } p.\text{emps} \text{ exists } e' \text{ in } d'.\text{emps})
where $e'.\text{ename} = e.\text{ename} \land e'.\text{salary} = e.\text{salary}$

Notice the use of the Skolem function $E$ to express that employees must be grouped by department name when moved into the target schema. A straightforward reformulation of this mapping scenario is shown in Figure 8.1(b). Since we do not consider function symbols, the source and target schemas must be reformulated in order to make explicit the semantics of the Skolem functions. More specifically, the function $E$ relation is introduced in the source schema in order to simulate the Skolem function $E$. Additional schema constraints are also needed to guarantee that $functionE$ is a functional relation; in particular, both attributes of this relation, namely input and output, must be keys. Also, two referential constraints are needed to state that $functionE$ is defined over the set of department’s names. The fact that $functionE$ generates a unique id for a set of employees is expressed by the following reformulated mapping assertion, which maps the output of $functionE$ into the $\text{empSetId}$ attribute that has been introduced into the target schema to make explicit the semantics that the set of employees has an id:

\[
\begin{align*}
\text{Attribute } \text{empSetId} \text{ must therefore be a key of } \text{empSet}; \text{ since } \text{empSet} \text{ is a nested record, we clarify that it means there may not be two } \text{empSet} \text{ records in the whole target instance with the same value for } \text{empSetId}.
\end{align*}
\]

### 8.1.2 Schema-Based Approaches

Schema-based approaches are those that check certain properties of the mappings by reasoning on the mapped schemas and the mapping definition. Our approach is clearly a member of this group, and it is inspired by the work of Madhavan et al. [MBDH02] (see Section 1.2). Other existing
approaches that are close to ours are those of Sotnykova et al. [SVC+05], Cappellari et al. [CBA10], Amano et al. [ALM09] and Bohannon et al. [BFFN05].

Sotnykova et al. propose a mapping validation phase as a part of their approach to the integration of spatio-temporal database schemas. They use Description Logics (DLs) to represent both the schemas to be integrated and the mappings between them. Description Logics are a family of formalisms for knowledge representation and reasoning, with formal logic semantics [BCM+03]. A DL schema defines the relevant concepts of a domain and the relationships between them (typically, roles, role hierarchies and concept inclusions). A mapping expressed in DL consists of a set of general concept inclusions (GCIs) and concept equivalences, in the form of $C \sqsubseteq D$ and $C \equiv D$, respectively, where concepts $C$ and $D$ are from different schemas. Sotnykova et al. use the description logic SHIQ [HST00] to describe schemas and mappings without any spatial and temporal features, and the description logic ALCRP(D) [HLM99] to specify the spatio-temporal aspects. They rely on the DL reasoning services in order to validate the mappings against the schemas. The validation they perform consists in checking concept satisfiability, that is, checking for each concept whether or not it describes an empty set of instances.

Concept satisfiability relates with our mapping satisfiability property; in particular, it implies weak mapping satisfiability. The intuition is that if all concepts are satisfiable, then for each mapping assertion in the form of $C \sqsubseteq D$ or $C \equiv D$, there is an interpretation in which $C$ is not empty. That means each mapping assertion can be satisfied in a non-trivial way.

To compare our mapping formalism with that used by Sotnykova et al., we disregard the spatio-temporal aspects and focus on the description logic SHIQ. We show that the DL SHIQ is a subset of the relational formalism we consider. In particular, Table 8.1 shows how the SHIQ constructs and axioms can be rewritten as a set of disjunctive embedded dependencies (DEDs). The idea of the translation is to assign a unary relation symbol $P_C$ to each atomic and non-atomic concept $C$, and a binary relation symbol $P_R$ to each role $R$. A set of DEDs is used to explicit the semantics of the constructs and axioms that appear in the DL terminology. Note that although we are able to deal with DEDs extended with derived relation symbols and arithmetic comparisons, the use of derived symbols is not required by this translation and the DL does not allow for arithmetic comparisons; thus, it suffices to consider traditional DEDs with (dis)equalities in the form of $(w \neq w') = w'$, where $w$ and $w'$ are variables or constants [DT01]. As an example, consider the concepts Female, Employee and Department; the role worksIn; and the axiom

$$Employee \sqcap \neg Female \sqsubseteq \exists \text{worksIn}.Department$$
Table 8.1: Translation of DL SHIQ constructs and axioms into DEDs.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Translation into DEDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤ (universal concept)</td>
<td>{P_c(X) → P_r(X)</td>
</tr>
<tr>
<td>⊥ (bottom concept)</td>
<td>{P_c(X) → 1 = 0}</td>
</tr>
<tr>
<td>C,D (conjunction)</td>
<td>{P_c⊔D(X) → P_c(X) ∧ P_d(X), P_c(X) ∧ P_d(X) → P_c⊔D(X)}</td>
</tr>
<tr>
<td>C,D (disjunction)</td>
<td>{P_c⊔D(X) → P_c(X) ⊔ P_d(X), P_c(X) ⊔ P_d(X) → P_c⊔D(X), P_d(X) → P_c⊔D(X)}</td>
</tr>
<tr>
<td>¬C (negation)</td>
<td>{P_c¬C(X) ∧ P_c(X) → 1 = 0, P_c(X) → P_c(X) ⊔ P_c¬C(X)}</td>
</tr>
<tr>
<td>∃R.C (exists restriction)</td>
<td>{P_c∃R.c(X) → ∃Y P_c(Y, X) ∧ P_r(Y), P_c(Y, X) ∧ P_r(Y) → P_c∃R.c(X)}</td>
</tr>
<tr>
<td>∀R.C (value restriction)</td>
<td>{P_c∀R.c(X) → P_c∀R.c(X), P_c¬∀R.c(X) → P_c∀R.c(X)}</td>
</tr>
<tr>
<td>R ∈ R (transitive role)</td>
<td>{P_c(X, Y) ∧ P_c(Y, Z) → P_c(X, Z)}</td>
</tr>
<tr>
<td>R ⊆ S (role hierarchy)</td>
<td>{P_c(X, Y) → P_d(X, Y)}</td>
</tr>
<tr>
<td>≥nR.C (qualifying number restriction)</td>
<td>{P_c≥nR.c(X) → P_r(X) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_n) ∧ P_d(X, Y_1) ≠ Y_n ∧ Y_1 ≠ Y_n ∧ Y_m ≠ Y_n, P_r(X) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_n) ∧ P_d(X, Y_1) ≠ Y_n ∧ Y_1 ≠ Y_n ∧ Y_m ≠ Y_n → P_c≥nR.c(X)}</td>
</tr>
<tr>
<td>≤nR.C (qualifying number restriction)</td>
<td>{P_c≤nR.c(X) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_m) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_m) ∧ Y_1 ≠ Y_2 ∧ ... ∧ Y_m ≠ Y_1 ∧ Y_n ≠ Y_1 ∧ Y_n ≠ Y_1 → 1 = 0} ⊔ {P_r(X) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_m) ∧ P_d(X, Y_1) ∧ ... ∧ P_d(X, Y_m) ∧ Y_1 ≠ Y_2 ∧ ... ∧ Y_m ≠ Y_1 ∧ Y_m ≠ Y_1 → P_c≤nR.c(X) ⊔ Y_m ≠ Y_1 ∧ Y_m ≠ Y_1 ∧ Y_m ≠ Y_1 ∧ 0 ≤ m ≤ n}</td>
</tr>
<tr>
<td>Axiom</td>
<td>Translation into DEDs</td>
</tr>
<tr>
<td>C ⊆ D (general concept inclusion)</td>
<td>{P_c(X) → P_d(X)}</td>
</tr>
<tr>
<td>C ≡ D (C ⊆ D and D ⊆ C)</td>
<td>{P_c(X) → P_c(X), P_c(X) → P_c(X)}</td>
</tr>
</tbody>
</table>

Such a DL schema would be translated into a relational schema with unary relation symbols
P_Employee¬Female, P_Employee, P_Female, P_worksIn.Department and P_Department; binary relation symbol
P_worksIn; and the following set of DEDs:

{ P_Employee¬Female(X) → P_worksIn.Department(X), P_Employee¬Female(X) → P_Employee(X) ∧ P_Female(X), P_Employee(X) ∧ P_Female(X) → P_Employee¬Female(X), P_Female(X) ∧ P_Female(X) → 1 = 0, P_r(X) → P_Female(X) ∧ P_Female(X), P_worksIn.Department(X) → ∃Y P_worksIn(X, Y) ∧ P_Department(Y), P_worksIn(X, Y) ∧ P_Department(Y) → P_worksIn.Department(X), P_Employee¬Female(X) → P_r(X), P_Employee(X) → P_r(X), P_Female(X) → P_r(X), P_worksIn.Department(X) → P_r(X), P_Department(X) → P_r(X),
Checking the satisfiability of a concept $C$ would be thus equivalent to check whether $P_C(X)$ is satisfiable with respect to the set of DEDs. Such test could be performed with the CQC method [FTU05].

Cappellari et al. [CBA10] propose to check the semantic compatibility of schema mappings with respect to a domain ontology $O$. From each source-to-target TGD $q_S(\bar{X}) \rightarrow \exists Y q_T(\bar{X}, \bar{Y})$ in the mapping, they derive a set of verification statements in the form of $\Gamma_q(\bar{x}) \sqsubseteq \Gamma_{q'}(\bar{x})$ (i.e., concept subsumption), where $x$ is a variable from $\bar{X}$ and $\Gamma_q(\bar{x})$, $\Gamma_{q'}(\bar{x})$ denote the concept of the domain ontology assigned to $x$ in the context of the source and target schema, respectively. [CBA10] understands the verification of a mapping as checking, for each verification statement $u$, whether $O \models u$. This semantic compatibility property has a goal similar to that of our mapping satisfiability property, that is, the detection of inconsistencies within the mapping. The main difference is that Cappellari et al. check the consistency of the mapping against an external ontology that models the domain shared by the source and target schema, while we focus on detecting inconsistencies between the mapping assertions and the integrity constraints that are present in the mapped schemas. It is also worth noting the difference in the formalism, that is, Cappellari et al. reason on ontologies while we reason on a class of schemas and mappings that is based on first-order logic—see Table 8.1 for a comparison of our formalism with a set of constructs and axioms commonly used in Description Logics. Nevertheless, given the progressive expansion of the Semantic Web, it would be interesting to study if our approach to mapping validation can also take advantage from such external domain ontologies. As [CBA10] shows, the semantics expressed in this kind of ontologies may uncover mapping flaws that could not be detected by taking only into account the semantics explicitly stated in the mapped schemas.

Amano et al. [ALM09] study the consistency and absolute consistency checking problems for XML mappings that consist of source-to-target implications of tree patterns between DTDs. 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. A mapping is absolutely consistent if all trees that conform to the source DTD are mapped into a tree that conforms to the target DTD. This work extends the previous work of Arenas and Libkin [AL08], where mapping consistency is addressed for a simpler class of XML mappings.

The mapping consistency property of [ALM09] 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 in the relational setting, which we firstly address, the empty database instance is a consistent instance of any database schema; therefore, any mapping is trivially satisfied by an empty source instance and an empty target instance. Moreover, even when we focus on the XML setting, the class of mapping scenarios we consider—with integrity constraints, negations and arithmetic comparisons—makes more 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. Our mapping satisfiability property makes thus sense in both the relational and the XML setting. Another difference is that we consider two flavors of satisfiability: strong and weak. Remind that we say a mapping is weakly satisfiable if at least one mapping assertion can be satisfied in a non-trivial way, and strongly satisfiable if all mapping assertions can be satisfied in a non-trivial way at the same time.

Regarding the absolute consistency property, we do not address it yet, but we intend to do it in future research—see the “Conclusions and Further Research” chapter for some ideas.

Comparing our XML schemas with those addressed by Amano et al, we can say that the subset of the XSD language we consider corresponds to a subset of the DTD language extended with some specific XSD features. In particular, a general DTD is a set of productions in the form of \( A \rightarrow \alpha \), where \( A \) is an element type and \( \alpha \) is a regular expression over element types. Our XML schemas can also be seen as sets of productions that satisfy the following conditions:

1. Each regular expression \( \alpha \) is of the form:

\[
\alpha ::= \text{simple-type} | \varepsilon | B_1, ..., B_n | B_1 + ... + B_n | B^*
\]

where \text{simple-type} denotes the name of a simple type, e.g., string, integer or real; \( \varepsilon \) is the empty word; \( B \) is an element type (a child of \( A \)); ‘+’ denotes disjunction; ‘,’ denotes conjunction; and ‘*’ is the Kleene star.

2. Each element type \( B \) appears in the body of at most one production.

As pointed out in [BFFN05], the first condition does not suppose a loss of generality with respect to general DTDs, since any DTD \( S \) can be rewritten into an \( S' \) that fills this condition, and so the XPath queries over \( S \) can be rewritten into equivalent ones over \( S' \). Similarly, the second condition can be enforced by splitting the repeated element types, that is, if \( B \) is child of \( A_1 \) and \( A_2 \), and \( B \rightarrow \alpha \) is its production, then we can split \( B \) into \( B_{A1} \) and \( B_{A2} \) with productions \( B_{A1} \rightarrow \alpha \) and \( B_{A2} \rightarrow \alpha \).
and \( B_{42} \rightarrow \alpha \). Note that if \( \alpha \) is not a simple type name, then we will have to recursively split its components. Note also that this rewriting of DTD \( S \) into \( S' \) is only possible if \( S \) is not recursive. Regarding the rewriting of the XPath expressions over \( S \), they can be easily rewritten into equivalent ones over \( S' \) as long as they do not use the descendant axis (i.e., ‘/’) over element types that need to be split. Summarizing, our XML schemas are more restricted than those of Amano et al. in the sense that they do not allow for recursion or the use of the descendant axis on element types that appear (directly or indirectly) in the body of more than one production. Nevertheless, our XML schemas do allow for certain integrity constraints that can be found on XSD schemas; in particular, we consider keys, keyrefs and restrictions on the range of simple types (e.g., \( \text{salary} \rightarrow \text{real between 1000 and 5000} \)).

Regarding the mapping formalism, Amano et al. consider implications of tree patterns such as the following (adapted from [ALM09]):

\[
\begin{align*}
\text{Source DTD:} & \\
& \quad \rightarrow \\
& \text{Target DTD:} & \\
& \\
\end{align*}
\]

where the source and target DTD are as follows:

**Source DTD:**
- \( r \rightarrow \text{prof}^* \)
- \( \text{prof} \rightarrow \text{teach}, \text{supervise} \)
- \( \text{teach} \rightarrow \text{year} \)
- \( \text{year} \rightarrow \text{course}1, \text{course}2 \)
- \( \text{supervise} \rightarrow \text{student}^* \)

**Target DTD:**
- \( r \rightarrow \text{courses}, \text{students} \)
- \( \text{courses} \rightarrow \text{course}^* \)
- \( \text{students} \rightarrow \text{student}^* \)
- \( \text{course} \rightarrow \text{taughtby} \)
- \( \text{student} \rightarrow \text{supervisor} \)

This source-to-target dependency states that whenever a source XML document conforms to the premise of the implication, i.e., whenever it contains a professor named \( x \) who teaches courses numbered \( cn_1, cn_2 \) on the year \( y \) and supervises a student named \( s \), then the target document must conform to the pattern in the consequent. The variables in the patterns refer to attributes of the element types. Equalities (\( = \)) and inequalities (\( \neq \)) of such variables are also allowed on both sides.
of the implication, e.g., $c_1 \neq c_2$ could be added to the premise to avoid replicate the course on the target.

Implications of tree patterns can be converted to assertions of the form $Q_{source} \subseteq Q_{target}$. For example, the previous implication can be rewritten as:

```xml
for $c_1$ in /r/courses/course, $c_2$ in /r/courses/course,
    $s$ in /r/students/student
where not($c_1$ is $c_2$) and
$c_1$/@year/text() = $c_2$/@year/text() and
$c_1$/taughtby/@value/text() = $c_2$/taughtby/@value/text() and
$s$/supervisor/@name/text() = $c_1$/taughtby/@value/text()
return
<result>
<prof>{$c_1/@name/text()}</prof>
<year>{$c_1/@year/text()}</year>
<course1>{$c_1/@number/text()}</course1>
<course2>{$c_2/@number/text()}</course2>
<Student>{$s/@name/text()}</student>
</result>
```

Amano et al. allow the use of horizontal navigation axes in the patterns, that is, the “next-sibling” axis and the “following-sibling” axis. We only consider the traditional vertical axes, i.e., the child (’/') axis and the descendant (‘//’) axis, and we also assume set semantics, that is, we disregard the order in which the children of a single node that are of the same element type appear. We do allow for arithmetic comparisons and negations in the XPath expressions and in the “where” clause of the mapping’s queries.

Information preservation is studied by Bohannon et al. [BFFN05] for XML mappings between DTDs. A mapping is information preserving if it is invertible and query preserving. A mapping is said to be query preserving with respect to a certain query language if all the queries that can be posed on a source instance in that language can also be answered on the corresponding target instance. Bohannon et al. show that it is undecidable to determine whether any language subsuming first-order logic is information preserving with respect to projection queries. To address this, they propose the mapping formalism of schema embeddings, which is guaranteed to be both invertible and query preserving with respect to the regular XPath language [Mar04]. Note that query preservation is related to query answerability and mapping losslessness, but is a different property. Query preservation is checked with respect to an entire query language, while query answerability and mapping losslessness are checked with respect to a particular query. Moreover, query answerability and mapping losslessness are aimed at helping the designer to determine whether a mapping that is partial or incomplete—and thus not query preserving—suffices to perform the intended task [MBDH02] (remind that mapping losslessness generalizes
query answerability in order to deal with query inclusion assertions, and that both properties are equivalent when all mapping assertions are query equalities).

Looking at the mapped DTDs as graphs, a schema embedding is a pair of functions: one that maps each node \( A \) in the source DTD—an element type—into a node \( \lambda(A) \) in the target DTD, and another that maps each edge \((A, B)\) in the source DTD—a parent-child relationship—into a unique path from \(\lambda(A)\) to \(\lambda(B)\) in the target DTD.

Comparing with our XML mapping formalism, a schema embedding can be seen as a single query \(Q_{\text{source}}\) that produces a nested-type result that conforms to the target DTD, i.e., the result is the target instance. Such a mapping is a particular case of a mapping with a single assertion of the form \(Q_{\text{source}} = Q_{\text{target}}\). The main difference is that a schema embedding maps each source instance into a unique target instance, while a query equality in its general form maps a single source instance into a set of target instances. That is because although the extension for \(Q_{\text{source}}\) determines the extension of \(Q_{\text{target}}\), there may be more than one target instance in which \(Q_{\text{target}}\) has this extension, which is the well-known view updating problem [DTU96].

Bohannon et al. consider paths expressed in the regular XPath language [Mar04], which allows for qualifying conditions with negations and disjunctions, but not arithmetic comparisons. Regular XPath also allows for position qualifiers to distinguish between multiple appearances of a same element type \(B\) in the body of a single production. This feature is similar to the horizontal navigation considered in [ALM09]. As we already discussed, we only consider vertical navigation and set semantics. Another difference is that Bohannon et al. consider recursive DTDs, but not integrity constraints, while we consider integrity constraints, but not recursive schemas.

Information preservation is also addressed in [BFM04] for XML-to-relational mapping schemes. Such mapping schemes are mappings between the XML and the relational model, and not mappings between specific schemas as are the mappings we consider. It is also worth noting that the notion of lossless mapping scheme defined in [BFM04] corresponds to that of query preservation in [BFFN05] and not to our mapping losslessness property.

A kind of schema-based validation is also performed in the context of ontology matching (a.k.a. ontology alignment). Recall that a matching is a set of correspondences between two schemas, where a correspondence is typically a triplet of the form \((e_1, e_2, sim)\), where \(e_1\) and \(e_2\) are the entities being related (one from each schema), and \(sim\) is a similarity score that measures how likely it is that these two entities are actually related [RB01]. Some matching algorithms use some form of reasoning to improve the quality of the generated matching. That is the case of the
algorithms proposed in [UGM07] and [JSK09], which are algorithms aimed at ontology matching that use the reasoning capabilities of Description Logics.

In [UGM07], Udrea et al. present the ILIADS algorithm that aligns OWL Lite ontologies and makes use of logical reasoning to adjust the similarity measure of the correspondences. The algorithm performs a limited number of inference steps for each candidate correspondence (limited means a given finite number of steps); if it can infer equivalences that are probable, the similarity measure of the current candidate increases; otherwise, the similarity measure decreases.

[JSK09] proposes the ASMOV algorithm for OWL-DL ontology matching with semantic verification. The verification step checks whether certain axioms (of a prefixed kind) inferred from the candidate matching are true given the information in the ontologies. Correspondences whose inferred axioms can be verified are preferred; those whose inferred axioms cannot be verified are removed from the candidate matching.

Note that these approaches are not comparable with ours since they target a different kind of mappings, i.e., matchings, while we focus on logical mappings.

8.2 Computation of Explanations

Existing approaches to validate mappings by means of desirable-property checking focus only on determining whether the tested property holds or not, but do not address the question of what feedback is provided to the user, that is, they only provide a Boolean answer [MBDH02, SVC+05, ALM09]. We address this situation either by returning the (counter)example produced by the CQC method [FTU05] or by highlighting the schema constraints and mapping assertions responsible for the (un)satisfiability of the tested property. We refer to the latter task as computing an explanation.

Our notion of explanation is related to that of Minimal Unsatisfiable Subformula (MUS) in the propositional SAT field [GMP08], and to that of axiom pinpointing in Description Logics [SC03]. In the next subsections, we review the most relevant related work, and show that our approach adapts and combines the main techniques from these two areas.

8.2.1 Explanations in Propositional SAT

The explanation of contradictions inside sets of propositional clauses has received a lot of attention during the last years. A survey of existing approaches to this problem can be found in [GMP08]. The majority of these techniques rely on the concept of Minimal Unsatisfiable
Subformula (MUS) in order to explain the source of infeasibility. A set of propositional clauses $U$ is said to be a MUS of a CNF (Conjunctive Normal Form) formula $F$ if (1) $U \subseteq F$, (2) $U$ is unsatisfiable, and (3) $U$ cannot be made smaller without becoming satisfiable. Notice that what we call an *explanation* is basically the same as a MUS; the difference is that instead of propositional clauses we have schema constraints and mapping assertions.

It is well-known that there may be more than one MUS for a single CNF formula. The most efficient methods for the computation of all MUSes are those that follow the hitting set dualization approach—see the algorithm of Bailey and Stuckey [BS05], and the algorithms of Liffiton and Sakallah [LS05, LS08]. The hitting set dualization approach is based on a relationship that exists between MUSes and CoMSSes, where a CoMSS is the complementary set of a MSS, and a MSS is a Maximal Satisfiable Subformula defined as follows. A set of clauses $S$ is a MSS of a formula $F$ if (1) $S \subseteq F$, (2) $S$ is satisfiable, and (3) no clause from $F$ can be added to $S$ without making it unsatisfiable. The relationship between MUSes and CoMSSes is that they are “hitting set duals” of one another, that is, each MUS has at least one clause in common with all CoMSSes (and is minimal in this sense), and vice versa. For example, assume that $F$ is an unsatisfiable CNF formula with 6 clauses (denoted $c_1$ to $c_6$) and that it has the following set of MSSes and CoMSSes (example taken from [GMP08]):

- MSSes: $\{c_1, c_2, c_3, c_5, c_6\}$, $\{c_2, c_3, c_4, c_6\}$, $\{c_3, c_4, c_5\}$, $\{c_2, c_4, c_5\}$
- CoMSSes: $\{c_4\}$, $\{c_1, c_5\}$, $\{c_1, c_2, c_6\}$, $\{c_1, c_3, c_6\}$

The corresponding set of MUSes would be the following:

- MUSes: $\{c_2, c_3, c_4, c_5\}$, $\{c_1, c_4\}$, $\{c_4, c_5, c_6\}$

Notice that each MUS has indeed an element in common with each CoMSS, and that removing any element from a MUS would invalidate this property.

In order to find all MUSes, the algorithms that follow the hitting set dualization approach start by finding all MSSes, then compute the CoMSSes, and finally find the minimal hitting sets of these CoMSSes. The intuition of why this approach results more efficient than finding the MUSes directly is the fact that, in propositional SAT, finding a satisfiable subformula can be done in a more efficient way than finding an unsatisfiable one, mainly thanks to the use of incremental SAT solvers. Moreover, the problem of finding the minimal hitting sets is equivalent to computing all minimal transversals of a hypergraph; a well-known problem for which many algorithms have been developed.
The problem of applying hitting set dualization in our context is that, to our knowledge, there is no incremental method for query satisfiability checking, and, in particular, it is not clear how to make the CQC method incremental (that could be a topic for further research). Therefore, there is no advantage in going through the intermediate step of finding the MSSes, and finding the MUSes directly becomes the more efficient solution. This intuition is confirmed by the experiments we have conducted to compare our black-box approach with the algorithm of Bailey and Stuckey [BS05].

Since in the worst case the number of MUSes may be exponential w.r.t. the size of the formula, computing all MUSes may be costly, especially when the number of clauses is large. In order to combat this intractability, Liffiton and Sakallah [LS08] propose a variation of their hitting set dualization algorithm that does not compute all CoMSSes neither all MUSes, but a subset of them. This goes on the same direction that the phase 2 of our black-box approach, which is more than a mere intermediate step in the process of finding all explanations. It provides a maximal set of minimal explanations with only a linear number of calls to the CQC method. The idea is to relax completeness so the user can obtain more than one explanation without the exponential cost of finding all of them.

Also because of this intractability, many approaches to explain infeasibility of Boolean clauses focus on the easier task of finding one single MUS. The two main approaches are the constructive [SP88] and the destructive [BDTW93].

The constructive approach considers an initial empty set of clauses $F'$. It keeps adding clauses from the given formula $F$ to the set $F'$ while $F'$ is satisfiable. When $F'$ becomes unsatisfiable, the last added clause $c$ is identified as part of the MUS. The process is iterated with $F' - \{c\}$ as the new formula $F$ and $\{c\}$ as the new set $F'$. The process ends when $F'$ is already unsatisfiable at the beginning of an iteration, which means that $F'$ is a MUS.

The destructive approach considers the whole given formula, and keeps removing clauses until the formula becomes satisfiable. When that happens, the last removed clause is identified as part of the MUS. The process is iterated with the identified and remaining clauses as the new initial formula. The process ends when no new clause is identified.

The computation of one MUS relates with the phase 1 of our black-box method, which is aimed at computing one minimal explanation for the tested mapping property (reformulated as a query satisfiability problem). The phase 1 applies the destructive approach to our context, and combines it with our glass-box method. The idea is to take advantage of the fact that the modified
version of the CQC method—the CQC$_E$ method—does not only check whether a given query is satisfiable but also provides an approximated explanation (i.e., not necessarily minimal) when is not. The destructive approach is the one that is best combined with our glass-box approach since it is expected to perform a lot of satisfiability tests with negative result. Each time a constraint is removed and the tested query is still unsatisfiable w.r.t. the remaining constraints, the CQC$_E$ method will provide an approximated explanation in such a way that in the next iteration of the destructive approach we will be able to remove all remaining constraints that do not belong to the approximated explanation. This way, the number of calls that the phase 1 of our black-box approach makes to the CQC$_E$ method decreases significantly. Moreover, since phase 1 is reused by the two subsequent phases, we can benefit from this combination of the black-box and glass-box approaches not only when we are interested in computing one explanation but also when computing either a maximal set of disjoint explanations or all the possible explanations.

Given that computing one single MUS still requires multiple calls to the underlying satisfiability method, some approaches consider the approximation of a MUS a quicker way of providing the user with some useful insight on the source of infeasibility of the formula. This relates with our glass-box approach, which is also aimed at providing an approximated explanation without additional executions of the CQC method. The work that is most relevant to us is that of Zhang and Malik [ZM03]. They propose a glass-box approach that makes use of a resolution graph that is built during the execution of the SAT solver. The resolution graph is a directed acyclic graph in which each node represents a clause and the edges represent a resolution step. An edge from a node A to a node B indicates that A is one of the source clauses used to infer B via resolution. The root nodes are the initial clauses, and the internal nodes are the clauses obtained via resolution. In order to obtain an approximated MUS, the root nodes that are ancestors of the empty clause are considered. Such a MUS is approximated since it is not guaranteed to be minimal. The drawback of this approach is the size of the resolution graph, which may be very large; actually, in most cases the resolution graph is stored in a file on disk. Besides the storage problems, the additional step that is required to obtain the approximated MUS from the resolution graph may also introduce a significant cost given the necessity of exploring the file in a reverse order. The main difference with respect to our glass-box approach is that our modification of the CQC method does not require keeping in memory the fragment of the search space that has been explored during the execution. The CQC$_E$ method only keeps in memory the current branch (the search space is tree-shaped). The nodes in the current branch store the explanations of their failed subtrees. When the CQC$_E$ method finishes without finding a solution (i.e., a (counter)example for the tested property) the union of the explanations stored in the root
node is the approximated explanation that will be returned to the user, so no additional step is required. Moreover, the modifications introduced to the CQC method do not only preserve the running time, but may reduce it dramatically.

8.2.2 Explanations in Description Logics

Axiom pinpointing is a technique introduced by Schlobach and Cornet [SC03] as a non-standard reasoning service for the debugging of Description Logic terminologies. The idea is similar to that of MUSes in the SAT field, and to our notion of explanations, that is, identify those axioms in a given DL terminology that are responsible for the unsatisfiability of its concepts. They define a MUPS (Minimal Unsatisfiability-Preserving Sub-TBox) as a subset $T'$ of a terminology $T$ such that a concept $C$ from $T$ is unsatisfiable in $T'$, and removing any axiom from $T'$ makes $C$ satisfiable.

Schlobach and Cornet propose a glass-box approach to calculate all the MUPSes for a given concept with respect to a given terminology. The algorithm works for unfoldable ALC terminologies [Neb90] (i.e., ALC terminologies whose axioms are in the form of $C \sqsubseteq D$, where $C$ is an atomic concept and $D$ contains no direct or indirect reference to $C$), although it has been extended to general ALC terminologies in [MLBP06]. The idea is to extend the standard tableau-like procedure for concept satisfiability checking, and decorate the constructed tableau with the axioms that are relevant for the closure of each branch. After the tableau has been constructed and the tested concept found unsatisfiable, an additional step is performed, which applies a minimization function on the tableau (it can also be applied during the construction of the tableau) and uses its result (a Boolean formula) to obtain the MUPSes. The MUPSes will be the prime implicants of the minimization function result, i.e., the smallest conjunctions of literals that imply the resulting formula. Comparing with our glass-box approach, the main difference is that the two approaches have different goals; while our modified version of the CQC method is aimed at providing one approximated explanation without increasing the running time, the algorithm of Schlobach and Cornet provides all the exact MUPS, but that requires and additional exponential cost.

A black-box approach to the computation of MUPSes is presented in [SHCH07]. Since it makes use of an external DL reasoner, it can deal with any class of terminology for which a reasoner exists. The algorithm uses a selection function to heuristically choose subsets of axioms of increasing size from the given terminology. The satisfiability of the target concept is checked against each one of these subsets. The approach is sound but however not complete, i.e., it does
not guarantee that all MUPSes are found. In this sense, it is similar to executing our black-box method until its phase 2, which results in an incomplete but sound set of minimal explanations.

Another way of explaining the result of the different reasoning tasks in Description Logics is explored by Borgida et al. in [BCR08]. Their notion of explanation is different from ours in the sense that they consider an explanation to be a formal proof, which can be presented to the user following some presentation strategy (e.g., tree-shaped proofs). They propose a set of inference rules for each reasoning task commonly performed in DL-Lite [CDL+07]. Then, a proof can be constructed from the premises by means of using the corresponding set of inference rules. As future research, it would be interesting to study how these proof-like explanations can be combined with ours. The motivation would be that highlighting the constraints and mapping assertions responsible for the (un)satisfiability of the tested property may not be enough to fully understand what the problem is, i.e., it may not be clear what the precise interaction between the highlighted elements is, especially if the constraints and assertions are complex. In this situation, providing some kind of proof that illustrates how the highlighted elements relate might be very useful.

8.3 Translation of XML Mapping Scenarios into Logic

In order to apply our validation approach to mappings between XML schemas, we translate the XML mapping scenarios into the first-order logic formalism used by the CQC method (step that is straightforward in the relational setting). This way, we can apply the same technique than with relational mappings, that is, reformulate the desirable-property checking in terms of query satisfiability and apply the CQC method to solve it.

In the next subsections, we firstly show that our translation adapts and combines existing approaches to the translation of XML schemas and queries. Secondly, we show how our translation of the mapping assertions differs from existing ones, which also deal with inclusion and equality assertions although not in the context of mappings but in the context of query containment and query equivalence checking.

8.3.1 Translation of XML Schemas and Queries

Our translation of XML schemas into first-order logic is based on the hierarchical representation used by Yu and Jagadish in [YJ08]. 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 following nested relational schema, which models data about an organization, its employees, and the projects each employee works on:

```
org: Rcd
   org-name

employees: Set Of
   employee: Rcd
      name
      address

projects: Set Of
   project: Rcd
      proj-id
      budget
```

Its hierarchical representation would be the following set of flat relations:

```
{org(@key, parent, org-name), employee(@key, parent, name, address),
   project(@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 the id of XML nodes; and the parent attribute, which references the @key attribute of the parent table, and models the parent-child relationship of XML nodes.

We adapt this hierarchical representation to the subset of the XSD language that we consider. We assign a flat relation in the form of `element(id, parent)` to each schema element of complex type (e.g., `employee(id, parent)`), and a flat relation in the form of `element(id, parent, value)` to each simple-type schema element (e.g., `name(id, parent, value)`, where `name.parent` references `employee.id`). The reason why we define simple-type schema elements as separated flat relations is to make easier the translation of the XSD’s choice construct, which is not considered in the nested relational formalism.

Our translation of the mapping’s XQueries adapts the one used by Deutsch and Tannen in [DT05], and combines it with the hierarchical representation from [YJ08].

Deutsch and Tannen address in [DT05] the problems of XML query containment and XML query reformulation (i.e., rewriting a query through a mapping) by means of reducing these problems into relational ones. This way, the problems can be solved with the `chase` procedure and with the `Chase&Backchase (C&B)` algorithm [DPT99], respectively. The mappings they consider
are in the form of GAV and LAV XQuery views. In order to translate these nested-type views into a flat formalism, Deutsch and Tannen encode each XQuery as a set of $XBind$ queries. They define $XBind$ queries as an analog of relational conjunctive queries; the difference is that instead of relational atoms they have predicates defined by XPath expressions. As an example, consider the following XQuery (taken from [DT05]), which returns a set of items, each item consisting of a writer’s name and a set of book titles written by that writer:

$$Q: \text{for } a \text{ in } //\text{author/text()} \text{ return } \langle \text{item} \rangle \langle \text{writer} \rangle \{a\} \langle /\text{writer} \rangle \{\text{for } b \text{ in } //\text{book}, a1 \text{ in } b/\text{author/text()}, t \text{ in } b/\text{title} \text{ where } a = a1 \text{ return } t \} \langle /\text{item} \rangle$$

An $XBind$ query would be associated to each query block as follows:

$$Xb_{outer}(a) \leftarrow [//\text{author/text()}](a)$$
$$Xb_{inner}(a, b, a1, t) \leftarrow Xb_{outer}(a) \wedge [//\text{book}](b) \wedge [//\text{author/text()}](b, a1) \wedge [./\text{title}](b, t) \wedge a = a1$$

Unary XPath atoms denote absolute paths. For example, $[//\text{author/text()}](a)$ is true if and only if there is an author node in the XML tree whose text is $a$. Similarly, $[//\text{book}](b)$ is true iff $b$ is a book node which is descendant of the root. Binary XPath atoms denote relative paths. For instance, $[./\text{author/text()}](b, a1)$ is true iff there is an author node that is child of node $b$ and whose text is $a1$.

The next step in the translation proposed by Deutsch and Tannen is to replace the XPath atoms with their definition expressed in the GReX ($\text{Generic Relational encoding for XML}$) encoding, in order to convert the $XBind$ queries into relational conjunctive queries. The GReX encoding uses a set of predefined predicates such as: $\text{root}$, $\text{child}$, $\text{desc}$ (descendant), $\text{tag}$ and $\text{text}$ (among others). As an example, the conjunctive queries that result from encoding the two $XBind$ queries above into GReX are the following:

$$B_{outer}(a) \leftarrow \text{root}(r) \wedge \text{desc}(r, d) \wedge \text{child}(d, c) \wedge \text{tag}(c, \text{“author”}) \wedge \text{text}(c, a)$$
$$B_{inner}(a, b, a1, t) \leftarrow B_{outer}(a) \land \text{root}(r) \land \text{desc}(r, d) \land \text{child}(d, b) \land \text{tag}(b, \text{“book”}) \land \text{child}(b, au) \land \text{tag}(au, \text{“author”}) \land \text{text}(au, a1) \land \text{child}(b, t) \land \text{tag}(t, \text{“title”}) \land a = a1$$
The semantics of the GReX predicates is partially modeled by a set of DEDs which the authors call TIX (True In XML). Although TIX does not capture the entire semantics of the GReX predicates, it suffices to solve the query containment and query reformulation problems via the chase and the Chase&Backchase, respectively. It is worth noting that Deutsch and Tannen address the containment of XBind queries only, and do not consider containment of XQueries. Similarly, they address the problem of reformulating an XBind query—not an XQuery—through a mapping that consists of XQuery views.

It is also worth noting that in order to solve the query reformulation problem, the conjunctive queries that result from translating the XQuery views into XBind queries and then encoding them into GReX must be materialized, that is, the mapping queries are replaced by DEDs which keep the materialized predicates updated. This makes possible the subsequent application of the C&B algorithm, whose inputs are a single conjunctive query and a set of DEDs.

Comparing with our approach, we also consider XQueries in the mapping assertions, but we allow for arithmetic comparisons and negations in both the XPath expressions’ conditions and the where clauses of the XQueries. Moreover, our mappings are not GAV or LAV, but GLAV, i.e., each mapping assertion consists of two queries instead of one. That means we need to introduce additional constraints (DEDs) to explicit the semantics of these GLAV assertions. To do that, we rely on the derived predicates that result from the translation of the mapping’s queries. Note that we do not require the mapping’s queries to be materialized, since the CQC method allows for derived predicates.

Nevertheless, one of the main differences of our approach with respect to that of Deutsch and Tannen is that we use the hierarchical representation from [YJ08] instead of the GReX encoding. We apply the hierarchical representation because the resulting translation is closer to the formalism we use when we focus on the relational setting. This way, we can take advantage of our contributions in the relational setting without major modifications.

Another difference is that we assume the schemas are given in a subset of the XSD language [W3C04], and focus on translating these schemas into logic. Deutsch and Tannen assume that the given schemas have been already encoded as sets of XML integrity constraints (XICs), which are in the form of DEDs with XPath atoms instead of relational atoms (like in the case of XBind queries).
8.3.2 Translation of XML Mapping Assertions

Since our mapping assertions are in the form of query inclusions and query equalities, the problem of translating these assertions into first-order logic matches the problem of reducing the containment and equivalence checking of nested queries to some other property checking over relational queries. The works in this area that are closer to ours are those of Levy and Suciu [LS97], Dong et al. [DHT04], and DeHaan [DeH09].

Levy and Suciu address in [LS97] 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 in the form of $Q(\bar{I}_{1}; ..., \bar{I}_{d}; V_{1}, ..., V_{n})$, where $\bar{I}_{1}, ..., \bar{I}_{d}$ denote sets of index variables, and variables $V_{1}, ..., V_{n}$ denote the resulting tuple. For example, consider the following COQL query, which computes for each project the set of employees that work on it:

$$Q: \text{select } [p\.proj-name, (\text{select } e\.name \text{ from Employee } e \text{ where } e\.project = p\.proj-id)] \text{ from Project } p$$

This query would be encoded by the following two indexed queries:

$$Q_{1}(\text{proj-id}; \text{proj-name}) \leftarrow \text{Project}(\text{proj-id}, \text{proj-name, budget})$$

$$Q_{2}(\text{proj-id}; \text{emp-name}) \leftarrow \text{Employee}(\text{emp-name, address, proj-id})$$

In the case of $Q_{1}$, it associates the index $\text{proj-id}$ to each project name; the intuition is that this index denotes the set of employees computed by the inner query. Query $Q_{2}$ indicates which employees are associated with each index. It is worth noting that although we mainly follow the query translation used by Deustch and Tannen [DT05], the idea of index variable has inspired us the concept of inherited variable, which we introduce in our translation in order to avoid the repetition of the outer query blocks in the inner query blocks (e.g., we would like to avoid atoms $X_{b_{\text{outer}}}(a)$ and $B_{b_{\text{outer}}}(a)$ in the example of the previous section)—see Chapter 6.

Relying on the concept of indexed query, Levy and Suciu define the property of query simulation. Let $Q$ and $Q'$ be two indexed queries, $Q$ simulates $Q'$ if for every database instance the following condition holds:

$$\forall \bar{I}_{1} \exists \bar{I}'_{1} \ldots \forall \bar{I}_{d} \exists \bar{I}'_{d} \forall V_{1} \ldots \forall V_{n} [Q(\bar{I}_{1}; ..., \bar{I}_{d}; V_{1}, ..., V_{n}) \rightarrow Q'(\bar{I}'_{1}; ..., \bar{I}'_{d}; V_{1}, ..., V_{n})]$$

They 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. $Q$ strongly simulates $Q'$ if:

$$
\forall \overline{I}_1 \exists \overline{I}'_1 \ldots \forall \overline{I}_d \exists \overline{I}'_d \forall V_1 \ldots \forall V_n [Q(\overline{I}_1; \ldots; \overline{I}_d; V_1, \ldots, V_n) \leftrightarrow Q'(\overline{I}'_1; \ldots; \overline{I}'_d; V_1, \ldots, V_n)]
$$

They 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).

Dong et al. [DHT04] adapt the technique proposed by Levy and Suciu 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. They show that the reduction of COQL query containment proposed by Levy and Suciu is insufficient, since it only considers a subset of the query simulations that should be checked. 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.

DeHaan [DeH09] 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). The idea is to follow the approach proposed by Levy and Suciu, that is, encode the nested queries into flat queries and then reduce the equivalence problem to some property checking over the flat queries. DeHaan shows that the reduction of nested query equivalence to strong query simulation proposed by Levy and Suciu is not correct. He 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 (Levy and Suciu [LS97] and Dong et al. [DHT04] 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 containment and query equivalence into first-
order logic, and then rewrites it into the syntax required by the CQC method by means of some algebraic manipulation. Our goal is to obtain a set of constraints (DEDs) that model the semantics of the mapping assertions, since that is the way in which we encode the mappings when we reformulate the mapping validation tests in terms of query satisfiability in the relational setting. Therefore, translating the XML mapping assertions in this way makes easier to reuse the techniques we have proposed for the case of mappings between relational schemas.
Conclusions and Further Research

Mappings between schemas are key elements in any system that requires interaction of heterogeneous data and applications. A lot of research has focused on the goal of making the mapping design process as automatic as possible, since manually designing mappings is a labor-intensive and error-prone process. However, the design of a mapping always requires the participation of a human engineer to solve the semantic heterogeneities and further refine the proposed mapping. Mapping designers need thus to validate the produced mapping in order to see if it is what was intended.

In this thesis, we have proposed an approach to mapping validation that allows the designer to check whether the mapping satisfies certain desirable properties. We have proposed a reformulation of this desirable-property checking problem in terms of the problem of checking the satisfiability of a query on a database schema. This way, we can take advantage of an existing validation technique that has been successfully used on the area of database schema validation, i.e., the CQC method [FTU05], which allows solving such a query satisfiability problem.

Moreover, we have proposed to provide the mapping designer with a richer feedback for the desirable-property checking than just a Boolean answer. We have proposed to either provide a (counter)example for the tested property, or highlight the mapping assertions and schema constraints that are responsible for the (not) satisfaction of the tested property. Since the former task is already addressed by the CQC method, we have focused on the latter, which we refer to as computing an explanation. To this end, we have adapted and combined techniques from the propositional SAT and Description Logics areas. We have firstly proposed a black-box method that computes all minimal explanations. This method, however, may lead to high running times, especially when the schemas and the mapping are large. To address this, we have also proposed a glass-box approach, that is, a modification of the CQC method such that it produces an
approximated explanation (i.e., not necessarily minimal) as a result of its single execution. We have also combined our glass-box and black-box approaches in order to get the benefits from both.

Since checking the desirable properties of mappings that we consider in this thesis is an undecidable problem, we have proposed to perform a termination test as a previous step to the validation. If the answer of the test is positive, then we can be sure that the corresponding desirable-property checking will terminate. In order to do this, we have adapted and extended a termination test proposed in the area of reasoning on UML/OCL conceptual schemas [QT08]. In particular, we have extended the termination test to handle multiple levels of negation and overlapping cycles of constraints.

Finally, we have gone beyond the relational setting and applied our approach to the validation of mappings between XML schemas. Such mappings have received a growing attention during the last years, especially since the emergence of the Web. Our idea has been to reuse as much as possible the techniques we have developed for the validation of relational mappings, so we can take advantage from them. In order to do so, we have translated the XML mapping scenarios into the first-order logic formalism used by the CQC method (this step was straightforward in the relational setting). This way, we can then apply the same technique than with relational mappings, i.e., reformulate the desirable-property checking as a query satisfiability problem and apply the CQC method to solve it.

As further research, we plan to study a desirable property that we have not considered here: absolute consistency of a mapping, which has been identified in [ALM09]. In a data exchange scenario, a mapping is said absolutely consistent if all valid source instances are mapped into valid target instances. This property is more complex to check than the ones we have considered in this thesis, but yet we think we can adapt the approach we have proposed here to deal with it. The idea would be to address first the case of full mappings, that is, mappings in which the target instance is filled only with data from the source and not with values invented by the mapping (in other words, the mapping does not have existentially quantified variables or Skolem functions). The reason is that the problem of checking absolute consistency of full mappings seems easier to reformulate in terms of a query satisfiability problem. Then, for the case of mappings that are not full, we could compute the “full fragments” of these mappings, i.e., a simplified version of each mapping which has the property of being full, and such that if this fragment is not absolutely consistent, neither is the original mapping. This way, we would have a sufficient condition for a mapping to be not absolutely consistent. Finally, we would like to identify and characterize
classes of non-full mappings in which the former condition is not only sufficient but also necessary.

Also in the context of data exchange, we would like to study whether the CQC method can be used to compute universal solutions for the class of mapping scenarios considered here or at least for the language fragments in which the semantics of data exchange has clearly been established. That would give us a better understanding of the relationship between the CQC method and the well-known chase procedure [FKMP05], which is the procedure typically used to compute universal solutions for data exchange problems. As a starting point, it is easy to see that the application of the CQC method with its Simple VIP (i.e., the Variable Instantiation Pattern that instantiates each variable with a fresh constant) when all schema constraints and mapping assertions are TGDs is equivalent to the application of the standard chase [FKMP05].

Another line for future research is that of improving the efficiency of the CQC method. This is important since our mapping validation approach relies on this method to perform the validation tests. More specifically, there is work to do with the way in which the integrity constraints are evaluated during the method’s execution. Currently, they are evaluated for the whole database instance each time a new tuple is inserted. This situation could be improved by adapting the technique proposed in [MT03] for a view updating and integrity maintenance method. This technique makes explicit an order for the integrity constraints to be handled. The order is provided by a precedence graph in order to minimize the number of times that each constraint is checked during the integrity maintenance process. For instance, assume that you have to check the constraints $ic_1$ and $ic_2$ and that you know the repair of $ic_2$ may lead to the violation of $ic_1$. Then, it is more efficient to check $ic_2$ first, and then check $ic_1$; otherwise you may need to check $ic_1$ two times: before and after the check of $ic_2$. A similar technique is also used in [QT08] which takes advantage of the dependency graph that has been already constructed as a part of the proposed termination test (see Section 1.3.3 and Chapter 5).

It would be also interesting to study the applicability of our mapping validation approach in the field of conceptual modeling. Conceptual schemas are usually richer in semantics than relational or XML schemas, and therefore the ability of our approach to deal with expressive schemas should be especially useful. It should be studied whether existing mapping formalisms between conceptual schemas (e.g., QVT [OMG08]) can be translated into first-order logic, and whether the desirable properties of mappings we consider here still make sense in these formalisms. Related with this, a tool to check desirable properties of UML/OCL conceptual schemas—called AuRUS (Automated Reasoning on UML/OCL Schemas)—is being developed
inside our research group [QRT+10]. This tool uses the CQC method to reason on a first-order logic translation of the conceptual schema; therefore, it would be a natural starting point to be extended in order to address the validation of mappings between conceptual schemas.

Also with the aim of going beyond the relational setting, but in a more generic way, we would like to explore the application of model management in order to translate a given mapping scenario expressed in some formalism into a relational scenario on which we could perform the validation by means of the techniques presented in this thesis. Let us assume we have a mapping from schema $A$ to schema $B$; the model management operators that would be necessary to translate this scenario into a relational one are: ModelGen [ACT+08], in order to translate schema $A$ and $B$ into the relational model; the inversion of a mapping [Fag07], which we would apply to the mapping that relates $A$ with its relational version; and the composition of mappings [BGMN08], which we would use to compose the previously inverted mapping with the mapping that goes from $A$ to $B$ and with the mapping from $B$ to its relational version. The key point here would be to see whether the resulting relational mapping scenario would be equivalent from the point of view of validation to the original one.
References

[ABBG09a] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, Giorgio Gianforme: A runtime approach to model-independent schema and data translation. EDBT 2009: 275-286


[ACB06] Paolo Atzeni, Paolo Cappellari, Philip A. Bernstein: Model-Independent Schema and Data Translation. EDBT 2006: 368-385


[AT96] Paolo Atzeni, Riccardo Torlone: Management of Multiple Models in an Extensible Database Design Tool. EDBT 1996: 79-95


[Ber03] Philip A. Bernstein: Applying Model Management to Classical Meta Data Problems. CIDR 2003


[Cla77] Keith L. Clark: Negation as Failure. Logic and Data Bases 1977: 293-322


[DeH09] David DeHaan: Equivalence of nested queries with mixed semantics. PODS 2009: 207-216


[GMP08] Éric Grégoire, Bertrand Mazure, Cédric Piette: On Approaches to Explaining Infeasibility of Sets of Boolean Clauses. ICTAI (1) 2008: 74-83


[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, Mary Roth: Clio grows up: from research prototype to industrial tool. SIGMOD Conference 2005: 805-810


[KQLJ07] David Kensche, Christoph Quix, Yong Li, Matthias Jarke: Generic Schema Mappings. ER 2007: 132-148

[KQLL07] David Kensche, Christoph Quix, Xiang Li, Yong Li: GeRoMeSuite: A System for Holistic Generic Model Management. VLDB 2007: 1322-1325


[MH03] Jayant Madhavan, Alon Y. Halevy: Composing Mappings Among Data Sources. VLDB 2003: 572-583


184


[PB08] Rachel Pottinger, Philip A. Bernstein: Schema merging and mapping creation for relational sources. EDBT 2008: 73-84


[QKL07] Christoph Quix, David Kensche, Xiang Li: Generic Schema Merging. CAiSE 2007: 127-141


[SV05] Luc Segoufin, Victor Vianu: Views and queries: determinacy and rewriting. PODS 2005: 49-60


[YMHF01] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, Ronald Fagin: Data-Driven Understanding and Refinement of Schema Mappings. SIGMOD Conference 2001: 485-496

[ZM03] Lintao Zhang, Sharad Malik: Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula. SAT 2003