Incremental Checking and Maintenance of UML/OCL Integrity Constraints

Xavier Oriol Hilari

Advised by Ernest Teniente

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor per la UPC

2017

This book is dedicated to its readers.

Acknowledgements

I am afraid that, if I said that my advisor, Ernest Teniente, is an extraordinary hard-working researcher, I would not be the first one, neither the last one, but one of the most sincerest. I would like to thank him for everything I have learnt, including what he has taught me in the research discipline, and computer science, and appreciate the patience he has showed towards me so many times.

In addition, I want to highlight that many of the contributions of this thesis have been developed working with him and other researchers. In particular, Enrico Franconi, Alessandro Mosca and Guillem Rull (for the OCL expressiveness study part), Albert Tort (for the maintenance part), and Giuseppe De Giacomo, Domenico Fabio Savo, and Riccardo Rosati (for the application in DL-Lite).

I have also to thank my family for their support materialized under the mandatory easily-replicable question ‘when are you going to finish?’, and all the (office and non-office) friends with whom I have shared so many coffees, pizzas and beers avoiding such topic in a friendly manner.

Finally, I would like to thank her, Cristina, for showing me that everything in this life is perishable and pointless... until you make it funny. You do not know how much I laugh you.

Abstract

Ensuring the data correctness of some information system is a crucial task. So, software engineers specify sets of integrity constraints that should be satisfied by the system's data. These constraints, however, can be violated every time a user modifies the data state. To avoid so, the system should either *check* that the current update does not cause any violation before applying it, or *maintain* the constraints after applying the update (i.e., apply the update together some additional corrective measures to avoid the violation).

This thesis contributes on both problems, i.e., how to automatically *check* and/or *maintain* a set of integrity constraints after a user update, considering information systems and constraints described with two of the currently most spread conceptual modeling languages: UML for describing the information schema, and OCL for defining its constraints.

In the first part of the thesis, we start analyzing the expressiveness of OCL for defining integrity constraints. In this analysis, we observe that a broad subset of the OCL language is expressively equivalent to relational algebra. Consequently, in the second part, we tackle the integrity checking/maintenance problem departing from a currently existing method for relational databases (the *event rules*) which we extend in several directions. For the case of checking constraints, we extend the *event rules* to deal with aggregation functions, thus, pushing the expressiveness of the constraints they can deal with, and exploit this extension to increase the performance of the original method. For the case of maintenance, we show that, with a slight modification of the *event rules*, we can maintain the constraints using a simple complete implementation of the well-known *chase* algorithm. Furthermore, in order to show the high applicability of our approach, in the third part, we export our work into the context of Description Logics, in which we present a variant of the event rules that permits *maintaining* a DL-Lite ontology in polynomial time.

Contents

Preface

Chapter 1

Introduction

In the developed world, everybody is almost constantly interacting with some information system. Indeed, people communicate, work, buy, get informed, organized and entertained, through data intensive applications we refer as information systems.

The data stored by such information systems changes fast and worldwide. Clearly, messages, purchases, commands, etc. are constantly created, updated and deleted by simple clicks in our PCs or smartphones.

This fact leads to a very simple question: how can we ensure that all these updates are done properly in the information systems? That is, how can the information systems ensure that a user only sends messages to their contacts? How can they ensure that the purchases are done by users with sufficient credit? etc.

For the moment, all these checks are manually implemented in the applications, which might be time consuming and error prone. So, the true question is, can we automatize them? This is basically, the main question that has motivated the development of this PhD Thesis.

In the following, we first discuss the problem we want to tackle with the intention to give the reader an intuitive understanding of our main target. Then, we give an overview of our solution that sketches the full approach that is developed in this thesis. Afterwards, we summarize all our contributions, and finally, we describe the whole document structure.

1.1 The Checking/Maintenance Problem

One of the main functions of any information system is to store the current state of the real world domain we want to capture [12]. In order to guarantee the correctness of the data underlying the information system, the system should be only capable to store *valid* states of the domain [84], that is, states that might *potentially* occur in the real world. This is a crucial point since the capability to store invalid states lets the user store incorrect data in it, either by accident or not.

We can force an information system to store only valid states of the domain by means of integrity constraints. An integrity constraint is a condition that the data of the information system should always satisfy. In this way, if our information system admits some invalid data state regarding the real world, we can prune it by adding a convenient constraint. For example, assume that we want to build an information system for storing TV shows and its episodes. Clearly, in the real world, any TV show has, at least, one episode. Thus, we should define a constraint specifying that every TV show stored in the information system should have, at least, one episode stored.

Such integrity constraints can be *violated* when a user changes the data of the system by means of executing some operation. Continuing the previous example, a user might try to store the TV show *Modern Family* without storing any of its episodes, thus, causing the violation of the previous constraint.

To avoid this violation, the system has mainly two options. The first option consists in *checking* whether the update requested by the user cause an integrity constraint violation before applying it (and cancel its application if a violation is found). The second option consists in applying the update and *maintaining* the violated integrity constraints, i.e., applying additional updates to avoid the constraint violation. Following the previous example, the system can either reject the addition of the TV show *Modern Family*, or add it in the system and additionally associate to it a new episode (e.g. *episode 1*).

Unfortunately, such policies are, for the moment and to the best of our knowledge, manually implemented.

Manually implementing the treatment of integrity constraint violations is an error prone practice [110]. Given that, and considering that industrial-size information systems might store hundreds of concepts [69], it becomes clear that manually implementing the integrity constraints policy associated to them can be a tedious time consuming effort.

This problematic phenomenon leads to the main topic we tackle in this thesis. Indeed, we are interested on how to automatically perform checking/maintenance of constraints in some information system.

Automatically dealing with integrity constraints is an old challenge, present in different communities, still under research. For instance, in the context of databases, the problem appeared when the CREATE ASSERTION statement was defined in the SQL-92 standard [7] to give support to *checking* integrity constraints. However, despite more than 20 years has passed, none of the releases of the current most popular relational database management systems deals with them (Oracle 12, MySQL 5.7, SQLServer 2014, PostgreSQL 9.5, DB2 10.5).

This thesis focuses in the problem of checking/maintenance in the context of *conceptual modelling*, where the problem appears in its research agenda figuring as a grand challenge [85].

1.1.1 Checking/Maintenance in Conceptual Modelling

The target of *conceptual modeling* is to build a description of an information system called *conceptual schema* [84]. A conceptual schema consists of two parts: a structural schema describing the data structure and its constraints, and a behavioral schema describing the operations available for the user to modify such data. Structural schemas are mainly defined by means of a *class diagram* complemented with *textual integrity constraints*, and behavioral schemas are mainly composed of a set of *operation contracts* defining the pre/postconditions of the operations.

A remarkable pair of modeling languages for defining conceptual schemas is UML and OCL, which are standard languages maintained by the OMG group [86, 87]. Whereas UML permits defining the class diagram, and thus, the data to store in the system, OCL allows defining its constraints and operations, and thus, the constraints to check/maintain and the operations that might violate those constraints.

Possibly, one of the greatest software engineering dreams is to exploit these UML/OCL conceptual schemas to automatize the development of the information systems they describe [45, 83]. This goal can be achieved either by directly interpreting the UML/OCL schema, or by (semi)automatically compiling them into actual executable code. In this line, the well-known Model-Driven Architecture approach was proposed more than 10 years ago [79].

However, when pursuing this target, the checking/maintenance problem appears immediately. That is, assuming that we can effectively execute the information system specified by the UML/OCL schema, either by interpreting the schema or by implementing it in code, how can we check/maintain the constraints specified in it? Moreover, how can we efficiently do so?

In response, one promising approach to the *checking/maintenance* of UML/OCL constraints is the so called *incremental* approach [18, 60]. Briefly, incrementally *checking/maintaining* a constraint consists into (1) only apply the already defined checking/maintenance policy to the constraints when the user has applied an operation that might violate them, and (2) only check the values affected by the operation. E.g., in our previous example, we would like to only check/maintain the constraint asserting that each TV show has at least one episode only for the newly inserted TV show *Modern Family* (thus, skipping the rest of TV shows stored in the system), and we do not want to check this constraint for some operation that adds new TV show episodes (since this data update cannot cause the violation of the constraint).

In this thesis, we contribute on the problem of integrity checking/maintenance of constraints, following an incremental approach, and considering data and constraints specified in the UML/OCL language. Nevertheless, as we will see in next chapters, our results can be exported into different languages.

In the following, we state such problems formally. To do so, we assume a function *apply* that given a set of structural events E (i.e., a set of insertions/deletions of data), and some finite data state I, returns the new data state I^n corresponding to apply the insertions/deletions E into it.

Problem 1. UML/OCL Incremental Integrity Checking

Given a UML schema S, a finite data state I, a set of UML/OCL constraints C, and a set of structural events E (i.e., a set of insertions/deletions into I), where $I \models C$, the problem of incremental integrity checking consists in assessing whether $apply(E, I) \models C$.

Problem 2. UML/OCL Incremental Integrity Maintenance

Given a UML schema S, a data state I, a set of UML/OCL constraints C, and a set of structural events E (i.e., a set of insertions/deletions of instances into I), where $I \models C$, the problem of incremental integrity maintenance consists in finding the minimal sets of structural events R s.t. $apply(E \cup R, I) \models C$. We call R to be a *repair* for E, I, and C.

1.2 Our Approach Overview

In the following, we first give a UML/OCL Schema example that will be used along all the thesis to explain our approach. Afterwards, we give a brief overview of our method with the idea to give the reader an intuitive understanding of it.

1.2.1 The Online TV Service UML/OCL Schema

Consider the UML class diagram depicted in Figure 1.1 representing the information system of some streaming TV content service. In this service, *users* can *buy* and *visualize* some *contents*, where some of these contents might be *movies* or *series episodes*, and some of the users might be *premium*.

We show several OCL constraints for such schema in Figure 1.2. The first two define the primary key attributes of content and user to be *code* and *name*, respectively. The *SeenIsBought* and *BoughtContentIsAppropiate* constraints specify that the contents seen by a user should be bought contents, and that their minimal recommended age should be lower than the age of the user. The *AllEpisodesMaxPrice*

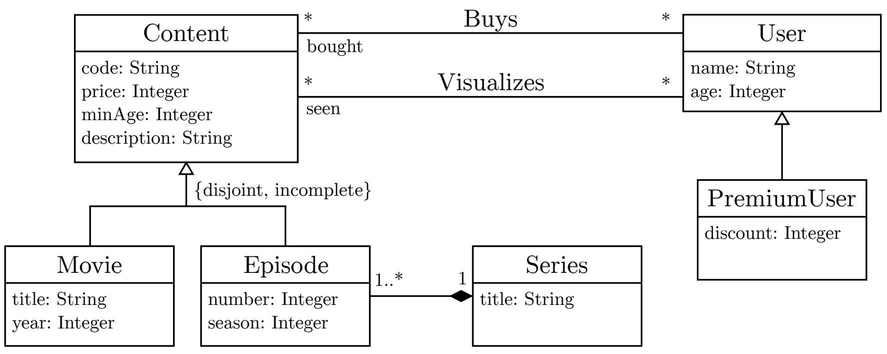


Figure 1.1: UML Schema of an online TV service

limits the sum of the prices of all the episodes of the same series to be, at most, 100, and *PremiumUserBoughtCompleteSeries* establish that a premium user should have bought, at least, all the episodes of some series.

context *Content* **inv** *ContentID* **:**
Content.allInstances()->forAll(c1,c2|c1<>c2 implies c1.code <> c2.code)
context *User* **inv** *UserID* **:**
User.allInstances()->forAll(u1,u2|u1<>u2 implies u1.name<>u2.name)
context *User* **inv** *SeenIsBought* **:**
self.seen->includesAll(self.bought)
context *User* **inv** *BoughtContentIsAppropriate* **:**
self.seen->forAll(c|c.minAge <= self.age)
context *Series* **inv** *AllEpisodesMaxPrice* **:**
self.episode.price->sum() < 100
context *PremiumUser* **inv** *PremiumBoughtCompleteSeries* **:**
Series.allInstances()->exists(s|self.bought->includesAll(s.episode))

Figure 1.2: OCL Constraints for an online TV service

1.2.2 Our Approach To Checking/Maintenance

Our final goal is to provide some method/s to perform incremental integrity checking and maintenance of the constraints defined in some UML/OCL schema like the one presented in Section 1.2.1.

To do so, our first step is to analyze the expressiveness of OCL constraints, which determines the complexity of the problems to solve. Then, we define what we call the *Event-Dependency Constraints*, which are some logic rules that permits solving the problem of checking. Next, we transform the *Event-Dependency Constraints* into another logic rules we call *Repair-Generating Dependencies* to solve the problem of maintenance. Finally, to show the applicability of our method, we adapt the Repair-Generating Dependencies for solving the problem of integrity maintenance in DL-Lite.

In the following we briefly overview each of this steps separately.

OCL Expressiveness Analysis

Interestingly, as we are going to show, OCL is so expressive that *checking* an arbitrary OCL constraint is not even semidecidible. That is, there cannot be any algorithm that can check an arbitrary OCL constraint ensuring its termination. Moreover, termination cannot be ensured not even in the case where the constraint is, in fact, being satisfied.

To tackle this high complexity, we characterize an OCL subset for which the problem of checking is tractable. This approach, of course, encompasses loosing part of the OCL expressive power, so, the idea is to find a balance between expressiveness and checking complexity.

For achieving so, our strategy consists into look for the subset of OCL equivalent to another well studied language: relational algebra. We call such subset OCL_{FO}[1]. Indeed, most OCL constraints can be translated into a relational algebra query that looks for the objects that violates it. For instance, given the OCL constraint *SeenIs-Bought*, we can build the SQL query looking for those users who has seen some content without buying it:

```
SELECT Visualizes.user
FROM Visualizes
        LEFT JOIN Buys ON (
                 Visualizes.user = Buys.user AND
                 Visualizes.seen =  Buys.bought)
WHERE Buys.user IS NULL
```

The OCL subset equivalent to relational algebra is a good starting point for our work since it offers several nice properties. First, its equivalence with relational algebra makes it expressive because it guarantees that any (domain independent) first-order logic constraint can be encoded in it. Second, it ensures good computational properties since checking these constraints can be reduced to executing relational algebra queries. Third, it permits us to build the algorithms for checking/maintaining them departing from current solutions on constraints checking/maintenance in relational databases. Actually, once we have this subset at hand, we continue our work by studying how to solve the incremental integrity checking/maintenance problem adapting a technique whose origins are in the database field.

[1]FO stands for First-Order, since relational algebra is essentially equivalent to first-order logics.

Checking Through Event-Dependency Constraints

Intuitively, our idea to perform incremental integrity checking is to build some logic rules we call Event-Dependency Constraints (EDCs, for short). An EDC is a logic rule that states under which conditions some insertions/deletions causes the violation of some integrity constraint. For instance, for the *SeenIsBought* constraint, we build the following rule (among others):

$$\iota visualizes(u, c) \wedge \neg buys(u, c) \wedge \neg \iota buys(u, c) \rightarrow \bot$$

Roughly speaking, this rule states that, if we insert that some user u visualizes some content c that u has not bough neither we are inserting as bough, then, there is a constraint violation.

Interestingly, such EDCs can be easily implemented as SQL queries. For instance, assume that the previous UML/OCL schema is stored as an SQL schema, and that, for each SQL table *T*, we have two additional tables (*ins_T* and *del_T*) containing the tuples we want to insert/delete in *T*. Given this situation, we can implement the previous EDC as the following SQL query:

```
SELECT ins_Visualizes.user
FROM ins_Visualizes
LEFT JOIN Buys ON (
        Visualizes.user = Buys.user AND
        Visualizes.seen =  Buys.bought)
LEFT JOIN ins_Buys ON (
        Visualizes.user = Buys.user AND
        Visualizes.seen =  Buys.bought)
WHERE Buys.user IS NULL AND ins_Buys.user IS NULL
```

In this manner, we can incrementally check the previous constraint through executing such SQL queries. Indeed, this query is retrieving those users for which we insert some visualized content c, where c is not bought, neither we insert as bought. In other words, the query returns data that witnesses the constraint violation. Note that by using SQL queries, we can exploit all the optimizations present in current relational databases in our favor.

Since there are multiple ways to violate some constraint, each constraint requires several EDCs to incrementally check it. To automatically obtain the complete set of EDCs required for some constraint, we use the so called *event rules* [82], which is a method that was designed for deductive databases to automatically find all the possible ways a constraint can be violated through insertions/deletions. This immediately permits us to incrementally check all the constraints in OCL$_{\text{FO}}$.

Now, we want to push a little bit further the expressiveness of the constraints we can deal with. In particular we show in this thesis how to extend the *event rules* to be able to generate EDCs for constraints involving aggregate operators, such as the constraint *AllEpisodesMaxPrice* of our example. Since, in general, constraints involving aggregate operations are not encodable in relational algebra, we are pushing the expressiveness of the constraints we can deal with beyond OCL_{FO}.

Moreover, as we are going to see, we can exploit these extension for OCL aggregates to improve the treatment of OCL constraints involving existential variables. Indeed, as we are going to see, the EDCs generated by the original *event rules* for OCL constraints involving existential variables lead to some rules whose evaluation, intuitively, encompassed inspecting almost all the data state. However, we improve such behavior by treating existential variables as aggregates.

Additionally, we show in this thesis how to reduce the hight number of EDCs generated by the event rules. Indeed, the event rules generate an exponential number of EDCs for each constraint c, with the length of c. However, we illustrate in this thesis that, if we abstract the common parts of the different generated EDCs, we can generate a linear number of EDCs for each constraint, which suppose an improvement of orders or magnitude with respect to the original proposal.

Maintenance Through Repair-Generating Dependencies

To illustrate our integrity maintenance approach, consider the constraint *SeenIs-Bough*, and the previous EDC we have obtained for it. Now, instead of translating this EDC into SQL as we do for integrity checking, suppose that we move its negated literals representing insertions/deletions from the left-hand side to the right-hand side of the rule:

$$\iota visualizes(u, c) \wedge \neg buys(u, c) \rightarrow \iota bought(u, c)$$

Intuitively, this new rule says that, if some user u visualizes some content c which he has not bought, *then*, we have to insert that u buys c. That is, this rule is telling which are the events (i.e., the insertions/deletions) that we have to apply to satisfy the constraint. We call these rules that tells which are the events that we have to apply to satisfy some constraint Repair Generating Dependencies (RGDs, for short).

Then, the basic idea is that we can repair some integrity constraints by means of *chasing* its RGDs. That is, each time that the left-hand side of some RGD holds, we have to instantiate its right-hand side to avoid a constraint violation. When doing so, it might happen that the left-hand side of another RGD turns to evaluate to true, indicating thus a new constraint violation, which requires instantiating its right-hand side too. Thus, the *chasing* process follows iteratively until no constraint is violated.

However, at this point we identify two new challenges that gains the thesis focus.

First, the problem of maintaining OCL constraints is not decidable, not even in the case for OCL_{FO}, which implies that the previous *chase* procedure cannot ensure termination. To deal with such problem, we look for some OCL fragment in which maintenance is, at least, decidable. This lead to the definition of a new OCL fragment we call OCL_{UNIV}. Intuitively, OCL_{UNIV} is a fragment of OCL in which all the variables are *universally* quantified. Thus, since the source of undecidability are the existential variables [103], and no existential variables occurs in OCL_{UNIV}, chasing the OCL_{UNIV} RGDs ensures termination.

Second, the number of different ways that some violated OCL constraints can be maintained might be exponential. To deal with this problem, we develop some techniques to reduce such exponential search space. In particular, we show that RGDs can be easily customized by the users to permit/disallow some concrete repairs to some constraint violations. For instance, consider that we violate the minimum cardinality constraint stating that each series should have, at least, one episode, because of deleting some episode in some series. In such case, the RGDs tells us that we can maintain such constraint by means of either adding a new episode to such series, or removing such series. However, we can easily customize the RGD to decide one of the two options, in function of the user requirements.

Application in DL-Lite

Finally, we export our technique for integrity maintenance into the context of ontologies.

Differently from the community of conceptual modelling and databases, ontology practitioners usually works with the so called open-world assumption instead of closed-world. That is, whereas in closed-world assumption, some fact is true if and only if it is present in the DB, under the open-world, some fact might be true even when it is not present in the data (since the database is considered that might be incomplete). For instance, consider that the movie *Suicide Squad* is not present in some movie database. According to the open-world assumption, a film called *Suicide Squad* might exists although we do not know. In contrast, under the closed-world assumption, it is considered that *Suicide Squad* is not an existing movie (definitely and for sure).

This different assumption adds some crucial differences in the notion of constraint violation. For instance, under closed-world, a data state with some series without episodes is considered to violate the min. cardinality constraint between series and episodes. However, under the open-world semantics, the same data state is considered not to violate such constraint (since the data is considered to be just incomplete). It might be argued that this problem can be solved by just completing the data before checking the constraints, however, the completion process might never terminate which makes it unfeasible in the general case.

To avoid all this difficulties, we focus our work in DL-Lite language, which is the basis for the OWL 2 QL standard [97]. Indeed, in DL-Lite it is possible to reduce the problem of open-world integrity checking/maintenance into closed-world integrity checking/maintenance [20]. Thus, intuitively, we can check and maintain DL-Lite ontologies by means of checking/maintaining some closed world constraints with our approach. Moreover, DL-Lite ensures that checking/maintenance is decidable. In fact, in this thesis we show that the additional insertions/deletions that we might need to incrementally repair a DL-Lite ontology can be computed through SQL queries, that is, incrementally repairing a DL-Lite ontology is not only decidable, but polynomial (with respect to data complexity).

1.3 Contributions

In the following, we briefly review our contributions by topic, we enumerate the prototype tools we have developed for this thesis, and finally, we list all its related publications.

1.3.1 Contributions by Topic

OCL Constraints Expressiveness

- **Proving OCL checking non-semidecidability** To the best of our knowledge, this is the first study on the complexity of checking OCL constraints. Surprisingly, we find that general OCL constraints checking is not even semidecidable. That is, no algorithm can check a general OCL constraint ensuring its termination, not even when the constraint is, in fact, satisfied.

- **Defining OCL$_{\text{FO}}$** To tackle the OCL checking non-decidability, we suggest a new OCL subset we call OCL$_{\text{FO}}$. OCL$_{\text{FO}}$ is the subset of OCL equivalent to relational algebra (RA). That is, every constraint in OCL$_{\text{FO}}$ can be checked through checking the emptiness of some relational algebra query (which guarantees OCL$_{\text{FO}}$ checking efficiency), and every constraint that can be evaluated through checking the emptiness of a relational query can be written in OCL$_{\text{FO}}$ (which guarantees OCL$_{\text{FO}}$ expressiveness). OCL$_{\text{FO}}$ is formally and concisely defined through set-theory semantics.

- **Defining OCL$_{\text{CORE}}$** To make OCL$_{\text{FO}}$ an easy object of study, we present OCL$_{\text{CORE}}$, a minimal subset of OCL$_{\text{FO}}$ expressively equivalent to it.

UML/OCL Inc. Integrity Checking

- **Definition of EDCs to deal with aggregations** We depart from the work of [82] to perform incremental integrity checking in deductive databases, and extend it to deal deal with constraints with certain aggregations (i.e., *size*, *sum*, and *count*). In this manner, we are able to efficiently incrementally check constraints beyond OCL_{FO}, since such aggregations are not encodable in relational algebra.

- **Improving the event rules treatment of existentials** We show that we can translate constraints involving existentials to constraints involving aggregations. Thus, we can apply the previously incremental techniques developed for aggregations to improve the performance of constraints with existentials, which was one of the most complex cases to treat in [82]

- **Reducing the number of generated EDCs** We improve the event rules technique to generate a linear number of queries to check a constraint. Taking in account that in the original proposal the number of generated queries was exponential with the length of the constraint, this suppose an improvement of orders of magnitude with respect the original proposal.

UML/OCL Inc. Integrity Maintenance

- **Definition of RGDs** We show that with a slight modification of the EDCs, we can obtain some new rules we call RGDs. Using the RGDs, we can compute the repairs (i.e., the additional set of insertions/deletions required to maintain a set of constraints) with a chase procedure, thus, simplifying the previous maintenance methods which were based on first-order resolution techniques [109]

- **Defining OCL_{UNIV}** Since the problem of maintaining a set of OCL constraints is not decidable, and not even in the case of OCL_{FO}, we define a new OCL subset we call OCL_{UNIV} for which the problem of maintenance is decidable. That is, maintaining OCL_{UNIV} constraints by means of chasing its corresponding RGDs ensures termination.

- **Techniques for dealing with multiple solutions** We show that, with the RGDs, we can easily customize the method to reduce the inherent combinatorial explosion in the solution search space of the maintenance problem.

Application in DL-Lite

- **Adapting the EDCs/RGDs to DL-Lite** We show that EDCs/RGDs can be effectively adapted to the DL-Lite language to incrementally check/maintain DL-Lite ontologies.

- **Computing DL-Lite repairs through SQL queries** Using the RGDs, we show that the problem of DL-Lite incremental maintenance can be solved through executing SQL queries.

1.3.2 Implemented Tools

During the development of this thesis, we have implemented the following prototype tools to show the feasibility of our techniques. In particular, we have implemented *TINTIN* for the case of integrity checking, and *IDEFIX* for the case of integrity maintenance.

TINTIN A Tool for INcremental INTegrity checking [92]. TINTIN receives as input a set of constraints and an SQL schema, and automatically generates all the necessary SQL code to automatically check if some data update violates the given constraints. The expressiveness of the constraints TINTIN can deal with is the one of relational algebra (and thus, OCL_{FO}).

IDEFIX IDEntifying missing structural events to FIX-up operation contracts. IDEFIX receives as input a UML class diagram, some OCL constraints, and a set of OCL operation contracts, and returns the set of structural events not specified in the OCL operation contracts that should be included in each operation to ensure their executability.

1.3.3 List of Publications

In the following, we include the list of our publications classified in JCR-indexed journals, international conferences, international workshops/seminars, national conferences, and final degree projects directed.

JCR-Indexed Journals

1. Xavier Oriol and Ernest Teniente. "Simplification of UML/OCL Schemas for Efficient Reasoning". Journal of Systems and Software 2017 (JCR-Q1) [91]

2. Xavier Oriol, Ernest Teniente and Albert Tort. "Computing Repairs for Constraint Violations in UML/OCL Conceptual Schemas". Data & Knowledge Engineering 2015 (JCR-Q2) [95]

3. Enrico Franconi, Alessandro Mosca, Xavier Oriol, Ernest Teniente and Guillem Rull. "OCL_{FO}: Expressive OCL Constraints for Efficient Integrity Checking" (Submitted to SoSyM)

International Conferences

We include, for each publication, the grade of the international conference according to the GII-GRIN Computer Science and Computer Engineering Rating[1] which aggregates several rankings and ratings such as *CORE*, *Microsoft Academic Research Ranking*, and the *Google-Scholar-Based Conference Ranking*.

4. Giuseppe de Giacomo, Xavier Oriol, Montserrat Estañol, and Ernest Teniente. "Linking Data and BPMN Processes to Achieve Executable Models". CAiSE 2017 (A Conference) [36]

5. Giuseppe de Giacomo, Xavier Oriol, Riccardo Rosati, and Fabio Savo. "Updating DL-Lite Ontologies through First-Order Queries". ISWC 2016 (A+ Conference) [37]

6. Xavier Oriol, Ernest Teniente, and Guillem Rull. "TINTIN: a Tool for INcremental INTegrity checking of Assertions in SQL Server". EDBT 2016 (demo track) (A Conference) [92]

7. Xavier Oriol and Ernest Teniente. "Incremental Checking of OCL Constraints with Aggregates through SQL". ER 2015 (A- Conference) [90]

8. Xavier Oriol, Ernest Teniente and Albert Tort. "Fixing up Non-executable Operations in UML/OCL Conceptual Schemas". ER 2014 (**Best Student Paper Award**) (A- Conference) [94]

9. Enrico Franconi, Alessandro Mosca, Xavier Oriol, Ernest Teniente and Guillem Rull. "Logic Foundations of the OCL Modelling Language". JELIA 2014 (short) (B Conference) [54]

10. Xavier Oriol and Ernest Teniente. "OCLuniv: Expressible UML/OCL Conceptual Schemas Finite Reasoning". (Submitted to ER 2017)

11. Giuseppe de Giacomo, Domenico Lembo, Xavier Oriol, Domenico Fabio Savo, and Ernest Teniente. "Practical Update Management in Ontology-based Data Access". (Submitted to ISWC 2017)

[1] http://valutazione.unibas.it/cs-conference-rating/ratingSearch.jsf

International Workshops/Seminars

12. Xavier Oriol and Ernest Teniente. "Incremental Checking of OCL Constraints through SQL Queries". OCL Workshop 2014 [89]

13. Xavier Oriol, Albert Tort and Ernest Teniente. "Reasoning about the Effect of Structural Events in UML". Dagstuhl Seminar on Automated Reasoning on Conceptual Schemas 2013 [22]

National Conferences

14. Xavier Oriol, Ernest Teniente, and Guillem Rull. "TINTIN: comprobación incremental de aserciones SQL". JISBD 2016 [93]

15. Xavier Oriol and Ernest Teniente. "Ejecución de Operaciones de un Esquema Conceptual de forma Persistente y Consistente". Doctoral Consortium - Sistedes 2014 [88]

Final-Degree Projects Direction

- Maria Claver. "SafeEx: Eina per a l'execució d'esquemes conceptuals en UML" (codirected with Ernest Teniente) [26]

1.4 Research Methodology

Some result is *scientific* if it has been obtained by means of a *scientific method*. However, there is not a single definition of a scientific method, but several.

Indeed, pure formal sciences like mathematics base his knowledge acquisition by means of *demonstrations*. Thus, starting from a simple set of definitions (the *axioms*), some new assertions are proofed (the *theorems*). In contrast, some other sciences such as medicine base his knowledge on *experimentation*. Thus, starting from some observations, some hypothesis are made, and such hypothesis are then tested by means of well-defined and replicable experiments.

We argue that software engineering, is in the middle between formal and experimental sciences. Indeed, to *know*, for instance, to what extend is *efficient* some *software application*, it is important not only to give the theoretical computational complexity, which can be obtained by means of formal proofs based on well defined concepts, but also to *test* that *program* in some *realistic* scenario.

With this in mind, we have applied the research methodology which Hevner defined under the name of *Design Science Research*. Briefly, we have spotted a real world important problem to tackle (mainly, integrity checking/maintenance of constraints and subproblems related to it), we have checked the literature for current solutions, and we have pushed them to new limits (as in the case of the *event rules*), and then, we have published our results to make them accessible to the scientific community. These basic interconnected processes are the iterative cycles that Hevner named as *Relevance Cycle*, *Design Cycle*, and *Rigor Cycle*. For more details on the definition and activities of such cycles, we invite the reader to check its publication [64].

Thus, following the previous processes we have: (1) defined the relevant problems we wanted to tackle, (2) specified new solutions to them, (3) mathematically proofed our intended solutions, (4) shown its feasibility by means of tool prototypes, and (5) empirically evaluated our solutions by testing such prototypes. Finally, the major part of these thesis contributions have been reviewed and approved by high-quality conferences and journal committees (including core A conferences and indexed journals), or are in process of revision.

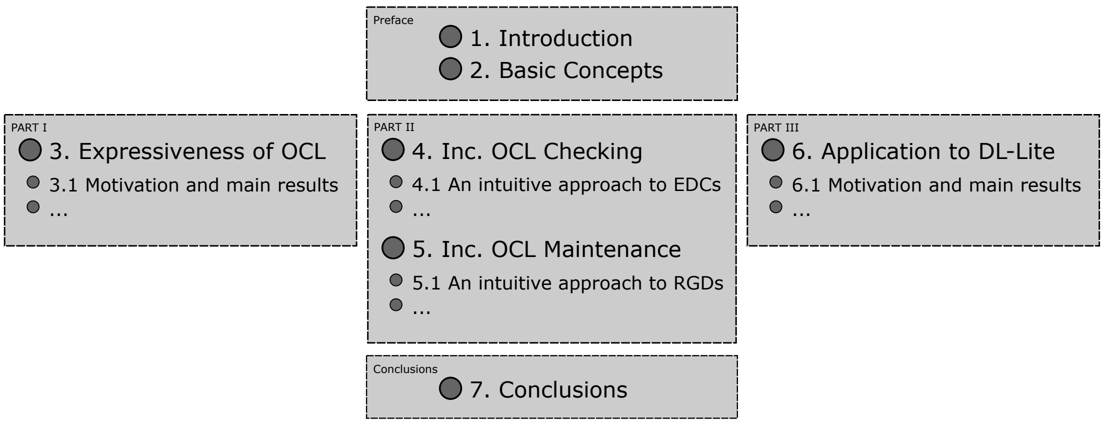


Figure 1.3: Structure of the document

1.5 Document Structure

Figure 1.3 shows the structure of the document. As it can be seen, the thesis starts with some Introduction and Basic concepts chapters. These chapters bring some background and terminology used during the thesis. Afterwards, the document continues with three self-contained parts:

(I) *OCL Constraints Expressiveness*: studying the expressiveness of the OCL language for defining constraints.

(II) *Inc. Integrity Checking/Maintenance in UML/OCL*: showing our technique for incrementally checking/maintaining UML/OCL constraints.

(III) *Application in Description Logics*: showing how similar ideas for maintaining UML/OCL constraints can be applied in DL-Lite.

A reader should be able to read and understand any of these parts separately, thus, he/she can focus on the part that catch his/her interests the most. In any case, we strongly recommend to take a look, at least, to the first section of each chapter before skipping it. This first section is meant to bring the main intuitions and results that might make the following chapters more understandable, apart from bringing the whole picture that has directed this thesis.

Finally, some conclusions and further research are discussed in the last chapter.

Chapter 2

Basic Concepts

Terms, atoms and literals A *term* t is either a variable or a constant. An *atom* is formed by a n-ary *predicate* p together with n terms, i,e., $p(t_1, ..., t_n)$. We may write $p(\bar{t})$ for short. If all the terms $\bar{t}$ of an atom are constants, we call the atom to be *ground* (and to be an instance of p). A literal l is either an atom $p(\bar{t})$, a negated atom $\neg p(\bar{t})$, or a built-in literal $t_i\ \omega\ t_j$, where ω stands for $<$,$\leq$,$=$, or $\neq$.

Derived/base/aggregate predicates A predicate p is said to be *derived* if the boolean evaluation of an atom $p(\bar{t})$ depends on some derivation rules, otherwise, it is said to be *base*. A *derivation rule* has the form: $\forall\bar{t}.\ p(\overline{t_p}) \leftarrow \phi(\bar{t})$ where $\overline{t_p} \subseteq \bar{t}$. In the formula, $p(\overline{t_p})$ is an atom called the *head* of the rule and $\phi(\bar{t})$ is a conjunction of literals called the *body*. We restrict all derivation rules to be safe (i.e., any variable appearing in the head or in a negated or built-in literal of the body also appears in a positive literal of the body) and non-recursive. Given several derivation rules with predicate p in its head, $p(\bar{t})$ is evaluated to true if and only if one of the bodies of such derivation rules is evaluated to true.

An aggregate predicate p_a (aka aggregate query/rule [3, 28, 29]) is a predicate defined over some predicate p that aggregates one of the terms of p with some aggregation function f. An aggregate predicate is defined by means of a rule: $\forall\bar{t}.\ p_a(\overline{t_p}, f(x)) \leftarrow p(\bar{t})$ where $\overline{t_p} \subseteq \bar{t}$ and $x \in \bar{t}$. An atom $p_a(\overline{t_p}, x_f)$ evaluates to true if and only if x_f equals to aggregating all values x in $p(\bar{t})$ by means of f. E.g. given the aggregate predicate $sumSalaries(e, x)$ defined by $sumSalaries(e, sum(s)) \leftarrow salary(e, s)$, $sumSalaries(e, x)$ evaluates to true if and only if x is equal to the sum of all salaries s such that $salary(e, s)$.

We also extend the notion of base/derived/aggregate predicate to atoms and literals. I.e., when the predicate of some atom/literal is base/derived/aggregate, we say that such atom/literal is base/derived/aggregate respectively.

Substitution A *substitution* θ is a set of the form $\{x_1/t_1, ..., x_n/t_n\}$ where each variable x_i is unique. The domain of a substitution is the set of all x_i and is referred as $dom(\theta)$. We say that θ is ground if every t_i is a constant. The literal $l\theta$ is the literal resulting from simultaneously substituting any occurrence of x_i in l for its corresponding t_i. Similarly, we define the conjunction $\phi\theta$ as the conjunction resulting from simultaneously applying the substitution θ to all the literals of ϕ.

Logic formalization of the UML Schema. As proposed in [103] we formalize each class *C* in the class diagram with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $c(Oid)$ together with n atoms of the form $cA_i(Oid, A_i)$, each association *R* between classes $\{C_1, \ldots, C_k\}$ by means of a base atom $r(C_1, \ldots, C_k)$, and, similarly, each association class *R* between classes $\{C_1, \ldots, C_k\}$ with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $r(Oid, C_1, \ldots, C_k)$ together with n atoms $rA_i(Oid, A_i)$. In the context of relational algebra, we might refer to such formalization as the *relational view* of the UML schema.

Data state A data state of some schema is a finite representation of the state of its domain. We represent a data state I as a finite set of ground base atoms.

Constraints A constraint is a logic assertion posed over a schema S that must be satisfied by the data state. Given a constraint c defined over a schema S and an *I* of S, we say that *I* *satisfies* c, i.e., *I*$\models c$, if and only if c evaluates to true in *I*. Otherwise, we say that *I* violates c (*I*$\not\models c$). We naturally extend this notion for a set of constraints C, i.e., *I*$\models C$ iff $\forall c \in C$. *I*$\models c$.

Dependencies. A *Tuple-Generating Dependency (TGD)* is a formula of the form $\forall \overline{x}, \overline{z}.\ \varphi(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\ \psi(\overline{x}, \overline{y})$. For our purposes, $\varphi(\overline{x}, \overline{z})$ will be a conjunction of literals and $\psi(\overline{x}, \overline{y})$ will be a conjunction of positive base atoms and optionally some built-in literals constraining its terms. A *denial constraint* is a special type of TGD of the form $\forall \overline{x}\big(\varphi(\overline{x}) \rightarrow \bot\big)$, in which the conclusion only contains the $\bot$ atom, which cannot be made true. A *Disjunctive Embedded Dependency (DED)* is a variation of TGDs where disjunctions are admitted in the conclusion of the rule. In particular, they follow the form: $\forall \overline{x}, \overline{z}.\ \phi(\overline{x}, \overline{z}) \rightarrow \bigvee \exists \overline{y}.\ \psi(\overline{x}, \overline{y})$. From now on, we omit the logic quantifiers since they can be understood by context.

Structural events and event literals. A *structural event* is an elementary change in the population of a class or association of the schema [84]. That is, a change in the contents of the data state. We consider six kinds of structural events: class instance insertion/deletion, association instance insertion/deletion and attribute instance insertion/deletion. Attribute updates are simulated by means of a simultaneous deletion and insertion of the old and new value respectively.

We denote atom insertions by ι and deletions by δ. That is, given a base atom $p(\overline{x})$, we denote its insertion using the atom $\iota p(\overline{x})$ and its deletion by the atom $\delta p(\overline{x})$. Base atoms of this kind are called *event literals*. Moreover, derived literals containing event literals in their bodies are also considered to be *event literals*.

Part I

OCL Constraints Expressiveness

Chapter 3

Expressiveness of OCL

In this chapter, we start by studying the expressiveness of the OCL language. In particular, we focus on the expressiveness of OCL for defining constraints over UML schemas.

As we are going to see, OCL is so expressive that we can define non-decidable constraints, that is, constraints that cannot be checked in finite time by any algorithm. Moreover, we are going to see that checking general OCL constraints is, in fact, not even semidecidible. In other words, no algorithm can assess the satisfaction of some OCL constraints, not even in the case where these constraints are satisfied.

Motivated by this issue, we define $\mathsf{OCL}_{\mathsf{FO}}$, a fragment of OCL that is equivalent to relational algebra (RA). That is, any constraint in $\mathsf{OCL}_{\mathsf{FO}}$ can be checked through a RA query, and any constraint that can be checked through RA can be defined in $\mathsf{OCL}_{\mathsf{FO}}$. Thus, $\mathsf{OCL}_{\mathsf{FO}}$ constraints are not only decidable, but can be checked in polynomial time (w.r.t. data complexity) and using SQL implementations. This (sub)language is syntactically defined through a formal grammar, and its semantics are given through standard set theory.

To complete the study, we identify the minimal subset of $\mathsf{OCL}_{\mathsf{FO}}$. That is, the minimal set of $\mathsf{OCL}_{\mathsf{FO}}$ constraints that has the full expressive power of RA. We call such subset $\mathsf{OCL}_{\mathsf{CORE}}$.

In the following, we first motivate our OCL expressiveness study, and continue by showing that checking general OCL constraints is not decidable (neither semidecidable). Afterwards, we bring the syntax and semantics of $\mathsf{OCL}_{\mathsf{FO}}$. The next sections prove that $\mathsf{OCL}_{\mathsf{FO}}$ is expressively equivalent to RA, and identify the $\mathsf{OCL}_{\mathsf{CORE}}$. Finally, we discuss some related work and bring some conclusions.

3.1 Motivation and Main Results

Since the definition of the Entity-Relationship (ER) language by Peter Chen in his seminal paper of 1976 [24], several new graphical modeling languages have been proposed so far by different researchers and institutions. Some prominent examples might be the Object Role Modeling language (ORM)[63], and the Unified Modeling language (UML) [86].

Using these languages, a software engineer can specify the conceptual schema of an information system. That is, the relevant concepts of the domain of interest, and how these concepts are related.

For instance, in Figure 3.1 we show a UML conceptual schema for some messaging application. The domain of such application consists of *users*, *conversation groups* (which can be divided into *pairs*, and *groups*) and *messages*. In this schema, users belong to conversation groups, and messages are sent by users to these groups.

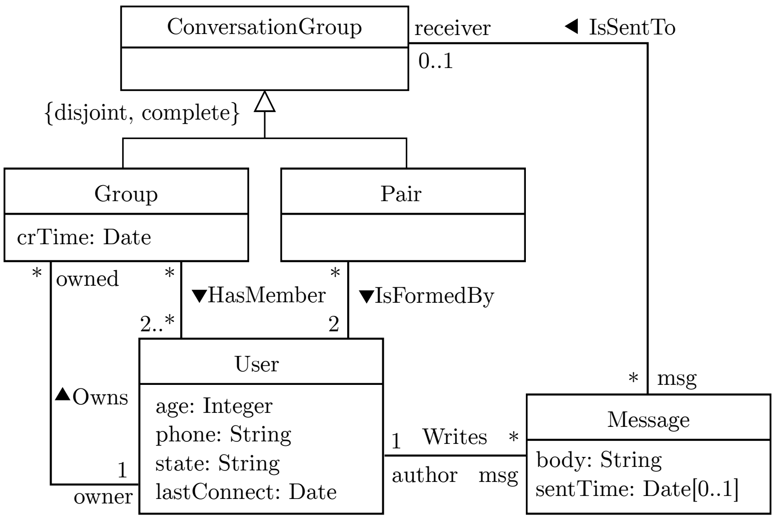


Figure 3.1: UML class diagram

In order to make these schemas precise, conceptual schemas may include *constraints*, i.e., conditions that the data of the schema should fulfill to be considered valid. Then, modeling languages provide themselves some graphical constructs that allow the definition of some frequent constraints. For instance, in our running example, we have stated the constraint that each message is sent to exactly one conversation group by means of two graphical UML cardinalities.

However, due to the limited expressiveness of graphical constraints, we also need in general to adopt a *textual language* to express some more sophisticated constraints. Indeed, constraints such as *Users only send messages to groups they belong to*, and *Messages of a group are sent after the group creation* cannot be graphically expressed in the previous UML schema.

Currently, OCL (Object Constraint Language) [87] is probably the most popular notation to specify textual constraints and an ISO/IEC standard. Roughly, OCL permits defining constraints by means of building navigations from classes/associations, and applying some OCL operators to such navigations. For instance, the previously mentioned constraints can be specified in OCL as:

```
context Group inv MessagesAreFromGroup:
self.user->includesAll(self.msg.author)

context Group inv MessagesAreSentAfterCreation:
self.msg->forAll(m|m.sentTime > self.crTime)
```

Figure 3.2: OCL constraints

However a natural question arising is to what extent is OCL expressive for defining constraints. That is, is OCL expressive enough for defining all the constraint we might require? or, on the contrary, is it even excessively expressive? In fact, the expressive power of a language determines the complexity to check or to analyze its expressions. Therefore, we may want to avoid a language which is too expressive because of its lack of efficiency, but it may happen also that a restrictive language might be efficiently checked but useless.

This is an important question, which has no answer yet in the OCL literature. Probably, the closest study in this area is the one by Mandel and Cengarle in [75] but it was performed more than 15 years ago, and for an old version of OCL which did not include new constructs and capabilities that have been released since then. This is why we understand that a new and more careful analysis must be done. This is specially important if we take in account that, in the OCL panel hold in the OCL Workshop 2014, the OCL community discussed about whether OCL should include more operations (and which ones), or if OCL should just reorganize the current existing ones [13].

In this context, the first contribution of this chapter is to show that full OCL is currently so expressive that it is able to encode non-decidable constraints (and even non-semidecidable ones). That is, we can write OCL constraints for which there is no algorithm able to check them in finite time. That means that OCL interpreters might not be able to assess whether an OCL constraint is satisfied by some data, even in the case that the data is indeed satisfying the constraint. Clearly, this is causing serious difficulties to OCL tool developers like [1, 8, 58] to provide support for the full OCL language.

To overcome this situation we should restrict the expressiveness of the OCL expressions used to specify the constraints. With this purpose, we identify a fragment of OCL which is expressive enough to write the most typically used textual constraints but without loosing good computational properties for checking them. The key idea is to look for the OCL fragment whose expressions can be checked through relational algebra (RA) queries in the sense that the constraint is violated iff the RA query returns a non-empty answer.

For instance, we have that the OCL constraints in Figure 3.2 can be checked by means of the following RA queries:

1. $\pi(\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Writes}) \setminus \pi(\text{Group} \bowtie \text{Member})$

2. $\sigma_{\text{sentTime} \leq \text{crTime}}(\text{Group} \bowtie \text{IsSentTo} \bowtie \text{Message})$

Intuitively, the first query looks for the users which send messages to groups they do not belong to. In this way, one can check if the *MessagesAreFromGroup* constraint is satisfied by checking if this query is empty. Similarly, the second query looks for the messages whose sent time is previous to its group creation time.

We name this fragment of OCL as OCL_{FO}[1]. We define the syntax of OCL_{FO} with a formal grammar and determine its semantics by means of set theory. In this way, the language is fully described in an unambiguous and concise way so that such concise description may be easily understood and adopted by current practitioners. Note that, those approaches aimed at bringing formal semantics to the full OCL require much larger definitions, and even relying on third party languages [14, 76, 87].

Regarding its expressiveness, we show that OCL_{FO} is not only *expressible* in relational algebra, but *equivalent* to relational algebra. That is, every OCL_{FO} constraint can be checked by means of a relational algebra query, and every constraint that can be checked by means of a relational algebra query can be written in OCL_{FO}. This basic property ensures that OCL_{FO} is as expressive as the main language of relational databases and guarantees that the complexity of checking the constraints is exactly the complexity of executing relational queries, that is, polynomial in data complexity (and in particular, AC^0). Moreover, it opens the door to reuse all the accumulated knowledge for efficient query answering in relational databases into efficiently checking of OCL_{FO} constraints, as proposed by incremental approaches like the one in [90].

Finally, in order to make OCL_{FO} an easy object of study, we also identify a *core* of the language that we call OCL_{CORE}. Indeed, OCL_{FO} is targeted to include the fragment of OCL that can be encoded into relational algebra. On the one hand, this makes OCL_{FO} a language containing most of the operators that an OCL practitioner might use. On the other, this makes OCL_{FO} a difficult object of study since it inherits a lot of the OCL syntactic sugar. The *core* of the language is aimed at overcoming this situation since it is a minimal fragment of OCL_{FO} (consisting only of 5 operations) able to express any constraint written in the whole OCL_{FO}.

The identification of this core of the language is also an important issue since it provides two significant contributions. First, it allows to easily state the relationship of

[1]FO stands for First-Order, since relational algebra is, essentially, first-order logic

any fragment of OCL with OCL_{FO}, by determining whether this fragment incorporates or not the five operations in OCL_{CORE}. Second, it entails that any implementation handling OCL_{CORE} will also be able to deal with OCL_{FO}.

3.2 Undecidability of OCL Constraint Checking

It is well known that there is a trade-off between the expressiveness of a language and the computational complexity to *reason* with it. I.e., the greater its expressiveness, the difficult is to *evaluate/analyze* its expressions.

Bearing this in mind, we are interested to know the computational complexity of checking a general OCL constraint in terms of *data complexity*. That is, how difficult is to evaluate an OCL constraint regarding the size of a UML instantiation. Surprisingly, to our knowledge, although there are similar studies on other problems like OCL *maintenance* [95] or *reasoning* [101], this analysis has not been yet performed for integrity checking of OCL constraints.

In fact, it turns out that, unfortunately, current full OCL is so expressive that it is not decidable. That is, it is impossible to build an algorithm guaranteed to terminate and correctly assessing whether an OCL constraint is satisfied or not in an arbitrary UML instantiation. Things get even worse because we can also prove that OCL is not even semidecidable. Therefore, the algorithm is not ensured to terminate even in the case of UML instantiations that satisfy the constraint. Clearly, this entails a huge problem to OCL constraint evaluator techniques since this result implies that they might hang infinitely when checking that some valid UML instantiation is valid.

We prove the previous results by means of reducing the problem of checking an OCL constraint into a non-decidable problem. In particular, we reduce the problem of checking whether some word is accepted by a 0-type grammar to OCL constraint checking by exploiting OCL recursion, and OCL string operators. In the following, we formally state and proof such result.

Property 1. OCL checking is undecidable

Checking whether an OCL constraint is satisfied in an arbitrary UML instantiation is not decidable, and not even semidecidable.

Proof. First, we prove non-decidability. Then, we prove non-semidecidability.

To prove non-decidability, we bring a UML schema that describes a 0-type grammar structure and the notion of accepted/non-accepted words. Then, we add an OCL constraint assessing that instances of accepted words should be produced by the grammar instance. In this way, the problem of checking whether some word is produced by the grammar is reduced to instantiate this grammar and word in our UML schema and then checking whether this OCL constraint is satisfied. Since the former, is a non-decidable problem, the latter turns out to be non-decidable too.

Assume the UML schema of Figure 3.3. This UML schema contains the concepts of a 0-type grammar. That is, *symbols* (distinguishing whether they are terminal, non-terminal, or the start symbol), and *production rules* over such symbols. Moreover, the UML schema also contains the concept of *word*, distinguishing between *accepted* and *non-accepted* words.

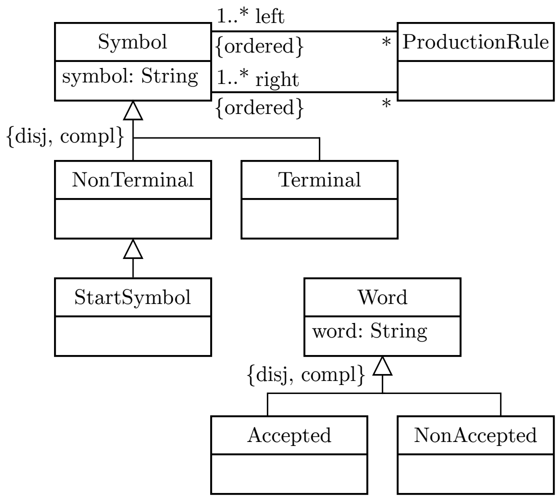


Figure 3.3: UML class diagram for a 0 type grammar

From here, we build an OCL constraint asserting that the instances of accepted words should be produced by the grammar instance. That is, it should exist some production rule that produces the accepted word from the start symbol (either directly, or by means of subsequent production rule applications). This can be defined as follows:

context *Accepted* **inv** *IsProducedWord*:
ProductionRule.allInstances()->exists(p| StartSymbol.allInstances()->exists(s|
        p.produces(s.symbol,self.word)))

where *produces* is an OCL operation that returns true if the first input string directly produces the second one, or if it produces some other word from which we can produce it. This operation can be defined in OCL as follows:

context *ProductionRule* **def** *produces (current: String, target: String): Boolean =*
self.replacements(current)->exists(newWord|
    newWord = target or

ProductionRule.allInstances()->exists(p| p.produces(newWord, word))

This definition makes use of the *replacements* operation, which returns the different words we can obtain by applying the production rule. *Replacements* can be defined recursively in OCL. Intuitively, we need to first compute the strings representing the left and right part of the production rule. Then, if the input word is empty, there is no replacement to apply and thus, we can return the empty set. Otherwise, we have to recursively compute the replacements of the word without the first character, and iterate the given result to concatenate this first character at the beginning of each returned word. Finally, we have to check if the word's beginning matches the left part of the rule, and if so, add into the result the word corresponding to replace the beginning with the right part of the production rule. Formally:

```
context ProductionRule def replacements(current: String): Set(String) =
let leftW: String = self.left->iterate(s; l: String = "" | l+s.symbol) in
let rightW: String = self.right->iterate(s; r: String = "" | r+s.symbol) in
if current = "" then Set{}
else self.replacements(
        current.substring(2,current.size())
    )->iterate(i; acc: Set{}|
        acc->including(current.substring(1,1)+i)
    )->union(
        if current.substring(1,leftW.size())=leftW
            then Set{rightW+current.substring(leftW.size(),current.size())}
        else Set{} endif )
endif
```

Thus, we can check whether some word is rejected by a 0-type grammar by instantiating the word and the grammar in the previous UML schema and checking the satisfaction of the given *IsProducedWord* OCL constraint. Since checking whether some word is rejected by a 0-type grammar is not decidable, checking OCL constraints is not decidable either.

Now, to prove that checking OCL constraints is not even semidecidable, we reduce the problem of checking whether some word is rejected by some 0-type grammar, which is a well known non-semidecidable problem.

Let us consider a constraint specifying that non-accepted words are words that cannot be produced by the grammar. In OCL, this constraint can be stated by simply negating the previous one:

```
context NonAccepted inv IsNotProducedWord:
not ProductionRule.allInstances()->exists(p|StartSymbol.allInstances()->exists(s|
        p.produces(s.symbol,word.word)))
```

We can check now whether some word is rejected by a 0-type grammar by instantiating the word and the grammar in the previous UML schema and checking the satisfaction of the given *IsNotProducedWord* OCL constraint. Therefore, checking OCL constraints is not even semidecidable.

□

3.3 The OCL$_{\text{FO}}$ Fragment of OCL

We provide in this section the syntax and the semantics of OCL$_{\text{FO}}$. Since our goal is to use OCL$_{\text{FO}}$ as a language for specifying constraints, we make special emphasis on OCL$_{\text{FO}}$ boolean statements.

The syntax is defined by means of a formal grammar which limits the standard OCL boolean statements to those that can be computed through relational algebra queries. Since, intuitively, relational algebra is known to be equivalent to (domain independent) first-order logic, such grammar leaves out the OCL higher-order operators like transitive closure, or (most) aggregation functions.

Semantics is defined by means of set theory. Roughly, OCL navigations are interpreted as sets, or single objects/values, and OCL boolean operators are interpreted as checks over them (e.g. the semantics of *includes* consists in checking whether some object/value belongs to a set, etc.). Therefore, this semantics does not distinguish among different collection types, as OCL does (e.g. it does not contemplate bags, nor ordered sets). However, this is not a limitation since RA only supports sets and most OCL operators regarding collections brings the same results when interpreting an OCL collection as a set.

After defining the syntax and the semantics of OCL$_{\text{FO}}$, we make a brief discussion about the OCL operations outside OCL$_{\text{FO}}$ while distinguishing whether they could be effectively emulated in OCL$_{\text{FO}}$, or not.

3.3.1 OCL$_{\text{FO}}$ Syntax

The grammar of OCL$_{\text{FO}}$ is stated in Figure 3.4. Briefly, an OCL$_{\text{FO}}$ constraint is an *OCL-Bool* statement written in some class context as shown in Figure 3.2. Such boolean statement might make use of *navigations*, i.e., *OCL-Set*, *OCL-Object*, or *OCL-Value* statements. Intuitively, the first kind of statements describe a set of objects, whereas the last two determine a single object/value, respectively. Such navigations are then used as the input of some OCL$_{\text{FO}}$ operator to obtain the *OCL-Bool* statement that defines the constraint.

```
OCL-Bool      ::=  OCL-Bool BoolOp OCL-Bool | not OCL-Bool |
                   OCL-Set ->includesAll(OCL-Set) | OCL-Set ->excludesAll(OCL-Set) |
                   OCL-Set ->includes(OCL-Single) | OCL-Set ->excludes(OCL-Single) |
                   OCL-Set ->forAll(VarL | OCL-Bool) | OCL-Set ->exists(VarL | OCL-Bool) |
                   OCL-Set ->isEmpty() | OCL-Set ->notEmpty() |
                   OCL-Set ->size() CmpOp Integer | OCL-Set ->one(Var | OCL-Bool) |
                   OCL-Set ->isUnique(attr) |
                   OCL-Object▪oclIsKindOf(Class) | OCL-Object▪oclIsTypeOf(Class) |
                   OCL-Object = null | OCL-Object <> null |
                   OCL-Navig = OCL-Navig | OCL-Navig <> OCL-Navig |
                   OCL-Value CmpOp OCL-Value |
                   OCL-Object▪bAttr | Var
OCL-Navig     ::=  OCL-Set | OCL-Single
OCL-Set       ::=  OCL-Set ->union(OCL-Set) | OCL-Set ->intersection(OCL-Set) |
                   OCL-Set ->select(Var | OCL-Bool) | OCL-Set ->reject(Var | OCL-Bool) |
                   OCL-Set ->selectByKind(Class) | OCL-Set ->selectByType(Class) |
                   OCL-Set▪role [ [role] ] | OCL-Set▪assoClass [ [role] ] |
                   OCL-Object▪nfRole [ [role] ] | OCL-Object▪nfAssoClass [ [role] ] |
                   OCL-Set▪attr | OCL-Object▪nfAttr |
                   Class▪allInstances() | OCL-Single
OCL-Single    ::=  OCL-Object | OCL-Value
OCL-Object    ::=  OCL-Object▪oclAsType(Class) |
                   OCL-Object▪fRole | OCL-Object▪fAssoClass | Var | self
OCL-Value     ::=  Constant | Var
                   OCL-Object▪fAttr |
                   OCL-Set->min() | OCL-Set->max() |
BoolOp        ::=  and | or | xor | implies
CmpOp         ::=  < | <= | = | >= | > | <>
VarL          ::=  Var (,Var)*
Var           ::=  ⟨a variable name⟩
Class         ::=  ⟨a class name⟩
assoClass     ::=  ⟨an association class name⟩
fAssoClass    ::=  ⟨an association class name of a functional role⟩
nfAssoClass   ::=  ⟨an association class name of a non functional role⟩
role          ::=  ⟨a role name⟩
fRole         ::=  ⟨a functional role name⟩
nfRole        ::=  ⟨a non functional role name⟩
attr          ::=  ⟨an attribute name⟩
bAttr         ::=  ⟨a boolean attribute name⟩
fAttr         ::=  ⟨a functional attribute name⟩
nfAttr        ::=  ⟨a non functional attribute name⟩
Integer       ::=  ⟨an integer number⟩
Constant      ::=  ⟨a constant name⟩
```

Figure 3.4: Syntax of OCL$_{\text{FO}}$

These OCL$_{\text{FO}}$ statements are built over a *signature* consisting of a set of class names, role/attribute names (where some might be functional -i.e. with a maximum cardinality of 1-), association class names, and constant names. Typically, this signature is provided by an associated UML class diagram.

For the seek of simplifying the language, we limit OCL$_{\text{FO}}$ statements to evaluate to valid results. Thus, we require the expressions to apply the proper safety checks to avoid rising the OCL *invalid* value in runtime. For instance, if we have some object of type T_1, and we want to cast it to T_2, we might need to check that the object has also the type T_2 (unless T_1 is a subclass of T_2). Note that we can analyze in which cases are these safety checks necessary with a syntactic inspection of the OCL expression and the class diagram.

3.3.2 OCL$_{\text{FO}}$ Semantics

To define the semantics of OCL$_{\text{FO}}$ we interpret the *OCL-Bool* statements as TRUE or FALSE values, the *OCL-Set* statements as sets of objects/values, and the *OCL-Object*/*OCL-Value* statements as a single object/value respectively.

We define first the semantics of OCL$_{\text{FO}}$ without considering NULL values. That is, assuming that when interpreting an *OCL-Object* or *OCL-Value* expression, we always reach some defined object/value. We will include the treatment of *nulls* later on.

OCL$_{\text{FO}}$ semantics without nulls

The semantics of an OCL$_{\text{FO}}$ statement is defined through the *interpretation* of its signature. Such interpretation represents a specific *database state* of the class diagram where the constraint is attached to. Namely, it indicates the classes each object is instance of, the relations between objects via associations, and the values objects have via their attributes. Since the interpretation of an association name determines the interpretation of its role names, instead of considering the interpretation of the roles, we assume a function ass : *role* $\mapsto$ *Assoc*, for retrieving the association name of some role name. In this way, the interpretation of some role r is obtained by taking the interpretation of its association $ass(r)$.

Formally, an *interpretation* is a pair $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is a non-empty set of object identifiers and values referred as *the interpretation domain*, and $\cdot^{\mathcal{I}}$ is a function, referred as *interpretation function*, that maps each element in the signature of the OCL$_{\text{FO}}$ statements to $\Delta^{\mathcal{I}}$ tuples. In particular, class names are mapped to a set of domain elements, attribute names are mapped to a set of domain element pairs, k-ary association (class) names are mapped to a set of k-ary (k+1-ary) domain element tuples, and constant names are interpreted to domain elements with the same name (i.e., we follow the standard name assumption).

For example, an *interpretation* $\mathcal{I}_0$ for the signature defined by our running UML schema example might be:

$$\begin{aligned}
&\Delta^{\mathcal{I}_0} = \{\#\mathit{user1}, \#\mathit{user2}, \#\mathit{group1}, \#\mathit{msg1},\\
&\quad 1/1/2016, 12/12/2015, \text{‘}\mathit{Happy\ new\ year!}\text{’}, \ldots\}\\
&\mathit{User}^{\mathcal{I}_0} = \{\#\mathit{user1}, \#\mathit{user2}\}\\
&\mathit{Group}^{\mathcal{I}_0} = \{\#\mathit{group1}\}\\
&\mathit{Message}^{\mathcal{I}_0} = \{\#\mathit{msg1}\}\\
&\mathit{crTime}^{\mathcal{I}_0} = \{< \#group1, 12/12/2015 >\}\\
&\mathit{sentTime}^{\mathcal{I}_0} = \{< \#msg1, 1/1/2016 >\}\\
&\mathit{isSentTo}^{\mathcal{I}_0} = \{< \#msg1, \#group1 >\}
\end{aligned}$$

OCL-Bool^I	$\in$	{TRUE, FALSE}
(OCL-Bool$_1$ BoolOp OCL-Bool$_2$)^I	$\equiv$	OCL-Bool$_1{}^I$ BoolOp OCL-Bool$_2{}^I$
(not OCL-Bool)^I	$\equiv$	$\neg$OCL-Bool^I
(OCL-Set$_1$->includesAll(OCL-Set$_2$))^I	$\equiv$	OCL-Set$_1{}^I$ $\supseteq$ OCL-Set$_2{}^I$
(OCL-Set$_1$->excludesAll(OCL-Set$_2$))^I	$\equiv$	OCL-Set$_1{}^I$ $\cap$ OCL-Set$_2{}^I$ $= \emptyset$
(OCL-Set->includes(OCL-Single))^I	$\equiv$	OCL-Single^I $\in$ OCL-Set^I
(OCL-Set->excludes(OCL-Single))^I	$\equiv$	OCL-Single^I $\notin$ OCL-Set^I
(OCL-Set->forAll(Var $\mid$ OCL-Bool))^I	$\equiv$	(not OCL-Bool)$^{I,\texttt{Var},\textit{OCL-Set}}$ $= \emptyset$
(OCL-Set->forAll(VarL, Var $\mid$ OCL-Bool))^I	$\equiv$	OCL-Set->forAll(Var $\mid$ OCL-Set->forAll(VarL $\mid$ OCL-Bool))^I
(OCL-Set->exists(Var $\mid$ OCL-Bool))^I	$\equiv$	OCL-Bool$^{I,\texttt{Var},\textit{OCL-Set}}$ $\neq \emptyset$
(OCL-Set->exists(VarL, Var $\mid$ OCL-Bool))^I	$\equiv$	OCL-Set->exists(Var $\mid$ OCL-Set->exists(VarL $\mid$ OCL-Bool))^I
(OCL-Set->isEmpty())^I	$\equiv$	OCL-Set^I $= \emptyset$
(OCL-Set->notEmpty())^I	$\equiv$	OCL-Set^I $\neq \emptyset$
(OCL-Set->size() CmpOp n)^I	$\equiv$	$\lvert\lvert$OCL-Set^I$\lvert\lvert$ CmpOp n
(OCL-Set->one(Var $\mid$ OCL-Bool))^I	$\equiv$	$\lvert\lvert$OCL-Bool$^{I,\texttt{Var},\textit{OCL-Set}}$$\lvert\lvert$ $= 1$
(OCL-Set->isUnique(attr))^I	$\equiv$	OCL-Set->forAll(v1,v2 $\mid$ v1 = v2 or v1.attr <> v2.attr)^I
(v.oclIsKindOf(Class))^I	$\equiv$	$v \in$ Class^I
(v.oclIsTypeOf(Class))^I	$\equiv$	$v \in$ Class^I $\setminus$ Subclasses(Class)^I
(OCL-Single = null)^I	$\equiv$	OCL-Single^I = NULL
(OCL-Single <> null)^I	$\equiv$	OCL-Single^I $\neq$ NULL
(OCL-Set$_1$ = OCL-Set$_2$)^I	$\equiv$	OCL-Set$_1{}^I$ = OCL-Set$_2{}^I$
(OCL-Set$_1$ <> OCL-Set$_2$)^I	$\equiv$	OCL-Set$_1{}^I$ $\neq$ OCL-Set$_2{}^I$
(OCL-Object$_1$ = OCL-Object$_2$)^I	$\equiv$	OCL-Object$_1{}^I$ = OCL-Object$_2{}^I$, or OCL-Object$_i$ = NULL
(OCL-Object$_1$ <> OCL-Object$_2$)^I	$\equiv$	OCL-Object$_1{}^I$ $\neq$ OCL-Object$_2{}^I$, or OCL-Object$_i$ = NULL
(OCL-Value$_1$ CmpOp OCL-Value$_2$)^I	$\equiv$	OCL-Value$_1{}^I$ CmpOp OCL-Value$_2{}^I$, or OCL-Value$_i{}^I$ = NULL
(OCL-Object▪bAttr)^I	$\equiv$	OCL-Object▪bAttr^I= TRUE, or OCL-Object▪bAttr^I = NULL
$(v)^I$	$\equiv$	v

OCL-Bool$^{I,\texttt{Var},\textit{OCL-Set}}$	=	$\{v \in$ OCL-Set^I $\mid$ (OCL-Bool$_{[Var/v]}$)^I = TRUE$\}$

OCL-Set^I	$\subseteq$	Δ^I
(OCL-Set$_1$->union(OCL-Set$_2$))^I	=	OCL-Set$_1{}^I$ $\cup$ OCL-Set$_2{}^I$
(OCL-Set$_1$->intersection(OCL-Set$_2$))^I	=	OCL-Set$_1{}^I$ $\cap$ OCL-Set$_2{}^I$
(OCL-Set->select(Var $\mid$ OCL-Bool))^I	=	OCL-Bool$^{I,\texttt{Var},\textit{OCL-Set}}$
(OCL-Set->reject(Var $\mid$ OCL-Bool))^I	=	OCL-Set^I $\setminus$ OCL-Bool$^{I,\texttt{Var},\textit{OCL-Set}}$
(OCL-Set->selectByKind(Class))^I	=	OCL-Set^I $\cap$ Class^I
(OCL-Set->selectByType(Class))^I	=	OCL-Set^I $\cap$ Class^I $\setminus$ Subclasses(Class)^I
(OCL-Set▪role)^I	=	π_{role}(OCL-Set^I $\bowtie_{\text{ns}}$ ass(role)^I)
(OCL-Set▪assoClass)^I	=	$\pi_{assoClass}$(OCL-Set^I $\bowtie_{\text{ns}}$ (assoClass)^I)
(OCL-Object▪nfRole)^I	=	π_{nfRole}({OCL-Object^I} $\bowtie_{\text{ns}}$ ass(nfRole)^I)
(OCL-Object▪nfAssoClass)^I	=	$\pi_{nfAssoClass}$({OCL-Object^I} $\bowtie_{\text{ns}}$ (nfAssoClass)^I)
(OCL-Set▪attr)^I	=	π_{attr}(OCL-Set^I $\bowtie_{\text{oid}}$(attr)^I)
(OCL-Object▪nfAttr)^I	=	π_{nfAttr}({OCL-Object^I} $\bowtie_{\text{oid}}$(nfAttr)^I)
(Class.allInstances())^I	=	Class^I
(OCL-Single)^I	=	{OCL-Single^I} if OCL-Single^I $\neq$ NULL, $\emptyset$ otherwise

OCL-Object^I	$\in$	$\Delta^I \cup$ {NULL}
(OCL-Object▪oclAsType(Class))^I	=	(OCL-Object)^I
(OCL-Object▪fRole)^I	=	π_{fRole}({OCL-Object}^{I} $\bowtie_{\text{ns}}$ ass(fRole)^I), or NULL
(OCL-Object▪fAssoClass)^I	=	$\pi_{fAssoClass}$({OCL-Object^I} $\bowtie_{\text{ns}}$ ass(fAssoClass)^I), or NULL
$(v)^I$	=	v

OCL-Value^I	$\in$	$\Delta^I \cup$ {NULL}
$(v)^I$	=	v
(OCL-Object▪fAttr)^I	=	π_{fAttr}({OCL-Object^I} $\bowtie_{\text{oid}}$(fAttr)^I) , or NULL
(OCL-Set->min()^I	=	(OCL-Set)^I $\setminus$ (π ($\sigma_>$(OCL-Set^I $\times$ OCL-Set^I)), or NULL
(OCL-Set->max()^I	=	(OCL-Set)^I $\setminus$ (π ($\sigma_<$(OCL-Set^I $\times$ OCL-Set^I)), or NULL

Figure 3.5: Semantics of OCL$_{\text{FO}}$

Given an interpretation $\mathcal{I}$, an OCL$_{\mathsf{FO}}$ statement is interpreted according to the recursive definition specified in Figure 3.5. Such definition is mainly provided in terms of set theory, together with some relational algebra operators (such as join $\bowtie$, project π, or select σ), to easily define the interpretation of OCL$_{\mathsf{FO}}$ navigations. Moreover, to define these navigations, we assume that ns is the role name of the navigation source which corresponds to the *[role]* expression of a navigation, or the opposite of some role in navigations through binary associations. Lastly, we use the expression *Subclasses(Class)*^{I} to refer to those objects belonging to a subclass of *Class*.

Thus, given an OCL$_{\mathsf{FO}}$ constraint ϕ of the form:

context *R* **inv** *ConstraintName*: *OCL-Bool*

We say that an interpretation $\mathcal{I}$ satisfies the constraint ϕ, and write $\mathcal{I} \models \phi$ if and only if it evaluates to TRUE for all the objects of its context class R. More formally:

$$\mathcal{I} \models \phi \textbf{ iff } \forall_{v \in R^I} \textit{OCL-Bool}^I_{[\textit{self}/v]} = \text{TRUE}$$

For example, $\mathcal{I}_0$ satisfies the OCL$_{\mathsf{FO}}$ constraint *MessagesAreSentAfterCreation* since for its unique group *#group1* it holds that:

$$\{v \in \pi_{msg}(\{\#group1\} \bowtie \textit{isSentTo}^{I_0}) \mid \\ \pi_{sentTime}(\{v\} \bowtie \textit{sentTime}^{I_0}) < \\ \pi_{crTime}(\{\#group1\} \bowtie \textit{crTime}^{I_0})\} = \emptyset$$

In case $\mathcal{I}$ satisfies ϕ, we say that $\mathcal{I}$ is a *model* of ϕ, otherwise, we say that $\mathcal{I}$ violates ϕ. We naturally extend the notions of model, satisfaction, and violation to sets of OCL$_{\mathsf{FO}}$ constraints Φ. E.g. $\mathcal{I}$ satisfies a set of constraints Φ if and only if $\mathcal{I}$ satisfies each $\phi \in \Phi$.

OCL$_{\mathsf{FO}}$ semantics with nulls

Sometimes, the interpretation of some *OCL-Object* or *OCL-Value* results into no value. Indeed, consider that in our running example, some message *#msg1* has no value defined for the attribute *sentTime*. In this case, when navigating from the user *#msg1* to its *sentTime*, we obtain no value. More formally, we obtain $\emptyset$.

In such case, we define the *OCL-Object*/*OCL-Value* to be interpreted as a new value called NULL not present in Δ^I. In particular, we cast the $\emptyset$ to NULL (and viceversa) depending on the OCL expression it appears. Note that such interpretation corresponds to the one given in the OCL standard in [87].

Thus, when $\emptyset$ appears when interpreting some *OCL-Object*/*OCL-Value*, we automatically cast it to the new value NULL. Since the NULL value is not present in Δ^I, the NULL value does not join any value in the signature interpretation $\mathcal{I}$. That is, it does not join any value present in the interpretation of any association or attribute. This implies that, when navigating from a NULL value to obtain another object/value, we obtain again $\emptyset$, which is cast to NULL if this navigation is an *OCL-Object/OCL-Value*. This behavior perfectly emulates the standard OCL semantics proposal [87].

Finally, we need to extend the interpretation of the *OCL-Bool* to determine if they are evaluated to TRUE or FALSE when they use some *OCL-Object/OCL-Value* that evaluates to NULL. This differs from standard OCL since OCL considers that an *OCL-Bool* expression might return NULL or even INVALID. However, since we want OCL_{FO} to be a two-valued logic language, we restrict *OCL-Bool* values to either TRUE or FALSE.

Thus, and following the criteria already used in [103], we consider that an *OCL-Bool* is true if some of its *OCL-Object/OCL-Value* subexpression are evaluated to NULL. The idea behind this interpretation is that an OCL_{FO} constraint is not violated unless the values that determine its satisfaction/violation are defined. For instance, the constraint *MessagesAreSentAfterCreation* would be satisfied in the previous example if *#msg1* had not *sentTime* defined yet, since then, its sent time would not be previous to the creation group time.

Note that this interpretation is just a default behaviour to apply in case of finding a NULL value. However, note that it is possible to write a constraint that it is violated when some of its expressions evaluate to NULL by simply adding the boolean subexpression *and OCL-Single/OCL-Object* <> *null* to the OCL constraint.

3.3.3 OCL Operations not in OCL_{FO}

As we will prove in the next two sections, an OCL_{FO} constraint is equivalent to relational algebra query. For this reason, we know that OCL_{FO} is also equivalent to (domain independent) first-order logic. Therefore, it can be stated straightforwardly that all OCL operations which are not first-order cannot be included in OCL_{FO}. Examples of such kind are *closure*, or aggregation functions such as *count*. It is worth noting, however, that the aggregation function *size* can be used in OCL_{FO} to compare the size of some set with some fixed integer, but not with another set size neither with the value obtained from some attribute.

Additionally, basic type operations such as $+$, -, *, and $/$ cannot be included in OCL_{FO} expressions since they are not supported by relational algebra. Finally, operations which are only used to express boolean conditions for other artifacts such as behavioural models but which cannot appear in a constraint definitions are not included either in OCL_{FO}. Examples of such operations may be *oclIsNew* and *oclIsInState*.

As a final remark to this section, it is worth mentioning that we have intentionally left out from OCL_{FO} some OCL operations that can also be translated into relational algebra such as *let ... in*, *if ... then ... else ... endif*, *including* or *excluding*. This has been done for the sake of having a compact fragment of the language and because these operations are not frequently used when defining OCL constraints. However, not including these operations in OCL_{FO} does not suppose a limitation of our choice because their consideration would not increase the expressive power of OCL_{FO} and because they can be translated into equivalent OCL_{FO} expressions according to the OCL equivalences among operations.

3.4 Checking OCL_{FO} Constraints by RA Queries

The main goal of this section is to show that any OCL_{FO} constraint can be checked by means of a RA query since given an OCL_{FO} constraint we can build a RA query that retrieves the objects that violate it. Therefore, the OCL_{FO} constraint is satisfied if and only if its corresponding query is empty.

This result entails that checking an OCL_{FO} constraint is at most as difficult as executing a RA query and, thus, checking an OCL_{FO} constraint can be solved in polynomial time with regarding to data complexity (and in particular, it belongs to the AC^0 complexity class). Moreover, this also enables reusing current techniques developed in the community of relational databases for the treatment of OCL_{FO} constraints. For instance, incremental OCL_{FO} constraints checking can be solved by means of incremental query answering (e.g. following [90], or [11]), and repairing OCL_{FO} constraints can be solved by view updating techniques (as studied for instance in [95]).

The RA query is built from an OCL_{FO} constraint in two steps. First, we normalize the OCL_{FO} constraint into an equivalent one that uses a lower number of OCL_{FO} operators. Then, we translate the normalized OCL_{FO} constraint into a RA query that returns the objects that violate the constraint.

3.4.1 Normalizing OCL_{FO} Constraints

The grammar presented in Figure 3.4 admits very complex OCL expressions, using lots of different OCL operations. Therefore, defining directly a translation from pure OCL_{FO} constraints into RA queries becomes very cumbersome. To make things simpler, we initially translate each OCL_{FO} constraint into a *normalized* one, whose expression is defined only by means of a small number of operations.

We say that an OCL_{FO} constraint is normalized if it is defined only by means of the following operations: *and*, *not*, *forAll*, $=$, and $<$ for *OCL-Bool* expressions; and *union*, *intersection*, -, *select* for *OCL-Set* expressions; although their expression can also contain the usual navigations through roles and attributes, and the *oclAsType* cast operation.

To normalize an OCL_{FO} constraint, we recursively apply the rewritings defined in Figure 3.6. It is not difficult to verify that such rewriting preserve the original

OCL-Bool		
OCL-Bool$_1$ or OCL-Bool$_2$	=	*not(not OCL-Bool$_1$ and notOCL-Bool$_2$)*
OCL-Bool$_1$ implies OCL-Bool$_2$	=	*not OCL-Bool$_1$ or OCL-Bool$_2$*
OCL-Set$_1$->includesAll(OCL-Set$_2$)	≡	*OCL-Set$_2$->forAll(e$_2$| not OCL-Set$_1$->forAll(e$_1$| e$_1$<>e$_2$))*
OCL-Set$_1$->excludesAll(OCL-Set$_2$)	≡	*OCL-Set$_2$->forAll(e$_2$| OCL-Set$_1$->forAll(e$_1$| e$_1$<>e$_2$))*
OCL-Set->includes(OCL-Single)	≡	*not OCL-Set->forAll(e | e <> OCL-Single))*
OCL-Set->excludes(OCL-Single)	≡	*OCL-Set->forAll(e | e <> OCL-Single))*
OCL-Set->exists(Var | OCL-Bool)	≡	*not OCL-Set->forAll(Var | OCL-Bool)*
OCL-Set->isEmpty()	≡	*OCL-Set->forAll(e | 1 $\neq$ 1)*
OCL-Set->notEmpty()	≡	*not OCL-Set->forAll(e | 1 $\neq$ 1)*
OCL-Set->size() < n	≡	*OCL-Set->forAll(e$_1$, ..., e$_n$ | e$_1$=e$_2$ or e$_1$=e$_3$ or ... e$_{n-1}$=e$_n$)*
OCL-Set->size() <= n	≡	*OCL-Set->forAll(e$_1$, ..., e$_{n+1}$ | e$_1$=e$_2$ or e$_1$=e$_3$ or ... e$_n$=e$_{n+1}$)*
OCL-Set->size() = n	≡	*OCL-Set->size() ¡= n and not OCL-Set->size() ¡ n-1*
OCL-Set->one(Var | OCL-Bool)	≡	*OCL-Set->select(Var | OCL-Bool)->size() = 1*
OCL-Set->isUnique(attr)	≡	*OCL-Set->forAll(v1,v2 | v1 = v2 or v1.attr <> v2.attr)*
v.oclIsKindOf(Class)	≡	*Class->forAll(e | e = v)*
v.oclIsTypeOf(Class)	≡	*Class->forAll(e | e = v) and Subclass->forAll(e | e <> v) ...*
OCL-Single = null	≡	*OCL-Single->forAll(e | 1 <> 1)*
OCL-Single <> null	≡	*not OCL-Single->forAll(e | 1 <> 1)*
OCL-Set$_1$ = OCL-Set$_2$	≡	*OCL-Set$_1$->includesAll(OCL-Set$_2$) and*
		OCL-Set$_2$->includesAll(OCL-Set$_1$)
OCL-Set$_1$ <> OCL-Set$_2$	≡	*not OCL-Set$_1$ = OCL-Set$_2$*
OCL-Set		
OCL-Set->reject(Var | OCL-Bool)	=	*OCL-Set->select(Var | not OCL-Bool)*
OCL-Set->selectByKind(Class)	=	*OCL-Set->select(e | e.oclIsKindOf(Class))*
OCL-Set->selectByType(Class)	=	*OCL-Set->select(e | e.oclIsTypeOf(Class))*
OCL-Value		
OCL-Set->min()	=	*OCL-Set->select(min | OCL-Set->forAll(e | min <= e))*
OCL-Set->max()	=	*OCL-Set->select(max | OCL-Set->forAll(e | max >= e))*

Figure 3.6: OCL$_{\text{FO}}$ normalization rewrittings

semantics of the constraint by means of directly inspecting the operation semantics defined in Figure 3.5.

In our running example, the normalized version of the constraint *MessagesAreFromGroup* is:

self.msg.author->forAll(a|
  not self.user->forAll(u|not a = u))

3.4.2 Drawing RA Queries from Normalized Constraints

To obtain the RA query of a normalized OCL$_{\text{FO}}$ constraint, we first translate each OCL$_{\text{FO}}$ navigation (i.e., *OCL-Set*, *OCL-Object* and *OCL-Value* expressions) into RA queries retrieving its corresponding set, object, or value. Then, the resulting RA query is obtained by translating the *OCL-Bool* expressions through the composition of the translation of its navigations.

More precisely, the crucial point for translating the navigations are the OCL$_{\text{FO}}$ variables (i.e, the *self* variable and the iteration variables appearing in *forAll* and *select* expressions) since our goal is to build a RA query with one relational attribute for each OCL_{FO} variable alive in the OCL_{FO} navigation, together with one additional attribute containing the result of the navigation. For instance, when translating the normalized *MessagesAreFromGroup* constraint, the navigation *self.msg.author* is translated as a relational query with the attributes *self* and *result*.

Intuitively, when executing such queries over some interpretation $\mathcal{I}$, each retrieved tuple t represents a combination of values that the OCL_{FO} variables may take when evaluating the OCL_{FO} navigation with $\mathcal{I}$. For instance, if in the interpretation $\mathcal{I}$ we have that a group *#group1* has some message *#msg1* written by *#John*, then, the row <*#group1*, *#John*> appears in the query result that translates the navigation *self.msg.author*.

Then, the idea of composing those translations to obtain the translation of the whole *OCL-Bool* expression defining the constraint is to select those tuples that witness the violation of the boolean condition. That is, we want to select the row <*#group1*, *#John*> from the previous example in case that *#John* is not a member of *#group1*.

All translations are recursive and use the input variable q_c, the *context query*, which is the relational query that retrieves the values for the alive OCL_{FO} variables defined in the upper expression of the expression being translated. For instance, to translate *self.msg.author*, we need a context query q_c retrieving the values that the variable *self* might take, e.g. $q_c = \textit{Group}$.

In the rest of this section, we first present the algorithms for translating each OCL_{FO} navigation, while discussing their intuition and providing a formal proof of their correctness. Afterwards, we show and prove how to use these algorithms to translate the whole *OCL-Bool* expression defining the OCL_{FO} constraint.

For the seek of simplicity, we omit some relational algebra low-level details such as relational algebra attribute renamings and some selection conditions since they can be easily understood from the context.

OCL-Set Translation

Algorithm 1 translates an *OCL-Set* into a relational query retrieving the same values/objects than the ones in the *OCL-Set*. Since the *OCL-Set* expression might have OCL free variables (such as *self*), we need the context query q_c to bring the different possible substitutions to apply to such variables. Thus, each tuple t in the result of the query has the form $< t_1, ..., t_n, v >$, where $t_1, ..., t_n$ represents a substitution for the OCL_{FO} variables appearing in the *OCL-Set* (which are taken from q_c), and v a value appearing in the *OCL-Set* according to such substitution for the free variables.

The idea behind the algorithm is to use the relational operation corresponding to each OCL_{FO} set operation. I.e., *union* is translated into $\cup$, role navigations as $\bowtie$, etc. The major difficult, however, relies on the translation of the *select* operation, which is translated using the translation of *OCL-Bool* expressions. In this case, the idea is to first, translate the *OCL-Set* source expression, and then, remove all those rows not satisfying the inner *OCL-Bool* expression.

As an example, consider the *OCL-Set* expression *self.msg.author* with a context query q_c=*Group* defining the values that the OCL_{FO} variable *self* might take. Such expressions is translated as:

$$\pi_{group,author}(Group \bowtie IsSentTo \bowtie Writes)$$

Intuitively, the translation just translates the role navigations to RA joins, and then, projects the result to only retrieve the reachable *messages* for each *group*.

In the following we formally proof the correctness of this algorithm.

Algorithm 1 raTranslation(*OCL-Set*, q_c)

```
if OCL-Set = OCL-Set_1->union(OCL-Set_2) then
    q_1 := raTranslation(OCL-Set_1, q_c)
    q_2 := raTranslation(OCL-Set_2, q_c)
    return $q_1 \cup q_2$
else if OCL-Set = OCL-Set_1->intersection(OCL-Set_2) then
    q_1 := raTranslation(OCL-Set_1, q_c)
    q_2 := raTranslation(OCL-Set_2, q_c)
    return $q_1 \Join q_2$
else if OCL-Set = OCL-Set_1 − OCL-Set_2 then
    q_1 := raTranslation(OCL-Set_1, q_c)
    q_2 := raTranslation(OCL-Set_2, q_c)
    return $q_1 \setminus q_2$
else if OCL-Set = OCL-Set_1->select(Var|OCL-Bool) then
    q_s := raTranslation(OCL-Set_1, q_c)
    q_r := raTranslation(OCL-Bool, q_s)
    return $q_s \setminus q_r$
else if OCL-Set = OCL-Set_1.role then
    q_1 := raTranslation(OCL-Set_1, q_c)
    return $\pi(q_1 \bowtie ass(role))$
else if OCL-Set = OCL-Set_1.assoClass then
    q_1 := raTranslation(OCL-Set_1, q_c)
    return $\pi(q_1 \bowtie assoClass)$
else if OCL-Set = OCL-Object_1.nfRole then
    q_1 := raTranslation(OCL-Object_1, q_c)
    return $\pi(q_1 \bowtie ass(nfRole))$
else if OCL-Set = OCL-Object_1.nfAssoClass then
    q_1 := raTranslation(OCL-Object_1, q_c)
    return $\pi(q_1 \bowtie nfAssoClass)$
else if OCL-Set = OCL-Set_1.attr then
    q_1 := raTranslation(OCL-Set_1, q_c)
    return $\pi(q_1 \bowtie attr)$
else if OCL-Set = OCL-Object_1.nfAttr then
    q_1 := raTranslation(OCL-Object_1, q_c)
    return $\pi(q_1 \bowtie nfAttr)$
else if OCL-Set = R.allInstances() then
    return $q_c \times R$
else if OCL-Set = OCL-Object_1 then
    return raTranslation(OCL-Object_1, q_c)
end if
```

Property 2. OCL-Set raTranslation correctness

Let ϕ be an *OCL-Set* over a UML conceptual schema S_{UML}, $\overline{q}$ a relational algebra expression defined over the relational view of S_{UML}, and q_c a context query such that $\overline{q} = \mathsf{raTranslation}(\phi, q_c)$ (Algorithm 1). Then,

$$v \in \phi^{I}_{[S_t]} \ \textbf{iff} \ \ < t[0], ..., t[n], v > \in \overline{q}(I)$$

for any value v, any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$.

Proof. The proof is based on structural induction over the $\mathsf{OCL}_{\mathsf{FO}}$ grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. Consider the base case:

$$\phi = \textit{R.allInstances()}$$

According to Algorithm 1,

$$\overline{q} = \mathsf{raTranslation}(\phi, q_c) = q_c \times R$$

Clearly, according to the semantics of $\mathsf{OCL}_{\mathsf{FO}}$, $v \in$ *R.allInstances()*^{I} iff $v \in R^I$. Moreover, it is guaranteed that S_t is obtained from $q_c(I)$. Hence, $v \in$ *R.allInstances()*$^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \in q_c(I)\times\ R^I$.

Consider the inductive case:

$$\phi = \textit{OCL-Set.select(s} \mid \textit{OCL-Bool)}$$

According to Algorithm 1,

$$\overline{q} = \mathsf{raTranslation}(\phi, q_c) = q_s \setminus q_r$$

where

$$q_s = \mathsf{raTranslation}(\textit{OCL-Set}, q_c)$$
$$q_r = \mathsf{raTranslation}(\textit{OCL-Bool}, q_s)$$

According to the semantics, we have that $v \in$ *OCL-Set.select(s* $\mid$ *OCL-Bool)*$^I_{[S_t]}$ if and only if $v \in$ *OCL-Bool*$^{I,s,\textit{OCL-Set}}_{[S_t]}$. This is the case if and only if $v \in$ *OCL-Set*$^I_{[S_t]}$ and *OCL-Bool*$^I_{[S_t \cup \{s/v\}]} = \text{TRUE}$. Equivalently, this is the case iff $v \in$ *OCL-Set*$^I_{[S_t]}$ and not *OCL-Bool*$^I_{[S_t \cup \{s/v\}]} = \text{FALSE}$. Now, by induction hypothesis, we have that $v \in$ *OCL-Set*$^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \in\ q_s(I)$, and *OCL-Bool*$^I_{[S_t \cup \{s/v\}]} = \text{FALSE}$ iff $< t[0], ..., t[n], v > \in q_r(I)$. Hence, $v \in$ *OCL-Set.select(s* $\mid$ *OCL-Bool)*$^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \in \overline{q}(I)$. □ □

OCL-Object and OCL-Value Translation

Algorithm 2 defines the translation of an *OCL-Object* or *OCL-Value* expression into a relational query that, intuitively, returns the object/value referred by the expression. Similarly as before, each tuple $< t_1, ..., t_n, v >$ returned by the query represents an evaluation that the OCL$_{\text{FO}}$ variables appearing in the expression might take (values $t_1, ..., t_n$ taken from a context query q_c), together with the value retrieved by the *OCL-Object*/*OCL-Value* expression according to that evaluation (v).

Again, the idea behind the translation is to use the relational algebra operator that corresponds to those defined in the OCL$_{\text{FO}}$ semantics. E.g, role navigations are translated by means of the same join we have defined in the OCL$_{\text{FO}}$ semantics.

For instance, consider the *OCL-Single* expression *m.sentTime* with a context query $q_c = \pi_{msg}(Group \bowtie IsSentTo)$ defining the values for the OCL$_{\text{FO}}$ variable *m*. Such expression is translated as:

$$(\pi_{msg}(Group \bowtie IsSentTo)) \bowtie SentTime$$

Intuitively, the translation converts the attribute navigation as a new join to retrieve the *sentTime* attribute for each value that *e* might take.

In the following, we formally proof the correctness of Algorithm 2.

Algorithm 2 raTranslation(*OCL-Single*, q_c)

```
if OCL-Single = Constant then
    return $q_c \times \{\text{Constant}\}$
else if OCL-Single = Variable then
    return $q_c$
else if OCL-Single = OCL-Object$_1$.fAttr then
    $q_1$ := raTranslation(OCL-Object$_1$, $q_c$)
    return $\pi\ (q_1 \bowtie fAttr)$
else if OCL-Single = OCL-Object$_1$.oclAsType(Class) then
    return raTranslation(OCL-Object$_1$, $q_c$)
else if OCL-Single = OCL-Object$_1$.fRole then
    $q_1$ := raTranslation(OCL-Object$_1$, $q_c$)
    return $\pi(q_1 \bowtie ass(fRole))$
else if OCL-Set = OCL-Object$_1$.fAssoClass then
    $q_1$ := raTranslation(OCL-Object$_1$, $q_c$)
    return $\pi(q_1 \bowtie fAssoClass)$
end if
```

Property 3. OCL-Single raTranslation correctness

Let ϕ be an *OCL-Single* over a UML conceptual schema S_{UML}, $\overline{q}$ a relational algebra expression defined over the relational view of S_{UML}, and q_c a context query such that $\overline{q} = \text{raTranslation}(\phi, q_c)$ (Algorithm 2). Then, for any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$, we have:

$$\text{NULL} = \phi^I_{[S_t]} \ \textbf{iff} \ \neg\exists v. < t[0], ..., t[n], v > \in \overline{q}(I)$$

and, for any value v different from NULL:

$$v = \phi^I_{[S_t]} \ \textbf{iff} \ < t[0], ..., t[n], v > \in \overline{q}(I)$$

Proof. The proof is based on structural induction over the OCL_{FO} grammar. In the following we bring the proof for one base case and one inductive case. The rest of cases follows analogously. Consider the base case:

$$\phi = \textit{self}$$

According to Algorithm 2,

$$\overline{q} = \text{raTranslation}(\phi, q_c) = q_c \tag{3.1}$$

According to the semantics, we have that $v = \textit{self}^I_{[S_t]}$ iff $\textit{self}_{[S_t]} = v$. This is the case if and only if we have $< t[0], ..., v, ..., t[n] > \in \ q_c$.

Consider the inductive case:

$$\phi = \textit{OCL-Object.fAttr}$$

According to Algorithm 2,

$$\overline{q} = \pi(q_1 \bowtie \textit{fAttr})$$

where

$$q_1 = \text{raTranslation}(\textit{OCL-Object}, q_c)$$

According to the semantics, we have that $v = (\textit{OCL-Object.fAttr})^I_{[S_t]}$ iff $v = \pi$ $(\textit{OCL-Object}^I_{[S_t]} \bowtie \textit{fAttr}^I)$. This is the case if and only if there exists some value v' in $\textit{OCL-Object}^I_{[S_t]}$ whose join with $\textit{fAttr}^I$ retrieves v. By induction hypothesis we know that $< t[0], ..., t[n], v' > \in \ q_1(I)$ iff $v' = \textit{OCL-Object}^I_{[S_t]}$. Thus, $v = (\textit{OCL-Object.fAttr})^I_{[S_t]}$ iff $< t[0], ..., t[n], v > \in \ \overline{q}(I)$. □

□

OCL-Bool Translation

In Algorithm 3 we show how to make use of the previous translations to obtain the values that cause the violation of the *OCL-Bool* condition. The output of this algorithm is a query that returns the evaluation of the OCL$_{\text{FO}}$ variables alive in *OCL-Bool* that make the expression evaluate to $\textsc{false}$.

The intuition behind the translation is to use the relational algebra selection σ to select those values that contradict the *OCL-Bool* expression.

For instance, consider the *OCL-Bool* expression *self.user->forAll(u|not a = u)* with the context query $q_c = \pi_{\mathit{group,author}}(\mathit{Group} \bowtie \mathit{IsSentTo} \bowtie \mathit{Writes})$ defining the values for the OCL$_{\text{FO}}$ variable *a* depending on the value given to *self*. Such expression would be translated as:

$$\sigma_{\mathit{author=user}}(q_c \bowtie (q_c \bowtie \mathit{HasMember}))$$

Intuitively, q_c retrieves the values that *a* might take for every value of *self* (that is, all the authors of messages sent to some group *self*), then, $(q_c \bowtie \mathit{IsMemberOf})$ retrieves the values that *u* might take for every value of *self* (that is, all the users of some group *self*). Then, the join of both expressions retrieves the values that *a* and *u* might take for the same value of *self* (that is, all the authors and members of some group *self*). Finally, the selection picks those tuples in which the value for *a* is equal to the value for *u*.

Note that, to translate the other normalized OCL$_{\text{FO}}$ boolean operations such as *and* and *not* we just need to compose the previous translation pattern. That is, *and* is translated by unifying the set of rows that causes the violation of the first condition, with those causing the violation of the second one; and *not* is translated by computing those rows violating the inner expression (in other words, those rows satisfying its negation), and removing them from the context query (so we have those rows violating the negated statement).

In the following, we formally proof the correctness of Algorithm 3.

Property 4. OCL-Bool raTranslation correctness

Let ϕ be an *OCL-Bool* over a UML conceptual schema S_{UML}, $\overline{q}$ a relational algebra expression defined over the relational view of S_{UML}, and q_c a context query such that $\overline{q} = \text{raTranslation}(\phi, q_c)$ (Algorithm 3). Then, for any interpretation $\mathcal{I}$, and any substitution S_t obtained from $q_c(I)$, we have:

$$\phi^{I}_{[S_t]} = \textsc{false} \textbf{ iff } t \in \overline{q}(I)$$

Algorithm 3 raTranslation(*OCL-Bool*, q_c)

```
if OCL-Bool = OCL-Bool_1 and OCL-Bool_2 then
    $q_1$ = raTranslation(OCL-Bool_1, $q_c$)
    $q_2$ = raTranslation(OCL-Bool_2, $q_c$)
    return $q_1 \cup q_2$
else if OCL-Bool = not OCL-Bool_1 then
    return $q_c \setminus$ raTranslation(OCL-Bool_1, $q_c$)
else if OCL-Bool = OCL-Set->forAll(Var|OCL-Bool_1) then
    $q_s$ := raTranslation(OCL-Set, $q_c$)
    $q_b$ := raTranslation(OCL-Bool_1, $q_s$)
    return $\pi\ q_b$
else if OCL-Bool = OCL-Single_1 CompOp OCL-Single_2 then
    $q_1$ = raTranslation(OCL-Single_1, $q_c$)
    $q_2$ = raTranslation(OCL-Single_2, $q_c$)
    return $\pi\sigma\ q_1 \bowtie q_2$
else if OCL-Bool = OCL-Value_1 then
    $q_1$ := raTranslation(OCL-Value_1, $q_c$)
    return $\pi\sigma\ q_1$
end if
```

Proof. The proof is based on structural induction over the OCL_{FO} grammar. In the following we bring the proof for the base case and one inductive case. The rest of cases follows analogously. For the base case, we consider the expression:

$$\phi = \textit{OCL-Single}_1 \ \textit{CompOp} \ \textit{OCL-Single}_2$$

According to Algorithm 3,

$$\overline{q} = \pi\sigma(q_1 \bowtie q_2)$$

where

$$q_1 = \text{raTranslation}(\textit{OCL-Single}_1, q_c)$$
$$q_2 = \text{raTranslation}(\textit{OCL-Single}_2, q_c)$$

According to the semantics, $\phi^I_{[S_t]} = \text{FALSE}$ iff the values $v_1 = \textit{OCL-Single}_1{}^I_{[S_t]}$, and $v_2 = \textit{OCL-Single}^I_{[S_t]}$ do not satisfy the comparison operator *CompOp*. By induction hypothesis we have that $< t[0], ..., t[n], v_1 > \in q_1(I)$, and $< t[0], ..., t[n], v_2 > \in q_2(I)$. Thus, we can obtain the values v_1 and v_2 by joining q_1 and q_2, and select the row $< t[0], ..., t[n] >$ iff v_1 and v_2 do not satisfy the corresponding *CompOp*. Thus, $\phi^I_{[S_t]} = \text{FALSE}$ iff $t \ \in \overline{q}(I)$.

For the inductive case, we consider:

$$\phi = \textit{OCL-Set->forAll(Var|OCL-Bool)}$$

According to Algorithm 3,

$$\overline{q} = \pi q_b$$

where

$$q_b = \text{raTranslation}(\textit{OCL-Bool}, q_s)$$
$$q_s = \text{raTranslation}(\textit{OCL-Set}, q_c)$$

According to the semantics, we have that $\textit{OCL-Set->forAll(Var|OCL-Bool)}^{I}_{[S_t]} =$ FALSE iff $(\textit{not OCL-Bool})^{I,\textit{Var},\textit{OCL-Set}}_{[S_t]} \neq \emptyset$. This is the case iff $\exists v.v \in \textit{OCL-Set}^{I}_{[S_t]}$ and $\textit{OCL-Bool}^{I}_{[S_t \cup \{\textit{Var}/v\}]} =$ FALSE. By induction hypothesis, we have that, for any value v, $v \in \textit{OCL-Set}^{I}_{[S_t]}$ if and only if $< t[0], ..., t[n], v > \in\ q_s(I)$. Moreover, by induction again we know that $v \in \textit{OCL-Set}^{I}_{[S_t]}$ and $\textit{OCL-Bool}^{I}_{[S_t \cup \{\textit{Var}/v\}]} =$ FALSE if and only if $< t[0], ..., t[n], v > \in\ q_b(I)$. Thus, $\textit{OCL-Set->forAll(Var|OCL-Bool)}^{I}_{[S_t]} =$ FALSE if and only if $t\ \in\ \overline{q}(I)$. □ □

Translating an OCL$_{\text{FO}}$ Constraint

To translate an OCL$_{\text{FO}}$ constraint, it is enough to invoke Algorithm 3 with the body of the constraint as the *OCL-Bool* parameter and the context class in which the constraint is defined as the context query q_c, so that, the variable *self* is going to be evaluated to all the objects of the given context class.

For instance, consider the constraint of our running example *MessagesAreFromGroup*. This constraint would be translated into:

$$\begin{aligned}&\pi(\textit{Group} \bowtie \textit{IsSentTo} \bowtie \textit{Writes})\setminus\\ &\quad \pi\sigma((\textit{Group} \bowtie \textit{IsSentTo} \bowtie \textit{Writes}) \bowtie\\ &\quad (\textit{Group} \bowtie \textit{IsSentTo} \bowtie \textit{Writes} \bowtie \textit{HasMember}))\end{aligned}$$

Intuitively, the query picks up all *users* who have written in some group (first line of the translation), and it takes out all those *users* who are indeed members of such group (second and third line of the translation). Thus, note that the constraint is satisfied if and only if the previous query is empty.

In the following we formally proof the correctness of the translation.

Property 5. Any OCL constraint can be translated into an equivalent RA query

Let ϕ be an OCL$_{\text{FO}}$ constraint over a UML conceptual schema S_{UML}, defined on the context class R, and $\overline{q}$ a relational algebra expression, defined over the relational view of S_{UML}, such that $\overline{q} = \text{raTranslation}(\phi, R)$ (Algorithm 3). Then, for any interpretation $\mathcal{I}$, we have:

$$I \models \phi \textbf{ iff } \overline{q}(I) = \emptyset$$

Proof. Directly from Property 4 we have that $\overline{q}(I)$ retrieves those values for the OCL$_{\text{FO}}$ variable *self* from R^I s.t. that makes the *OCL-Bool* in ϕ evaluate to FALSE. Thus, $I \models \phi$ iff $\overline{q}(I) = \emptyset$. □ □

.

3.5 RA queries to OCL$_{\text{FO}}$ Constraints

Now, we show that any constraint that can be checked by means of a relational algebra query can be encoded as an OCL$_{\text{FO}}$ constraint. That is, for any relational query q, we can build an OCL$_{\text{FO}}$ constraint ϕ such that, for any interpretation $\mathcal{I}$, we have that $q(\mathcal{I}) = \emptyset$ iff $I \models \phi$.

This result implies that the language of OCL$_{\text{FO}}$ is as expressive for defining constraints as relational algebra. Taking in account that in Property 5 we showed that any OCL$_{\text{FO}}$ constraint can be checked by means of a relational query, we may conclude that OCL$_{\text{FO}}$ is exactly as expressive for defining constraints as relational algebra. Thus, checking OCL$_{\text{FO}}$ constraints is as difficult as executing a relational query, that is polynomial with regarding to data complexity (and AC^0 in particular).

We define the *oclTranslation* from a RA query to an OCL$_{\text{FO}}$ constraint in Algorithm 4. This algorithm receives three input parameters: a context query q_c, an OCL$_{\text{FO}}$ boolean statement *OCL-Bool*, and a mapping $\mathcal{M}$ that makes explicit which attributes from q_c are mapped to which OCL$_{\text{FO}}$ variables from *OCL-Bool*. Then, the idea is that the algorithm returns a new OCL$_{\text{FO}}$ boolean statement ϕ such that, for any given interpretation $\mathcal{I}$, $\mathcal{I} \models \phi$ iff *OCL-Bool*$^I_{[S_t]} = \text{TRUE}$ for any given substitution S_t obtained from $q_c(\mathcal{I})$.

Then, we can then obtain the OCL$_{\text{FO}}$ constraint corresponding to a relational query q by invoking *oclTranslation(*q*, false,*$\emptyset$*)*. Indeed, the unique way that for all substitutions S_t obtained from q we can have *false*$^I_{[St]} = \text{TRUE}$ is that $q(\mathcal{I}) = \emptyset$.

Intuitively, the algorithm works by recursively removing relational operators from the input query q and placing them in the OCL-boolean expression given as a parameter. For instance, if we invoke:

oclTranslation(R $\setminus$ S, false, $\emptyset$)

We first recursively translate:

oclTranslation(S, r <> s, {s $\rightarrow$ S})

to obtain an OCL$_{\text{FO}}$ boolean expression that characterizes those values of some variable *r* that are different to any element in *S*. In particular, we obtain the new OCL-boolean ϕ:

S.allInstances()->forAll(s|r <> s)

Algorithm 4 oclTranslation(q_c, *OCL-Bool*, $\mathcal{M}$)

```
if $q_c = q_1 \cup q_2$ then
    $OCL\text{-}Bool_1 = oclTranslation(q_1, OCL\text{-}Bool, \mathcal{M})$
    $OCL\text{-}Bool_2 = oclTranslation(q_2, OCL\text{-}Bool, \mathcal{M})$
    return $OCL\text{-}Bool_1$ + ‘ and ’ + $OCL\text{-}Bool_2$
else if $q_c = \pi_a q_1$ then
    return oclTranslation($q_1$, OCL-Bool, $\mathcal{M}$)
else if $q_c = \sigma_{a \omega b}\ q_1$ then
    $\mathcal{M}'$ := getCompleteMap($\mathcal{M}$, $q_1$)
    OCL-Bool' := $\mathcal{M}'$.getVar($q.a$) $\omega$ $\mathcal{M}'$.getVar($q.b$) + ‘ implies ’ + OCL-Bool
    return oclTranslation($q_1$, OCL-Bool', $\mathcal{M}'$)
else if $q_c = q_1 \times q_2$ then
    $OCL\text{-}Bool_2$ := oclTranslation($q_2$, OCL-Bool, $\mathcal{M}$)
    return oclTranslation($q_1$, $OCL\text{-}Bool_2$, $\mathcal{M}$)
else if $q_c = q_1 \setminus q_2$ then
    $\mathcal{M}_1$ := getCompleteMap($\mathcal{M}$, $q_1$)
    $\mathcal{M}_2$ := getCompleteMap($\mathcal{M}$, $q_2$)
    ocl-ineq := getInequalities($\mathcal{M}_1$, $\mathcal{M}_2$, $q_1$, $q_2$)
    OCL-Bool' := oclTranslation($q_2$, ocl-ineq, $\mathcal{M}_2$) + ‘ implies ’ + OCL-Bool
    return oclTranslation($q_1$, OCL-Bool', $\mathcal{M}_1$)
else if $q_c$ = R then
    $\mathcal{M}_2$ := getCompleteMap($\mathcal{M}$, $R$)
    return ‘R.allInstances()->forAll(’+$\mathcal{M}$.getVar($R.id$)+‘|’ + OCL-Bool+‘)’
end if
```

Then, we recursively translate:

$$oclTranslation(R,\ \phi \text{ implies false},\ \{r \rightarrow R\})$$

obtaining:

R.allInstances()->forAll(r |
S.allInstances()->forAll(s | r <> s) implies false)

This boolean expression iterates through all elements *r* of *R*, it checks whether *r* is different from every *s* in *S*, and, if so, it returns false. Clearly, such OCL-boolean statement only evaluates to true iff the relational query $R \setminus S$ evaluates to the empty set.

Algorithm 4 makes use of some auxiliary functions. Function *getCompleteMap(*$\mathcal{M}$*, q)* returns a copy of the map $\mathcal{M}$ but adding some new correspondences between relational attributes in *q* and new fresh OCL_{FO} variables. For instance, this function allowed us to create the new OCL free variables *r* and *s*, and map them to the relational tables *R* and *S* respectively. Function *getInegualities*($\mathcal{M}_1$, $\mathcal{M}_2$, q_1, q_2) returns a conjunction of OCL_{FO} variable inequalities. In particular, one inequality for each pair of OCL_{FO} variables that are mapped to the same i-th relational attribute in q_1, and q_2, respectively. This function allowed us to build the inequality *r <> s* in our previous example.

Property 6. RA oclTranslation correctness

Let *OCL-Bool* be an OCL_{FO} boolean statement defined over a UML conceptual schema S_{UML}, q_c a relational algebra expression defined over the relational view of S_{UML}, and $\mathcal{M}$ a mapping from q_c attributes to OCL_{FO} variables in ϕ_1. Let ϕ be the OCL_{FO} statement such that $\phi = \text{oclTranslation}(q_c, \textit{OCL-Bool}, \mathcal{M})$ (Algorithm 4). Then, for any interpretation $\mathcal{I}$, and any substitution S, we have:

$$\mathcal{I} \models \phi_{[S]} \ \textbf{ iff } \ \forall_{S_t} \textit{OCL-Bool}^{I}_{[S_t][S]} = \text{TRUE}$$

where the substitutions S_t are obtained from the context query q_c and the mapping $\mathcal{M}$.

Proof. The proof is inductive on the number of relational algebra operators present in the input context query q, thus, following the recursive nature of the Algorithm.

For the base case, consider a context query with the following form, where *R* is the name of some relation:

$$q_c = R$$

Applying Algorithm 4 we get:

$$\phi = \textit{R.allInstances()->forAll(r} \mid \textit{OCL-Bool)}$$

Then, according to the semantics, for any substitution S, we have that $\mathcal{I} \models$ *R.allInstances()->forAll(r* $\mid$ *OCL-Bool)*$_{[S]}$ if and only if it does not exists a value $v \in R^I$ s.t. $\textit{OCL-Bool}^{I}_{[r/v][S]} = \text{FALSE}$. This is the case iff for all the values $v \in R^I$ we have $\textit{OCL-Bool}_{r}{}^{I}_{[r/v][S]} = \text{TRUE}$. Equivalently, this is the case iff for all the possible substitutions S_t we can obtain from *R*, we have that $\textit{OCL-Bool}^{I}_{[S_t][S]} = \text{TRUE}$.

We deal now with the inductive cases. Consider first:

$$q_c = q_1 \cup q_2$$

Applying Algorithm 4 we get:

$$\phi = \textit{OCL-Bool}_1 \textit{ and } \textit{OCL-Bool}_2$$

where

$$\textit{OCL-Bool}_i = \text{oclTranslation}(q_i, \textit{OCL-Bool}, \mathcal{M})$$

According to the semantics, $\mathcal{I} \models \phi_{[S]}$ if and only if *OCL-Bool*$_1{}^I_{[S]} = \text{TRUE}$ and *OCL-Bool*$_2{}^I_{[S]} = \text{TRUE}$. By induction we know that *OCL-Bool*$_i{}^I_{[S]} = \text{TRUE}$ iff for every substitution S_t that can be obtained from q_i we have *OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ if and only if *OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$ for every substitution that can be obtained from $q_1 \cup q_2$.

Consider now the case:

$$q_c = \pi q_1$$

Applying Algorithm 4 we get:

$$\phi = \textit{OCL-Bool}_1 = \text{oclTranslation}(q_1, \textit{OCL-Bool}, \mathcal{M})$$

And we have that $\mathcal{I} \models \phi_{[S]}$ iff *OCL-Bool*$_1{}^I_{[S]} = \text{TRUE}$. By induction we know that *OCL-Bool*$_1{}^I_{[S]} = \text{TRUE}$ iff for every substitution S_t that can be obtained from q_1 we have *OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ if and only if *OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$ for every substitution that can be obtained from πq_1.

Consider the case:

$$q_c = \sigma_{a\omega b} q_1$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \textit{OCL-Bool'}, \mathcal{M}')$$

where

$$\begin{aligned}
\textit{OCL-Bool'} &= \text{`}v_a\ \omega\ v_b\ \textit{implies'} + \textit{OCL-Bool} \\
v_a &= \mathcal{M}'.getVar(q_1.a) \\
v_b &= \mathcal{M}'.getVar(q_1.b) \\
\mathcal{M}' &= getCompleteMap(\mathcal{M}, q_1)
\end{aligned}$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from q_1 it holds that *OCL-Bool'*$^I_{[S_t][S]} = \text{TRUE}$. Unfolding *OCL-Bool'* we get: $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from q_1 it holds that $v_a\ \omega\ v_b$ *implies OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$. According to the OCL$_{\text{FO}}$ semantics, we know that for every substitution S_t obtained from $\sigma_{a\omega b} q_1$, it holds that $v_a\ \omega\ v_b{}^I_{[S_t][S]} = \text{TRUE}$. Thus, $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t obtained from $\sigma_{a\omega b} q_1$ we have *OCL-Bool*$^I_{[S_t][S]} = \text{TRUE}$.

Consider the case:

$$q_c = q_1 \times q_2$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \textit{OCL-Bool}_2, \mathcal{M})$$

where

$$\textit{OCL-Bool}_2 = \text{oclTranslation}(q_2, \textit{OCL-Bool}, \mathcal{M})$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_{t1} from q_1 we have $\textit{OCL-Bool}_2{}^I_{[S_{t1}][S]} = \text{TRUE}$. By induction again, we see that $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_{t1} from q_1, and every substitution S_{t2} from q_2, we have $\textit{OCL-Bool}^I_{[S_{t2}][S_{t1}][S]} = \text{TRUE}$. Taking in account that any substitution S_t from q_c is obtained from any pair of substitutions S_{t1} and S_{t2} from q_1, and q_2 (respectively), we finally get $\mathcal{I} \models \phi_{[S]}$ iff for every substitution S_t from q_c we have $\textit{OCL-Bool}^I_{[S_t][S]} = \text{TRUE}$.

Consider the case:

$$q_c = q_1 \setminus q_2$$

Applying Algorithm 4 we get:

$$\phi = \text{oclTranslation}(q_1, \textit{OCL-Bool'}, \mathcal{M}_1)$$

where

$$\begin{aligned}
\textit{OCL-Bool'} &= \text{oclTranslation}(q_2, \textit{ocl-ineq} + \text{`}\textit{implies}\text{'}+\\
&\qquad \textit{OCL-Bool}, \mathcal{M}_2)\\
\text{ocl-ineq} &= \text{getInequalities}(\mathcal{M}_1, \mathcal{M}_2, q_1, q_2)\\
\mathcal{M}_1 &= \text{getCompleteMap}(\mathcal{M}, q_1)\\
\mathcal{M}_2 &= \text{getCompleteMap}(\mathcal{M}, q_2)
\end{aligned}$$

By induction we know that $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t obtained from q_1 it holds that $\textit{OCL-Bool'}^I_{[S_t][S]} = \text{TRUE}$. Unfolding *OCL-Bool'* using the induction hypothesis we have that $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t from q_1, and any substitution S_{t2} from q_2, it holds that $(\textit{ocl-ineq implies OCL-Bool})^I_{[S_{t2}][S_t][S]} = \text{TRUE}$. Equivalently, $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t from q_1, and any substitution S_{t2} from q_2, it holds that $\textit{ocl-ineq}^I_{[S_{t2}][S_t][S]} = \text{FALSE}$ or $\textit{OCL-Bool})^I_{[S_t][S]} = \text{TRUE}$. Since we know that, for any substitution S_t obtained from $q_1 \setminus q_2$ there is no substitution S_{t2} obtained from q_2 for which $\textit{ocl-ineq}^I_{[S_{t2}][S_t][S]} = \text{FALSE}$, we see that , $\mathcal{I} \models \phi_{[S]}$ iff for any substitution S_t obtained from $q_1 \setminus q_2$ we have $\textit{OCL-Bool})^I_{[S_t][S]} = \text{TRUE}$.
□ □

With this algorithm at hand, we now proof that any constraint that can be checked by means of relational algebra can be written in OCL$_{\text{FO}}$.

Property 7. Any RA query can be translated into an equivalent OCL$_{FO}$ constraint

Let S_{UML} be a UML conceptual schema, and q a relational algebra expression defined over the relational view of S_{UML}. Then, consider the OCL$_{\text{FO}}$ constraint ϕ s.t. $\phi = \text{oclTranslation}(q, \textit{false}, \emptyset)$ (Algorithm 4). Then, for any interpretation $\mathcal{I}$, we have:

$$\mathcal{I} \models \phi^I \ \textbf{iff} \ q(\mathcal{I}) = \emptyset$$

Proof. From Property 6 we know that $\mathcal{I} \models \phi^I$ **iff** for every substitution S_t obtained from q, $\textit{false}^I_{S_t} = \text{TRUE}$. This is the case if and only if there is no substitution S_t obtained from q. That is $\mathcal{I} \models \phi^I$ **iff**$q(\mathcal{I}) = \emptyset$. □ □

3.6 OCL$_{\text{CORE}}$

To conclude our analysis, we identify a minimum subset of OCL$_{\text{FO}}$ that we call OCL$_{\text{CORE}}$. OCL$_{\text{CORE}}$ is minimum in the sense that any of its proper subsets is not sufficient to encode the whole OCL$_{\text{FO}}$. Several minimum core fragments might exist.

Finding the smallest equivalent fragment of a language is crucial since it allows focusing on the relevant expressions of the language without considering cumbersome syntactic sugar. Therefore, it facilitates determining the relationship with OCL$_{\text{FO}}$ of any fragment of OCL that might be proposed and it entails that any implementation handling OCL$_{\text{CORE}}$ will also be able to deal with OCL$_{\text{FO}}$.

In Figure 3.7 we define OCL$_{\text{CORE}}$. which only contains the operations: *allInstances*, *forAll*, *implies*, $<$, and $=$, together with the operation to navigate from a variable to a role/attribute.

Next property states that this fragment is able to encode all constraints in OCL$_{\text{FO}}$.

OCL-Bool	::=	*OCL-Bool implies OCL-Bool* |
		OCL-Set->forAll(Var | *OCL-Bool)* |
		OCL-Object = *OCL-Object* |
		OCL-Value < *OCL-Value*
OCL-Set	::=	*Class*▪*allInstances()*
OCL-Object	::=	*Var* |
		Var▪*role*
OCL-Value	::=	*Var*▪*fAttr* |
		⟨a constant name⟩

Figure 3.7: Syntax of OCL_{CORE}

Property 8. $\text{OCL}_{\textit{CORE}}$ captures $\text{OCL}_{\textit{FO}}$

For any OCL_{FO} constraint ϕ, there is an OCL_{CORE} constraint ϕ_c such that, for any interpretation $\mathcal{I}$:

$$\mathcal{I} \models \phi \textbf{ iff } \mathcal{I} \models \phi_c$$

Proof. Given any ϕ constraint written in OCL_{FO}, we can obtain its corresponding constraint ϕ_c written in OCL_{CORE} by first translating ϕ to a RA query q, and then, translating q to OCL_{CORE}. Any OCL_{FO} constraint can be translated into a relational algebra query q following the process described in Section 3.4. Then, we can translate q back into OCL using the process described in Section 3.5, thus, obtaining an OCL_{FO} constraint ϕ'. By construction, this ϕ' already accommodates the OCL_{CORE} syntax described in 3.7, except for operations *and* and *not* that might appear in ϕ' but are not included in OCL_{CORE}. However, we can easily get rid of these operations using common boolean equivalences with *implies* (e.g. *not OCL-Bool* is equivalent to *OCL-Bool implies 1=2*). □ □

Now, it only lacks to show that OCL_{CORE} is minimal since we cannot remove any operation from it without loosing expressiveness. We cannot remove *allInstances* since it is the only operation that allows obtaining a set of instances from a UML class. We cannot take out *forAll* because it is the only operation that can be applied after *allInstances*. The *implies* operation is mandatory to encode, for instance, the OCL *not*. Finally, without $<$, or $=$, we would not be able to encode $<=$ (among others).

3.7 Related Work

The expressiveness of OCL and its relationship with Relational Algebra has been previously discussed by Mandel and Cengarle in [75]. However, whereas Mandel and Cengarle addressed the expressive power of OCL as a query language, we look at OCL as a constraint language. The main difference is that a constraint language deals only with boolean expressions, so when looking at the equivalence of OCL w.r.t. RA, we focus on whether we can check some OCL constraint through checking the emptiness of a RA query q, and viceversa. In contrast, Mandel and Cengarle investigate whether for every RA query q we can build an OCL expression that returns the same tuples as q. In particular, the authors argue that this is impossible since (1) OCL has no tuple constructor, and (2) OCL has no way to dynamically create new types. This implies that an OCL expression returns either a value of a primitive type, or an object of a previously defined UML class; thus, it is not possible in OCL to formulate RA queries that produce arbitrary structures. However tuple constructors were later introduced in OCL 2.0.

Since OCL 2.0 introduced tuple facilities, Balsters argued that OCL is able to encode any RA query composed of the basic RA operations [9], that is: union, difference, product, renaming, selection and projection operations. However, Balsters stressed in his work that, still, OCL is not equivalent to RA in a *maximal sense* since it is impossible in OCL to define a new operation that receives as input two arbitrary sets of tuples, and outputs the natural join of them. Note that this supposed impediment does not affect us since our goal is to show that any given RA query can be rewritten into an equivalent OCL_{FO} (not necessarily in a generic way).

From a more practical point of view, Queralt and Teniente proposed in [103] a translation from OCL to domain-independent first-order logics, which is equivalent to relational algebra. The fragment covered in their translation is expressively equivalent to OCL_{FO} since they cover OCL_{CORE}. Interestingly, their translation is also based on first normalizing an OCL constraint into another one composed of less expressions. Probably, a further study of such normalization might bring another OCL_{CORE} for OCL_{FO}. However, since their intention was to apply first-order reasoners on OCL rather than discussing OCL expressiveness, they did not prove that domain-independent first-order logic statements were expressible in OCL, neither that such normalization could bring a core, as we have done.

Another translation of a fragment of OCL into first-order logic is proposed by Clavel et al in [25]. By naturally extending their translation of inequalities to inequalities with objects, we can see that their OCL fragment covered is expressively equivalent to OCL_{FO}. In contrast, the translation given by Beckert et al in [10] seems to deal with a broader subset of OCL. However, their translation is not pure first-order logics since, for instance, it uses some built-in functions to count the number of times an object appears in a collection unrestrictedly, which is not a first-order capability.

It is important to note that none of these proposals departs from an OCL formal semantics. Thus, none of the previous translations has a proof of soundness. It can be argued that no soundness proof is required since, in the absence of formal OCL semantics, the semantics of OCL turns to be the translation itself. However, using such translations as the semantics for OCL is cumbersome and error prone. Indeed, they are defined by means of multiple algorithms and functions. Thus, it is extraordinarily difficult to assess, for instance, whether the semantics given by Queralt and Teniente [103] is equivalent to the one given by Clavel et al [25]. In contrast, the OCL_{FO} semantics we provide in this paper is concise and based on basic set theory (i.e., set inclusion, exclusion, etc). Then, it could be used to proof that some translation is *sound* w.r.t. OCL_{FO} semantics, and thus, two OCL translations would be equivalent if they are both sound with respect to OCL_{FO} semantics.

There are some tools that implement translations from OCL into SQL. Egea et al introduced MySQL4OCL[41], which generates MySQL code for a subset of OCL expressions. However, the translation defined clearly falls out of RA since it uses MySQL specific procedures. Another tool, part of the well-known Dresden OCL Toolkit [8], is OCL2SQL. It produces a translation in standard SQL, but lacks theoretical basis for sophisticated cases. Indeed, the translation is based on some straightforward patterns without any formal proof [39], thus, it is not clear the correctness of the translation when dealing with, for instance, NULL values. Indeed, OCL2SQL makes use of SQL NOT EXISTS expressions which is known to have spurious behavior when dealing with NULL values, but no discussion on this aspect is given.

OCL_{FO} is a fragment of OCL defined for ensuring efficient integrity checking. However, other fragments of OCL has been defined pursuing different objectives like OCL-Lite [101]. OCL-Lite is an OCL fragment designed to ensure satisfiability checking decidability. Indeed, checking whether there exists some instance I that satisfies a set of OCL constraints is known to be undecidable. This suppose a problem to early artifact verification/validation approaches [103]. Thus, Queralt et al defined the OCL-Lite fragment of OCL, an expressive fragment of OCL for which satisfiability checking is decidable. Inspecting its syntax, we can see that OCL-Lite is a subset of OCL_{FO}. Moreover, we know that it is a proper subset since we know that OCL_{FO} satisfiability checking is undecidable.

3.8 Conclusions

OCL is a formal language for defining constraints that serves as a complement for graphical modelling languages such as UML. However, full OCL is so expressive that checking OCL constraints is not even semidecidible. That is, no algorithm can check whether a general OCL constraint is satisfied in an arbitrary UML instance in finite time, not even in the case that the OCL constraint is, in fact, satisfied.

To tackle this issue, we have identified OCL_{FO}, the fragment of OCL equivalent to relational algebra. That is, any OCL_{FO} constraint can be evaluated by checking if some RA query is empty (which guarantees efficiency), and any RA query can be translated into OCL_{FO} (which guarantees expressiveness).

The syntax and semantics of OCL_{FO} are defined in a concise way and thus, we argue that can be easily adopted by OCL practitioners. Moreover, we identify the minimal subset of OCL_{FO} with its same expressive power, OCL_{CORE}, which makes OCL_{FO} an easy object of study.

As further work, we would like to identify the subset of OCL equivalent to first-order logics with least fixed points, since it is known that such language captures exactly the constraints that can be checked in polynomial time w.r.t. data complexity [65].

Part II

Incremental Integrity Checking/Maintenance in UML/OCL

Chapter 4

Incremental Checking of UML/OCL Constraints

In the previous chapter we have identified OCL_{FO}, a subset of OCL such that can be evaluated efficiently. Now, our intention is to build the necessary mechanisms to perform such efficient evaluation.

Briefly, our method for incrementally checking UML/OCL constraints is based on building a set of logic rules called Event-Dependency Constraints which, roughly speaking, captures the different ways some structural events can cause the violation of an integrity constraint. Then, such EDCs can be easily implemented as SQL queries. In this manner, we can incrementally evaluate some constraint by executing some SQL queries and checking whether they retrieve the empty set, or not.

Formally, our presented method is an extension of an already existing proposal for incremental integrity checking in the context of databases (i.e., the *event rules* [82]). In this thesis, we extend this method to deal with constraints using aggregation, exploit such extension to better treat constraints involving existential variables, and reduce the number of EDCs generated. For doing so, we slightly modify the event rules basic definitions and proofs, so that it can be easily extended for our purposes while proving the overall method soundness and completeness.

In the following, we start giving a first intuitive approach on how to build the EDCs and how to implement them in SQL. Then, we formalize the method and prove its soundness and completeness. Afterwards, we show some experiment results based on a SQL implementation of the method, and present TINTIN, a tool for incremental integrity checking assertions in databases based on this technique. We continue by discussing the related work and finish discussing some future work and conclusions.

4.1 An Intuitive Approach to EDCs

In this section, we give a first approach on building the EDCs for UML/OCL constraints, and how to implement them in SQL. To do so, we start showing how to build and implement the EDCs for the easiest possible constraints, that is, constraints not involving aggregations nor existential variables (e.g. OCL constraints 1-4 of our running example). Then, we move to constraints with aggregate operations (e.g. OCL constraint 5), and finally we deal with constraints involving existential variables (e.g. OCL constraint 6).

4.1.1 EDCs for Simple Constraints

The process to build the EDCs from some UML/OCL constraint has two steps. In the first step, we write the UML/OCL constraints as logic denials, and then, we translate such logic denials to EDCs.

To write the UML/OCL constraints as denials, we can either use the translation we have defined in Chapter 3, together with a translation from relational algebra to predicate logic (e.g. [27]), or we can directly rely on a UML/OCL constraints to denials translation such as [103]. We decide for the latter since it has already been implemented, optimized, and proved in other applications [106]. However, any other approach to obtain logic denials would also suit our purposes.

Using this translation, we can encode UML min/max cardinality constraints, hierarchies, disjoint/complete constraints, and OCL_{FO} textual constraints as denials. That is, we are able to encode any graphical UML constraint and almost all the OCL constraints we want to tackle with the exception of the OCL constraints involving aggregations. We briefly exemplify such denials encoding in the next lines.

The basic idea to encode each UML/OCL constraint as a denial is to write a logic rule that states the condition that causes the constraint violation. Indeed, it is well-known that every first-order constraint can be written as a denial [74].

Logic denials are written over the logic signature determined by the UML class diagram as in Chapter 3. For instance, the logic denials corresponding to our OCL constraints in Figure 1.2 are written over the following signature:

$$\begin{aligned}
&content(c), contentPrice(c, price), contentAge(c, age),\\
&movie(m), episode(e), series(s), episodeSeries(e, s),\\
&user(u), userAge(u, age), premiumUser(p),\\
&buys(u, c), visualizes(u, c)
\end{aligned}$$

For the sake of simplicity, and without loss of generality, we omit the *contentCode* and *userName* predicates since we use the *code/name* attributes directly as the OIDs of *Content*/*User* instances.

Given the previous logic signature, the constraint *SeenIsBought* can be written as the following denial:

$$visualizes(u, c) \wedge \neg buys(u, c) \rightarrow \bot$$

Intuitively, this denial states that, if there is some user u visualizing some content c that u does not buy, *then*, there is a constraint violation.

Obtaining EDCs for Logic Denials

An event dependency constraint (EDC) identifies a particular situation in which the logic denial would be violated in a data state I^n resulting from applying some set of structural events to some initial data state I. Therefore, each denial constraint obtained in the previous step is translated into several EDCs, each one corresponding to a different way in which the constraint may be violated.

The main idea for obtaining the EDCs is to replace each literal in the denial constraint by some expression that evaluates it in the new state I^n. Positive and negative literals in the denial are handled in a different way according to the event rules equivalences [82]:

$$\forall \overline{x}.\ p^n(\overline{x}) \equiv (\iota p(\overline{x})) \vee (\neg \delta p(\overline{x}) \wedge p(\overline{x})) \tag{4.1}$$
$$\forall \overline{x}.\ \neg p^n(\overline{x}) \equiv (\delta p(\overline{x})) \vee (\neg \iota p(\overline{x}) \wedge \neg p(\overline{x})) \tag{4.2}$$

Rule 4.1 states that an atom $p(\overline{x})$ will be true in the new state I^n if its insertion structural event has been applied or if it was already true in the initial state I and its deletion structural event has not been applied. In an analogous way, rule 4.2 states that $p(\overline{x})$ will not hold in I^n if it has been deleted or if it was already false and it has not been inserted.

By applying the substitutions according to the above equivalences, we get a set of EDCs. Each EDC states a different way to violate a constraint by means of applying structural events, and the whole set of EDCs obtained covers all the possibilities. From the previous denial constraint we get:

$$\iota visualizes(u, c) \wedge \delta buys(u, c) \rightarrow \bot \tag{4.3}$$
$$\iota visualizes(u, c) \wedge \neg buys(u, c) \wedge \neg \iota buys(u, c) \rightarrow \bot \tag{4.4}$$
$$visualizes(u, c) \wedge \neg \delta visualizes(u, c) \wedge \delta buys(u, c) \rightarrow \bot \tag{4.5}$$
$$visualizes(u, c) \wedge \neg \delta visualizes(u, c) \wedge \neg buys(u, c) \wedge \neg \iota buys(u, c) \rightarrow \bot \tag{4.6}$$

Intuitively, the EDC 4.3 states that there is a constraint violation if we insert that some user u visualizes some content c and, at the same time, we delete that u buys c. EDC 4.4 says that a violation also occurs if u does not buy c in the current data state, and we are not inserting that u buys c with the structural events. Similarly, the

EDC 4.5 states that a violation takes place when deleting that u buys c in case u has visualized c and we do not delete such visualization. Finally, the last EDC 4.6 says that there is a violation if, in the current data state, u visualizes c without buying it, and we do not apply any structural event to delete the visualization nor insert the purchase.

Note that, the last generated EDC corresponds to the case where a violation occurs in the current data state I, and we do not apply any structural event that might change the constraint violation. Since we assume that the current data state satisfies all the constraints, we can remove the last generated EDC from our set without compromising the completeness of our method.

In its original proposal, the number of generated EDCs from a given denial grew exponentially with the length of the denial. This is because each literal is replaced with two different possibilities, thus, if the original denial has k literals, we end up with 2^k-1 EDCs.

However, in this thesis we propose to introduce *disjunctions* in the EDCs to avoid such problem. By introducing disjunctions, we can abstract the common part of two different EDCs and thus, generate a linear number of them. For instance, the previous EDCs, could be written as:

$$\iota visualizes(u,c) \wedge (\delta buys(u,c) \vee \neg buys(u,c) \wedge \neg \iota buys(u,c)) \rightarrow \bot \tag{4.7}$$

$$visualizes(u,c) \wedge \neg \delta visualizes(u,c) \wedge \delta buys(u,c) \rightarrow \bot \tag{4.8}$$

That is, using disjunctions we build, for a given denial constraint with k constraints, exactly k denials with at most $2k$ literals each.

Implementing EDCs as SQL queries

Now, we show how to implement EDCs as SQL queries. In this manner, we can use any relational database management system to incrementally check integrity constraints. In other words, we can exploit query optimization techniques to solve the problem of incrementally checking constraints (e.g., query execution planners, different join algorithms, cache memories, indexes, etc).

To do so, we assume that we have an SQL schema version of the UML schema. That is, we have an SQL schema in which there is a table T for each UML class/association in the UML schema. For our purposes, and without loss of generality, we also assume that each attribute/role of a class/association is codified as an SQL attribute of its corresponding SQL table.

Moreover, we suppose that, for each class/association/attribute of the UML schema, we have two additional tables *ins_T/del_T*. Intuitively, each table *ins_T/del_T* contains the facts about T that we insert/delete, that is, *ins_T/del_T* tables contain the structural events. Attribute updates are codified using the traditional technique of considering a deletion and an insertion of a new value.

In this manner, we can implement EDCs as SQL queries by mapping the literals l representing UML classes/associations/attributes to SQL tables T representing the same concept, and mapping the literals $\iota l/\delta l$, to the tables ins_T/ins_T representing the same structural event insertion/deletion.

In particular, each literal from the EDC is mapped into its corresponding SQL table reference, and such table reference is placed in the FROM clause of the query being build. When doing so, we define an SQL JOIN for the table reference when the original literal is positive and it has some variable in common with another previously translated literal. In contrast, we define an SQL *antijoin* (by means of a LEFT JOIN together a IS NULL condition) for negative literals. Built-in literals and constant bindings are directly translated in the WHERE clause. Following our previous example, we would translate EDC 4.8 as:

```
SELECT V.user, V.content
FROM visualizes AS V
    LEFT JOIN del_visualizes AS dV
        ON (V.user = dV.user AND V.content = dV.content)
    JOIN del_buys AS dB
        ON (dB.user = v.user AND dB.content = v.content)
WHERE dV.cast IS NULL
```

Intuitively, this SQL query looks for those users u in the table *visualizes* such that: (1) they visualize some content c, (2) there is no structural event deleting such visualization of c, (3) there is some structural event deleting that u has bought c.

When executing this query, the query planner specifies its start from the *del_buys* table, rather than *visualizes*. This is because the cardinality of *del_buys* is expected to be much lower than the one of *visualizes*. Indeed, *del_buys* only contains insertion structural events, whereas *visualizes* contains all the current visualization relationships of the current data state. In this way, the DBMS does not look through all current data (i.e., all data in *visualizes*), but only to the data that joins the applied structural events. Moreover, if *del_buys* has no tuples, the query returns the empty set without accessing any other data. This is because any join with no tuples trivially returns the empty set. In this way, our queries behave incrementally since they only look to the data related to the update, and only when the update may cause a violation.

4.1.2 The Event-Dependency Constraints for Aggregations

We extend now the previous approach to deal with aggregates.

There exist several kinds of aggregates according to the complexity to incrementally update them when some structural event is applied [59]. In this work, we focus on *distributive* aggregation. Intuitively, distributive aggregates are those that can be updated by taking into account the current aggregated value of the data, the aggregated value of the data inserted, and the aggregated value of the data deleted. The distributive aggregates of OCL are: *sum*, *size* and *count*.

As we did before, we first translate any OCL constraint into a logic denial constraint; then, we translate this denial constraint into several EDCs and, finally, we translate each EDCs into an SQL query.

Logic Denial Encoding of OCL Aggregations

Any OCL aggregation expression is defined by means of a *source* (i.e., a navigation) and an *aggregation operation* (e.g. *sum*), where the resulting aggregate value is normally used in some arithmetic comparison. We translate the *source* of the expression following the same lines as [103], and from there, we use an *aggregate predicate* to aggregate the required value. Once we obtain the aggregated required value, we can define the built-in literal encoding the arithmetic comparison.

For instance, given the *AllEpisodesMaxPrice* constraint, the source of the OCL aggregation expression is *self.castMember.episodes.price*. This expression is translated as the following conjunction of literals:

$$episodeSeries(e, s) \wedge contentPrice(e, p)$$

From there, we can aggregate the prices (i.e., the p term) by means of defining an aggregate predicate:

$$sumPrices(s, sum(p)) \leftarrow episodeSeries(e, s) \wedge contentPrice(e, p)$$

In this way, the atom *sumPrices(s, x)* indicates that the sum of prices of the episodes of s is x. Thus, we can use x to check whether the sum of the episode prices for some series is greater than 100 with the following logic denial:

$$sumPrices(s, x) \wedge x >= 100 \rightarrow \bot$$

Obtaining EDCs for Denial Constraints with Aggregates

The most important issue for obtaining the EDCs in the presence of aggregate predicates relies on how to compute the aggregate value in the new data state I^n. We make use of two aggregate event predicates for this purpose: one for computing the aggregated value x_ι for the data being inserted, and another one for computing the aggregated value x_δ for the data being deleted. Since we focus on OCL distributive aggregation functions, we know that the aggregated value in the new state I^n equals to the current aggregated value x plus x_ι minus x_δ.

For instance, the previous denial would be translated as the following EDC:

$$\begin{aligned}&sumPrices(s,x) \wedge \iota sumPrices(s,x_\iota) \wedge \delta sumPrices(s,x_\delta) \wedge x_\iota > x_\delta \wedge \\ &\quad x < 100 \wedge x + x_\iota - x_\delta >= 100 \rightarrow \bot \end{aligned} \tag{4.9}$$

Intuitively, $\iota sumPrices(s,x_\iota)$ and $\delta sumPrices(s,x_\delta)$ computes the sum of the prices of the new episodes being added/deleted to some series s; $x_\iota > x_\delta$ ensures that the sum of prices of the new episodes is greater than the sum of the removed episodes, which means that the aggregation variable x changes its value in I^n in a manner that might violate the constraint; finally, $x <= 100 \wedge x + x_\iota - x_\delta >= 100$ ensures that the old aggregated value was satisfying the constraint, but the new one is not.

Now, we need to define the aggregate event predicates ι*sumPrices* and δ*sumPrices*. Recall that, for ι*sumPrices*, we want to sum the prices of the new episodes being added to the source *self.castMember.episodes.price*. Again, we can compute the new instances added to the *source* by replacing their literals according to the formulas 4.1 and 4.2:

$$\begin{aligned}&\iota sumPrices(s,sum(p)) \leftarrow \iota episodeSeries(e,s) \wedge \iota contentPrice(e,p)\\ &\iota sumPrices(s,sum(p)) \leftarrow \iota episodeSeries(e,s) \wedge contentPrice(e,p) \wedge \neg\delta contentPrice(e,p)\\ &\iota sumPrices(s,sum(p)) \leftarrow episodeSeries(e,s) \wedge \neg\delta episodeSeries(e,s) \wedge \iota contentPrice(e,p)\end{aligned}$$

Similarly, we can define δ*sumPrices*. In this case, we have to replace insertions by deletions since we are looking for instances which are deleted from the *source*:

$$\begin{aligned}&\delta sumPrices(s,sum(p)) \leftarrow \delta episodeSeries(e,s) \wedge \delta contentPrice(e,p)\\ &\delta sumPrices(s,sum(p)) \leftarrow \delta episodeSeries(e,s) \wedge contentPrice(e,p) \wedge \neg\delta contentPrice(e,p)\\ &\delta sumPrices(s,sum(p)) \leftarrow episodeSeries(e,s) \wedge \neg\delta episodeSeries(e,s) \wedge \delta contentPrice(e,p)\end{aligned}$$

Note that, in both cases, the different rules form a partition of the instances being inserted/deleted in the *source* expression. In this way, we can compute the total aggregated value x_ι and x_δ by the sum of the aggregated values obtained from the various rules.

Implementing EDCs with Aggregated Events into SQL

Implementing EDCs with aggregates into SQL queries follows the same principles as before: each literal is translated as a table reference in the FROM clause possibly with a JOIN condition.

However, in this case, and for the sake of efficiency, we propose to materialize the aggregated value referred by the EDC. That is, for instance, to implement the EDC 4.9 we assume that we have some table *sumPrices* containing the sum of the episode prices for each series. In this manner we avoid to recompute such aggregate each time we need to verify that constraint.

Thus, the rule EDC 4.9 can be implemented as the SQL query:

```
SELECT sumPrices.series, sumPrices.X + ins_sumPrices.X - del_sumPrices.X
FROM sumPrices
    LEFT JOIN ins_sumPrices ON(sumPrices.series = ins_sumPrices.series)
    LEFT JOIN del_sumPrices ON(sumPrices.series = del_sumPrices.series)
WHERE ins_sumPrices.X > del_sumPrices.X AND sumPrices.X < 100 AND
    sumPrices.X+ins_sumPrices.X-del_sumPrices.X >= 100
```

Where *ins_sumPrices* and *del_sumPrices* are two views computing the aggregation of the prices of the episodes being inserted and deleted for the different series. Note that we can update the materialized aggregate value by means of such views in case we finally commit the structural events.

Now, we need to define the SQL views *ins_sumPrices* and *del_sumPrices*. Such views are defined by means of translating into SQL the different definition rules of the predicates ι*sumPrices* and δ*sumPrices* specified in the EDCs. For instance, the second definition rule of ι*sumPrices* is translated as:

```
CREATE VIEW ins_sumPrices2 AS
SELECT iES.series, SUM(CP.price) AS X
FROM ins_episodeSeries AS iES
    LEFT JOIN contentPrice AS CP ON (CP.content = iES.episode)
    LEFT JOIN del_contentPrice AS dCP ON (dCP.content = CP.content)
WHERE dCP.content IS NULL
GROUP BY iES.series
```

These views are obtained by translating the body of the rule into SQL following the same principles as before, then applying a GROUP BY with the attributes corresponding to the terms of the rule's head, and finally aggregating the corresponding term.

At this point, we only need to define a view combining all the different episode prices sums corresponding to the different definition rules. For instance, in the case of *ins_sumPrices*, we define the SQL query:

```
CREATE VIEW ins_sumPrices AS
SELECT iSP1.series, iSP1.X + iSP2.X + iSP3.X AS X
FROM ins_sumPrices AS iSP1
    FULL OUTER JOIN ins_sumPrices2 AS iSP2 ON (iSP1.series = iSP2.series)
    FULL OUTER JOIN ins__sumPrices3 AS iSP3 ON (iSP1.series = iSP3.series)
```

4.1.3 The EDCs for Existential Variables

We refer as *existential variables* to those variables that appear in the body of some derivation rule, but not in its head. For instance, consider the derivation rule: [1]

$$existsUnboughtEpisode(u, s) \leftarrow episodeSeries(e, s) \land \neg buys(u, e)$$

Intuitively, such derivation rule returns true, if and only if for the given user u and series s, there exists some episode e of s such that u has not bought. Thus, e is an existential variable of this derivation rule.

We naturally extend the notion of existential variables of a derivation rule to derived literals and predicates. That is, we refer as existential variables of some derived literal l (or predicate p) to the existential variables of the derivation rule of l (or p). In the previous example, e is an existential variable of any literal with predicate $existsUnboughtEpisode$.

Dealing with existential variables for incremental checking can be challenging. Indeed, when deleting some data, we might loose the value that witnessed the truth evaluation of the derivation rule, thus, forcing the incremental checking engine to essentially look through all the data to find another witness, if it exists. However, as we are going to see, we can treat existential variables using aggregate predicates, and thus, increase the efficiency to deal with them in case we decide to materialize the aggregation in the SQL implementation.

In the following, we first argue the difficulty to deal with existential variables through an illustrative example, and then, we show how this difficulty can be avoided using the aggregate predicates.

The Existential Variables Problem

To illustrate the difficulty to deal with existential variables, consider the *Premium-BoughtCompleteSeries* OCL constraint of our running example. Such OCL constraint,

[1]This derivation rule is not *safe* because of the term u, but it is still *admissible*. A rule is called *admissible* if its *unsafe* terms appear in the head of the rule. It is known that a non-recursive logic program can be correctly (top-down) evaluated through an interpretation I, if the positive literals built from *admissible* predicates do not contain unsafe terms [38].

when encoded into logics, gives rise to the following denial:

$$premiumUser(u) \wedge \neg existsCompleteSeries(u) \rightarrow \bot$$
$$existsCompleteSeries(u) \leftarrow series(s) \wedge \neg existsUnboughEpisode(u, s)$$
$$existsUnboughtEpisode(u, s) \leftarrow episodeSeries(e, s) \wedge \neg buys(u, e)$$

Intuitively, the logic denial checks if for all the premium users u, there exists some series s, for which there is no episode unseen by u. Note that there are two existential variables in this logic formalization: s (ranging the *series*), and e (ranging the episodes of some series).

Now, consider the case in which some premium user, e.g. *Phil*, has bought all the episodes of the series *Modern Family*. Thus, *Modern Family* witnesses that *Phil* satisfies the *PremiumBoughtCompleteSeries* constraint. Now, assume that a new episode of *Modern Family* is added in the data state, so, we no longer know if *Phil* satisfies the *PremiumBoughtCompleteSeries* constraint. Thus, to check this constraint we have to, essentially, iterate through all the series and all its episodes to verify if *Phil* has another series for which he has bought all its episodes. That is, we need to almost look through all the data of the system, which might be prohibitive. Moreover, we will have to do so for each premium user who had bought all the episodes of *Modern Family*.

Dealing with Existential Variables Through Aggregates

The basic idea to efficiently deal with existential variables is to aggregate the number of witnesses they have. That is, in the case of the *PremiumBoughtCompleteSeries* OCL constraint, we can count, for each premium user u, the number of series such that u has seen all its episodes, and state in the denial that there is a violation in case such number is 0.

In this manner, we can use the previous mechanism defined for dealing with aggregate values. In particular, we can implement such EDC into SQL by materializing the aggregate value, which means that, in case that some premium user u looses some witness for the existential variable, we only need to check if the counter of witnesses is 0 to assess if there is a constraint violation. We argue that this is a much more efficient behavior rather than apply a complete search for a new witness.

For instance, the previous OCL constraint can be reformulated as:

$$premiumUser(u) \wedge completeSeries(u, x) \wedge x <= 0 \rightarrow \bot$$
$$completeSeries(u, count()) \leftarrow series(s) \wedge unboughtEpisodes(u, s, y) \wedge y <= 0$$
$$unboughtEpisodes(u, s, count()) \leftarrow episodeSeries(e, s) \wedge \neg buys(u, e)$$

Note that we have aggregated two existential variables, that is, we count the number x of series that each user u has seen all its episodes, and also the number y of episodes each user has not seen from each series, with the purpose to materialize them in the SQL implementation. This means materializing and maintaining a polynomial amount of aggregates. However, we argue that this materialization is still feasible and worthwhile. Indeed, as we are going to see in the experiments, maintaining such aggregates have very low time penalties.

4.2 Formalizing EDCs

In the previous section we have given an intuitive understanding of what EDCs are, how can they be obtained, and how can they be implemented in SQL. Now, we formalize the basic notions of EDCs to unambiguously describe our method, and to give a proof for its correctness.

As we have previously argued, obtaining the EDCs of some given UML/OCL constraint is a two steps process: first, we encode the UML/OCL constraint as logic denials, and then, we translate the denials into EDCs. In the following, we explain both steps separately.

4.2.1 Encoding UML/OCL Constraints as Logic Denials

Our intention here is to encode any UML minimum/maximum cardinality, hierarchy, disjoint/complete constraint, and OCL_{FO} constraint including distributive aggregations into an equivalent logic denial. Moreover, as discussed in the previous section, we want to avoid the existential variables of such logic denial encoding, and replace them with aggregates.

To do so, we depart and extend the current encoding defined in [103]. Such translation is able to deal with any UML and OCL_{FO} constraint, however, it was not though to give support to OCL aggregation operators, and might bring to several existential rules in their derivation rules. So, we extend such encoding to: (1) deal with OCL distributive aggregates, and (2) apply a postprocess to replace any existential variable with an aggregate predicate.

To encode the OCL distributive aggregates (i.e., *size*, *sum*, *count*) into denials, we bring 3 different algorithms, each one for each OCL aggregate.

Algorithm 5 translateSize(*OCL-Set* set)

```
derivationRule := translateSet(set)
derivationRule.getHead().putLastTerm(count())
return derivationRule
```

Algorithms 5, 6, 7 receive as input an *OCL-Set* expression (and an *OCL-Single* in the case of *count*) and return the derivation literal whose last term represents

Algorithm 6 translateSum(*OCL-Set* set)

```
derivationRule := translateSet(set)
$x$ := derivationRule.getHead().popLastTerm()
derivationRule.getHead().putLastTerm(sum($x$))
return derivationRule
```

Algorithm 7 translateCount(*OCL-Set* set, *OCL-Single* single)

```
derivationRuleSet := translateSet(set)
derivationRuleSingle := translateSingle(single)
$x$ := derivationRuleSet.getHead().popLastTerm()
$y$ := derivationRuleSingle.getHead().popLastTerm()
derivationRuleSet.addLiteral(derivationRuleSingle.getHead())
derivationRuleSet.addLiteral($x = y$)
derivationRule.getHead().putLastTerm(count())
return derivationRuleSet
```

the aggregated value of the set. These algorithms makes use of the *translate-Set*/*translateSingle* functions, which receive as input an *OCL-Set*/*OCL-Single* expression and return a derived literal whose last term represents any value that can be obtained from such expression. These functions are already given in [103].

Due to the absence of concise formal semantics for OCL aggregations in its standard [87], it is not possible to give any formal proof of the correctness for such algorithms. Nevertheless, we argue that the intuitive intended meaning of the OCL aggregates corresponds to the semantics underlying our translations. E.g. an OCL *size* corresponds to *count* the cardinality of the OCL source set.

Up to here, we have extended the encoding of OCL constraints in [103] to deal with aggregations. Now, our intention is to use the aggregate predicates to aggregate the existential variables that might appear in it. Indeed, as we have already discussed in the previous section, aggregating the existential variables permits us to improve the performance of the whole method.

Without loss of generality, we assume that any derived literal present in the input is defined by only one derivation rule. Indeed, given any predicate p defined with $n > 1$ derivation rules of the form $p(\overline{x}) \leftarrow body_i(\overline{x}, \overline{y})$, we can rename the predicate p of all its heads. For instance, we can rename them to $p_i(\overline{x}) \leftarrow body_i(\overline{x}, \overline{y})$ for each derivation rule $i \in \{0..n\}$. Then, any other rule containing a positive literal of p should be replaced with n rules, each one replacing p for p_i; and any rule containing a negative rule of p should be replaced for the conjunction $\neg p_1(\overline{x}), ..., \neg p_n(\overline{x})$. In this way, the semantics of all the rules are preserved and the all the derived predicates are defined with a single derivation rule.

To aggregate the existential variables appearing in this rules, we bring Algorithm 8. Intuitively, this algorithm receives as input a conjunction of literals, and returns a conjunction of literals where every existential variable has been replaced with an aggregate predicate of counting, together with a built-in literal comparing such aggregate value with 0. That is, to check whether some x exists, it is sufficient to count the number of x and comparing such number with 0.

Algorithm 8 existentialVariablesAggregation(Literals *lits*)

```
result := $\emptyset$
for all Literal $l$ in $lits$ do
  if $l$ is base or built-in then
    result.addLiteral($l$)
  else
    ulits := existentialVariablesAggregation($l$.unfoldLiterals())
    if $l$ is positive then
      result.addLiterals(ulits))
    else
      if $l$ has existential variable then
        $l'$ := createLiteral($l$.getPredicateName()+'Count', $l$.getTerms())
        createDerivationRule($l'$.getPredicateName(), $l'$.getTerms() $\cup$
                {'count()'}, ulits)
        $l'$.putLastTerm($x$)
        result.add($x$ <= 0)
      else
        result.addLiteral($l$)
      end if
    end if
  end if
end for
return result
```

The idea is thus, to apply such algorithm to every denial to aggregate every existential variable that might appear in it. In the following, we prove the algorithm correctness for performing such task.

Property 9. ExistentialVariablesAggregation correctness

Consider any denial ϕ. Then, applying Algorithm 8 to the body of ϕ we obtain a new denial ϕ' s.t. (1) all existential variables are aggregated by counting, and (2) for any given data state I, we have that:

$$I \models \phi \textbf{ iff } I \models \phi'$$

Proof. Since all predicates are non-recursive, assign to each predicate the following *strata*: 0 for base predicates, and $i+1$ for derived predicates defined over predicates whose maximum strata is i.

Given this strata, it is clear that the algorithm terminates (since at each recursive call, the maximum strata of the literals being treated decreases, and cannot decrease forever), and it is immediate to realize, by induction over the maximum strata of the literals, that all the existential variables are aggregated.

Thus, we concentrate into proving that, for any I, $I \models \phi$ iff $I \models \phi'$. We do so by showing that each literal l in ϕ is translated into an equivalent set of literals l' in ϕ'. Such proof is by induction on the maximum strata of the input literals. In the base case, the maximum strata is 0, so, all the literals are base, thus, for each literal l in ϕ we add l in ϕ'. Since l is equivalent to itself, this concludes the base case proof.

For the inductive case, for every derived literal l, the algorithm obtains the unfolding of l, and then, recursively aggregate the existential variables of its unfolded literals to obtain a new set of literals *ulits*. Clearly, l is equivalent to its unfolding by definition. Moreover, the unfolding is equivalent to *ulits* by induction hypothesis. Thus, *ulits* is equivalent to l. If, l is positive, we add *ulits* in ϕ' which concludes the proof with regarding the positive literals. If, we have $\neg l$ in ϕ, we show that the formula $lCount(..., x) \wedge x <= 0$ we add in ϕ' is equivalent to $\neg l$. Indeed, we know that, for any data state I and substitution σ, $I \models \neg l_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$. Equivalently, $I \models \neg l_{\sigma}$ iff $I \not\models$ *ulits*$_{[\sigma][\sigma_E]}$ for any substitution σ_E for the existential variables of l. By definition, $I \models lCount(..., x)_{[\sigma]}$ iff such number of substitution σ_E is x. Thus, $I \models \neg l_{[\sigma]}$ iff $I \models lCount(..., x)_{[\sigma]} \wedge x <= 0$, which concludes the proof.

□

4.2.2 From Logic Denials to EDCs

Now, we formally define how to translate the previous logic denials into EDCs.

The core to obtain the EDCs are the so called event rules [82]. The event rules are a set of equivalences that maps any literal l from some signature S to some formula ψ of an augmented signature S'. Such augmented signature contains all the predicates in S and the necessary predicates to represent the structural events, thus, intuitively, ϕ is true in any data state I with some structural events E iff l is true after applying the structural events E in I.

For our purposes, we bring a new definition and demonstration of the event rules in comparison with its original proposal in [82]. Indeed, the event rules were thought for deductive databases in which only base and derived literals existed, so, no framework for extending the event rules was given since they were already complete in their own context. However, since we want to export the event rules into another context and to include the treatment for more kinds of literals (in our case, aggregate literals) we define them in a new way that permits easily extending them and proving their correctness.

In particular, we define the event rules by means of two mappings: *new*/*old*. The mapping *new* maps each literal l to some formula ψ in S' s.t. new(l) evaluates to true iff l was false in I and becomes true after applying the structural events E (so that, l is *new*). Similarly, the mapping *old* maps each literal l to some formula ψ s.t. old(l) evaluates to true iff l was true in I and remains true after applying the structural events E (so that, l is *old*). Thus, l is true in the new state after applying E iff new(l) or old(l) are true. We would like to highlight that such mappings were already appearing in [82] as some shortcut notation, however, we give them a key role for our demonstrations.

So, the idea is to define the *new*/*old* mappings for base literals, derived literals (without existential variables since all of them have been replaced for aggregations as discussed in Section 4.2.1), and aggregate literals recursively. The base literals serve as base case and thus, its definition and proof is non-recursive. In contrast, the definition of the *new*/*old* mappings for derived literals is going to be defined through the *new*/*old* mapping itself, and its correctness proved by induction. In this manner, when defining the *new*/*old* mappings for aggregate literals recursively, and proving its correctness by induction, we get for free that the mappings for derived literals are well-defined and correct when containing aggregate literals in its body, and aggregate literals containing derived literals in its body are also well-defined and correct. Note that, in this way, extending the event rules to deal with other kinds of literals can be done by simply recursively defining such mappings for the intended new kind of literals, and proving their correctness by induction.

The *new*/*old* mappings for base and derived literals are extracted from [82], but we rewrite and repeat the demonstration here to bring the inductive proof that permits easily extending them. The definition and proof of the *new*/*old* mappings for aggregate literals is a pure new contribution of this thesis.

In the following, we first bring the basic definitions from which to define the new/old mapping. Then, we show the mappings for base literals, derived literals (without existential variables), and aggregate literals separately. Finally, we show how to build the EDCs using such mappings.

Augmented Signature, Update, New/Old Mappings

The augmented signature S' of some signature S is the signature resulting from adding a couple of predicates $\iota p/\delta p$ for each predicate $p \in S$. Roughly speaking, such couple of predicates are used to represent insertions/deletions of p.

Definition 1. Augmented signature of a signature

Given some signature S, its augmented signature S' is the signature:

$$S' = \bigcup_{p \in S} \{p, \iota p, \delta p\}$$

where ιp and δp have the same arity as p.

We now define a function to update a data state. In particular, consider a function *apply* that receives a set of instances of the structural events E written over S', some instances I over S, and returns a new set of instances I^n over S. We say that such function *apply* is an *update* function if, intuitively, it applies the insertions/deletions denoted by $\iota p/\delta p$ described in E into I and does not perform any other change. More formally:

Definition 2. Updating function

Given a signature S and its augmented signature S', consider $\mathcal{E}$ to be the universe of structural events sets that can be defined in S', and $\mathcal{I}$ to be the universe of instances that can be defined in S.
Then, a function *apply*:$\mathcal{E} \times \mathcal{I} \to \mathcal{I}$ is an *update function* if and only if for any data state $I \in \mathcal{I}$, structural events $E \in \mathcal{E}$, base predicate p from S and array of constants $\overline{X}$:

$$apply(E, I) \models p(\overline{X}) \textbf{ iff } E \models \iota p(\overline{X}) \textit{ or } (I \models p(\overline{X}) \textit{ and } E \not\models \delta p(\overline{X}))$$
$$apply(E, I) \not\models p(\overline{X}) \textbf{ iff } E \models \delta p(\overline{X}) \textit{ or } (I \not\models p(\overline{X}) \textit{ and } E \not\models \iota p(\overline{X}))$$

In the following, we assume that *apply* is an update function.

Note that such definition of update function permits applying *redundant* structural events. That is, if E contains some instance $\iota p(\overline{X})$ when $p(\overline{X})$ is already present in I, the update function leads to a new state I^n containing $p(\overline{X})$. However, note that there is no *true* insertion since $p(\overline{X})$ was already true in the initial data state I. To avoid redundant structural events, we define the concept:

Definition 3. Non-redundant structural events

A set of structural events E is non-redundant with respect to a data state I of some signature S iff, for each base predicate p in S, we have that:

$$E \models \iota p(\overline{X}) \quad \textit{only if} \quad I \models \neg p(\overline{X})$$
$$E \models \delta p(\overline{X}) \quad \textit{only if} \quad I \models p(\overline{X})$$

In the following, we assume our structural events to be *non-redundant* with respect to the data state.

Now, we define the notions of *New/Old* mappings. We define a mapping *new: Literals(S) $\rightarrow$ Formulas(S')* to be a *New* mapping if the formula new(l) evaluates to true iff l was false in the initial data state but, because of the structural events, it becomes true in the new data state. Similarly, a mapping *old: Literals(S) $\rightarrow$ Formulas(S')* is an *Old* mapping if, the formula old(l) evaluates to true iff l was true in the initial data state, and, after applying the structural events, it is still true in the new data state. Formally:

Definition 4. New/Old Mapping

Consider two mappings *new, old: Literals(S) $\rightarrow$ Formulas(S').*
new is a *New* mapping iff, for any literal l, ground substitution σ, data state I, and set of structural events E:

$$E \cup I \models \text{new}(l)_{[\sigma]} \quad \textit{iff} \quad I \not\models l_{[\sigma]} \text{ and } \mathit{apply}(E, I) \models l_{[\sigma]}$$

old is an *Old* mapping iff, for any literal l, ground substitution σ, data state I, and set of structural events E:

$$E \cup I \models \text{old}(l)_{[\sigma]} \quad \textit{iff} \quad I \models l_{[\sigma]} \text{ and } \mathit{apply}(E, I) \models l_{[\sigma]}$$

New/Old Mappings for base literals

Now, we define a pair of *New/Old* mappings for the base literals.

Intuitively, the new map just maps each literal $p(\overline{x})/\neg p(\overline{x})$ to the structural event that makes the literal true in the new data state (e.g. $\iota l/\delta l$). In contrast, the old map just maps the literal to itself together the negation of the structural event that would falsify it (e.g. $p(\overline{x})$ is mapped to $p(\overline{x}) \wedge \neg\delta p(\overline{x})$). Formally:

Definition 5. A New/Old mapping for base literals

Given any base predicate p and array of terms $\overline{x}$, we define the maps:

$$\begin{aligned}
&\text{new}(p(\overline{x})) = \iota p(\overline{x})\\
&\text{new}(\neg p(\overline{x})) = \delta p(\overline{x})\\
\\
&\text{old}(p(\overline{x})) = p(\overline{x}) \wedge \neg\delta p(\overline{x})\\
&\text{old}(\neg p(\overline{x})) = \neg p(\overline{x})) \wedge \neg\iota p(\overline{x})
\end{aligned}$$

We now prove that such mappings are, indeed, *New/Old* mappings:

Proof. We make the proof by cases. In particular, we start proving the new mapping for positive/negative literals. Then, we move to the old mapping.

We start proving that $E \cup I \models \text{new}(l)_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$. By definition of *new*, $E \cup I \models \text{new}(l)_{[\sigma]}$ iff $E \cup I \models \iota l_{[\sigma]}$. By non-redundancy of E, and the definition of *update* function, $E \cup I \models \iota l_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$. Thus, $E \cup I \models \text{new}(l)_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$.

We continue proving that $E \cup I \models \text{new}(\neg l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \not\models l_{[\sigma]}$. By definition of *new*, $E \cup I \models \text{new}(\neg l)_{[\sigma]}$ iff $E \cup I \models \delta l_{[\sigma]}$. By non-redundancy of E, and the definition of *update* function, $E \cup I \models \delta l_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \not\models l_{[\sigma]}$. Thus, $E \cup I \models \text{new}(\neg l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \not\models l_{[\sigma]}$.

Until here we have proved that *new* is a correct *New* mapping, we now prove the correctness of the *old* mapping.

We start proving that $E \cup I \models \text{old}(l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$. By definition of *old*, $E \cup I \models \text{old}(l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $E \cup I \not\models \delta l_{[\sigma]}$. Since $\delta l_{[\sigma]}$ is, actually, $\text{new}(\neg l)$, we have $E \cup I \models \text{old}(l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $E \cup I \not\models \text{new}(\neg l_{[\sigma]})$. Thus, by correctness of *new* we see: $E \cup I \models \text{old}(l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$.

We conclude proving that $E \cup I \models \text{old}(\neg l)_{[\sigma]}$ iff $I \models \neg l_{[\sigma]}$ and $apply(E, I) \models \neg l_{[\sigma]}$. By definition of *old*, $E \cup I \models \text{old}(\neg l)_{[\sigma]}$ iff $I \models \neg l_{[\sigma]}$ and $E \cup I \not\models \iota l_{[\sigma]}$. Since $\iota l_{[\sigma]}$ is, actually, $\text{new}(l)$, we have $E \cup I \models \text{old}(\neg l)_{[\sigma]}$ iff $I \models \neg l_{[\sigma]}$ and $E \cup I \not\models \text{new}(l_{[\sigma]})$. Thus, by correctness of *new* we see: $E \cup I \models \text{old}(\neg l)_{[\sigma]}$ iff $I \models \neg l_{[\sigma]}$ and $apply(E, I) \models \neg l_{[\sigma]}$.

□

New/Old Mappings for Derived Literals without Exists. Vars.

In case p is a derived predicate in S, we define its corresponding ιp and δp to be derived in its augmented signature S'. In particular, we only deal with the case in which p has no existential variables since we have already seen that existential variables can be eliminated through aggregation.

The idea is to define the derivation rules of $\iota p/\delta p$ in such a way that $\iota p(\overline{X})/\delta p(\overline{X})$ is true if and only if there is an insertion/deletion of the derived instance of $p(\overline{X})$ after applying the structural events. In this manner, the semantics of $\iota p(\overline{X})$ when p is a derived predicate coincides with the semantics of $\iota p(\overline{X})$ when p is base.

To define such derivation rules we assume that we already have, for its inner literals, two New/Old mappings: *new/old*. Moreover, we assume a new map *all*:$Literals(S) \rightarrow Formulas(S')$ which maps each literal l to the formula new(l) $\vee$ old(l). That is, all(l) is true iff l becomes true after applying the structural events (either because the structural events make l true, or it was already true and the structural events do not falsify l).

In particular, the derivation rules defined when p has no existential variables are the following:

Definition 6. Ins. Rules for Derived Predicates

Given p a derived predicate whose derivation rule body is $l_1 \wedge ... \wedge l_n \wedge b$, where each l_i is a literal and b is a conjunction of built in literals, we define n derivation rules for ιp, where each ιp_i rule has the form:

$$\iota p_i(\overline{x}) \leftarrow \left[\bigwedge_{j=1..i-1} old(l_j)\right] \wedge new(l_i) \wedge \left[\bigwedge_{k=i+1..n} all(l_k)\right] \wedge b$$

For instance, given the derived predicate $p(\overline{x}) \leftarrow q(\overline{x}) \wedge r(\overline{x}) \wedge \neg s(\overline{x})$, we define the following derivation rules:

$$\begin{aligned}
\iota p(\overline{x}) &\leftarrow \textit{new}(q(\overline{x})) \wedge \textit{all}(r(\overline{x})) \wedge \textit{all}(\neg s(\overline{x}))\\
\iota p(\overline{x}) &\leftarrow \textit{old}(q(\overline{x})) \wedge \textit{new}(r(\overline{x})) \wedge \textit{all}(\neg s(\overline{x}))\\
\iota p(\overline{x}) &\leftarrow \textit{old}(q(\overline{x})) \wedge \textit{old}(r(\overline{x})) \wedge \textit{new}(\neg s(\overline{x}))
\end{aligned}$$

Equivalently:

$$\begin{aligned}
\iota p(\overline{x}) &\leftarrow \iota q(\overline{x}) \wedge (\iota r(\overline{x}) \vee r(\overline{x}) \wedge \neg\delta r(\overline{x})) \wedge (\delta s(\overline{x}) \vee \neg s(\overline{x}) \wedge \neg\iota s(\overline{x}))\\
\iota p(\overline{x}) &\leftarrow q(\overline{x}) \wedge \neg\delta q(\overline{x}) \wedge \iota r \wedge (\delta s(\overline{x}) \vee \neg s(\overline{x}) \wedge \neg\iota s(\overline{x}))\\
\iota p(\overline{x}) &\leftarrow q(\overline{x}) \wedge \neg\delta q(\overline{x}) \wedge r(\overline{x}) \wedge \neg\delta r(\overline{x}) \wedge \delta s(\overline{x})
\end{aligned}$$

In a very similar way, we define the δp derivation rules.

Definition 7. Del. Rules for Derived Predicates

Given p a derived predicate whose derivation rule body is $l_1 \wedge ... \wedge l_n \wedge b$, where each l_i is a literal and b is a conjunction of built in literals, we define n derivation rules for δp, where each δp_i rule has the form:

$$\delta p_i(\overline{x}) \leftarrow \left[\bigwedge_{j=1..i-1} old(l_j)\right] \wedge new(\neg l_i) \wedge \left[\bigwedge_{k=i+1..n} l_k\right] \wedge b$$

For instance, given the derived predicate $p(\overline{x}) \leftarrow q(\overline{x}) \wedge r(\overline{x}) \wedge \neg s(\overline{x})$, we define the following derivation rules:

$$\begin{aligned}
&\delta p(\overline{x}) \leftarrow \textit{new}(\neg q(\overline{x})) \wedge r(\overline{x}) \wedge \neg s(\overline{x})\\
&\delta p(\overline{x}) \leftarrow \textit{old}(q(\overline{x})) \wedge \textit{new}(\neg r(\overline{x})) \wedge \neg s(\overline{x})\\
&\delta p(\overline{x}) \leftarrow \textit{old}(q(\overline{x})) \wedge \textit{old}(r(\overline{x})) \wedge \textit{new}(s(\overline{x}))
\end{aligned}$$

Equivalently:

$$\begin{aligned}
&\delta p(\overline{x}) \leftarrow \delta q(\overline{x}) \wedge r(\overline{x}) \wedge \neg s(\overline{x})\\
&\delta p(\overline{x}) \leftarrow q(\overline{x}) \wedge \neg \delta q(\overline{x}) \wedge \delta r \wedge \neg s(\overline{x})\\
&\delta p(\overline{x}) \leftarrow q(\overline{x}) \wedge \neg \delta q(\overline{x}) \wedge r(\overline{x}) \wedge \neg \delta r(\overline{x}) \wedge \iota s(\overline{x})
\end{aligned}$$

Using these derivation rules, we can now define the New/Old mapping for derived literals. Such mappings follows the same pattern as the base predicates. E.g. a derived instance of p is true after applying the structural events if its derivation rules ιp is true, or p was already true before applying the structural events, and δp is not derived. Formally:

Definition 8. A New/Old mapping for derived literals

Given any derived predicate p and array of terms $\overline{x}$, we define the maps:

$$\begin{aligned}
&\text{new}(p(\overline{x})) = \iota p(\overline{x})\\
&\text{new}(\neg p(\overline{x})) = \delta p(\overline{x})\\
\\
&\text{old}(p(\overline{x})) = p(\overline{x}) \wedge \neg \delta p(\overline{x})\\
&\text{old}(\neg p(\overline{x})) = \neg p(\overline{x})) \wedge \neg \iota p(\overline{x})
\end{aligned}$$

In the following we prove that such mappings are, indeed, correct *New*/*Old* mappings.

Proof. We start proving the correctness of the *new* mapping. Then, we move to *old*. In both cases, we distinguish between the map for positive/negative literals.

We start proving that $E \cup I \models \text{new}(l)_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$. Assume that we have $E \cup I \models \text{new}(l)_{[\sigma]}$. This is the case iff for some derivation rule we have $E \cup I \models \iota l_{[\sigma]}$. By construction of the derivation rules, this is the case iff for some literal lb_i inside the derivation rule body of l we have $E \cup I \models \text{new}(lb_i)_{[\sigma]}$, and for the rest of literals lb_j in the body we have $E \cup I \models \text{all}(lb_j)_{[\sigma]}$. Thus, we see that for all literals lb in the body of l we have $apply(E, I) \models lb_{[\sigma]}$, thus, $apply(E, I) \models l_{[\sigma]}$. Moreover, because $E \cup I \models \text{new}(lb_i)_{[\sigma]}$, we see that $I \not\models lb_{i[\sigma]}$, thus, $I \not\models l_{[\sigma]}$. The proof the other direction of the iff follows similarly.

We continue proving that $E \cup I \models \text{new}(\neg l)_{[\sigma]}$ iff $I \not\models \neg l_{[\sigma]}$ and $apply(E, I) \models \neg l_{[\sigma]}$. Assume that we have $E \cup I \models \text{new}(\neg l)_{[\sigma]}$. This is the case iff for some derivation rule we have $E \cup I \models \delta l_{[\sigma]}$. By construction of the derivation rules, this is the case iff for some literal lb_i inside the derivation rule body of l we have $E \cup I \models \text{new}(\neg lb_i)_{[\sigma]}$, and for the rest of literals lb_j in the body we have $E \cup I \models lb_{j[\sigma]}$. Thus, we see that for all literals lb in the body of l we have $I \models lb_{[\sigma]}$, thus, $I \models l_{[\sigma]}$, and therefore $I \not\models \neg l_{[\sigma]}$. Moreover, because $E \cup I \models \text{new}(\neg lb_i)_{[\sigma]}$, we see that $apply(E, I) \models \neg lb_{i[\sigma]}$, thus, $apply(E, I) \models \neg l_{[\sigma]}$. The proof the other direction of the iff follows similarly.

Until here we have proved that *new* is a correct New mapping. We now prove that *old* is a correct Old mapping.

We start proving that $E \cup I \models \text{old}(l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$. Assume that we have $E \cup I \models \text{old}(l)_{[\sigma]}$. By definition, this is the case iff $E \cup I \models l_{[\sigma]}$ and $E \cup I \not\models \delta l_{[\sigma]}$. Thus, this is the case iff $I \models l_{[\sigma]}$ and $E \cup I \not\models \text{new}(\neg l)_{[\sigma]}$. Using the correctness of the *new* mapping, this is the case iff $I \models l_{[\sigma]}$ and $(I \models \neg l_{[\sigma]}$ or $apply(E, I) \not\models \neg l_{[\sigma]})$. Equivalently, this is the case iff $I \models l_{[\sigma]}$ and $apply(E, I) \models l_{[\sigma]}$.

We conclude by proving that $E \cup I \models \text{old}(\neg l)_{[\sigma]}$ iff $I \models \neg l_{[\sigma]}$ and $apply(E, I) \models \neg l_{[\sigma]}$. Assume that we have $E \cup I \models \text{old}(\neg l)_{[\sigma]}$. By definition, this is the case iff $E \cup I \models \neg l_{[\sigma]}$ and $E \cup I \not\models \iota l_{[\sigma]}$. Thus, this is the case iff $I \models \neg l_{[\sigma]}$ and $E \cup I \not\models \text{new}(l)_{[\sigma]}$. Using the correctness of the *new* mapping, this is the case iff $I \models \neg l_{[\sigma]}$ and $(I \models l_{[\sigma]}$ or $apply(E, I) \not\models l_{[\sigma]})$. Equivalently, this is the case iff $I \models \neg l_{[\sigma]}$ and $apply(E, I) \models \neg l_{[\sigma]}$. □

Finally, we introduce some properties that will turn out to be crucial when dealing with aggregations.

By construction, ιp derivations rules are disjoint. That is, any instance of ιp that is true in $E \cup I$ is derived by one and only one derivation rule.

Property 10. Insertion rules disjointness

For any two different derivation rules ιp_i ιp_j of the same derived literal ιp, for any data instance I, and structural events E, we have that:

$$I \cup E \models \iota p_i(\overline{X}) \textbf{ implies } I \cup E \not\models \iota p_j(\overline{X})$$

Proof. Consider any two different derivation rules ιp_i and ιp_j for the same predicate ιp. By construction, there is, at least, one literal l s.t. new(l) appears in ιp_i but old(l) appears in ιp_j (or viceversa). This implies that both rules cannot be true at the same time since $E \cup I$ cannot satisfy new(l) and old(l) together. □

Similarly, δp derivations rules are disjoint by construction. That is, any instance of δp that is true in $E \cup I$ is derived by one and only one derivation rule.

Property 11. Deletion rules disjointness

For any two different derivation rules δp_i δp_j of the same derived literal δp, for any data instance I, and structural events E, we have that:

$$I \cup E \models \delta p_i(\overline{X}) \textbf{ implies } I \cup E \not\models \delta p_j(\overline{X})$$

Proof. Consider any two different derivation rules δp_i and δp_j for the same predicate δp. By construction, there is, at least, one literal l s.t. new($\neg l$) appears in ιp_i but old(l) appears in ιp_j (or viceversa). This implies that both rules cannot be true at the same time since $E \cup I$ cannot satisfy new($\neg l$) and old(l) together. □

New/Old Mappings for Aggregation Literals

In case p is an aggregate predicate, then, we define $\iota p/\delta p$ to be derived literals too. In this case, however, we define $\iota p/\delta p$ to represent the increasing/decreasing of the aggregated value. That is, ιp represents the aggregated value of the instances that are inserted in the body of p, and δp represents the aggregated value of the instances that are removed from the body of p. Since we work with distributive aggregations, note that the value of the aggregate value in $apply(E, I)$ is, precisely, the one in I plus the one of the instances being inserted, and minus the aggregated value of those being deleted.

To achieve so, we define $\iota p/\delta p$ for aggregate predicates as we do for normally derived predicates, but adding the aggregating function f in the head of the derivation rule. That is:

Definition 9. Ins. Rules for Aggregate Predicates

Given p an aggregate predicate with aggregation function f whose derivation rule body is $l_1 \wedge ... \wedge l_n \wedge b$, where l_i is a literal and b is a conjunction of built in literals, we define n derivation rules for ιp, where each ιp_i rule has the form:

$$\iota p_i(\overline{x}, f()) \leftarrow \left[\bigwedge_{j=1..i-1} old(l_j) \right] \wedge new(l_i) \wedge \left[\bigwedge_{k=i+1..n} all(l_k) \right] \wedge b$$

Definition 10. Del. Rules for Aggregate Predicates

Given p an aggregate predicate with aggregation function f whose derivation rule body is $l_1 \wedge ... \wedge l_n \wedge b$, where l_i is a literal and b is a conjunction of built in literals, we define n derivation rules for δp_i, where each δp_i rule has the form:

$$\delta p_i(\overline{x}, f()) \leftarrow \left[\bigwedge_{j=1..i-1} old(l_j) \right] \wedge new(\neg l_i) \wedge \left[\bigwedge_{k=i+1..n} l_k \right] \wedge b$$

Now, we give a *New*/*Old* mappings for p in terms of such ιp and δp. To do so, we first introduce some notation to make easier the following definitions.

We use the abbreviation $p(\overline{x})\ \omega\ z$ to refer to $p(\overline{x}, y) \wedge y\ \omega\ z$. Certainly, we know that any aggregate literal $p(\overline{x}, y)$ with aggregation variable y is always accompanied by some built-in literal $y\ \omega\ z$ constraining the value for y. Thus, we simply abbreviate both literals in a unique one. Note that this notation corresponds to use $p(\overline{x}, y)$ as a function rather than a predicate. Indeed, $p(\overline{x}, y)$ behaves as a function since, for each array of constants $\overline{X}$, there is exactly one value Y s.t. $p(\overline{X}, Y)$.

In addition, we use the symbol ω' to refer to the arithmetic comparison that assess the relation that should have two values X_ι, and X_δ, to achieve that $X + X_\iota - X_\delta\ \omega\ Y$ when $X\ \overline{\omega}\ Y$. For instance, if we have $X < 100$, we clearly need that $X_\iota > X_\delta$ to achieve $X + X_\iota - X_\delta \geq 100$, thus, the symbol $(\geq)'$ refers to $>$. It is easy to show that the corresponding arithmetic comparison to $(<)'$, and $(\leq)'$ is $>$; the corresponding arithmetic comparison to $(\neq)'$, and $(=)'$ is $\neq$; and that the corresponding arithmetic comparison to $(>)'$, and $(\geq)'$ is $<$.

Using this notation we can define the *new/old* mappings for some aggregate predicate p as follows:

Definition 11. A New/Old mapping for aggregated literals

Given any aggregated literal with arithmetic comparison $p(\overline{x})\ \omega\ z$, we define the maps:

$$\begin{array}{lcl} \text{new}(p(\overline{x})\ \omega'\ z) & = & \iota p(\overline{x}) \neq \delta p(\overline{x}) \wedge p(\overline{x})\ \overline{\omega}\ z \wedge p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z \\ \text{old}(p(\overline{x})\ \omega\ z) & = & p(\overline{x})\ \omega\ z \wedge p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z \end{array}$$

The idea is that $p(\overline{x})\ \omega\ z$ becomes true because of the structural events if the aggregated value of the instances of p being inserted/deleted contributes to violate the constraint, the current aggregated value does not satisfy the arithmetic comparison, and after adding/subtracting the aggregated value of the instances being inserted/deleted it does. Similarly, $p(\overline{x})\ \omega\ z$ remains true despite the structural events if the arithmetic comparison was satisfied by the old aggregated value, and remains satisfied after adding/subtracting the aggregated value for the instances being inserted/deleted.

In the following, we proof that such *new/old* mappings are correct.

Proof. We prove that the *new* map is a correct *New* mapping. The prove that *old* is a correct *Old* mapping follows analogously.

We want to prove that $E \cup I \models \text{new}(p(\overline{x})\ \omega\ z)_{[\sigma]}$ iff $I \not\models (p(\overline{x})\ \omega\ z)_{[\sigma]}$ and $apply(E, I) \models (p(\overline{x})\ \omega\ z)_{[\sigma]}$. Assume that, $E \cup I \models \text{new}(p(\overline{x})\ \omega\ z)_{[\sigma]}$. According to our map definition, this is the case iff $E \cup I \models (\iota p(\overline{x})\ \omega'\ \delta p(\overline{x}))_{[\sigma]}$ and $E \cup I \models (p(\overline{x})\ \overline{\omega}\ z)_{[\sigma]}$ and $E \cup I \models (p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z)_{[\sigma]}$. Clearly, we see that $E \cup I \models (p(\overline{x})\ \overline{\omega}\ z)_{[\sigma]}$ and $E \cup I \models (p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z)_{[\sigma]}$ implies $E \cup I \models (\iota p(\overline{x})\ \omega'\ \delta p(\overline{x}))_{[\sigma]}$, thus, we can safely omit such condition. So we have, $E \cup I \models \text{new}(p(\overline{x})\ \omega\ z)_{[\sigma]}$ iff $I \not\models (p(\overline{x})\ \omega\ z)_{[\sigma]}$ and $E \cup I \models (p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z)_{[\sigma]}$. Since we limit to distributive aggregations, $apply(E, I) \models (p(\overline{x})\ \omega\ z)_{[\sigma]}$ is equivalent to $E \cup I \models (p(\overline{x}) + \iota p(\overline{x}) - \delta p(\overline{x})\ \omega\ z)_{[\sigma]}$ if $\iota p(\overline{x})$ and $\delta p(\overline{x})$ correctly computes the aggregation value of the instances being inserted/deleted in the body of p. By correctness of the mapping for derived literals, ιp and δp captures any instance being inserted/deleted in the body of p. Moreover, by construction, the rules of ιp and δp are disjoint, thus, any instance being inserted/deleted from the body of p is derived through one and exactly one derivation rule. Thus, aggregating

$\iota p/\delta p$ correctly computes the aggregated value of the instances being inserted/deleted respectively. $\square$

4.2.3 Obtaining EDCs from New/Old Mappings

Until here we have defined two maps from literals l in S to formulas ψ in S' that permits evaluating the truth of l after applying some structural events E. In particular, the *new* map retrieves a formula that evaluates to true if and only if the application of some structural events has made l become true (when it was false), and the *old* map retrieves the formula that evaluates evaluates to true if and only if l was already true, and despite the application of the structural events it remains true.

Now, we want to move this mapping from literals to constraints. That is, we want to map a denial constraint ϕ in signature S into a set of denial constraints Ψ in some augmented signature S' s.t. Ψ is violated iff ϕ is violated after applying the structural events.

This leads to the definition of EDCs:

Definition 12. EDCs

Given a denial ϕ over some signature S, we refer as *the EDCs of* ϕ to the set of logic denials Ψ such that, for any given data state I satisfying ϕ, and structural events E, we have:

$$apply(E, I) \models \phi \textbf{ iff } E \cup I \models \Psi$$

We can build the EDCs by means of the *New*/*Old* map we have built in the previous section. Indeed, assume that our denial ϕ has the form $l_0 \wedge ... \wedge l_n \wedge b \rightarrow \bot$. In this case, to obtain the EDCs Ψ we simply need to replace each literal l of the denial for the formula that permits evaluating l in the data state after applying the structural events, that is, new(l) $\vee$ old(l). Since we suppose that I satisfies ϕ, we know that, if ϕ is violated in I^n, it is because some literal l_i in ϕ was not true in I but becomes true in I^n, that is, because some literal l_i is *new*. More formally, we can obtain the EDCs in the following manner:

Property 12. EDCs can be obtained through the New/Old mappings

Given a denial ϕ with the form $l_1 \wedge ... \wedge l_n \wedge b$, where l_i is a literal and b a conjunction of built-in literals, consider the following n denials ψ:

$$\psi_i = \left[\bigwedge_{j=1..i-1} old(l_j)\right] \wedge new(l_i) \wedge \left[\bigwedge_{k=i+1..n} all(l_k)\right] \wedge b \rightarrow \bot$$

Then, the set $\Psi = \cup_{i=1..n}\{\psi_i\}$ is the set of EDCs of ϕ.

Proof. We start proving that when some EDC ψ_i is violated in $E \cup I$, then, ϕ is violated in $apply(E, I)$ (i.e., *soundness*). Then, we prove than when ϕ is violated in $apply(E, I)$, then, some ψ_i is violated in $E \cup I$ (i.e., *completeness*).

Assume that, for some ψ_i, $E \cup I \not\models \psi_i$. By definition of ψ_i we have $E \cup I \models \left[\bigwedge_{j=1..i-1} old(l_j)\right] \wedge new(l_i) \wedge \left[\bigwedge_{k=i+1..n} all(l_k)\right] \wedge b$. Thus, by correctness of the *old/new* mapping we have: $apply(E, I) \models l_1 \wedge ... \wedge l_n \wedge b$. Hence, $apply(E, I) \not\models \phi$, which concludes the proof for *soundness*.

Assume that, $apply(E, I) \not\models \phi$, but $I \models \phi$. Then, for all literal l_j we have $apply(E, I) \models l_j$ and for some literal l_i we have $I \not\models l_i$. Thus, for all literals l_j we have that $E \cup I \models$ all(l_i), and for some literal l_i, $E \cup I \models$ new(l_i). So, for some i, we have $E \cup I \models \left[\bigwedge_{j=1..i-1} old(l_j)\right] \wedge new(l_i) \wedge \left[\bigwedge_{k=i+1..n} all(l_k)\right] \wedge b$. Hence, $E \cup I \not\models \psi_i$, which concludes the proof for *completeness*. □

By construction, it is clear that each EDC ψ_i always contains some formula *new(l_i)*. This property is the key for incrementallity and, thus, efficiency, when implementing EDCs through SQL.

Indeed, the EDCs guarantees that we only check some constraint when some structural event might cause its violation. This is because the *new(l_i)* is only satisfied when some literal l_i was not true in the initial data state I but becomes true in I^n because of the structural events. Note that checking whether *new(l_i)* is true is *efficient* since it can be assessed by inspecting the structural events alone (e.g. checking if there is a ground substitution σ s.t. $\iota p(\overline{x})_{[\sigma]}$, or $\iota p(\overline{x})\ \omega'\ \delta p(\overline{x})$ in case of aggregates). Thus, if it is false, we know for sure that the EDC is satisfied without the need to inspect the data in I. In this manner, we only need to inspect the data in I if, intuitively, the structural events can cause the violation of the constraint.

Moreover, using the EDCs we only need to inspect the data in I that might rise a violation according to the structural events applied. Indeed, we expect new(l_i) to have some variables in common with other literals in the same EDC (otherwise, the original constraint would speak about totally disconnected objects). Thus, we can check an EDC by means of first, looking for those ground substitutions σ that makes new$(l_i)_{[\sigma]}$ true. These substitutions σ are limiting the instances in I that we need to inspect. Indeed, we do not need to inspect all the instances in I, but only those instances that joins the values appearing in σ, which are, actually, the instances that might violate the constraint according to the structural events applied.

If we implement the EDCs through SQL, all this behavior is achieved through the query planner capabilities of the relational database management system.

Indeed, the query planner is going to first compute the tuples that satisfies the formula *new(l_i)*. This is because, to do so, it only needs to inspect the tables containing the structural events, and the cardinality of these tables are much lower than the cardinality of the tables containing the actual data.

Moreover, at this point, if the computed set of tuples is the empty set, the query execution stops since any join with the empty set trivially returns the empty set. Thus, the query engine do not look to the actual data if there is no structural events that might potentially violate the constraint.

In addition, once the query engine has computed the set of tuples satisfying *new(l_i)*, the query engine is going to use such tuples to compute the join with the rest of tuples from the database that might violate the constraint. However, note that this is a join of tuples rather than a complete full scan, that is, the inspection of the actual data is limited to the data joining the structural event tuples. Finally, note that we can always enhance the performance of such joins by means of incorporating indexes.

4.3 SQL Implementation Experiments

As we have already discussed, EDCs can be implemented through SQL queries in case we have an SQL translation of the UML schema, and some tables to materialize the structural events we want to apply. Intuitively, each EDC can be translated as an SQL query that looks for the instances that violates the constraint according to the contents in the structural event tables.

When limited to base/derived literals (without existential variables), the implementation of the EDCs is based on translating positive literals as joins, and negative literals as anti-joins. When including aggregation literals we need to include subqueries to compute the aggregation variables. In order to avoid a hard penalization for computing such aggregation variables, we propose to materialize the current aggregated value, and maintain such value according to the queries that computes its increasing/decreasing.

To show the feasibility of this approach, we have conducted an illustrative exper-

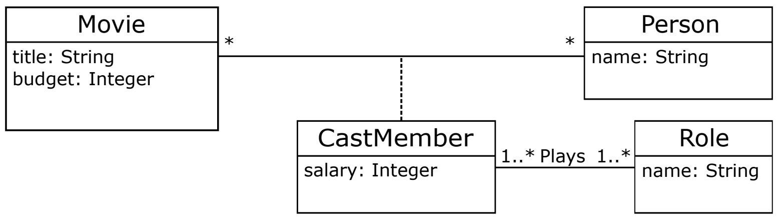


Figure 4.1: Simplified UML class diagram for the IMDb Information System

iment. To do so, we have first looked for some UML/OCL schema that could be populated with real data, thus, enhancing the reliability of our results. In particular, we have built a simple UML/OCL schema of the IMDb database, an information system of movies and cast members, and we have populated it with the real data available in its public interface.

In the following, we first explain the UML/OCL schema used and its initial data state, then, we discuss the experiment design, and finally, we discuss the results.

4.3.1 Experimenting UML/OCL Schema

Consider the UML schema in Figure 4.1. This schema specifies an information system storing data about *movies* and the *people* participating in them as *cast members*, where each cast member exercises a *role* (e.g. director, actor, actress, etc.).

In this schema, we have defined the integrity constraints shown in Figure 4.2. The first one states, there should not be any cast member playing the role of *actor* and *actress* at the same time; the second forces the sum of any movie budget to be higher than the salaries of its cast members; the last one, ensures that there is one cast member playing the role of *director* for each movie. Note that aggregate operations such as *sum* or *count* are used in the constraint definitions.

context *CastMember* **inv** *NotActorAndActress*:
self.role.name->excludes('actor') or self.role.name->excludes('actress')
context *Movie* **inv** *BudgetIsHigher*:
self.budget >= self.castMember.salary->sum()
context *Movie* **inv** *HasSomeDirector*:
self.castMember.role.name->count('director') > 0

Figure 4.2: OCL Constraints for the simplified UML class diagram of IMDb

In this schema, we have loaded the IMDb public interface available data (about 13*10^4 movies and 2.5*10^6 cast members).

4.3.2 Experiment Design

The goal of this experiments is to show the benefits of our approach. In order to be able to compare the performance of our technique with other proposals, we have decided to compare our results with those obtained with Dresden OCL [8]. Dresden OCL is a tool that translates an OCL constraint into an SQL query that retrieves the violating instances of the constraint (thus, in a similar way as we do). However, and different from us, Dresden OCL has no incremental capabilities. Thus, we expect to see, via this experiment, the benefits of the incremental approach we propose.

The experiment consisted in measuring the execution time of our method to check the OCL constraints of Figure 4.2 in three different scenarios: adding new movies, modifying salaries of cast members, and deleting movie directors. All these experiments have been conducted in MySQL 5.6 running on Windows 8 in an Intel i7-4710HQ up to 3.5GHz machine with 8GB of RAM.

For each scenario, we have executed our method several times increasing the number of structural events applied in each case. Note that inserting a movie requires several structural events: inserting the movie, its budget, its cast members, etc.; updating a salary requires two events: deleting the old salary and inserting the new one; and deleting a director from a movie requires three: deleting its cast membership, its role and its salary.

4.3.3 Experiment Results

In Table 4.1 we show the execution times in seconds for checking each constraint of the example, using our technique, in function of the number of structural events applied to each scenario.

From these results we can see that the time to check any constraint increases with the number of movie insertions. This is because all three constraints can be violated in this scenario. Insertions of 1000 movie had better response times than those of inserting 500 due to the cache memories of MySQL. When analyzing salary updates, we see that only the constraint *BudgetIsHigher* gets worse results when increasing the number of events considered, while the other two remain almost constant. This is because it is impossible to violate them when updating salaries. The same phenomena occurs with the constraint *NotActorAndActress* in the third scenario since it is impossible to violate it by deleting cast members.

It is worth noting that most of the experiments took less than one second and that only one of them was over 30s. Moreover, the cache memories improved the results of the last experiments with the largest number of structural events. In the case with most number of data changes (22,037 structural events), it took 12.37s to check one constraint in a database with more than 3 million rows.

We also show in Table 4.2 the execution time in seconds required to update the materialized aggregates for each scenario once the constraint check has been performed. Note that updating the materialized aggregates does not suppose any scalability problem since none of them takes more than 0.5 seconds.

Regarding the results obtained by Dresden OCL, the execution time to check *NotActorAndActress* was 21.47s, while checking *HasSomeDirector* and *BudgetIsHigher* did not finish within two hours. We could improve these last execution times after manually rewriting the automatic translation provided by the tool, but their results were still hight: 238.33s and 79.44s. Note that, since this method is not incremental, its execution time is independent of the events applied, thus, it takes these times even when the events applied cannot violate any of the constraints.

Table 4.1: Time in seconds to check constraints

#Movie Insertions	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	28	149	272	1254	2059	10876	22037
NotActorAndActress	0.36	0.75	5.31	3.51	5.54	9.39	9.13
HasSomeDirector	0.28	0.08	0.33	0.41	0.56	1.42	0.38
BudgetIsHigher	0.41	0.90	5.37	5.54	9.82	30.34	12.37
#Salary Updates	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	2	10	20	100	200	1000	2000
NotActorAndActress	0.14	0.05	0.05	0.03	0.05	0.03	0.06
HasSomeDirector	0.31	0.00	0.00	0.02	0.02	0.00	0.00
BudgetIsHigher	0.17	0.09	0.30	0.69	1.16	1.00	1.44
#Director Deletions	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	3	15	30	150	300	1500	3000
NotActorAndActress	0.19	0.20	0.17	0.13	0.16	0.70	0.12
HasSomeDirector	0.45	0.53	0.95	1.79	2.07	13.09	3.93
BudgetIsHigher	0.30	0.47	0.37	0.44	2.38	0.60	0.48

Table 4.2: Time in seconds to update the materialized aggregates

	1	**5**	**10**	**50**	**100**	**500**	**1000**
Movie Insertions	0.05	0.14	0.12	0.15	0.11	0.48	0.35
Salary Updates	0.08	0.08	0.12	0.08	0.06	0.11	0.10
Director Deletions	0.05	0.05	0.08	0.06	0.42	0.06	0.14

4.4 TINTIN: A Tool for INcremental INTegrity Checking

TINTIN is a tool designed to give support to standard SQL assertions by means of the EDCs technique already exposed. Indeed, we can encode SQL assertions as denials, and thus, apply all the previous exposed technique to incrementally check SQL assertions.

Basically, a database user can provide TINTIN with a database and a set of SQL assertions, and TINTIN builds in the database all the necessary queries and procedures to incrementally check the given assertions. The unique requirement for the user is to call the automatically created procedure *safeCommit()* at the end of its transactions.

When invoking *safeCommit()*, the automatically generated code checks if the tuple insertions/deletions specified since the last call to *safeCommit* violates or not the provided SQL assertions. If such insertions/deletions cause some assertion violation, the insertions/deletions are rejected and the user receives a message with the conflicting assertion. Otherwise, the insertions/deletions are committed into the database.

In the following, we first briefly review the problem of implementing SQL assertions, then, we describe the TINTIN under the perspective of a user, that is, its input and output. Finally, we present the tool architecture.

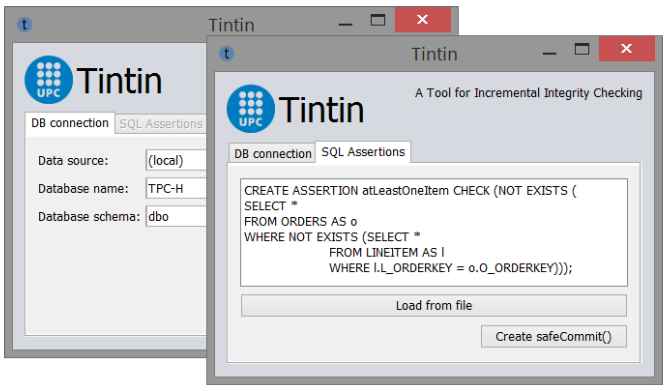
4.4.1 The Problem: Implementing SQL Assertions

In standard SQL, users can specify general constraints using the CREATE ASSERTION statement. The basic technique for writing assertions is to specify a query that selects those tuples that violate the desired condition. By including this query inside a NOT EXISTS clause, the assertion will specify that the query result must be empty. Thus, the assertion is violated if and only if the query result is not empty [44].

Assertions were initially defined in SQL-92 [7] and they serve as a means for expressing global integrity constraints not tied to a particular table, but ranging over several ones. They are sufficient for expressing most constraints since almost the full expressiveness of SQL can be used to define the query inside the NOT EXISTS clause. It is also well known that many integrity constraints can only be expressed via assertions since the other constructs provided by SQL are not powerful enough. Thus, assertions provide an elegant way to define general constraints in SQL.

However, assertions are still not supported by any of the most well-used commercial RDBMS (Oracle, MySQL, SQL Server, PostgreSQL, DB2). It might be argued that assertions can be emulated via manually writing a set of triggers, which is a widely supported feature of RDBMS. However, its manual definition is error prone and the whole set of necessary triggers to write might not be evident when given a complex constraint, thus, compromising the integrity of the data if just one trigger is missing or ill-defined. Hence, it is better to delegate this complex checking code to RDBMS capabilities [110], as we do in TINTIN[1].

[1]http://www.essi.upc.edu/~xoriol/tintin/



Data source: (local)
Database name: TPC-H
Database schema: dbo

```
CREATE ASSERTION atLeastOneItem CHECK (NOT EXISTS (
SELECT *
FROM ORDERS AS o
WHERE NOT EXISTS (SELECT *
                FROM LINEITEM AS l
                WHERE l.L_ORDERKEY = o.O_ORDERKEY)));
```

Figure 4.3: TINTIN Graphical User Interface

4.4.2 TINTIN: A User's Perspective

TINTIN is a Java standalone tool that provides incremental integrity checking capabilities of assertions in a SQL Server database specified by the user.

In particular, a user must provide to TINTIN with a DB connection (that is, a connection string, a database name and a database schema), and a set of SQL assertions. In Figure 4.3 we show the graphical user interface provided to the user to bring such input.

TINTIN accepts assertions to be defined through selection, projection, join, subselect (exists, in), negation (not exists, not in) and union. Future releases will allow functions (e.g. aggregates, arithmetic functions).

As a result, TINTIN creates all the necessary triggers and procedures to transparently capture all the insertions/deletions specified by the user. That is, if a user applies a INSERT INTO or DELETE FROM statement, such insertions/deletions are not committed to the database, but are captured and stored in some auxiliary tables storing the tuples that are being inserted/deleted from the database.

In addition, TINTIN builds a procedure called *safeCommit* that is responsible to commit such captured insertions/deletion only if they do not cause any constraint violation. That is, users need to invoke such automatically created procedure to try to commit their insertions/deletions. When doing so, *safeCommit* executes several queries that corresponds to the EDCs of the different specified SQL assertions. If the queries returns the empty set, this means that the captured insertions/deletions do not cause any constraint violation, and thus, are committed by the procedure. If some query returns different from the empty set, such query result is shown to the user together with the name of the SQL assertion corresponding to such query. In any case, after invoking *safeCommit*, the auxiliary tables storing the captured insertions/deletions are emptied.

Note that TINTIN is only necessary in *compilation time*. In other words, TINTIN

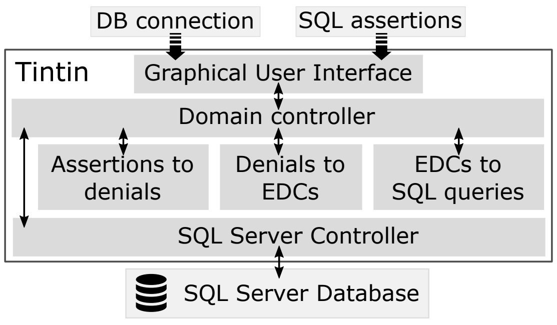


Figure 4.4: TINTIN architecture

is only used to *install* in the database the necessary triggers and procedures to check some SQL assertions. However, once this step is done, the user can normally operate with the database with its favorite database client with the unique condition to invoke *safeCommit* at the end of his/her transactions, and no more need of the TINTIN tool is required. In this manner, TINTIN offers a decoupled, transparent usage to the database users.

4.4.3 TINTIN: Tool Architecture

The architecture of TINTIN is shown in Figure 4.4.

When the user introduces the DB connection, the SQL Server Controller creates a new auxiliary database event_DB to store the different events applied to it; that is, for each table T in DB, the SQL Server Controller builds two new tables (*ins_T* and *del_T*) to store the different tuples being inserted and deleted in T. In order to capture these tuples, the SQL Server Controller creates two different INSTEAD OF triggers, which capture the tuple insertions/deletions and place them in the corresponding *ins_T* or *del_T* table.

Afterwards, when the user introduces the SQL assertions, they are firstly encoded as logic denials reusing a component already implemented in . Then, these denials are translated into EDCs and, finally, the EDCs are implemented as SQL queries. Each of these steps is implemented in a different module following the previously presented method.

The resulting SQL queries are stored as views in event_DB. Then, the SQL Server Controller builds the *safeCommit* procedure. This procedure, when called, performs the following:

1. Queries the previous views.
2. If all queries are empty, it disables the triggers capturing the insertions/deletions, applies the update (insert in the DB the tuples contained in the *ins* tables, and

remove from the DB the tuples contained in the *del* tables), and enables again the triggers.

3. Truncates the *ins*/*del* tables.

The prototype has been developed in Java, with the exception of the *Assertions to denials* translator component, for which we have reused a previously existing C# software.

4.5 Related Work

We review the related work found in the literature in three main blocks:

- *OCL evaluators*: approaches relying on OCL interpreters.
- *OCL to database techniques*: approaches based on checking OCL constraints through DB mechanisms such as SQL queries/procedures.
- *OCL to graph patterns*: approaches based on checking OCL constrains through graph pattern queries.

As we are going to see, although there are some approaches for making *OCL evaluators* follow an incremental checking policy, their techniques are not fully incremental. That is, they repeat some evaluations that, because of the nature of the update applied, cannot cause the violation of any constraint. Moreover, these approaches intrinsically suffer from the lack of OCL interpreter support tools.

The approaches based on translating OCL to DB technologies appeared to, essentially, make database servers capable of evaluating OCL constraints. Unfortunately, none of the presented translations permits an incremental evaluation of the OCL constraints, which drastically penalizes their execution times.

Finally, the OCL to graph patterns approach tries to evaluate OCL constraints incrementally using graph database technology. In particular, they propose the usage of the RETE algorithm [53] to perform this incremental evaluation. Unfortunately, the RETE algorithm basically consists in materializing all the relational algebra intermediate results necessary to evaluate an OCL constraint, which involves a huge memory cost to store these auxiliary data, and the time penalty necessary to maintain it.

In contrast, our proposal permits incrementally evaluating OCL constraints with relational databases, which are still one of the most well-known and wide spread database technologies. Moreover, our approach based on materializing aggregates (and existential variables), permits benefiting from the good incremental checking without incurring in a prohibitive maintainment cost of such materialized information.

In the following, we review all these approaches by blocks.

4.5.1 OCL Evaluators

Several OCL evaluator tools can be found in the literature. Some prominent examples are USE [58], the Dresden OCL Toolkit [8], the Eclipse Modelling Framework OCL evaluator [61], or mOdCL [105]. Anyway, although these tools are capable of checking whether a data state satisfies a given set of OCL constraints, none of them implements an incremental approach for doing so. Thus, whenever a change in the data state occurs, they must check again all the constraints over the whole new data state. In order to solve this inconvenience, several approaches for incremental OCL evaluation have been suggested.

In this line, the work in [60] is based on, for each OCL constraint, mapping the different context elements of the OCL constraint to the related data required to perform such evaluation. Thus, when any of these related data is modified, the corresponding OCL constraint is reevaluated for such context element. This approach, although intuitive, makes an intensive usage of memory which has already been criticized [48].

Differently, the work presented in [111] is based on, given a data update, compute the context elements from which we should reevaluate the whole OCL constraint. Intuitively, the method is based on tracing back the source of the OCL operations affected by the update. Note that, in this case, no auxiliary data structures are required.

However, both techniques are thought to reevaluate an OCL constraint whenever its truth/false evaluation changes. That is, they do not only reevaluate an OCL constraint when it could be violated, but also when some update might repair the constraint. For instance, in our example, they do not only evaluate the *SeenIsBought* constraint when we apply insertions of *visualizes* (which might make the OCL constraint evaluate to false, and thus, raise a violation), but also when applying insertions of *bought* (which might make the OCL constraint evaluate to true, and thus, repair the violation). Therefor, these approaches are not optimal for incremental integrity checking as we have defined, although they would be for other problems such as incremental OCL query evaluation.

Another approach specifically targeted to only evaluate an OCL constraint when it can be violated is the one explained in [18]. This approach computes: (1) which OCL constraints should be evaluated because of some data update, (2) for which context element, and (3) it rewrites the OCL constraint to change the context element if doing so improves its evaluation.

Unfortunately, all the presented approaches seen so far share the same weakness: they can only determine one object for which to reevaluate the whole OCL constraint, but not the set of its related objects for which to perform the evaluation. For instance, in our example, if we add the fact that *John sees 'Some like it hot'*, clearly, we should check if *John has bought 'Some like it hot'* in order to incrementally check whether the *SeenIsBought* constraint has been violated. However, all the previous approaches reevaluate the *SeenIsBought* constraint for the context element *John*. That is, they check the constraint for all the contents seen by *John*, instead of only checking so for the new one *'Some like it hot'*.

4.5.2 OCL to Database Techniques

In order to bring support to OCL, the OCL community has built several tools that translate OCL to other languages that can be executed. In this section, we discuss the tools aimed at translating OCL to SQL/NoSQL languages.

The OCL2SQL is a component inside the Dresden OCL Toolkit that translates OCL constraints into SQL queries [8]. Similarly to our approach, the Dresden OCL returns a query such that returns the values that violate the input OCL constraint. That is, the query evaluates to the empty set if and only if the OCL constraint is satisfied. However, we have not been able to find the formalization of the Dresden OCL to SQL translation, which makes difficult to assess its mathematical correctness and also the expressiveness of the OCL constraints they can deal with.

In contrast, the MySQL4OCL tool is defined through a formal translation from OCL to MySQL queries/procedures [41]. In this case, the tool is intended to build some MySQL code that returns exactly the same values returned by the input OCL expressions. That is, for a given OCL constraint, its generated code returns true if and only if the constraint is satisfied. Differently from ours, this approach deals with a three-valued logic. That is, the code for some OCL boolean expression might return true, false, or null. Regarding its expressiveness, the OCL constraints they can deal with are the same as ours with the exception that they can also deal with OCL Bags.

However, none of the previously discussed tools behaves incrementally, which makes them unsuitable for efficient integrity checking. Following the same lines, the work in [34] presents a translator from OCL to a NoSQL query language (Gremlin), and [114] presents a translation from an extended OCL (UnifiedOCL) to several languages (including SQL), again without incremental checking capabilities.

4.5.3 OCL to Graph Patterns

The work of [11] consists in translating OCL constraints into graph patterns to benefit from graph-pattern incremental queries. Such method uses the RETE algorithm to achieve the incremental behavior, which, intuitively, materializes every relational algebra operation performed by the queries [53]. Such materialization has already been criticized because of its huge memory usage and the penalization time required to maintain such materialization.

To solve such issues, other algorithms have been suggested to replace RETE, such as TREAT [80]. Intuitively, TREAT is based on storing only the final query result (and not the intermediate results), and incrementally update this result according to the changes applied in the data state. However, this algorithm suffers from the existential variables problem we have discussed in Sections 4.1.3. That is, given a constraint involving an existential variable, whenever we delete a witness of such existential variable, the TREAT algorithm requires inspecting essentially the whole data set to look for another witness for the existential variable.

Thus, it seems that there is a trade-off between materializing/no-materializing intermediate information. In this thesis, we defend a balanced approach consisting in materializing uniquely the number of witnesses each existential variable has. Indeed, in this manner we keep the advantages found in both algorithms. That is, we avoid the existential variables problem (like in RETE), without having to maintain a materialized version of every intermediate result (like in TREAT).

4.6 Conclusions

In this chapter, we have presented a method for incrementally checking UML/OCL constraints. This method is based on what we call *event dependency constraints* (EDCs), which are some rules that state all the possible ways that some structural events can cause a constraint violation according to the data. To build the EDCs, we have augmented the signature of the logic schema by introducing two new predicates (ιp and δp) for each predicate p to represent the insertions and deletions of p instances, and rewrite the original constraints in terms of these new predicates.

Using this technique we can deal with OCL_{FO} constraints extended with OCL distributive aggregation (i.e., *sum*, *count*, *size*). To deal with these aggregates, we have proposed to materialize the current materialized value of such aggregates into the data, and to incrementally maintain such materialization. In this way, we skip iterating through all the current data to compute the aggregate value to check the constraint, which would be prohibitive. Moreover, we have also proposed to use such aggregation technique to deal with the existential variables. That is, we have proposed to aggregate the number of *witnesses* that each existential variable has, so that, in case some witness is deleted, we do not need to inspect all the data to look for another witness.

We have mathematically proved the correctness of this method, and experimentally shown its benefits. In particular, we have seen that some constraints whose non-incremental evaluation using automatic tools could not finish within hours (but whose execution took minutes after manual optimization) reduced its execution time to seconds with this automatic incremental technique. Such experiments were performed using real data coming from the IMDb system consisting of 3 million rows, and considering sets of 22,000 structural events applied at the same time.

Given these promising results, we have developed the TINTIN tool, a tool for incrementally checking SQL assertions. With this tool, a user can specify which SQL assertions does he/she want to check in some relational database, and TINTIN builds all the necessary triggers/procedures to do so. Then, a user only needs to invoke the automatically generated *safeCommit* procedure at the end of his/her transactions. After invoking *safeCommit*, the structural events applied in the transaction are committed unless an SQL assertion violation is detected. In such case, the events are rejected and a message to the user with the conflicting data and violated SQL assertion is shown.

As future work, we would like to extend the current method to deal with even more expressive constraints. In particular, right now it only works with a two-valuated logic, thus, it would be interesting to deal also with *null* values as it happens with standard OCL and SQL. Another future work would be to consider 1st order constraints with *least fixed points*, as in the case of *datalog*.

Chapter 5

Incremental Maintenance of UML/OCL Constraints

Our method for incrementally maintaining UML/OCL constraints is based on modifying the previously described EDCs to obtain a new set of logic rules that we call Repair-Generating Dependencies (RGDs). Roughly speaking, the RGDs do not only capture the different ways some structural events can cause the violation of a constraint, but also determine which structural events can repair it. Then, such RGDs can be chased to find the repairs R i.e., the missing structural events that, when applied together with the initial structural events, lead the current data state to a new consistent state.

Formally, our method is a variation of an already existing proposal for integrity maintenance based on the *event rules* [109]. In this thesis, however, we vary such proposal to reduce the process of computing R to a chase of RGD logic rules. Moreover, we permit a domain expert to customize these RGDs to control the generated solutions, and thus, avoid maintaining some constraints in undesired manners.

The constraints we can maintain with this method are pure OCL_{FO} constraints. That is, we do not consider constraints involving aggregations. This is because, maintaining OCL_{FO} constraints is already non decidable, which means that chasing the previous RGDs might never terminate. To deal with the non-decidability problem, we are going to define a subset of OCL_{FO} where such problem is, actually, decidable, and reuse some decidability results already present in the literature which we redefine under the name of *Finite Canonical Property*.

In the following, we start with an intuitive approach on the RGDs method. Then, we formalize the method, prove its soundness and completeness, and show its customization possibilities. Afterwards, we show some experiment results based on a C# implementation of the method. Next, we present IDEFIX, a tool for identifying missing structural event in OCL operation contracts based on this technique. Next, we discuss the related work. Finally, we present some future work and conclusions.

5.1 An Intuitive Approach to RGDs

In this section, we first exemplify how to obtain and chase the RGDs of some constraints distinguishing whether the original constraint has (or not) existential variables. Afterwards, we show how to customize RGDs to control the solutions found by the chase.

5.1.1 RGDs without Existential Variables

In the previous chapter we have seen that EDCs point out the situations where an integrity constraint is violated as a consequence of the application of a set of structural events. However, EDCs do not directly provide any information on the repairs for that violated constraint. We transform EDCs into RGDs for this purpose.

Obtaining RGDs

We obtain the RGDs by removing the negated structural events of the EDCs, and placing them positively in the right-hand side of the rule. Indeed, each negated structural event in the premise of an EDC is a structural event that, if applied, avoids the violation of its corresponding constraint. Thus, these structural events repair the constraint in case of violation. If there is more than one negated structural event in the EDC, then there is more than one possible repair. Thus, all these events are all placed in the conclusion of the RGD as a disjunction.

For instance, the EDCs corresponding to the *SeenIsBought* constraint (4.3, 4.4, 4.5, and 4.6) give rise to the following RGDs:

$$\iota visualizes(u,c) \wedge \delta buys(u,c) \rightarrow \bot \tag{5.1}$$

$$\iota visualizes(u,c) \wedge \neg buys(u,c) \rightarrow \iota buys(u,c) \tag{5.2}$$

$$visualizes(u,c) \wedge \delta buys(u,c) \rightarrow \delta visualizes(u,c) \tag{5.3}$$

$$visualizes(u,c) \wedge \neg buys(u,c) \rightarrow \delta visualizes(u,c) \vee \iota buys(u,c) \tag{5.4}$$

Note that we obtain exactly one RGD for each EDC.

RGD 5.1 states that if we insert that u visualizes c whereas we delete, at the same time, that u buys c, then, there is a constraint violation that cannot be repaired. Indeed, there is no structural event that can be applied to compensate such constraint violation.

In a different way, RGD 5.2 states that, if we insert that u visualizes c when u has not bought c, then, there is a constraint violation unless we also insert that u buys c. Similarly, RGD 5.3 states that, if we delete that u buys c when u has not visualized c, then, we should delete that u has visualized c.

Finally, RGD 5.4 states that, if u has visualized some content that he has not bought, then, we should either delete such visualization or add that u buys such content to avoid a constraint violation. Again, this RGD corresponds to the case in which the current data state I violates the constraint, and thus, using the assumption that the current data state satisfies all the constraints, we can safely remove such RGD without compromising the completeness of the method.

Chasing RGDs

Suppose that we have some set of RGDs of some constraints, the initial data state I, and some initial structural events E. Now we want to compute those sets of extra structural events R s.t. when applying $E \cup R$ into I, leads to a new data state satisfying all the constraints.

To compute these repairs R, we are going to *chase* the RGDs. Chasing the RGDs consists into, first, set R to be the empty set. Then, we look which instances from $I \cup E \cup R$ violate some RGD, and repair such violation by adding into R the structural events specified in the right-hand side of the RGD.

For instance, consider the previous given RGDs for the constraint *SeenIsBought*, and the following initial data state and structural events:

$$I = \{userAge(Luke, 14), contentAge(\text{“}The\ Shining\text{”}, 18)\}$$
$$E = \{\iota visualizes(Luke, \text{“}The\ Shining\text{”})\}$$

Clearly, $I \cup E$ violates the RGD 5.2, thus, to repair such violation, we need to add in R the instance $\iota buys(Luke, \text{“}The\ Shining\text{”})\}$ specified in the RHS of the RGD.

However, when adding such new instances, new RGDs might be violated. Indeed, consider the following RGD corresponding to the *BoughtContentIsAppropiate* constraint:

$$\iota buys(u, c) \wedge contentAge(c, a) \wedge userAge(u, a_2) \wedge a > a_2 \rightarrow \delta contentAge(c, a) \vee \delta userAge(u, a_2)$$

After adding the instance $\iota buys(Luke, \text{“}The\ Shining\text{”})\}$, we need to add in R either $\delta contentAge(c, a)$ or $\delta userAge(u, a_2)$ to avoid the violation of *BoughtContentIsAppropiate* constraint. These two different possibilities will lead to the different ways to maintain some constraint.

Note that chasing is an iterative process. Indeed, every time we repair some RGD, we might violate some other RGD that might require being repaired. Moreover, since some violations might have several repairs, the *chase* execution follows a tree structure where each branch represents a different repair selection for each violation. Note also that some violations cannot be repaired (such as RGD 5.1). When these RGDs are violated, no structural event can be applied to repair them, which means that no solution exists in such branch. In this manner, depending on the number of solutions found in the different branches, the chase might found 0, 1, or several sets R of missing structural events.

5.1.2 RGDs with Existential Variables

We see now how to obtain the RGDs for existential variables.

In summary, this encompasses to first building the EDCs for the constraint involving existential variables, and then, applying again the process of moving the negated structural events from the LHS of the EDC to its RHS. Remember that, in the previous chapter, we have not built the EDCs involving existential variables. This is because, whenever we found an existential variable, we used the technique of aggregating it for improving the performance of integrity checking. However, we cannot longer use such technique since we restrict ourselves to OCL$_{\text{FO}}$ constraints, which do not contain this kind of aggregation capabilities.

In the following we first exemplify how to obtain the EDCs for denials with existential variables, and then, how to obtain the RGDs from them. Next, we show how to chase RGDs with existential variables.

Obtaining EDCs with existential variables

Consider the following denial encoding the minimum cardinality constraint stating that each *Series* have at least one *Episode* e:

$$\begin{aligned} &series(s) \land \neg hasEpisode(s) \rightarrow \bot \\ &hasEpisode(s) \leftarrow episodeSeries(e, s) \end{aligned} \tag{5.5}$$

Stating the EDCs for such constraint encompasses a careful treatment of the existential variable e. This is because, whenever we delete some episode e for some series s, this does not mean that $hasEpisode(s)$ becomes necessarily false in the new data state I^n. Indeed, it might also happen that we insert, at the same time, some new episode for such series, or it might also happen that there is another *old* episode for such series which is not being removed.

Thus, for such constraint we define the following EDCs:

$$\iota series(s) \land \delta hasEpisode(s) \land \neg \iota hasEpisode(s) \land \neg hasEpisode^o(s) \rightarrow \bot \tag{5.6}$$

$$\iota series(s) \land \neg hasEpisode(s) \land \neg \iota hasEpisode(s) \rightarrow \bot \tag{5.7}$$

$$series(s) \land \neg \delta series(s) \land \delta hasEpisode(s) \land \neg \iota hasEpisode(s) \land \neg hasEpisode^o(s) \rightarrow \bot \tag{5.8}$$

$$series(s) \land \neg \delta series(s) \land \neg hasEpisode(s) \land \neg \iota hasEpisode(s) \rightarrow \bot \tag{5.9}$$

Intuitively, EDC 5.6 tells us that there is a constraint violation if we insert a new series s whereas, at the same time, the following 3 things occur: (1) we delete some episode in s, (2) we do not insert any new episode for s, and (3) there is no other old episode for s not being removed. So, the idea is to ensure each of these conditions by means of the derived literals $\delta hasEpisode(s)$, $\neg\iota hasEpisode(s)$, and $\neg hasEpisode^o(s)$, respectively. Bearing this in mind, we define the derivation rules of such literals as follows:

$$\iota hasEpisode(s) \leftarrow \iota episodeSeries(e, s) \tag{5.10}$$

$$\delta hasEpisode(s) \leftarrow \delta episodeSeries(e, s) \tag{5.11}$$

$$hasEpisode^o(s) \leftarrow episodeSeries(e, s) \wedge \neg\delta episodeSeries(e, s) \tag{5.12}$$

The EDCs 5.7 and 5.8 define two other possible ways to violate the constraint. In particular, the former states that there is a constraint violation if we insert a new series s which has no episodes and for which we do not insert new episodes; while the latter raises a violation if we delete some episode of some series s that is not being deleted, where s has no other episode neither we insert some other episode for it.

EDC 5.9 represents, again, that some violation remains in the new data state I^n if it existed in the initial data state I, and no structural event is applied to avoid it. Thus, assuming that the initial data state satisfies all the constraints, we can also safely remove this last EDC.

Obtaining the RGDs

The idea to obtain RGDs for EDCs with existential variables is to move the negated literals defined with events (that is, negated structural event literals, or negated derived literals defined over event literals) from the LHS to the RHS. However, in this case, some of these literals are *derived* literals instead of *base*, so, we need to unfold them after placing them in the RHS of the rule.

For instance, consider the EDC 5.7 which contains the $\neg\iota hasEpisode(s)$ event. After moving such literal into the RHS we obtain:

$$\iota series(s) \wedge \neg hasEpisode(s) \rightarrow \iota hasEpisode(s) \tag{5.13}$$

Since RGDs require all the literals in the conclusion to be base, we need to unfold the literal $\iota hasEpisode(s)$. This is a necessary condition to ensure that chasing the RGDs computes repairs that contain only structural events and no derived events. So, after unfolding we obtain:

$$\iota series(s) \wedge \neg hasEpisode(s) \rightarrow \iota episodeSeries(e, s) \tag{5.14}$$

Moreover, in some cases we might need to apply a postprocess to explicitly forbid the chase to apply some structural events. For instance, assume that we delete some episode e from some series s, then, we have 3 possible ways to avoid violating its minimum cardinality constraint: (1) deleting s, (2) inserting a new episode e' in s, (3) in case s has already some episode e', forbid removing such episode.

This is, actually, the case of the EDC 5.8. Certainly, after moving to its RHS the negated event literals and the negated derived literals containing events in its derivation rule, we obtain:

$$series(s) \wedge \delta hasEpisode(s) \rightarrow \delta series(s) \vee \iota hasEpisode(s) \vee hasEpisode^{o}(s)$$

And then, after unfolding the derived literals, we have the following rule:

$$\begin{aligned} series(s) \wedge \delta hasEpisode(s) \rightarrow & \delta series(s) \vee \\ & \iota episodeSeries(e, s) \vee \\ & (episodeSeries(e', s) \wedge \neg \delta episodeSeries(e', s)) \end{aligned}$$

The first two disjuncts represents, respectively, repairing the constraint by means of deleting the series, and repairing the constraint by inserting a new episode. The third disjunct represents the possibility that the series has already some other episode avoiding the violation, and we do not delete it.

The *chase* should remember not to delete $\delta episodeSeries(e', s)$ only if it decides to avoid such constraint violation by means of preserving this episode e'. To do so, it should first ensure that such episode e' exists, and then, ensure that it is never deleted. We do so by introducing two new literal types, *ιcontain* and *ιforbid*, replacing the previous ones in the following way:

$$\begin{aligned} series(s) \wedge \delta hasEpisode(s) \rightarrow & \delta series(s) \vee \\ & \iota episodeSeries(e, s) \vee \\ & \iota containEpisodeSeries(e', s) \wedge \iota forbidDeltaEpisodeSeries(e', s) \end{aligned} \tag{5.15}$$

now we define two new rules to ensure that if $\iota containEpisodeSeries(e', s)$ is true, then there is some episode e' for the series s, and that if $\iota forbid\delta episodeSeries(e', s)$ is true, then, we do not delete e' from s:

$$\iota containEpisodeSeries(e', s) \wedge \neg episodeSeries(e', s) \rightarrow \bot \tag{5.16}$$

$$\iota forbidDeltaEpisodeSeries(e', s) \wedge \delta episodeSeries(e', s) \rightarrow \bot \tag{5.17}$$

The first rule needs no longer transformation, but the second one needs to reapply recursively all the process for obtaining RGDs in case $\delta episodeSeries(e', s)$ is derived (i.e., unfold such literal, move negated events appearing in it to the RHS, etc). This recursion is guaranteed to terminate since we only deal with non-recursive literals.

Chasing RGDs

Chasing RGDs with existential variables encompasses the difficulty of inventing new values for them. Indeed, when the LHS of some RGD holds for some ground substitution σ, we need to include in R the literals of its RHS after replacing its variables according to the ground substitution σ. However, σ does not give a value for the existential variables in the RHS of the RGD. Thus, the chase needs to invent some values for them.

For instance, consider the data state $I = \{content(\textit{PilotEpisode})\}$, and the initial set of the structural events $E = \{\iota series(\textit{Modern Family})\}$. In this situation, the RGD 5.14 says that we need to include in R some literal of the form $\iota episodeSeries(e, \textit{Modern Family})$, but it does not specify which value should take the existential variable e.

Our strategy to give a value to the existential variables is to consider, for each existential variable, all the values currently present in $I \cup E \cup R$ together with some new fresh value. Each of these values leads to a different possible repair R. Such strategy is based on the *Variable Instantiation Patterns* [50], which are grounded on the concept of canonical databases [112]. A more difficult treatment of the existential variables with the presence of $<$, $\leq$ comparisons is explained in [50].

In the previous example, we would consider to give e the value *PilotEpisode* (since it is already present in I), together some new fresh value such as *Episode 1*. Both possibilities lead to two different sets R of repairing structural events.

In this case, note that *chasing* RGDs with existential variables might never terminate. Indeed, it is possible that, to repair some constraint, we need to give a fresh new value to some existential variable, that is, we need to create a new object. Such new object, in its turn, might violate another constraint that might need to be repaired by means of creating a new object. Thus, the chase might be infinitely creating new objects, and new objects to repair such objects. As we are going to see, this non-terminating phenomenon is due to the fact that maintaining OCL constraints with existential variables is non-decidable.

To solve this undecidability, we propose two alternatives: (1) limiting the OCL language to avoid bringing RGDs with existential variables, and (2) reuse some decidability results coming from the world of conceptual schema reasoning [103, 107].

For the first option, we define a new OCL_{FO} subset we call OCL_{UNIV} which, basically, consists in removing from OCL_{FO} the *exists* operation, and any other operation that can be used to emulate it. The complete grammar of OCL_{UNIV} together the demonstration that it brings to a set of RGDs without existential variables can be seen in the formalization Section 5.2.3

For the second option, we show that that the decidability conditions stated in [103, 107] for showing decidability for specific conceptual reasoning tasks can be described in a concise logic way, which we call *Finite Canonical Property* (FCP), that can be exported to other reasoning tasks. Intuitively, a set of dependencies (such as RGDs) Σ enjoys FCP if for any initial data state I, all its possible repairs can be captured by a finite set of finite canonical models. Then, this FCP entails chase termination, as we show in the formal definitions and proofs of Section 5.2.4.

5.1.3 Customizing RGDs

Chasing RGDs permits computing all the possible ways to maintain all the constraints. That is, fixed some initial data state I, initial structural events E, and constraints C, chasing the RGDs of C permits computing all the possible missing structural events R s.t. *apply*$(E \cup R, I) \models C$.

However, it is possible that the information system should not maintain all the possible constraint violations, but combine checking and maintenance policies.

For instance, consider the constraint *SeenIsBought*. Recall that the RGDs of such constraint are the formulas 5.1, 5.2, and 5.3. Clearly, in order to repair the violation of this constraint, it might have sense for some TV-content providing company to automatically consider that some user u buys some content c if s/he starts visualizing it (RGD 5.2). In contrast, it might be undesired to repair the same constraint by deleting the fact that some user has seen some content c when such user cancels the purchase of c (RGD 5.3). Actually, in this case it is probably more desirable to cancel the deletion of the purchase since the user has already visualized the content. In summary, the information system should apply a maintenance policy for the first kind of violation of the *SeenIsBought* constraint, whereas it should probably apply a checking policy for the second one.

To achieve this combination of checking and maintenance policies, we can simply replace RGDs for its corresponding EDCs. That is, in the previous example, we should replace the RGD 5.3 for its corresponding EDC 4.5. Thus, when chasing the RGDs, we are not going to compute the repairs for such kind of violation.

Moreover, it might also be desirable to disallow some of the possible repairs pointed by the remaining RGDs. For instance, when deleting some episode for some series, it might not be desirable to delete the series to repair a minimum cardinality constraint violation, but it might be ok to insert a new episode to it.

In order to disallow some of the possible repairs pointed by some RGD, we can simply move the repair from the RHS to the LHS of the rule again. Following our previous example, we can disallow from RGD 5.15 the repair consisting in deleting the series s by moving the literal $\delta series(s)$ from the LHS to the RHS as follows:

$$\begin{aligned} &series(s) \wedge \delta hasEpisode(s) \wedge \neg\delta series(s) \rightarrow \iota episodeSeries(e,s) \vee \\ &\qquad \iota containEpisodeSeries(e',s) \wedge \iota forbidDeltaEpisodeSeries(e',s) \end{aligned} \tag{5.18}$$

Note that, if we remove all literals from the RHS to the LHS of the RGD, we obtain again the EDC rule.

5.2 Formalizing RGDs

In the previous section we have given an intuitive understanding of what RGDs are, how can they be obtained, and how can they be chased to compute the repairs R. Now, we formalize the basic notions of RGDs to unambiguously describe our method, and to give a proof for its correctness to compute repairs.

As we have previously shown, obtaining the RGDs from a given set of UML/OCL constraint departs from its translation to EDCs. In this case, however, we need to obtain the EDCs for constraints involving existential variables. In order to deal with these existential variables, we provide the *new*/*old* mappings for derived literals containing existential variables, and prove its correctness.

Then, we bring an algorithm to obtain the RGDs from these EDCs. Intuitively, the RGDs obtained are *equivalent* to the input EDCs in the sense that, roughly speaking, any model of the EDCs is a model of the RGDs. Thus, any set of structural events E and data state I satisfies the RGDs if and only if applying E in I satisfies the original constraints.

Afterwards, we show that by chasing RGDs we can compute the repairs R for some given data instance I and set of structural events E. To do so, we distinguish two cases: RGDs without existential variables, and RGDs with existential variables. Indeed, computing repairs with RGDs with no existential variable is decidable, whereas it turns to be undecidable when existential variables are involved.

In order to tackle such undecidability issue with RGDs, we define the concept of *Finite Canonical Property* (FCP). Briefly, a set of RGDs enjoys FCP if all its canonical models are finite, which is a situation that guarantees decidability for computing repairs.

Finally, we formally present the RGDs customization possibilities. To do so, we show how can we customize an RGD and redefine the concept of repair based on the notion of *well-supported models* [46], next we show how can we chase such RGD to obtain these new customized repairs.

5.2.1 Obtaining EDCs with Existential Variables from UML/OCL Constraints

The first step to obtain the RGDs consists in obtaining the EDCs from the UML/OCL constraints. To do so, we benefit from the method already explained in Section 4.2. That is, we first translate UML/OCL constraints into logic denials, and then, we apply the *new*/*old* mappings to the literals in such denials to obtain the EDCs.

The translation from UML/OCL constraints into denials is exactly the same as before, but without aggregating the existential variables (i.e., without applying the postprocessing Algorithm 8). This is because since we do not deal with aggregated literals for the problem of integrity maintenance, we no longer benefit from such transformation.

As a result, we need to extend the *new*/*old* mappings to deal with derived literals containing existential variables. Indeed, in Section 4.2 we only gave such mappings for derived literals without existential variables since all the derived literals with existentials were replaced by aggregations according to Algorithm 8.

In the following we present the *new*/*old* mapping for derived literals with existential literals. Since positive derived literals can be unfolded, this mapping only treats negated literals. Recall that the *new*/*old* mappings map a literal l from some signature S to a formula ϕ in an augmented signature S' s.t., for any given data state I and structural events E, $I \cup E \models \phi$ iff *apply*$(E, I) \models l$.

In particular, given a derived predicate p, and its corresponding derived predicates ιp and δp as defined in Section 4.2.2., we define the *new*/*old* mapping for p literals as follows:

Definition 13. A New/Old mapping for derived literals with exists.

Given any derived predicate p whose derivation rule body is $l_1 \wedge ... \wedge l_n \wedge b$, we define the maps:

$$\begin{aligned}\text{new}(\neg p(\overline{x})) &= \delta p(\overline{x}) \wedge \neg \iota p(\overline{x}) \wedge \neg p^o(\overline{x}) \\ \text{old}(\neg p(\overline{x})) &= \neg p(\overline{x})) \wedge \neg \iota p(\overline{x})\end{aligned}$$

where p^o is a derived predicate defined as follows:

$$p^o(\overline{x}) \leftarrow old(l_1) \wedge ... \wedge old(l_n) \wedge b$$

Intuitively, these mappings says that $p(\overline{x})$ becomes false when it was true when some deletion occurs in the body of p, and there is no insertion neither no tuple in the body of p survives the structural events being applied. Similarly, $p(\overline{x})$ remains false when it was false and no insertion occurs in its body.

In the following, we formally proof that both mappings are correct *New*/*Old* mappings:

Proof. We start proving the correctness of the *new* mapping. Then, we move to *old*.

We prove that $E \cup I \models \text{new}(\neg l)_{[\sigma]}$ iff $I \models l_{[\sigma]}$ and $apply(E, I) \not\models l_{[\sigma]}$. Assume that we have $E \cup I \models \text{new}(\neg l)_{[\sigma]}$. Then, for some derivation rule we have $E \cup I \models \delta l_{[\sigma]}$. By construction of the derivation rules, this is the case iff for some literal lb_i inside the derivation rule body of l we have $E \cup I \models$ new$(\neg lb_i)_{[\sigma]}$, and for the rest of literals lb_j in the body we have $E \cup I \models lb_{j[\sigma]}$. Thus, we see that for all literals lb in the body of l we have $I \models lb_{[\sigma]}$, thus, $I \models l_{[\sigma]}$. We now prove that $apply(E, I) \not\models l_{[\sigma]}$. We do it by contradiction. If $apply(E, I) \models l_{[\sigma]}$, then, for all the literals lb inside l, we would have $apply(E, I) \models lb_{[\sigma]}$. Thus, for each literal lb, either $E \cup I \models$ old$(lb_{[\sigma]})$ or $E \cup I \models$ new$(lb_{[\sigma]})$. If for all literals lb we have $E \cup I \models$ old$(lb_{[\sigma]})$, then, by construction $E \cup I \models l^o{}_{[\sigma]}$, and thus $apply(E, I) \not\models l_{[\sigma]}$, contradiction. If for some literal lb we have $E \cup I \models$ new$(lb_{[\sigma]})$, then, by construction, $E \cup I \models \iota l_{[\sigma]}$, and thus, $apply(E, I) \not\models l_{[\sigma]}$, contradiction.

The proof of the other direction of the iff follows similarly. We now prove that *old* is a correct Old mapping.

We prove that $E \cup I \models$ old$(\neg l)_{[\sigma]}$ iff $I \not\models l_{[\sigma]}$ and $apply(E, I) \not\models l_{[\sigma]}$. Assume that we have $E \cup I \models$ old$(\neg l)_{[\sigma]}$. By definition, this is the case iff $E \cup I \models \neg l_{[\sigma]}$ and $E \cup I \models \neg \iota l_{[\sigma]}$. Thus, we see that $I \models \neg l_{[\sigma]}$. We now proof that $apply(E, I) \models \neg l_{[\sigma]}$ by contradiction. If $apply(E, I) \models l_{[\sigma]}$, then, for all literals lb in the body of l we would have $apply(E, I) \models lb_{[\sigma]}$. Since it cannot be the case that for all literals lb, $I \models lb_{[\sigma]}$ (otherwise, $I \models l_{[\sigma]}$, which is a contradiction), we know that for some literal lb, $E \cup I \models$ new$(lb_{[\sigma]})$. Thus, by construction, $E \cup I \models \iota l_{[\sigma]}$, which implies $E \cup I \not\models$ old$(\neg l)_{[\sigma]}$, contradiction.

The proof of the other direction of the iff follows similarly. □

Finally, we need to remove the disjunctions used in the LHS of the EDCs. In Section 4.2 we used disjunctions in order to reduce the number of generated EDCs. However, now we need the body of the EDCs to be pure conjunctions of literals in order to be able to move the negated structural events to the RHS of the rule.

In order to remove such disjunctions it is sufficient to apply an *unfolding-like* procedure. For instance, given an EDC with disjunctions in its LHS like the following:

$$\iota p(x) \wedge (\iota q(x) \vee (q(x) \wedge \neg \delta q(x))) \rightarrow \bot$$

We obtain the following 2 EDCs without disjunctions:

$$\begin{aligned} &\iota p(x) \wedge \iota q(x) \rightarrow \bot \\ &\iota p(x) \wedge q(x) \wedge \neg \delta q(x) \rightarrow \bot \end{aligned}$$

5.2.2 Obtaining RGDs from EDCs

The basic idea to obtain the RGDs from EDCs consists into moving negated structural events from the LHS to the RHS. However, in order to have all the possible structural events that might repair a constraint in the RHS of the rule, we need to apply additional transformations such as literal unfolding.

In Algorithm 9, we formally define how to obtain the RGDs of a given EDC by applying all these transformations. It receives as input an EDC and returns as output a set of RGDs. The number of RGDs obtained depends on the derived literals appearing in the EDC.

Algorithm 9 getRGDs($premise \rightarrow \bot$)

```
$result := \emptyset$
if $premise$ contains a positive derived literal $l$ then
    for all $unfoldedPremise$ in unfoldingPremise($l$, $premise$) do
        $result := result \cup$ getRepairGeneratingDependencies($unfoldedPremise \rightarrow \bot$)
    end for
else
    $new\_Conclusion := \bot$
    $new\_Premise := \top$
    % First loop: Place negative literals containing structural events to the conclusion
    for all Literal $l$ in $premise$ do
        if $l$ is a negative event literal then
            $new\_Conclusion := new\_Conclusion \vee positive(l)$
        else
            $new\_Premise := new\_Premise \wedge l$
        end if
    end for
    % Second loop: Apply all the possible unfoldings
    for all Positive derived literal $l$ in $new\_Conclusion$ do
        $new\_Conclusion :=$ unfoldingConclusion($l$, $new\_Conclusion$)
    end for
    % Third loop: Remove negative literals from the conclusion
    for all Negative literal $l$ in $new\_Conclusion$ do
        $new\_Conclusion :=$ replace($\neg l$, $\iota forbidL$, $new\_Conclusion$)
        $result := result \cup$ getRepairGeneratingDependencies($\iota forbidL \wedge l \rightarrow \bot$)
    end for
    % Fourth loop: Remove non structural event literals from the conclusion
    for all Non structural event literal $l$ in $new\_Conclusion$ do
        $new\_Conclusion :=$ replace($l$, $\iota containL$, $new\_Conclusion$)
        $result := result \cup \{\iota containL \wedge \neg l \rightarrow \bot\}$
    end for
    $result := result \cup \{new\_Premise \rightarrow new\_Conclusion\}$
end if
return $result$
```

The algorithm starts by checking the existence of positive derived literals in the premise. If this is the case, the algorithm unfolds the literal and recursively calls the algorithm until all the literals are base. Since, in our settings, the predicates are non-recursive, this recursion is guaranteed to terminate.

Then, the algorithm performs four loops, each one corresponding to a different transformation.

The first loop moves the negative literals involving structural events (i.e., structural/derived event literals, or derived literals containing event literals in its body) from the premise to the conclusion.

The second loop applies the usual unfolding for derived literals placed in the conclusion, which is also guaranteed to terminate.

The third loop removes negative literals p from the conclusion by considering a new EDC. This transformation encompasses a recursive call to the algorithm to build the RGD for such new EDC. This recursion directly terminates if the predicate p is base since the formula $\iota forbidP \wedge p \rightarrow \bot$ would need no more transformations. Again, the absence of recursive predicates guarantees that at some point predicate p will be base and, thus, the recursion is finite.

The last loop removes non-structural event literals from the conclusion. We *replace* any non-structural event literal p for $\iota containP$ and add a new EDC $\iota containP \wedge \neg p \rightarrow \bot$. In this case, we do not need to call the algorithm recursively to translate this EDC to RGDs because we know that, since p is base, no transformation would be applied to the rule.

Since all the recursions and loops are guaranteed to terminate, the algorithm terminates.

Now, we proof that such algorithm is correct. That is, we first proof that the output are, indeed, RGDs, and then, that such RGDs are equivalent to the input EDCs.

Property 13. Algorithm's output are RGDs

For any given EDC, applying Algorithm 9 we obtain a set of RGDs, that is, a set of rules with the form:

$$l_1 \wedge \ldots \wedge l_m \wedge b \rightarrow \bigvee_{i=1..n} l_{i1} \wedge \ldots \wedge l_{im} \wedge b_i$$

where each l_i is a positive base event or a non-event literal; each l_{ij} is a positive base event literal, and b and b_i are conjunctions of built-in literals.

Proof. The proof is based on checking that every rule added in the *result* variable of the algorithm fits the previous RGD form. We do so by induction following the recursive nature of the algorithm.

The rules directly added from algorithm's recursive calls fits such form immediately by induction hypothesis. This is because the recursion is finite since all predicates are non-recursive, and thus, at some point the recursion reaches some base case. So, we only lack to prove these base cases. That is, the rules created and added in the *result* variable that do not directly come from recursion.

It is easy to see that there are only two kinds of rules added in *result* not coming from a recursive call of the algorithm: $\{\iota containL \wedge \neg l \rightarrow \bot\}$ (added in the fourth loop), and $\{new\_Premise \rightarrow new\_Conclusion\}$.

For the first one, l is ensured to be a positive non-event base literal. We now show so by contradiction. If l was negated, it would have been replaced in the third loop, so l must be positive. If l is positive but derived, it would have been unfolded in the second loop, so l must be positive and base. Finally, since l appears in the fourth loop, which iterates through non-event literals, then, it is for sure a positive non-event base literal. Thus, taking in account that $\iota containL$ is a positive event literal $\{\iota containL \wedge \neg l \rightarrow \bot\}$ fits the previous form.

For the latter, we first show that *new_Premise* literals are base literals or non-event literals. Then, we show that the *new_Conclusion* literals are all base event literals.

new_Premise cannot contain derived literals since all of them have been unfolded at the beginning of the algorithm. The unique exception are the negated literals, which cannot be unfolded. However, those negated literals that are event literals are removed from the premise in the first loop. Thus, since no more literals are added in the *new_Premise* after such loop, all its literals are base literals or non-event literals.

We now conclude the proof by showing that *new_Conclusion* literals are all base positive event literals. All the literals are base since all the derived literals are unfolded in the second loop and no derived literal is inserted in *new_Conclusion* after that loop. All the negated literals are removed in the third loop and no negated literal is inserted after that. Lastly, all the literals that are not events are removed in the last loop. □

Property 14. Algorithm's output is equivalent to EDCs input

For any given EDC ψ, applying Algorithm 9 we obtain a set of equivalent formulas Σ. Such formulas are equivalent in the sense that, for any data instance I and set of structural events E, we have that:

$$I \cup E \models \psi \textbf{ iff } \text{for some A, } I \cup E \cup A \models \Sigma$$

where A is a set of instances for the auxiliary predicates $\iota containX$ and $\iota forbidX$.

Proof. We proof so by showing that each transformation applied to the input EDC leads to an equivalent formula. In particular, we first show that the transformations applied by the first two loops preserve the equivalence in the classical sense (i.e., $\phi \equiv \Sigma$), whereas the last transformation is equivalent in the previously defined meaning.

The first loop moves literals from the LHS to the RHS of the rules. Such transformation leads to an equivalent logic formula because $\phi \wedge \neg l \rightarrow \psi \equiv \neg\phi \vee l \vee \psi \ \equiv \phi \rightarrow l \vee \psi$

The second loop only applies literal unfoldings, which are well-known to produce equivalent logic formulas.

The third loop replaces the negated literals l from the conclusion for a literal $\iota forbidL$ and adds the new formula $forbidL \wedge l \rightarrow \bot$. We now show that such transformation is equivalent to the original formula according to the previous equivalence definition. Assume that the initial formula is ψ, and that after applying such transformation for some negated literal l in the conclusion of ψ we obtain a new formula Σ. Consider some data state I and set of structural events E. In case $I \cup E \models \psi_{[\sigma]}$, for some substitution σ then, clearly, $I \cup E \not\models l_{[\sigma]}$. Consider now $A = forbidL_{[\sigma]}$. It is easy to see that $I \cup E \cup A \models \Sigma$. In case $I \cup E \not\models \psi_{[\sigma]}$, we now show by contradiction that, for any A, $I \cup E \cup A \not\models \Sigma[\sigma]$. Assume that we have an A s.t. $I \cup E \cup A \models \Sigma_{[\sigma]}$. Then, because of the rule $forbidL \wedge l \rightarrow \bot$ included in Σ, we know that $I \cup E \cup A \not\models l_{[\sigma]}$. Thus, $I \cup E \models \neg l_{[\sigma]}$. Moreover, $I \cup E \models \psi_{[\sigma]}$, contradiction.

The fourth loop replaces the non-event literals l from the conclusion for a literal $\iota containL$ and adds the new formula $\iota containL \wedge \neg l \rightarrow \bot$. We now show that such transformation is equivalent to the original formula according to the previous equivalence definition. Assume that the initial formula is ψ, and that after applying such transformation for some literal l in the conclusion of ψ we obtain a new formula Σ. Consider some data state I and set of structural events E. In case $I \cup E \models \psi_{[\sigma]}$, for some substitution σ then, clearly, $I \cup E \models l_{[\sigma]}$ if the LHS of $\psi_{[\sigma]}$ holds in $I \cup E$. Consider now $A = containL_{[\sigma]}$ if the LHS of $\psi_{[\sigma]}$ holds in $I \cup E$. It is easy to see that $I \cup E \cup A \models \Sigma$. In case $I \cup E \not\models \psi_{[\sigma]}$, we now show by contradiction that, for any A, $I \cup E \cup A \not\models \Sigma_{[\sigma]}$. Assume that we have an A s.t. $I \cup E \cup A \models \Sigma_{[\sigma]}$. Then, because of the rule $containL \wedge \neg l \rightarrow \bot$ included in Σ, we know that $I \cup E \cup A \models l_{[\sigma]}$. Thus, $I \cup E \models l_{[\sigma]}$. Moreover, $I \cup E \models \psi_{[\sigma]}$, contradiction. □

We summarize both properties in the following one:

Property 15. Algorithm's output are equivalent RGDs

For any given EDC ψ, applying Algorithm 9 we obtain a set of equivalent RGDs Σ, in the following sense:

$$I \cup E \models \psi \textbf{ iff } \text{for some A, } I \cup E \cup A \models \Sigma$$

where A is a set of instances for the auxiliary predicates $\iota containX$ and $\iota forbidX$.

Proof. Directly from the Properties 13, and 14. □

For our purposes, we strengthen the previous result to sets of EDCs and sets of RGDs. That is, a set of EDCs is *equivalent* to the set of RGDs obtained by applying Algorithm 9 to each EDC.

Property 16. A set of EDCs is equivalent to its set of RGDs

For any given set of EDC Ψ, the set of RGDs Σ obtained by applying Algorithm 9 to every $\psi \in \Psi$, we have:

$$I \cup E \models \Psi \textbf{ iff } \text{for some A, } I \cup E \cup A \models \Sigma$$

where A is a set of instances for the auxiliary predicates $\iota containX$ and $\iota forbidX$.

Proof. First, we show that if $I \cup E \models \Psi$, we can build a set of auxiliary instances A s.t. $I \cup E \cup A \models \Sigma$. Then, we proof that if $I \cup E \not\models \Psi$, then, for no A, $I \cup E \cup A \models \Sigma$.

Assume that $I \cup E \models \Psi$. Then, build A in the following way. Initialize $A = \emptyset$, and then, pick any $\xi \in \Sigma$ that is violated in $I \cup E \cup A$. Consider additionally ψ to be the EDC where ξ comes from. Because of Property 15, we know that, for this ξ, there are some minimal set of instances A_ξ s.t. makes $I \cup E \cup A_\xi \models$ getRGDs(ψ). Include such A_ξ into A until every ξ is satisfied. We now show that adding such minimal set of instances A_ξ into A does not encompass the violation of any other $\xi' \in \Sigma$, and thus, such process eventually terminates.

Assume that when adding the minimal set of instances A_ξ into A we cause $I \cup E \cup A$ to violate some other RGD ξ'. This is only the case if ξ' contains an auxiliary literal in its LHS. Clearly, $\xi' \notin$ getRGDs(ψ). However, all the RGDs that might unify some newly created auxiliary literal to repair ξ appears in getRGDs(ψ) by construction, contradiction.

Now, we proof that if $I \cup E \not\models \Psi$, then, for no A, $I \cup E \cup A \models \Sigma$. If $I \cup E \not\models \Psi$, then, for some $\psi \in \Psi$, $I \cup E \not\models \psi$. Thus, by Property 15, for no A, $I \cup E \cup A \models$ getRGDs(ϕ). Hence, for no A, $I \cup E \cup A \models \Sigma$.

□

5.2.3 Computing Solutions through RGDs

For our purposes, we define the notion of an RGD repair. Intuitively, given a set of RGDs Σ, some data state I, and structural events E, an RGD repair is a minimal set of structural events R that makes $I \cup E \cup R$ satisfy Σ. By minimal, we mean that no proper subset R' of R is itself a repair. Formally:

Definition 14. RGDs repair

Given a set of RGDs Σ, a data state I, and a set of structural events E, a set of structural events R is a *repair* for $I \cup E$ and Σ iff:

- There exists some set of instances A for the auxiliary predicates $\iota containX$ and $\iota forbidX$ s.t. $I \cup E \cup R \cup A \models \Sigma$, and
- There is no such set of instances A for a subset $R' \subset R$.

Now, the crucial point is that an RGD repair is the set of additional structural events such that, when applied to the initial data state I with the initial structural events E, permits reaching a new data state I^n that does not violate any integrity constraint. That is, given a set of integrity constraints written as denials Φ, consider the RGDs Σ obtained by translating the EDCs of Φ to RGDs through Algorithm 9. Then, given any data state I, and structural events E, a set of structural events R is an RGD repair for Σ and $I \cup E$ if and only if applying $E \cup R$ into I does not violate any constraint in Φ. In virtue of this result, we say that *RGD repairs* are (constraint) *repairs* as defined in the preliminaries. Formally:

Property 17. RGDs repairs are constraint repairs

Consider a set of denials Φ, and its corresponding set of RGDs Σ obtained by translating the EDCs of Φ through Algorithm 9. Then, for any data state I, and structural events E we have that:

R is a (constraint) repair for Φ,I,E **iff** R is an (RGD) repair for Σ,I,E.

Proof. Consider any set of of denials Φ, its corresponding set of EGDs Ψ, and its corresponding set of RGDs Σ. Moreover, consider also an arbitrary data state I and set of structural events E. We start proving that any constraint repair R for Φ,I,E is also an RGD repair for Σ,I,E and then, we proof the converse.

Suppose R is a (constraint) repair for Φ,I,E. Then, because of Definition 12, $I \cup E \cup R$ also satisfies the EDCs Ψ. By Property 16 we have that there is some set of auxiliary instances A s.t. $I \cup E \cup R \cup A \models \Sigma$. Moreover, R is minimal since, otherwise, it would not be a constraint repair. Thus, R is a (RGD) repair for Σ, I,E.

Conversely, assume that R is a (RGD) repair for Σ,I,E. Then, because of Property 16 we have that $I \cup E \cup R \models \Psi$. Thus, by Definition 12 we have *apply*$(E \cup R, I) \models \Phi$.

Moreover, R is minimal since, otherwise, it would not be an RGD repair. Thus, R is a (constrain) repair for Φ, $\square$

Thus, we can compute constraint repairs for Φ through computing RGD repairs for its corresponding Σ. For ease of presentation, in the following we are going to speak about computing repairs for RGDs, but we must keep in mind that this is equivalent to compute the repairs for the denials Φ and, in the last term, it is equivalent to compute the repairs for the UML/OCL$_{\text{FO}}$ constraints.

The interesting fact about working with RGDs is that its repairs can be computed using a chase algorithm. This is because RGDs follows the syntactic form of *dependencies* (with built-in literals, disjunctions, negation in the LHS, and $\perp$), and the chase is an algorithm though for computing the necessary literals to satisfy sets of dependencies. In this manner, we see that the chase algorithm, is not only suitable for solving the certain answers of some query, or computing data exchange universal solutions [40], but also to perform incremental integrity maintenance.

In the following, we explain how can we *chase* the previous RGDs to compute such repairs. To do so, we distinguish whether the RGDs have existential variables, or not.

In the case where no existential variable is present in the RGDs, we are going to see that: (1) any complete chase implementation can be used to compute all the sets of repairs R, (2) such computation always terminates, (3) there is an expressive fragment of OCL that only generates RGDs without existential variables, and thus, benefits from the previous two properties.

In the case where existential variables exists, we are going to see that: (1) the traditional chase implementations might bring results that are not repairs, (2) a new chase-like algorithm can be defined solving the previous problem, (3) chase termination cannot be guaranteed since the problem of computing the repairs R is not decidable.

Computing Repairs through RGDs without existential variables: the chase

The chase is an algorithm for computing universal model sets [40], that is, given an initial data state I and some set of dependencies Σ, compute a set $\mathfrak{U}$ of sets of instances U (possibly with *labelled nulls*) satisfying the following properties:

- (soundness) $I \cup U \models \Sigma$,
- (universality) for any M s.t. $I \cup M \models \Sigma$, there is some $U \in \mathfrak{U}$, and some ground substitution σ for the labelled nulls in U s.t. $U\sigma \subseteq M$
- (minimality) for each $U \in \mathfrak{U}$, there is no other $U' \in \mathfrak{U}$, and ground substitution σ for the labelled nulls in U' s.t. $U'\sigma \subseteq U$

Intuitively, the first property means that the models in $\mathfrak{U}$ satisfy the dependencies in Σ. The second, ensures that any other model M satisfying the dependencies Σ is *captured* in $\mathfrak{U}$. The third, guarantees that we cannot remove any model from $\mathfrak{U}$.

The idea is that, in the absence of existential variables, the *universal model set* of the RGDs is the set of all its possible repairs. We formally state so in the following property:

Property 18. Universal Model Sets of RGDs (without existential variables) are Repairs

Consider a set of RGDs Σ without existential variables. Then, for any data state I, and structural events E we have that:

$$R \text{ is a repair for } \Sigma, I, E \textbf{ iff } R \in \mathfrak{U}^*$$

where $\mathfrak{U}^*$ is the universal model set of Σ, I, E after removing, for each $U \in \mathfrak{U}$, the auxiliary literals $\iota containX$ and $\iota forbidX$, and removing any U_2^* from $\mathfrak{U}^*$ if $\mathfrak{U}^*$ contains some $U_1^* \in \mathfrak{U}^*$ s.t. $U_1^* \subsetneq U_2^*$.

Proof. We start proving that if R is a repair for Σ,I,E, then, $R \in \mathfrak{U}^*$. Afterwards, we prove the converse.

Assume that R is a repair for the RGDs Σ and data state I, with structural events E. In addition, consider $\mathfrak{U}$ the universal model set of Σ with I, E. Now, by Definition 14, we see that there is some minimal set of auxiliary instances A s.t. $I \cup E \cup R \cup A \models \Sigma$. Thus, by universality of models sets, there exists some $U \in \mathfrak{U}$ s.t. for some substitution σ for the *labelled null*, $U\sigma \subseteq R \cup A$. Since U has no *labelled nulls* due to the absence of existential variables in Σ, we can omit such substitution σ, thus, $U \subseteq R \cup A$. However, U cannot be a proper subset of $R \cup A$ (since this would imply A not to be minimal, or R not to be minimal, which would be a contradiction), thus, $U = R \cup A$. Hence, the set U^* corresponding to remove the A literals from U satisfies that $U^* \in \mathfrak{U}^*$, and $U^* = R$, thus $R \in \mathfrak{U}^*$.

We now proof the converse. Consider any set of structural events $R \in \mathfrak{U}^*$. Then, by construction of $\mathfrak{U}^*$ from $\mathfrak{U}$, and soundness of $\mathfrak{U}$, there is some set of auxiliary instances A s.t. $I \cup E \cup R \cup A \models \Sigma$. We lack to proof that R is minimal to show that it is an RGD repair. We do so by contradiction. Assume that R is not minimal, and thus, there is some other $R' \subset R$ for which there is another A' s.t. $I \cup E \cup R \cup A \models \Sigma$. Moreover, assume this R' to be minimal. In this case, by universality of $\mathfrak{U}$ we have, $R' \cup A' \in \mathfrak{U}$. Thus, R' would appear in $\mathfrak{U}^*$, which would mean that $\mathfrak{U}^*$ would contain both R', and R. But this would mean that $\mathfrak{U}^*$ would not be minimal, contradiction.

Thus, R is minimal, and thus, it is a repair of Σ for I, E. $\square$

Hence, any complete chase able to compute universal model sets can be used to compute denial repairs, if their RGDs have no existential variables.

Moreover, since there are no existential variables, the chase is going to terminate. Intuitively, the chase is going to create instances using a finite number of constants (those present in $I \cup E$), and a finite number of predicates (those appearing in the RHS of RGDs). Thus, the number of new instances a chase can create is finite, and since the chase terminates when it does not create a new instance, the chase is ensured to terminate.

In the following, we formally state and proof such statement. To do so, we use the notion of *complete chase*. By *complete chase* we mean any *chase-like* algorithm ensuring termination when the *universal model set* $\mathfrak{U}$ is finite (i.e. $||\mathfrak{U}||$ is finite, and for each $U \in \mathfrak{U}$, $||U||$ is finite).

Property 19. Computing repairs of RGDs (without existential variables) through chase terminates

Given a set of RGDs Σ without existential variables, computing the repairs of Σ for any data state I and structural events E through a complete chase terminates.

Proof. The proof is based on showing that the universal model set $\mathfrak{U}$ is finite.

Indeed, consider the set of all possible instances V we can build with (1) the set of predicates appearing in the RGDs Σ, and (2) the set of constants appearing in $I \cup E$. Since the number of predicates is finite, and the number of constants appearing in $I \cup E$ is finite, then, $||V||$ is finite.

Since there are no existential variables in Σ, for each $U \in \mathfrak{U}$, we have $U \subseteq V$, thus, $||U||$ is finite. In addition, $\mathfrak{U} \subseteq 2^V$, thus, $||\mathfrak{U}||$ is finite.

So, since a complete chase terminates if $\mathfrak{U}$ is finite, and we have shown that $\mathfrak{U}$ is finite, the chase is ensured to terminate. $\square$

Hence, any complete chase dealing with disjunctions, $\perp$, and built-in literals can be used to repair a set of RGDs (without existential variables). Examples of chase-like algorithms that are complete in such case are, for instance, the ded chase described in [78], and the *extended core chase* [40]. The last one does not natively support built-in literals, although, we argue that we can naturally extend it to do so. [1]

[1]Indeed, a built-in literal such as $x > 3$ is no more than a literal with a prefixed set of instances with the particularity that, instead of storing such instances in a data state (which would be im-

```
ExpBool   ::=  ExpBool and ExpBool        | ExpBool or ExpBool
               | ExpOp
ExpOp     ::=  Path->excludesAll(Path)    | Var.Member->includesAll(Path)
               | Path->excludes(Path)     | Var.Member->includes(Var)
               | Path->isEmpty()          | Path->forAll(Var| ExpBool)
               | Path OpComp Constant     | not Path.oclIsKindOf(Class)
               | Path OpComp Path         | Path.oclIsKindOf(Class)
Path      ::=  Var.Role                   | Class.allInstances().Nav
Nav       ::=  Role.Nav                   | oclAsType(Class).Nav
               | Role                     | Attribute
               | oclAsType(Class)
```

Figure 5.1: OCL_{UNIV} syntax

Finally, we show that there is an expressive subset of OCL_{FO} whose RGDs have no existential variables. That is, the constraints written in this language can be maintained through a chase procedure ensuring its termination. We call such OCL_{FO} subset as OCL_{UNIV}.

The syntax of OCL_{UNIV} is given in the grammar of Figure 5.1. Briefly, OCL_{UNIV} limits OCL_{FO} to avoid the *exists* operator, which is the cause for existential variables, and any other OCL operator that could be used to emulate the *exists*. For instance the use of *not* is restricted, since combining a *not* with a *forAll* would emulate the *exists* operator.

Property 20. Computing repairs of OCL_{UNIV} RGDs through chase terminates

Given a set of RGDs Σ coming from a set of OCL_{UNIV} constraints C, computing the repairs of C for any data state I and structural events E through a complete chase procedure terminates.

Proof. By construction, OCL_{UNIV} constraints only give rise to denials with atomic negation [103]. Thus, no derivation rule is required to define such denials, and hence, they do not have existential variables. By construction again, we see that denials without existential variables give raise to EDCs without existential variables. This is because the *New*/*Old* mappings defined for transforming denials into EDCs do not create derivation rules if the denials do not use derivation rules. Finally,

possible since they are infinite, e.g. there are infinitely many x with $x > 3$), they are computed on demand.

by construction of RGDs, EDCs without existential variables brings RGDs without existential variables. Thus, by Property 19 we see that chasing the RGDs of some OCL_{UNIV} constraints terminates, and its result is the set of all the possible repairs. $\square$

OCL_{UNIV} is expressive enough to codify most of the constraints appearing in our running UML/OCL schema example. In particular, it permits encoding the first 4 OCL constraints, all the UML maximum cardinality constraints, the hierarchy constraints, and the disjointness. That is, a total of 20 constraints out of 36 constraints.

Furthermore, OCL_{UNIV} is able to encode almost a superset of the first-order constraints patterns proposed in [31], which have been shown to be useful for defining around the 60% of the integrity constraints found in real schemas. The only exception is the path inclusion constraint pattern for which we can only specify the situations that are compliant to our grammar to avoid existential variables.

Computing Repairs through RGDs with existential variables: the vips-chase

In the previous subsection, we have computed the repairs of the RGDs (without existential variables) by *chasing* them.

Now we want to deal with RGDs with existential variables. Indeed, if we deal with RGDs with existential variables, we are going to be able to repair any set of OCL_{FO} constraints, for any initial data state, and any set of initial structural events.

However, when introducing existential variables, the result of chasing RGDs are not necessarily repairs since they might contain *labelled nulls*. Indeed, in the standard chase, existential variables are instantiated with *labelled nulls* rather than constants, and according to our definitions, an RGD repair should be ground, thus, no labelled nulls are allowed.

In order to obtain the repairs, we can apply substitutions for the *labelled nulls* appearing in the chase result. That is, we can obtain the repairs by replacing each *labelled null* for a constant, in all the sets of instances retrieved by the chase.

However, applying such substitutions only ensures completeness for finding all the repairs R, but not *soundness*. That is, although all the repairs R can be obtained through these replacements (*completeness*), some replacements might lead to sets of instances that are not repairs (*unsoundness*). In the following we first exemplify the problem, and then, go to the solution.

Consider the following set of dependencies Σ:

$$\iota P(x) \rightarrow \iota Q(x, y)$$
$$\iota Q(x, 1) \rightarrow \bot$$

And that the initial data state and structural events are $I = \emptyset$, $E = \iota P(1)$. Chasing Σ with I, E returns a universal model set composed of only one model U:

$$U = \{\iota Q(1, \textit{null}_1)\}$$

Certainly, a repair R such as $\iota Q(1,2)$ can be obtained from U by the substitution $\sigma = \{\mathit{null}_1/2\}$, but a non-repair such as $\iota Q(1,1)$ can also be obtained using the substitution $\sigma = \{\mathit{null}_1/1\}$. Note that $\iota Q(1,1)$ is not a repair because it directly violates the second dependency in Σ.

In order to restrict the universal model sets to only characterize repairs, our intention is to provide them with a set of built-in literals restricting the values that the labelled nulls might take in a similar way to the intensional solutions described in [32]. In the previous example, we would like to have:

$$U = \{\iota Q(1, \mathit{null}_1), \mathit{null}_1 \neq 1\}$$

Hence, for our purpose, we define the notion of *canonical model set*.

Definition 15. Canonic Model set

Given an initial set of instances I and some set of dependencies Σ, a canonical model set $\mathfrak{U}$ is a set of pairs $\langle U,\ B\rangle$, where U is a set of instances and B is a (consistent) set of built-in literals about its labeled nulls, satisfying the following properties:

- (soundness) For any $\langle U,\ B\rangle \in \mathfrak{U}$, and ground substitution σ s.t. $B\sigma$ is true, we have $I \cup U\sigma \models \Sigma$.
- (universality) For any M s.t. $I \cup M \models \Sigma$, there is some pair $\langle U,\ B\rangle \in \mathfrak{U}$, and some ground substitution σ for the labelled nulls in U s.t. $U\sigma \subseteq M$ and $B\sigma$ is true.
- (minimality) For every $\langle U_1,\ B_1\rangle \in \mathfrak{U}$ and substitution σ_1 for the labelled nulls in U_1 s.t. $B_1\sigma_1$ is true, we have that for any other $\langle U_2,\ B_2\rangle \in \mathfrak{U}$ and substitution σ_2 s.t. $B_2\sigma_2$ is true, $U_2\sigma_2 \not\subseteq U_1\sigma_1$.

Each pair $\langle U,\ B\rangle \in \mathfrak{U}$ is referred as a *canonical model* for Σ, and I.

Then, the idea is that each canonical model in $\mathfrak{U}$ captures a different repair. That is, each canonical model $\langle U,\ B\rangle$ defines a different way to repair the dependencies, up to renaming the *labelled nulls* with values satisfying the built in literals stated in B. Note that, in this way, although a set integrity constraints might have an infinite number of different repairs (e.g. by considering different values to their existential variables), we might be able to describe them all using a finite set of pairs $\langle U,\ B\rangle$.

Property 21. The Canonic Model Sets of RGDs capture its repairs

Consider a set of RGDs Σ. Then, for any data state I, and structural events E we have that:

R is a repair for Σ,I,E **iff**
there is some $\langle U,\ B\rangle \in \mathfrak{U}^*$ and substitution σ s.t. $R = U\sigma$, and $B\sigma$ evaluates to true.

where $\mathfrak{U}^*$ is the canonical model set of Σ, I, E after removing, for each $U \in \mathfrak{U}$, the auxiliary literals $\iota containX$ and $\iota forbidX$, and removing any U_2^* from $\mathfrak{U}^*$ if $\mathfrak{U}^*$ contains some $U_1^* \in \mathfrak{U}^*$ s.t. $U_1^* \subseteq U_2^*$.

Proof. First we proof that for any $\langle U,\ B\rangle \in \mathfrak{U}^*$ and substitution σ s.t. $B\sigma$ evaluates to true, then, $U\sigma$ is a repair for Σ, I, and E. Then, we proof the other direction.

Assume that we have some $\langle U,\ B\rangle \in \mathfrak{U}^*$ and substitution σ s.t. $B\sigma$ evaluates to true. By construction and soundness of $\mathfrak{U}^*$, we have that there exists some set of auxiliary instances A s.t. $I \cup E \cup U\sigma \cup A\sigma \models \Sigma$. We lack to prove that $U\sigma$ is minimal to proof that it is a repair. We proof so by contradiction. Assume that there is some $U' \subset U\sigma'$ s.t. $I \cup E \cup U' \cup A' \models \Sigma$, for some set of auxiliary instances A'. Then, by universality of $\mathfrak{U}$, there is some $\langle U_2,\ B_2\rangle \in \mathfrak{U}^*$ and substitution σ_2 s.t. $B_2\sigma$ is true and $U_2\sigma_2 \subseteq U'$. Moreover, $U_2\sigma_2 \subseteq U\sigma$. Thus, $\mathfrak{U}$ would not be minimal, contradiction.

Assume that we have some repair R for Σ, I, and E. Thus, there exists some set of auxiliary instances A s.t. $I \cup E \cup R \cup A \models \Sigma$. By universality and construction of $\mathfrak{U}^*$, we have that there exists some $\langle U,\ B\rangle$ and substitution σ s.t. $B\sigma$ evaluates to true, and $U\sigma \subseteq R$. We lack to prove that $U\sigma = R$. We do so by contradiction. Assume $U\sigma \subset R$, then, by soundness and construction of $\mathfrak{U}^*$, there would be some set of auxiliary instances such that $I \cup E \cup A \cup U\sigma \models \Sigma$. Thus, R would not be a repair since it would not be minimal, contradiction. □

Now, we define a new chase-like algorithm that computes the canonical models $\langle U,\ B\rangle$ for a given set of dependencies Σ and a set of instances $I \cup E$. Note that this chase needs to create, apart from the ordinary instances in U, the built in literals in B.

We compute the built-in literals present in B with a method based on the Variable Instantiation Patterns (VIPs) technique [50]. For the self-containment of the thesis, we quickly overview the VIPs approach, and how do we apply it in our thesis.

The core idea underlying VIPs consists in, for every labelled null created during the chase, considering all the relevant built-in literals relations we can build between the new labelled null, and the rest of constants and labelled nulls already present. For instance, assume the current state of the chase is the following: $U = \{P(1,5), Q(5, \mathit{null}_1)\}$, and $B = \{\mathit{null}_1 \neq 1, \mathit{null}_1 \neq 5\}$, and the chase needs to create a new labelled null *null*$_2$ for some instance $Q(1, \mathit{null}_2)$. Then, the chase has to consider four different relevant possibilities: $\mathit{null}_2 = 1$, $\mathit{null}_2 = 5$, $\mathit{null}_2 = \mathit{null}_1$, and $\mathit{null}_2 \neq 1 \wedge \mathit{null}_2 \neq 5 \wedge \mathit{null}_2 \neq \mathit{null}_1$. Thus, it needs to consider the following new chase states (or branches):

- $U = \{P(1,5), Q(5, \mathit{null}_1), Q(1,1)\}$, and $B = \{\mathit{null}_1 \neq 1, \mathit{null}_1 \neq 5\}$
- $U = \{P(1,5), Q(5, \mathit{null}_1), Q(1,5)\}$, and $B = \{\mathit{null}_1 \neq 1, \mathit{null}_1 \neq 5\}$
- $U = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_1)\}$, and $B = \{\mathit{null}_1 \neq 1, \mathit{null}_1 \neq 5\}$
- $U = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_2))\}$, and $B = \{\mathit{null}_1 \neq 1, \mathit{null}_1 \neq 5, \mathit{null}_2 \neq 1, \mathit{null}_2 \neq 5, \mathit{null}_2 \neq \mathit{null}_1\}$

Under the presence of order comparisons (i.e., $<$), the relevant relations are all the possible orders a labelled null might take. In the previous case, assume that our set B of current built-in literals of our instance being chased is $B = \{1 < \mathit{null}_1 < 5\}$. Then, the chase has to consider the following new chase states (or branches):

- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1,1)\}$, and $B = \{1 < \mathit{null}_1 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1,5)\}$, and $B = \{1 < \mathit{null}_1 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_1)\}$, and $B = \{1 < \mathit{null}_1 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_2)\}$, and $B = \{\mathit{null}_2 < 1 < \mathit{null}_1 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_2)\}$, and $B = \{1 < \mathit{null}_2 < \mathit{null}_1 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_2)\}$, and $B = \{1 < \mathit{null}_1 < \mathit{null}_2 < 5\}$
- $I = \{P(1,5), Q(5, \mathit{null}_1), Q(1, \mathit{null}_2)\}$, and $B = \{1 < \mathit{null}_1 < 5 < \mathit{null}_2\}$

The first three cases corresponds to equate *null*$_2$ with the current constants and labelled nulls (i.e., 1, 5, null_1). The last four correspond to consider all the possible orderings for the new labelled null with respect to $\{1 < \mathit{null}_1 < 5\}$.

It is worth to highlight that this procedure has its roots on the *canonical models* defined by Ullman to tackle the problem of query containment [112], but in this case the *canonical models* are computed on runtime rather than statically [50].

Algorithm 10 vips-chase(Σ, *I*, E, R_c, B_c, *Result*)

```
$\xi$ := getViolatedDependency($\Sigma$, $I$, $E$, $R_c$, $B_c$)
if $\xi = null$ then
    $Result$.add(removeForbidContain($R_c$), $B_c$)
else
    for all Structural events conjunction $R$ in (getRHS($\xi$)) do
        newPairs := getRepairsAndBuiltInLiterals($R$, $I$, $E$, $R_c$, $B_c$)
        for all $\langle \sigma_R, B_R \rangle$ in newPairs do
            if isMinimal($\Sigma$,$I$,$E$,$R_c \cup R\sigma_R$,$B_C$) then
                chaseRGDs($\Sigma$, $I$, $E$, $R_c \cup R\sigma_R$,$B_c \cup B_R$, $Result$)
            end if
        end for
    end for
end if
```

The following Algorithm 10 describes a chase-like procedure incorporating the VIPs approach we call *vips-chase*.

Initially, the algorithm is called with $R_c = \emptyset$, $B_c = \emptyset$ and $Result = \emptyset$. The first two are the variables containing the current set of instances and current set of built-in literals being built in the chase, and $Result$ is an input/output parameter that will contain the canonical model set for the given RGDs, I, and E. The *getViolatedDependency* function looks for a dependency being violated according to the contents of $I \cup E \cup R_c$, and the built-in literals stored in B_c, and returns the dependency after substituting its universal variables for the constants that witness the violation. We assume the function *getViolatedDependency* to be *fair*. That is, for any violated dependency ξ, ξ is going to be selected at some point of the chase. There are several ways to define a *fair* selection of the violated dependency (e.g., a *first-in-first-out* queue of violated dependencies).

If no dependency is violated, then $\langle R_c, B_c \rangle$ is added in *Result*, since this means that any substitution σ for the labelled nulls in R_c satisfying B_c, we have $I \cup E \cup R_c\sigma \models \Sigma$. Before adding R_c into $Results$, we apply the function *removeForbidContain*. This function returns the set of structural events R_c but removing those ground atoms corresponding to the auxiliary structural events $\iota forbidP$ and $\iota containP$.

For repairing the dependency, we try all the possible conjunctions of events R in the dependency conclusion. Moreover, for each one of these conjunctions, we try all the suitable substitutions and relevant built-in literals for their *labelled nulls* $\langle \sigma_R, B_R \rangle$ according to the VIPs. Then, we add $R\sigma_R$ into the repair R_c, B_R into B_c, and apply a recursive call to the same algorithm to continue the chase until no dependency is violated.

In order to ensure completeness, the vips-chase requires applying a minimality check at each chase-step. This is done with the *isMinimal* function invoked just before the recursive chase call. This function checks whether some proper subset of the currently built R_c is already a solution, and if so, it discards R_c. Intuitively, without such minimality check, the vips-chase might hang computing an infinite non-canonical model (recall that all canonics must be minimal). This situation is similar to the core-computation step required for the core-chase: if the core-chase does not make a core computation at each chase-step, the core-chase might hang computing an infinite non-core solution [40].

Since applying a minimality check at each chase-step might cause a bottle neck in the method, we are going to see that, in some cases, we can safely delay such check to the end of the chase without compromising completeness. For the moment, we proof that this version of the vips-chase is correct for any set of RGDs Σ, initial data state I, and structural events E.

Property 22. Vips-chase computes canonical model sets

Given a set of RGDs Σ, a data state I, a set of structural events E, assume that *Result* is the result of applying algorithm 11 with parameters Σ, I, E, $\emptyset$, $\emptyset$, *Result*. Then, *Result* is the canonical model set of $I \cup E$ w.r.t. Σ.

Proof. We need to prove that *Result* is sound, universal and minimal as defined in Definition 15. The soundness and minimality of *Result* are directly guaranteed because of the functions *getViolatedDependency* and *isMinimal*. The proper, avoids adding unsound solutions in the *Result*, and latter avoids adding non-minimal solutions. We lack to prove universality.

To do so, we benefit from a close relationship between vips-chase and the CQC algorithm [50]. Briefly, CQC is an algorithm that builds a tree search space for finding the models of a set of denials based on the vips-approach.

In particular, we prove *universality* through the following argument: (1) A CQC-tree for Σ and goal $I \cup E$ is universal, and (2) The vips-chase with parameters Σ, I, E is a traversal of a CQC tree for Σ and $I \cup E$. We prove all these steps separately.

A CQC-tree is universal for Σ and goal $I \cup E$ in the sense that every minimal model containing $I \cup E$ for Σ is contained in the CQC tree. We proof so by contradiction benefiting from the fact that the CQC-tree is known to be complete (i.e., if a finite model containing the goal exists, it appears in the CQC-tree). Suppose that $\mathcal{M}$ is the set of finite models appearing in the tree, and M a minimal model not appearing in $\mathcal{M}$. Now, consider the CQC-tree with the same goal but with constraints $\Sigma \cup \Sigma'$, where Σ' is a set of denial constraints forbidding any (superset) interpretation from $\mathcal{M}$. Note that such constraints can be written using denial constraints with only positive literals (and no negation). Thus, due to the absence of negative literals in Σ' we see that the new CQC-tree is a subset of the previous one (since this new constraints only cuts branches without creating new ones). Moreover, no model is present in the new CQC-tree since none of the previously found models $\mathcal{M}$ satisfies Σ'. However, M is finite model of $\Sigma \cup \Sigma'$ containing $I \cup E$. Thus, the CQC-tree would not be complete, contradiction.

The vips-chase can be seen as a depth traversal of a CQC-tree. In particular, the *getViolatedDependency* function is equivalent to apply and traverse all the possible CQC-expansion B-rules to a given constraint, and then, the different *getRepairsAndBuiltInLiterals* of such violation corresponds to apply all the possible CQC-expansion A-rules to maintain it, and traverse them in depth. Assuming that the *getViolatedDependency* function of the chase is *fair*, then, the CQC-tree generated and traversed by the chase is *fair* too according to the fair notion as defined in [50]. Since a *fair* CQC-tree guarantees that the unique infinite branches are those corresponding to infinite models, the depth CQC-tree traversal done by vips-chase finds all the possible models, unless some model is infinite. In this case, there are two possibilities: such infinite model is not minimal (and thus, the vips-chase cuts this branch through the *isMinimal* check), or such infinite model is minimal (and thus, it is an infinite canonical model). In the last case, the canonical model set is not finite, and thus, the vips-chase is infinitely computing such canonical model set. □

Similarly to the other chase algorithms, the vips-chase cannot guarantee its termination. The source of this phenomenon relies on the hight complexity of the problem the vips-chase can tackle. Indeed, the vips-chase is able to solve the problem of integrity maintenance, and such problem is undecidable, meaning that no terminating algorithm can be built to solve it. We show this in the following property and proof:

Property 23. Incremental Integrity Maintenance is undecidable

Given a set of UML/OCL$_{\text{FO}}$ constraints C, an initial data state I, and some set of structural events E, computing whether there exists a single repair R for C, I and E is undecidable.

Proof. The proof is based on a reduction from the *Class liveliness* problem, which is a well-known undecidable problem in UML/OCL$_{\text{FO}}$ [106].

The class liveliness problem consists in, given a UML/OCL schema and some class Cl, assessing whether there is a finite data state I_{Cl} of the whole UML/OCL schema containing one instance of Cl and satisfying all the constraints.

We can reduce the class liveliness problem to the problem of maintenance in the following way: Consider $I = \emptyset$, and $E = \{\iota Cl(c)\}$, where c is any constant. Now, if there is a repair R for the constraints of the schema, I, and E, we see that $I_{cl} = \mathit{apply}(E \cup R, I)$ is a new finite data state satisfying all the constraints of the schema, and containing one instance of C. Conversely, if there is a finite data state I_{Cl} satisfying all the constraints and containing one instance of $\{Cl(c)\}$, then the set $R = \{\iota A | A \in I_{Cl}\}$ is a repair for E, and I. Thus, we can decide if Cl is lively by checking the existence some repair R for some data state I and structural events E. Thus, since the problem of class liveliness is undecidable, computing whether there is some repair R is also undecidable. □

Property 24. Termination of vips-chase is not guaranteed

Given a set of RGDs Σ, an initial data state I, and a set of structural events E, it is undecidable to know if the vips-chase with Σ, I, E terminates.

Proof. Directly from 23 since we can use the output of vips-chase to know if there exists a single repair R. □

Another way to see such undecidability result is that the vips-chase might hang computing an *infinite canonical model* of the RGDs Σ, and the undecidability of the maintenance problem ensures the existence of such infinite canonics. That is, there exists infinite repairs, and the vips-chase gets stacked computing them.

5.2.4 Improving the vips-chase: the Finite Canonical Property

In the previous section we have defined the vips-chase, an algorithm able to compute the repairs of some constraint violations using the RGDs.

When defining the vips-chase, we have identified two main issues affecting it: non-termination, and the costly minimality computation step. That is, the vips-chase might hang computing a repair in some cases, and it requires applying a costly minimality check at each chase-step to ensure its completeness.

These two problems have the same flavor as the issues affecting the core-chase [40]. Indeed, the core-chase cannot ensure its termination either, and requires a *core* computation at each chase-step to ensure its completeness.

In the case of the core-chase, both problems (termination, and the costly core-computation step) can be solved if, for the given set of dependencies Σ, we can ensure that applying a *standard chase* to Σ terminates. That is, if the standard chase ensures its termination with such dependencies, then, not only the termination is ensured, but the core-computation step can be delayed to the end of the chase to obtain the core solution. Indeed, the core-chase step was introduced to compute some cores that are not detected when the standard chase does not terminate [40].

In a similar way, we show that if a standard chase ensures its termination with a set of RGDs, then, the vips-chase not only ensures its termination, but it can also delay the minimality check to the end of the chase without compromising completeness.

For our purposes, we define such concept as *Finite Canonical Property* (FCP). Intuitively, we say that a set of deds Σ satisfies the FCP if, for any possible data state I, we have that the canonical model set of Σ and I is finite. As we are going to see, this property is, roughly speaking, equivalent to chase-termination.

In the following, we first give the formal definition of FCP, and show that FCP entails vips-chase termination and the possibility to delay the minimality checking step. Next, we show some ways to identify when some schema enjoys FCP.

The Finite Canonical Property

Now we formally define the finite canonical property. Similarly as the *canonical model set* concept, we define this notion for general dependencies (rather than RGDs for augmented schemas), and then, we show its application to RGDs.

Definition 16. Finite Canonical Property

Given a set of dependencies Σ, we say that Σ has the finite canonical property (FCP), iff:

For any data state I, there is a finite canonical model set of I and Σ.

where a canonical model set is *finite* if it is a *finite* set of *finite* models.

Intuitively, the FCP captures the notion of *chase-termination*. Certainly, consider a chase dealing with disjunctions in the RHS of the dependencies, and thus, whose execution has the form of a tree rather than a sequence. In this situation, each branch of the chase builds a model of the dependencies, and thus, a branch might be infinite if it builds an infinite model. However, roughly speaking, the FCP ensures that for each initial data state to start the chase I, the chase-tree built has a finite number of branches of finite depth. Thus, the chase terminates.

For our purposes, we bring an alternative definition of FCP and prove that it is equivalent to the previous one:

Definition 17. Finite Canonical Property (alternative definition)

Given a set of dependencies Σ, we say that Σ has the finite canonical property (FCP), iff for any data state I:

> Given any infinite data state I' s.t. $I \subset I'$ and $I' \models \Sigma$, then, there is a finite data state M s.t. $I \subset M \subset I'$ and $M \models \Sigma$

Continuing with our previous analogy with the chase-tree, this alternative definition is saying that, for any data state I, any branch that might built an infinite model for Σ and I (such as I'), terminates building a finite model (M).

In the following, we formally proof that both definitions are equivalent:

Proof. It is easy to see from the definition of *canonical model set* that the first definition of FCP implies the second one. So, we concentrate on proving the converse. We do so by contradiction. Assume that we have a set of dependencies Σ satisfying Definition 17, but not satisfying Definition 16. That is, assume that there is some I s.t. Σ and I has an infinite canonical model set $\mathfrak{U}$, but for which for any $I' \supset I$ s.t. $I' \models \Sigma$, there is a finite M s.t. $M \models \Sigma$ and $I \subseteq M \subseteq I'$.

First, we are going to see that each canonical model in $\mathfrak{U}$ is finite. We prove so by contradiction. Assume that there is an infinite canonical model in $U \in \mathfrak{U}$. Then, by Definition 17, we know that there is another finite model M that is a subset of U and that satisfies Σ. By universality of $\mathfrak{U}$, such M is also captured in $\mathfrak{U}$. Assume M' to be the canonical model in $\mathfrak{U}$ capturing M. Then, clearly, M' also captures U. Since M' and U are both in $\mathfrak{U}$, $\mathfrak{U}$ is no longer minimal, contradiction.

Now, we are going to build a contradiction from the fact that every canonical model in $\mathfrak{U}$ is finite. In particular, we are going to see that, if $\mathfrak{U}$ is an infinite collection of finite canonical models, then, we contradict the compactness theorem of 1st order logics [113]. To do so, we build a first order logic theory Σ' in the following way: for each canonical model $U \in \mathfrak{U}$, add a new axiom forbidding the model U, and any captured model of U (broadly speaking, we can obtain such axiom by writing $\Sigma^C \wedge \neg U^C$, where Σ^C is the conjunction of all the dependencies in Σ, and U^C is the conjunction of all atoms in U replacing the labelled nulls for existentially quantified variables). Then, Σ' is an infinite inconsistent 1st order theory (it has no model since we forbid all its models), but any finite subset of it is consistent (indeed, if we remove the constraint $\Sigma^C \wedge \neg U^C$, U becomes a model of the theory). Since the compactness theorem says that any infinite inconsistent first-order logic theory has a finite inconsistent subset, we reach a contradiction. □

Once we have established these two definitions of FCP, we find necessary to stop in them to make a comparison with the *Finite Model Property* (FMP). Roughly speaking, a set of dependencies Σ has the finite model property if, for any data state I, if there is some infinite data state I' s.t. $I \subset I'$ and $I' \models \Sigma$, then, there is a finite M s.t. $I \subseteq M$ and $M \models \Sigma$. Thus, the unique difference between FCP and FMP is that FCP requires such M to be also a subset of I'. Thus, FCP implies FMP. However, the converse is not true. Indeed, consider the typical set of dependencies saying that each person has a father, and each father is a person. Such set of dependencies has the finite model property. Intuitively, from any possibly finite data state I, we can always build a finite model M by making some people be their own fathers. In contrast, note that for the infinite model I' in which everyone has a different father, and where the father relationship has no cycles, every finite subset of I' containing I is not consistent, thus, such set of dependencies does not enjoy FCP.

Now, we establish the two results we were pursuing: (1) FCP ensures vips-chase termination, and (2) FCP permits delaying the minimality check to the end of the chase.

Property 25. Vips-chase terminates for RGDs enjoying FCP

Given a set of RGDs Σ satisfying FCP, for any given initial data state I and set of structural events E, vips-chase with input Σ, I, E, terminates.

Proof. In the proof of vips-chase correctness, we have seen that vips-chase with input Σ, I, E terminates if the canonical model set is finite. So, if Σ enjoys FCP, then, by definition Σ, $I \cup E$ has a finite canonical model set. Thus, the vips-chase terminates with input Σ, I, E. $\square$

Property 26. Vips-chase can delay the minimality check for RGDs enjoying FCP

Given a set of RGDs Σ satisfying FCP, for any given initial data state I and set of structural events E, vips-chase with input Σ, I, E delaying the minimality check to the end of the chase terminates and correctly computes its canonical model set.

Proof. In the proof of vips-chase correctness, we have seen that the minimality check step was only necessary to deal with the non-minimal infinite chase branches. However, FCP ensures that all the branches are finite. This is because any infinite model I' that might be computed from a chase-branch eventually stops with a finite model M. Thus, we can remove such checking from the middle of the chase without compromising termination. However, in order to ensure that the result is a true *canonical model set*, we need to apply a postprocessing to check that each instance in the canonical model set is, indeed, minimal. □

Identifying conditions ensuring FCP

Identifying if a set of dependencies Σ enjoys FCP is clearly undecidable. This is because it is undecidable to know if the chase is going to terminate for a given set of dependencies Σ.

However, we can identify some sufficient conditions ensuring FCP (although such sets of conditions will never be complete).

In particular, we can see that the dependency acyclicity conditions stated in [103, 107] ensures FCP. Roughly speaking, their conditions are sufficient to state when chasing a set of dependencies Σ terminates for any initial data state I. Thus, since a set of dependencies can only ensure its chase termination if it enjoys FCP, these conditions ensure FCP. We briefly review such method for the self-containment of the thesis:

Intuitively, this approach builds a *dependency graph* from Σ. In this graph, nodes represents dependencies, and arcs represents the violations of some dependency that might occur when chasing another dependency. Then, a set of dependencies enjoys FCP if all its cycles of the dependency graph are *finite*. Roughly speaking, a cycle is said to be finite if chasing the cycle is ensured to terminate, which is ensured if one of the following conditions holds:

1. For each arc between dependencies, the existential variable of the first dependency are not propagated to the LHS of second dependency. Intuitively, this means that, during the chase, the newly generated objects to maintain the first constraint do not cause new extra dependency violations. Thus, the number of violated dependencies cannot increase during the chase, which means that the chase eventually terminates.

2. There is some predicate in the LHS of some dependency not present in the RHS of any dependency of the cycle. Intuitively, this means that, during the chase, the newly generated objects cannot cause the violation of one of the dependencies present in the cycle, which, eventually, breaks the maintenance/violation cycle that cause the non-termination of the chase.

3. Maintaining the dependencies using new fresh labelled nulls reaches a fixed-point in one iteration. Intuitively, this condition tries to emulate the worst

case of the chase in which new fresh values have to be generated to repair all the dependencies of the cycle. However, if during such process a fixed-point is reached (that is, no dependency is violated in the first cycle iteration), this guarantees that any chasing execution of such cycle always find such fixed-point, and thus, terminate too.

We encourage the interested reader to check [103, 107] to read a formal description of the conditions. In the following, we formally state that they are sufficient to entail FCP.

Property 27. Aciclicity conditions ensuring FCP

Consider a set of dependencies Σ satisfying the acyclicity conditions stated in [103, 107]. Then, Σ satisfies FCP.

Proof. The proof is based on the fact that such conditions are proved to be sufficient to ensure the CQC algorithm termination. In particular, we show that if they did not entail also FCP, then, the CQC algorithm would not ensure its termination, which would be a contradiction.

Assume that Σ does not enjoy FCP, thus, for some data state I there is some infinite canonical model $M \supset I$. Then, consider that we run the CQC method with input Σ and I. Because of the completeness of the CQC method, CQC is going to compute the canonical model M, thus, making CQC not to terminate. Nevertheless, such conditions have been proved to ensure CQC termination [107]. □

In Table 1 we summarize the most notable results we have stated so far w.r.t. integrity maintenance. In particular, we state, for each OCL subset we have considered (OCL_{UNIV} and OCL_{FO}) which logic rules do we create to maintain its constraints, which kind of model set do we compute as a solution, which algorithm do we use to compute such model set, and whether the termination of the algorithm is ensured or not.

Table 1. Integrity maintenance summary for OCL

Lang.	Logic Rules	Solution	Algorithm	Termination
OCL_{UNIV}	RGDs no exists.	Univ. Model Set	chase	ensured
OCL_{FO}	RGDs with exists.	Can. Model Set	vips-chase	if FCP

Note that, so far, our method is intended to compute all the possible repairs of the constraints. However, since it might be the case that a domain expert can decide that some of these repairs are not desired, we aim at customizing these RGDs so that they only compute desired repairs.

5.2.5 Customizing RGDs

Now, the idea is to customize the RGDs in order to avoid *undesired* repairs. Indeed, since we are focused on computing all the repairs R, we might find a repair R that repairs a constraint violation in an inappropriate way according to the domain. This is, for instance, the case in which we repair the *SeenIsBought* constraint by deleting the fact that some user u has seen some content c. In this domain, it is impossible to *unsee* some content once it has been watched.

As we have previously pointed, the basic idea to customize the RGDs is to move the undesired structural event literals from the RHS to the LHS of the RGDs, and adding a negation symbol to them. Recall that if we move all the repairs from the RHS to the LHS, we obtain again the original EDC. This is the case of the previous mentioned example, in which if we take out the structural event for deleting the *seen* fact from the RHS in RGD 5.3 and place it again in the LHS we obtain the original EDC 4.5. Another possibility is to move some of the structural events to the LHS, but leave some other in the RHS as we have seen in RGD 5.18.

Intuitively, a chase-like algorithm can deal with these customized RGDs. Indeed, it is just a matter of ignoring a possible repair of a violation (the one placed in the LHS) if several repairs are present in the RHS, or reporting that no solution is found in such branch if no other repairs are present in the RHS or the rule.

To prove so, we first require modifying the semantics of the concept of RGD repair to coincide with our intended semantics. Indeed, consider the following simplified singleton set of RGDs[1]:

$$\iota s \wedge \neg \iota u \rightarrow \iota r$$

Assume that $E = \{\iota s\}$. Then, according to the RGD repair definition given in Definition 14, $R = \{\iota r\}$ and $U = \{\iota u\}$ are both repairs since $E \cup R$ and $E \cup U$ satisfies the RGDs and are minimal. However, intuitively, only R is a *true* repair. This is because ιr is *supported* by ιs, whereas ιu is *unsupported*, that is, there is no RGD justifying its insertion since it does not appear in the RHS of any RGD whose LHS evaluates to true.

Moreover, in order to find all the possible repairs, we need to apply the chase in

[1]For the sake of simplicity, we have taken out from these RGDs all the non-structural literals, and consider literals with 0-arity.

a particular order. Indeed, consider the following set of RGDs:

$$\iota s \wedge \neg \iota u \rightarrow \iota r \tag{5.19}$$

$$\iota p \wedge \neg \iota r \wedge \neg \iota u \rightarrow \iota v \tag{5.20}$$

$$\iota q \wedge \iota v \rightarrow \bot \tag{5.21}$$

Assume that $E = \{\iota p, \iota q, \iota s\}$. If we chase such RGDs starting from RGD 5.20, we are not going to find any solution. Indeed, after instantiating ιv to repair such RGD, we violate the RGD 5.21, which cannot be repaired. In contrast, if we start chasing from RGD 5.19, we are going to find the solution $R = \{\iota q\}$.

In the following, we first change the semantics of an RGD repair to a new one based on the well-supported semantics notion of logic programs [46]. Afterwards, we show that his new kind of repair can be computed using a chase-like procedure that chases the RGDs in a prefixed order.

Well-supported definition of RGD repair

Intuitively, we want a set of structural events and auxiliary literals $R \cup A$ to be considered *well-supported* if they can be produced by means of chasing some RGDs Σ with an initial data state I and structural events E. That is, if any of the literals in $R \cup A$ appears in the RHS of some RGD whose LHS evaluates to true using those literals in $R \cup A$ created in previous chase steps. This is the case if and only if there can be a strict total order $<$ among the literals in $R \cup A$ recording the time precedence they have in the chase. This leads to the following definition (adapted from the well-supported model definition from logic programs [46] to RGDs):

Definition 18. Well-supported instances of RGDs

Given a set of customized RGDs Σ, a data state I, and a set of structural events E, a set of structural events and auxiliary predicates $R \cup A$ is *well-supported* with respect to $I \cup E$ and Σ iff there is a strict total order $<$ in $R \cup A$ s.t.:
For each $r \in R \cup A$ there is some RGD $\xi \in \Sigma$ s.t. for some ground substitutions σ, we have $I \cup E \cup R^{<r} \cup A^{<r} \models LHS(\xi)\sigma$ and for some ground substitution ρ, r appears in $RHS(\xi)\sigma\rho$, where $R^{<r} \cup A^{<r}$ is the set of instances from $r' \in R \cup A$ s.t. $r' < r$.

Now, we define a customized RGD repair to be a minimal set of well-supported structural events that makes all the RGDs evaluate to true.

Definition 19. Customized RGDs repair

Given a set of customized RGDs Σ, a data state I, and a set of structural events E, a set of structural events R is a *repair* for $I \cup E$ and Σ iff:

- There exists some set of instances A for the auxiliary predicates $\iota containX$ and $\iota forbidX$ s.t. the instances in $R \cup A$ are *well-supported* and $I \cup E \cup R \cup A \models \Sigma$
- There is no such set of instances A for a subset $R' \subset R$.

Computing repairs for customized RGDs: the stratified-chase

Similarly as before, because of the existential variables in RGDs, the number of RGD repairs might be infinite. Thus, it is better to capture them using some kind of *universal model set*, rather than trying to enumerate all the possibilities.

In the previous subsection, we have used the notion of *canonical model sets* for that purpose. However, the canonical model sets do not take in account the well-supported notions we have previously defined. That is, a canonical model set captures unsupported solutions, which we are not interested in.

Thus, we now define the concept of *well-supported canonical model set*. The definition only extends the previous definition of canonical model set to make every element in the set to be well-supported.

Definition 20. Well-Supported Canonic Model set

Given an initial set of instances I and some set of dependencies Σ, a well-supported canonical model set $\mathcal{U}$ is a set of pairs $\langle U,\ B\rangle$, where U is a set of instances and B is a set of built-in literals about its labeled nulls, satisfying the following properties:

– (soundness and well-supportedness) For any $\langle U,\ B\rangle \in \mathcal{U}$, and ground substitution σ s.t. $B\sigma$ is true, we have $I \cup U\sigma \models \Sigma$ and U is well-supported w.r.t. Σ and I.

– (universality) For any M s.t. $I \cup M \models \Sigma$, and M is well-supported, there is some pair $\langle U,\ B\rangle \in \mathcal{U}$, and some ground substitution σ for the labelled nulls in U s.t. $U\sigma \subseteq M$ and $B\sigma$ is true.

– (minimality) For every $\langle U_1,\ B_1\rangle \in \mathcal{U}$ and substitution σ_1 for the labelled nulls in U_1 s.t. $B_1\sigma_1$ is true, we have that for any other $\langle U_2,\ B_2\rangle \in \mathcal{U}$ and substitution σ_2 s.t. $B_2\sigma_2$ is true, $U_2\sigma_2 \not\subsetneq U_1\sigma_1$.

Similarly as before, we relate the notion of well-supported canonical model set to the notion of customized repairs.

Property 28. The Well-Supported Canonic Model Set of some RGDs captures its customized Repairs

Consider a set of customized RGDs Σ. Then, for any data state I, and structural events E we have that:

R is a customized repair for Σ,I,E **iff**
there is some $\langle U,\ B\rangle \in \mathcal{U}^*$ and substitution σ s.t. $R = U\sigma$, and $B\sigma$ evaluates to true.

where $\mathcal{U}^*$ is the well-supported canonical model set of Σ, I, E after removing, for each $U \in \mathcal{U}$, the auxiliary literals $\iota containX$ and $\iota forbidX$, and removing any U_2^* from $\mathcal{U}^*$ if $\mathcal{U}^*$ contains some $U_1^* \in \mathcal{U}^*$ s.t. $U_1^* \subseteq U_2^*$.

Proof. The proof is obtained by extending the proof for Property 21 carrying the concept of well-supported in all the steps. □

With this concept at hand, we only lack to compute the well-supported canonical model set of some customized RGDs to characterize their repairs. We are going to do so with a chase like procedure.

As we have seen before, when we have negated structural events in the LHS of some RGDs, the order in which the chase treats the RGDs matters. That is, starting the chase from some RGD might bring to one solution, whereas starting from another might bring to no solution. A possible way to deal with this phenomenon would be to inspect all the possible orders, but, clearly, this approach might become prohibitive due to its hight complexity.

So, the idea is to establish, at compilation time, which is the order in which the chase should treat the RGDs. We do so assuming a stratification of the RGDs. In the following, we continue by bringing the definition of RGD stratification, and then, give an algorithm that chases such stratified RGDs in a prefixed order:

Definition 21. Customized RGD Stratification

Given a set of RGDs Σ, we say that Σ_1, ..., Σ_n is a stratification of Σ if and only if:

- $\Sigma = \bigcup_{i=1..n} \Sigma_i$
- For each $\xi_i \in \Sigma_i$, no predicate in the RHS of ξ_i appears positively in the LHS of any $\xi_j \in \Sigma_j$ with $j < i$, neither negatively in the LHS of any $\xi_j \in \Sigma_j$ with $j \leq i$.

In Algorithm 11 we show how to chase a stratified set of RGDs $\Sigma = \Sigma_1 \cup$ ... $\cup \Sigma_n$ to compute the well-supported canonical model set, and thus, the customized repairs of Σ. Such algorithm uses the algorithm *vips-chase* to compute the canonical model set for each Σ_i. After this function, another function, minimize, removes those solutions that are not minimal according to Σ_1, ... Σ_n.

Algorithm 11 stratified-chase($\Sigma_1 \cup ... \cup \Sigma_n$, *I*, E)

```
Result := $\{\langle\emptyset, \emptyset\rangle\}$
for all $i$ in $1..n$ do
    newResult := $\emptyset$
    for all $\langle R, B\rangle$ in Result do
        vips-chase($\Sigma_i$, $I$, $E$, $R$, $B$, newResult)
        minimize($\Sigma_1 \cup$ ... $\cup\Sigma_i$,newResult)
    end for
    Result := newResult
end for
```

Now, we show that such algorithm is correct.

Property 29. Stratified-chase computes well-supported canonical model sets

Given a stratified set of RGDs Σ_1, ..., Σ_n, a data state I, a set of structural events E, assume that *Result* is the result of applying algorithm 11 to Σ_1, ..., Σ_n, I, E. Then, *Result* is the well-supported canonical model set of $I \cup E$ w.r.t. Σ.

Proof. The proof is based on induction on the number of strata n. For the base case $n = 0$ (that is, the empty set of RGDs), we have that *Result* is the empty set. Indeed, the empty set is the well-supported canonical model set of the empty set of RGDs (i.e., it is sound, well-supported, universal and minimal). This concludes the proof for the base case.

For the inductive case, we first show that for any $\langle R, B \rangle$ in *Result*, we have that $\langle R, B \rangle$ is indeed sound, well-supported and minimal w.r.t. Σ_1, ..., Σ_{n+1}. Next, we show that *Result* satisfies *universality* (i.e., any other model M is captured by some $\langle R, B \rangle$ in *Result*). In the following we assume that $\langle R, B \rangle$ an arbitrary solution in *Result*.

(Soundness) By construction, $\langle R, B \rangle$ is a superset of some well supported canonical model of Σ_1, ..., Σ_n. Because of the stratification, the newly added ground atoms in $\langle R, B \rangle$ cannot violate any RGD from Σ_1, ..., Σ_n (otherwise, there would be a predicate in the RHS of some $\xi \in \Sigma_{n+1}$ appearing in the LHS of some $\xi \in \cup_{1..n}\Sigma_i$), moreover, by construction, $\langle R, B \rangle$ satisfies every RGD in Σ_{n+1}, thus, $\langle R, B \rangle$ is sound w.r.t Σ_1, ..., Σ_{n+1}.

(Well-supportedness) By construction, $\langle R, B \rangle$ is a superset of some well supported canonical model of Σ_1, ..., Σ_n. Thus, all the instances in $\langle R, B \rangle$ that were already present in this well supported canonical model is still well supported. Now, pick any instance in l from $\langle R, B \rangle$ that is new. We show that it is well-supported by contradiction. If it is not well-supported, this means that by removing such literal l from $\langle R, B \rangle$, together all the literals supported by l, $\langle R, B \rangle$ would be sound, which makes $\langle R, B \rangle$ not minimal w.r.t Σ_{n+1}, but vips-chase only outputs minimal models, contradiction.

(Minimality) Minimality is ensured by virtue of the *minimize* postprocessing.

Now, we lack to show universality, that is, we need to proof that every well-supported canonical model of Σ_1, ..., Σ_{n+1} is captured by *Result*.

(Universality) Pick any well-supported canonical model $\langle R, B \rangle$ of Σ_1, ..., Σ_{n+1} and remove all the literals that are supported by Σ_{n+1} RGDs. In this manner, we obtain a new $\langle R', B' \rangle$ that is, by construction, a model of Σ_1, ..., Σ_n. By definition, such new model should be captured by the well-supported canonical model set of Σ_1, ..., Σ_n, which is computed, by induction hypothesis, in the *n* iteration of the algorithm. Assume that $\langle R_U,\ B_U\rangle$ is the well-supported canonical model of Σ_1, ..., Σ_n that captures $\langle R',\ B'\rangle$. Now, we conclude the proof by showing that $\langle R,\ B\rangle$ is a canonical model for the RGDs Σ_{n+1} and initial instance $\langle R_U,\ B_U\rangle$. Indeed, $\langle R,\ B\rangle$ is a superset of $\langle R_U,\ B_U\rangle$ and satisfies all the dependencies in Σ_{n+1}, moreover, it is minimal (otherwise, it would not be a well-supported minimal model of Σ_1, ..., Σ_{n+1}). Thus, $\langle R,\ B\rangle$ is captured by *Result*, since *Result* contains the canonical model set of Σ_{n+1} with $\langle R_U,\ B_U\rangle$ due to the vips-chase. □

Now, we show that such stratification requirement for the customized RGDs is feasible. That is, there are useful customizations of RGDs satisfying the stratification property. In particular, given any set of RGDs, we see that: (1) customizing the RGDs by moving to the LHS all the structural events of the same predicate is stratified and (2) customizing such RGDs by changing some of them to its EDC form is stratified. The first case, corresponds to assume that it is impossible to create/delete instances from some UML class/association (i.e., considering it *frozen*, or *permanent* [84]). The second case, corresponds to combine checking and maintenance policies.

Property 30. Moving all the structural events of the same predicates from the RHS to the LHS of the RGDs leads to a stratified set of RGDs

Given any set of RGDs Σ, and set of structural event predicates P we want to forbid, customizing the RGDs by moving to the LHS all the structural events of the predicates in P leads to a stratified set of RGDs of only 1 strata.

Proof. By construction, for each $\xi \in \Sigma$, all the predicates in its RHS only appears positively in the other RGDs, and no one appears negatively in any other RGD. Moreover, considering that there is only one strata, it is impossible to have some predicate in the RHS of ξ appearing positively in the LHS of dependency from a previous strata. □

This implies, for instance, that in our running example we could customize the repairs to avoid deleting *Visualizes* instances since, conceptually, the association visualizes is *permanent* (e.g. once some user has seen, for instance, *Suicide Squad*, it is impossible for such user to *unsee Suicide Squad*, no matter what he/she does). We can achieve such customization by moving from the RHS of the RGDs all the structural events of the predicate δ*see* to its LHS.

Moreover, since there is only one strata, we can compute the repairs of such RGDs directly computing its *canonical model set*, directly. This is because the *stratified-chase* is only going to do a single invocation to the vips-chase to compute the *canonical model set*.

We now move to the next useful stratified RGD customization:

Property 31. Changing some RGDs for its EDCs leads to a stratified set of customized RGDs

Given any set of RGDs Σ, any customization consisting in changing some RGDs for their original EDCs leads into a stratified customization of RGDs.

Proof. We proof so by constructing the stratification. In particular, consider the stratification consisting of 2 strata: the first strata is for the remaining RGDs, and the second strata is for the EDCs. It is easy to see that this is a correct stratification since the EDCs, by definition, do not have any literal in its RHS. □

Thus, we finally rich the following result:

Property 32. Checking and Maintenance Policies can be Combined

We can combine incremental integrity checking and maintenance policies using a single method.

Proof. Directly from Properties 31 and 29. □

In Table 2 we complete the previous summary Table 1 including the customization possibilities of our method. It is important to highlight here that the algorithm used to compute the solution for a set of customized RGDs depends on the kind of language used. That is, if the customized RGDs comes from a set of OCL_{UNIV} constraints, then, the stratified chase applies a stratified version of the normal chase, however, it applies a stratified version of the vips-chase if the RGDs comes from OCL_{FO}.

Table 2. Integrity maintenance summary for OCL with customization

Lang.	Logic Rules	Solution	Algorithm
OCL_{UNIV}	RGDs no exists.	Univ. Model Set	chase
OCL_{FO}	RGDs with exists.	Can. Model Set	vips-chase
$\text{OCL}_{\text{UNIV}}/\text{OCL}_{\text{FO}}$	Custom. RGDs	W.S. Model Set	stratified-chase

5.3 C# Implementation Experiments

To prove the feasibility of our approach and to analyze its efficiency, we have developed a prototype tool of our method and applied it to compute repairs in several situations related to a particular case study: the well-known EU-Car Rental UML/OCL schema [31]. We have considered two different scenarios to perform our experiments: the original version of the schema and also a simplified one, limited to constraints encodable in OCL_{UNIV} (that is, without existential variables). We refer to the former as EU-Car Rental schema and the latter as EU-Car Rental OCL_{UNIV} schema.

In particular, the goal of our experiments is to analyze the efficiency and the scalability of our approach, according to the following criteria:

- *Size of the initial data state*. We want to analyze to what extent our method scales up with the size of the data state.
- *Whether the applied structural events* do *or* do not *cause any violation*. We want to measure the time required for computing a repair when some constraint is violated, but also the time consumed when there is no constraint violation.
- *Whether the applied structural events are insertions or deletions*.
- *Whether the constraints considered have existential variables or not*.

In the following, we first explain the UML/OCL schema used and its initial data state, then, we discuss the experiment design, and finally, we discuss the results.

5.3.1 Experimenting UML/OCL Schema

The EU-Car Rental system is aimed at specifying a fictional car rental company with the purpose of managing the *rentals* agreed, its *customers*, the rented *cars* and the different *branches* of the company, among other concepts.

We have used the EU-Car Rental schema that appears in [31] as the first schema for our case study. This schema has 21 classes/associations, 17 attributes and 74 explicit constraints (15 OCL constraints, 2 subtyping constraints and 57 min/max cardinalities).

In addition, we have considered the EU-Car Rental OCL_{UNIV} schema. This schema is the same EU-Car Rental schema but removing those constraints that cannot be encoded in OCL_{UNIV}. This restriction only implied the deletion of all the minimum cardinality constraints, since all the rest were perfectly encodable in OCL_{UNIV}. Thus, we came up with a total of 47 explicit constraints (15 OCL constraints, 2 subtyping constraints and 30 max. cardinalities).

5.3.2 Experiment Design

To perform these experiments, we have implemented a vips-chase procedure by modifying a previously existing C# tool: SVTe [49]. Roughly speaking, SVTe is a first order logic satisfiability checker based on constructing a model of the logic constraints using the VIPs approach. For our purposes, we have adapted SVTe to receive as input EDCs, and interpret them as RGDs. In this manner, SVTe tries to build a model chasing the RGD version of each EDC[1]. Moreover, to avoid such chase to compute undesirable repairs, we have modified the tool to create new instances from a restricted set of desirable structural events. In this manner, we emulated the RGD customization capabilities seen in Section 5.2.5.

To create the input for such tool, we translated all the constraints of the EU-Car Rental and the EU-Car Rental OCL_{UNIV} schemas into EDCs using a self-made Java translator component. From this translation we obtained 437 EDCs and 393 EDCs in 2.58s and 2.71s respectively.

Then, we randomly built data states of increasing size together two sets of structural events: one to create a new rental in that state and another to delete a rental. Just to begin with, we ensured that both structural events sets included all the minimum necessary additional structural events to avoid the violation of any constraint.

Afterwards, we iteratively computed the repairs of applying such structural events set to its corresponding initial data state into both schemas: the original EU-Car Rental, and the OCL_{UNIV} version of it. At each iteration, we randomly removed some of the structural events, and computed the time it took for the chase to compute these missing events. In this way, we ensured that all the violations were repairable, and controlled the size of the required repair.

Moreover, we also restricted the structural events that the chase could use for computing such repairs. In particular, for the case of inserting a new rental, we

[1]Other approaches for building chase-like algorithms from scratch for computing repairs were considered [104], but a SVTe customization is the one that gave us the best results due to its already built-in optimizations.

limited the chase to repair the RGDs just by creating new rental instances, instances for its attributes and associations, and possibly new customers. For deletions, we allowed the chase to compute new rental deletions together with their corresponding associations/attributes.

All the experiments were performed in an Intel Core i7-4710HQ up to 3.5Ghz, 8GB of RAM, running Windows 8.

5.3.3 Experiment Results

The results of these experiments are shown in Tables 5.1 and 5.2. The first column in each table indicates the number of instances in the initial data state used. The following columns state the chase execution time in seconds for finding the first repair when the set of events lacks M structural events to be consistent. That is, for the M=0 column, we used a set of structural events which ensured that no constraint was violated. The other columns represents the cases that required computing repairs of size $M = \{2, 4, 6, 8\}$ instances.

Table 5.1 shows that our method scales well in the EU-Car Rental schema when inserting new rentals that do not cause any constraint violation ($M = 0$). As expected, the execution time increases when some violation occurs and additional structural events need to be computed (cases $M > 0$). In this situation, the execution time increases with the size of the repair to be computed.

For deletions, our method takes about 2–3 min to compute repairs for data states of 24,000 instances. Intuitively, these higher execution times are explained because, when some instance of an association is deleted, we need to check in the data state whether some minimum cardinality is violated. This encompasses looking through all the instances of that association in the data state. This phenomenon turns out to be the bottleneck of the chase rather than the size of the repair to be computed. As it can be seen in Table 5.2, the execution times remain almost constant among the size of the repair needed, that is, such execution times are dominated by the size of the data state rather than the number of missing structural events.

On the other side, our method scales nicely when dealing with the version of the schema limited to OCL_{UNIV} for both cases, insertions and deletions. This might be explained because the OCL_{UNIV} version of the EU-Car rental schema have no minimum cardinality constraints, and thus, removing structural events from the initial set of structural events cannot induce their violation, which we suspect is the bottleneck of the technique.

For this reason, we decided to perform a new experiment with the EU-Car Rental OCL_{UNIV} schema with a different strategy for generating the structural events. In particular, we generated totally random structural events sets of N instances, combining both insertions and deletions. We argue that this is the worst case since, when the structural events of an operation are randomly generated, the number of missing events to make it consistent (i.e., the number of structural events of the repair) may grow with each new structural event considered. Note that this case is just theoretical since operations should be cohesive [70], and thus, the events they encompass are not random.

The results of this new experiment are shown in Table 5.3. For each size N of structural events set considered, we show the size of the repair created and its execution time in seconds. In this table, *NR* stands for *no repair* found, meaning that the set of structural events set considered was inherently contradicting the constraints of the schema.

As it can be seen, most executions took less than one second, or just a few seconds. Execution times over 10s occurred with the largest data state of 24,000 instances (i.e., last row) and also with the largest number of structural events (i.e., last column values). Nevertheless, the maximum execution time was up to 1 min.

It is worth saying that we used a tool originally developed for satisfiability checking where only few instances needed to be taken in account. Thus, better results might be expected if considering a dedicated application with big data structure support.

Table 5.1: Execution time in seconds for new rental insertion

	EU-Car Rental					EU-Car Rental OCL$_{\text{UNIV}}$				
IB size	M=0	M=2	M=4	M=6	M=8	M=0	M=2	M=4	M=6	M=8
1,168	0.11	0.11	0.14	0.78	0.58	0.11	0.10	0.12	0.90	0.09
1,804	0.12	0.14	0.11	0.15	0.68	0.12	0.11	0.10	0.13	0.09
3,525	0.14	0.12	0.36	0.60	1.96	0.14	0.10	0.13	0.11	0.10
6,016	0.16	9.75	10.7	1.35	2.50	0.15	0.12	0.10	0.11	0.10
12,567	0.21	0.27	0.24	14.6	8.29	0.20	0.21	0.18	0.14	0.12
24,834	0.38	31.0	24.3	147	19.0	0.42	0.28	0.22	0.17	0.16

Table 5.2: Execution time in seconds for deleting a rental

	EU-Car Rental					EU-Car Rental OCL$_{\text{UNIV}}$				
I size	M=0	M=2	M=4	M=6	M=8	M=0	M=2	M=4	M=6	M=8
1,168	0.24	0.28	0.29	0.27	0.26	0.06	0.06	0.06	0.06	0.06
1,804	0.61	0.56	0.51	0.57	0.56	0.06	0.06	0.06	0.06	0.06
3,525	2.32	2.30	2.04	2.21	2.03	0.07	0.07	0.06	0.07	0.06
6,016	7.10	5.98	7.10	7.49	5.73	0.15	0.07	0.07	0.07	0.07
12,567	35.8	33.2	32.6	35.1	34.3	0.09	0.09	0.08	0.10	0.08
24,834	161	141	187	146	147	0.12	0.10	0.12	0.11	0.11

Table 5.3: EU-Car Rental OCL_{UNIV} results for random insertions/deletions

	N=2		N=4		N=6		N=8	
I size	Rep.	Time	Rep.	Time	Rep.	Time	Rep.	Time
1,052	11	0.70	4	0.07	NR	0.09	NR	0.11
1,877	5	0.14	2	0.15	NR	0.07	NR	0.10
3,292	11	1.30	NR	0.09	NR	0.07	15	0.55
6,539	3	0.12	NR	0.08	18	2.99	28	59.7
11,739	3	0.61	6	6.51	NR	2.51	22	55.5
24,272	3	14.1	0	0.04	11	19.2	NR	12.6

5.4 IDEFIX: A Tool for IDEntifying missing structural events to FIXing-up operation contracts

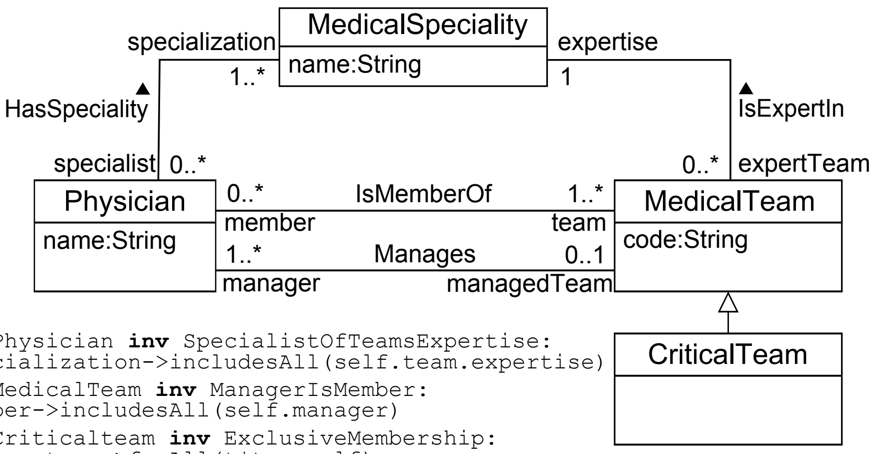
Up to here, we have developed and tested a method for incrementally maintaining constraints when updating data. However, as we have already discussed in the thesis Introduction, the *reasoning* ability to maintain constraints is a powerful tool that enables solving other *reasoning tasks*. In particular, as we are going to see, we can use such technique to help domain experts on fixing-up non-executable operation contracts. We explain the problem and contribution through an example.

Consider the UML class diagram in Fig. 5.2. This class diagram states information about medical teams, their expertise, and membership/management relations with medical teams. The OCL constraints provide additional semantics. *SpecialistOfTeamsExpertise* ensures that a physician is not a member of a team if he does not have the speciality of the team. *ManagerIsMember* states that all managers of a medical team must also be members of the team. Finally, *ExclusiveMembership* states that members of a critical team may not be members of other medical teams.

Consider now the following UML/OCL operation contracts aimed at inserting and at deleting an instance of a *Critical Team*, respectively:

Operation: newCriticalTeam(p: Physician, s: MedicalSpeciality, cd: String)
pre: MedicalTeam.allInstances()->forAll(m|m.code<>cd) and p.specialization->includes(s) and p.managedTeam->isEmpty()
post: CriticalTeam.allInstances()->exists(c|c.oclIsNew() and c.code = cd and c.expertise = s and c.manager->includes(p))

Operation: deleteCriticalTeam(criticalTeam: CriticalTeam)
post: CriticalTeam.allInstances()->excludes(criticalTeam)
–we assume that deleting an instance of a class also deletes its links to other instances.



```
context Physician inv SpecialistOfTeamsExpertise:
self.specialization->includesAll(self.team.expertise)
context MedicalTeam inv ManagerIsMember:
self.member->includesAll(self.manager)
context Criticalteam inv ExclusiveMembership:
self.member.team->forAll(t|t = self)
```

Figure 5.2: A UML/OCL schema for the domain of medical teams

Both operations are non-executable since their execution always violate some constraint. Indeed, trying to execute *newCriticalTeam* violates the *ManagerIsMember* constraint, and executing *deleteCriticalTeam* violates the minimum cardinality 1 of the team role. This last violation occurs because *ExclusiveMembership* forces all employees of a critical team to be members only of that team. So, those physicians will have no team when a critical team is deleted through this operation.

Several approaches have been proposed to identify non-executable operations [15, 17, 58, 102, 105], either by automatic reasoning or manual testing, and most of them should be able to determine the non-executability of the previous operations. However, to our knowledge, none of them is able to provide the designer with additional information on how to modify the operation contracts to make them executable.

This has lead us to the development of IDEFIX, a tool that uses our method for incremental integrity maintenance to identify non-executable operations and provide information about how to fix up the problem. This information is given in terms of the missing structural events on the operation postcondition that allow ensuring that all constraints are satisfied after executing the operation. For instance, IDEFIX can tell us that the *newCriticalTeam* operation lacks to specify that the manager of the new created team should also be inserted as a member of such team and, moreover, due to the *ExclusiveMembership* constraint, the operation should also specify the deletion of its previous team membership relations.

In the following, we first formalize the problem, then, we briefly describe the IDEFIX tool under the perspective of a user (that is, its input and output), and finally, we present the tool architecture.

5.4.1 The Problem: Fixing-up Operation Contracts

Pursuing the correctness of a conceptual schema is a key activity in software development since mistakes made during conceptual modeling are propagated throughout the whole development cycle, thus affecting the quality of the final product. The high expressiveness of conceptual schemas requires to adopt automated reasoning techniques to support the designer in this important task.

The conceptual schema includes structural as well as behavioral knowledge. The structural part of the conceptual schema consists of a taxonomy of classes with their attributes; associations among classes; and integrity constraints, which define conditions that the instances of the schema must satisfy [84]. In UML [86], we represent structural schemas by means of class diagrams, with its graphical constraints, and by a set of user-defined constraints, usually specified in OCL [87]. The data state (aka information base) contains the instances of the structural schema. We say that a data state is consistent if it does not violate any integrity constraint.

The behavioral part of a conceptual schema contains all operations required by the system. Each operation is defined by means of a contract, which states the changes that occur on the data state when the operation is executed. In UML, an *operation contract* is specified by a set of *pre/postconditions*, which states conditions that must hold in the data state before/after the execution of the operation. Such pre/postconditions are usually specified in OCL too, but using an heuristic interpretation of their postcondition we can obtain an imperative specification based on structural events [16].

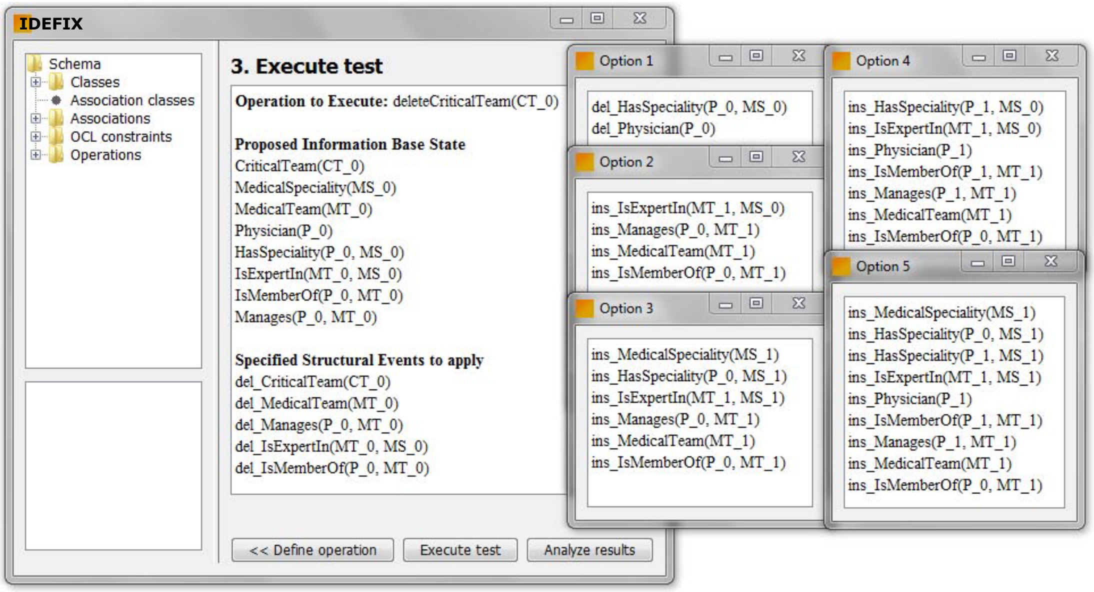
An operation is *executable* if there is at least one consistent data state I such that: I satisfies its preconditions, and applying in I the structural events specified in the postcondition leads to a new consistent state. A non-executable operation is useless since it can never be applied, and the designer should avoid them by modifying its contract.

5.4.2 IDEFIX: A User's Perspective

With IDEFIX, users can fix-up the operation contracts of some conceptual schema by means of the following steps: (1) loading a conceptual schema, (2) selecting an operation to analyze (3) optionally modifying the automatically generated initial data state in which to apply the operation, (4) inspecting the repairs proposed.

In the first step, a user loads into IDEFIX a conceptual schema written in the *XMI* format. In the current version, IDEFIX can read UML/OCL$_{\text{FO}}$ conceptual schemas written in the ArgoUML modelling tool[1].

[1] ArgoUML is an open source modelling tool available for free at http://argouml.tigris.org/



3. Execute test

Operation to Execute: deleteCriticalTeam(CT_0)

Proposed Information Base State
CriticalTeam(CT_0)
MedicalSpeciality(MS_0)
MedicalTeam(MT_0)
Physician(P_0)
HasSpeciality(P_0, MS_0)
IsExpertIn(MT_0, MS_0)
IsMemberOf(P_0, MT_0)
Manages(P_0, MT_0)

Specified Structural Events to apply
del_CriticalTeam(CT_0)
del_MedicalTeam(MT_0)
del_Manages(P_0, MT_0)
del_IsExpertIn(MT_0, MS_0)
del_IsMemberOf(P_0, MT_0)

Option 1
del_HasSpeciality(P_0, MS_0)
del_Physician(P_0)

Option 2
ins_IsExpertIn(MT_1, MS_0)
ins_Manages(P_0, MT_1)
ins_MedicalTeam(MT_1)
ins_IsMemberOf(P_0, MT_1)

Option 3
ins_MedicalSpeciality(MS_1)
ins_HasSpeciality(P_0, MS_1)
ins_IsExpertIn(MT_1, MS_1)
ins_Manages(P_0, MT_1)
ins_MedicalTeam(MT_1)
ins_IsMemberOf(P_0, MT_1)

Option 4
ins_HasSpeciality(P_1, MS_0)
ins_IsExpertIn(MT_1, MS_0)
ins_Physician(P_1)
ins_IsMemberOf(P_1, MT_1)
ins_Manages(P_1, MT_1)
ins_MedicalTeam(MT_1)
ins_IsMemberOf(P_0, MT_1)

Option 5
ins_MedicalSpeciality(MS_1)
ins_HasSpeciality(P_0, MS_1)
ins_HasSpeciality(P_1, MS_1)
ins_IsExpertIn(MT_1, MS_1)
ins_Physician(P_1)
ins_IsMemberOf(P_1, MT_1)
ins_Manages(P_1, MT_1)
ins_MedicalTeam(MT_1)
ins_IsMemberOf(P_0, MT_1)

Figure 5.3: Snapshot of IDEFIX after computing the missing structural events of the operation *deleteCriticalTeam*

In the second step, a user chooses which operation does he/she wants to fix. When selecting one operation, IDEFIX automatically builds a data state satisfying the precondition of such operation and all the constraints of the schema. This is the initial data state I we are going to use for computing the missing structural events.

Next, in the optional third step, a user can modify the initial data state I. The idea is to allow the user choose the data state I that witnesses the executability of the operation.

In the last step, the tool returns the user the different minimum sets of missing structural events that, if applied together the structural events specified in the post-condition in I, would lead to a new consistent data state. This steps corresponds to the snapshot shown in Figure 5.3.

Then, the user is expected to modify the operation postcondition to include, at least, one of the minimum sets of missing structural events returned by IDEFIX. When doing so, the user guarantees that the operation is executable and that I is a data state that witnesses so.

5.4.3 IDEFIX: Tool Architecture

IDEFIX has been implemented as a standalone Java program. As it can be seen in the architecture shown in Figure 5.4, IDEFIX follows a layered structure in the sense that all user interactions are managed by a *Graphical User Interface* who delegates all the logic application to a *Domain controller*. In the following, we explain the behavior of all the relevant components of the *domain layer* of IDEFIX step by step.

When the user loads the UML/OCL$_{\text{UNIV}}$ schema through an XMI file, IDEFIX loads this schema in its domain controller using the EinaGMC library [1]. Briefly, EinaGMC is a Java library implementing the UML/OCL metamodels, thus, bringing support to inspect the different elements composing a UML/OCL model. In this way, IDEFIX can retrieve all the classes/associations/attributes and constraints of a schema, and traverse them for its translation purposes.

Then, the *UML/OCL to denials* component translates the constraints of the loaded schema into logic denials. The result of such translation is then stored in the domain controller. Afterwards, when the user has selected the operation to fix, the domain controller asks the *Reasoner Controller* to return an initial data state satisfying the previous denials and the precondition posted in the operation selected by the user. For obtaining such data state, IDEFIX follows the process already described and successfully implemented in [96, 102]. Briefly, we translate the negation of the precondition as a new denial and ask a satisfiability checker to create a model for the whole set of denials. For this purpose, we use a customized version of SVTe [49] as our satisfiability checker tool.

Next, the *Denials to EDCs* component translates the denials into EDCs. These EDCs are then loaded into our adapted version of SVTe which interprets them as RGDs. In addition, the domain controller brings to SVTe the initial data state and the set of structural events specified in the operation postcondition. To obtain such structural events from the operation postcondition, IDEFIX uses the patterns described in [102].

Finally, the adapted version of SVTe chases the given constraints, with the provided data state and initial structural events, and computes the repairs. Such repairs are then shown to the user through the *GUI*.

5.5 Related Work

We review the related work found in the literature in three main blocks:

- *OCL approaches*: approaches based on maintaining the data consistent with regards to some OCL constraints.

- *Model Change Propagation Techniques*: approaches based on maintaining a UML/OCL specification consistent (with regards to its metamodel) after some change is applied in the model.

[1]http://guifre.lsi.upc.edu/eina_GMC/

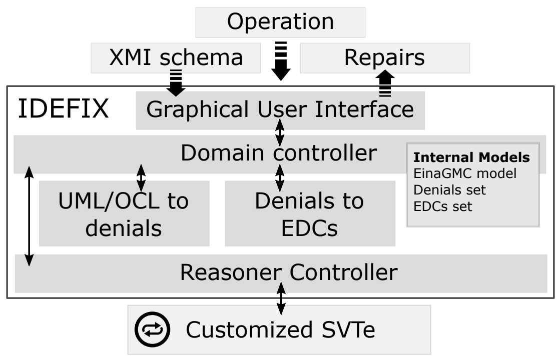


Figure 5.4: IDEFIX architecture

- *Integrity Enforcement Techniques*: approaches based on changing the operations of the behavioral model to ensure that the user cannot execute an operation that might cause a constraint violation.

As we are going to see, there are very few OCL approaches, and none of them is incremental and complete. That is, either they do not focus on repairing the data that might violate some constraint because of the last update, or they are unable to repair all the possible violations.

In contrast, there are some techniques developed for the problem of propagating the changes applied in some model. That is, given a model, some (metamodel) constraint it must fulfill, and some change applied in the model, these techniques are able to compute additional changes that must be applied in the model to satisfy the constraint. The drawbacks that we have seen in these techniques are that (1) they might not be incremental, (2) they might not consider repairs consisting of more than one event, or even (3) they might not take into account that a repair might violate another constraint, thus, bringing results which are not valid.

Then, there are some proposals based on changing the behavioral model rather than the data. The idea of these techniques is to ensure that the operations that the user might apply to modify the data never causes a constraint violation, and thus, no *runtime* integrity checking/maintenance policy is required. However, as we are going to see, these proposal consists in, in fact, *compiling* our integrity checking/maintenance policy in the pre/postcondition of the operations.

Moreover, although almost all the previous refereed work agrees on the importance of automatic mechanisms to ensure the satisfaction of the constraints, only a few make the effort of proving the correctness of their automatic approaches. With this related work, we aim at convincing the reader that, to the best of our knowledge, our proposed technique is the unique one that is fully incremental, complete, and with a formal proof of their correctness.

In the following, we review all these approaches by blocks.

5.5.1 OCL Approaches

The work in [68] consists in a OCL simulation tool called OCLexec. This tool is able to execute an operation specified by means of an OCL pre/postcondition contract without violating any constraint. The method is based on considering the OCL postcondition and the constraints of the schema as a Constraint Satisfaction Problem (CSP), which is then solved by means of a SAT solver. In order to increase the efficiency of the method, their proposal performs a syntactic analysis to find which constraints might be affected by some operation, thus, reducing the number of constraints of the CSP problem. However, they do not perform such analysis with regards to the data. That is, their proposal might check that previously existing data not affected by the operation conforms to the previous constraints. Thus, this approach is not fully incremental.

In contrast, the OCL2Trigger tool relies on triggers [6]. That is, given a set of OCL integrity constraints, they automatically build a set of relational database triggers which detects constraint violations and applies some repairs to maintain them. Clearly, this approach is fully incremental due to the trigger mechanisms. However, in this case, the work presented is only able to repair very simple constraints (essentially, constraints equivalent to disjoint/complete hierarchies), thus, the expressiveness they deal with is quite far from the full OCL_{FO} expressiveness we have tackled in this thesis.

Additionally, it is worth to mention that none of the previous methods offers a mathematical base for its correctness.

5.5.2 Model Change Propagation Techniques

In [81] the authors depart from a technique able to return which model elements violate some (metamodel) constraint, and extend it to return how should the user modify such conflicting elements to repair the constraint violation (i.e., which elements should be created/deleted/modified). To do so, the authors performs a static analysis of the constraints to determine which actions can repair them. However, this approach does not take into account that repairing some constraint might cause the violation of another constraint, neither the case that two repairs might contradict each other (e.g., some element might be proposed for deletion to repair one constraint, whereas it is proposed for modification for repairing another one). Essentially, this problem arises because they treat constraints individually.

In a different way, the Badger tool deals with all the metamodel constraints at the same time to repair some model [100]. Their method is based on the literature of automated planning. Intuitively, a plan is a sequence of repairs leading the model to satisfy all the constraints. Interestingly, the method is implemented using some heuristics to perform a guided search for the plan instead of a blind search. However, the method is not incremental since it takes as input the whole model, without considering which are the last modifications applied in it. A similar approach is also followed by [33], again without being incremental.

The work in [42] is incremental and deals with all the constraints at the same time. Briefly, it integrates the incremental integrity checking in [60] to detect constraint violations after some model change. Then, it tries to change 1 other model element to find some additional change that might make this violation disappear. Clearly, this approach do not consider those repairs composed of more than one event (i.e., changing more than 1 element in the model to repair the violation), and requires manually expressing which parts of the model can change to repair the constraints. Moreover, it cannot identify repairs based on creating new elements.

In contrast, our method is automatic, incremental, and considers all the possible repairs (including those consisting of more than one repair). Similarly as [33, 81], it also permits forbidding some particular ways to repair a constraint that might be undesired according to the domain.

5.5.3 Integrity Enforcement Techniques

In [98], the authors brings a method for, given the specification of some operation, return the missing effects to make it executable (i.e, not to violate any constraint). The method is *lightweight* in the sense that it is based on a syntactic analysis of the operation definition and constraints involved. On one hand, this makes the method for computing the missing effects efficient, on the other, the kind of constraints they can deal with is very limited in comparison to ours (mainly, they only deal with UML structural constraints such as min. cardinality or hierarchy completeness).

The work presented in [30] describes a method for, given one operation, add the minimal necessary preconditions that ensures that the execution of the operation will not rise any integrity violation. Note that the focus now is on changing the operation precondition, rather than its effects. Their method, however, is limited to deal with the constraint that were defined as constraint patterns in [31] rather than dealing with general constraints that can be written in languages such as OCL.

At this point, it is worth to realize that the previous approaches are essentially compiling an integrity maintenance policy (in the case of [98]) or a checking policy (in the case of [30]) directly in the operation definition. That is, they compile the maintenance policy in the operation effects, or compile the checking policy as preconditions. We argue that compiling the policies directly in the operation definition does not bring any benefit in comparison to our approach. Indeed, although it might be thought that having these policies compiled we might gain execution time efficiency, we deny such possibility since our proposal is incremental, and thus, only applies the necessary checks (and no more). Moreover, we have seen that these approaches cannot deal with the hight expressiveness of OCL.

Mixing both approaches, the work in [115] modifies preconditions and operation effects together to ensure that no operation execution violates any constraint. However, such work is only able to deal with schemas written in the Booster language, which is a much more restricted language for writing constraints compared to UML/OCL. In particular, this work only focuses on association related constraints (e,g., referential integrity constraints and association cardinalities).

Finally, it is worth to mention that there is also some work dealing with the opposite problem. That is, instead of accommodating the operations to the constraints, evolving the constraints to accommodate the operations. An example of this approach is followed in [77] in which they evolve functional dependencies.

5.6 Conclusions

In this chapter we have seen a method for incremental integrity maintenance based on RGDs. RGDs (*repair-generating dependencies*) are some rules that detects when some structural events cause a constraint violation, and which are the necessary additional structural events that repair it. The RGDs are obtained by modifying the previously seen EDCs by moving the negated event literals from the LHS of the rule to the RHS.

Then, a chase-like algorithm can be applied to compute all the possible repairs of a given set of initial structural events. The kind of chase depends on the kind of RGDs involved. When, dealing with RGDs without existential variables, the notion of RGD repair coincide with the notion of *Universal Model Set* [40], and thus, a classic chase dealing with disjunctions in the RHS can be applied. Moreover, such chase ensures its termination. Using this method we can deal with OCL_{UNIV} constraints, a subset of OCL_{FO}.

When dealing with RGDs with existential variables, the notion of RGD repair do not exactly coincide anymore with the *Universal Model Set*. Thus, we have defined the notion of *Canonical Model Set*, which, essentially, completes the Universal Model Sets by adding to each model a set of built-in literals restricting the values that the labelled nulls might take. To obtain such *Canonical Model Sets* we have defined a new chase-like algorithm we call vips-chase which is closely related to the VIPs method defined in [50], and permits creating this additional set of built-in literals for the labelled nulls. Using this method we can deal with OCL_{FO} constraints.

Since repairing OCL_{FO} constraints is not decidable, we have specified a property, *Finite Canonical Property* (FCP), that ensures chase-termination. Moreover, we have shown that the decidability conditions stated in [103, 107] are, in fact, conditions ensuring FCP.

Since the number of repairs for a given set of constraints and structural events might increase exponentially, we have defined a way to customize the way that the RGDs can be repaired. In particular, we permit customizing the RGD repairs by moving some structural events from the RHS of the rule to the LHS. In this manner, such structural events are not considered for repairing the RGD by the chase. In this case, the repairs of the RGDs are the *Well-Supported Canonical Model Sets* of them, which, roughly speaking, extends the notion of *Canonic Model Set* with the notion of *well-supportedness* taken from the logic programming literature. In order to obtain such well-supported canonical model sets, we have to apply a stratified version of the chase.

We summarize all the previous statements in the following Table 3. It is worth to say that the RGD customization can be applied to both, OCL_{UNIV} constraints and OCL_{FO} constraints, and thus, its termination depends on the kind of constraints involved (customizing OCL_{UNIV} RGDs ensures termination, but customizing OCL_{FO} RGDs only ensures termination if the original OCL_{FO} RGDs satisfy FCP).

Table 3. Integrity maintenance summary for OCL

Lang.	Logic Rules	Solution	Algorithm	Termination
OCL_{UNIV}	RGDs no exists.	Univ. Model Set	chase	ensured
OCL_{FO}	RGDs with exists.	Can. Model Set	vips-chase	if FCP
*	Customized RGDs	W.S. Model Set	strat-chase	*

The method has been mathematically proved and experimentally evaluated. In the experiments we have seen that the method is able to complete user transactions (i.e., sets of structural events), when the user forgets some of these events. Because of these results, we have implemented IDEFIX, a tool that helps a user to complete OCL operation contracts not to forget any necessary structural event according to the constraints.

As future work, we would like to improve the way to create the built-in literals of the *Canonical Model Sets*. Indeed, we consider that the vips-chase approach we currently apply cause a bottle neck due to the number of new chase-branches it might create. Additionally, more work on identifying *decidable* subsets of OCL (with respect to the problem of integrity maintenance) could be done, and new conditions ensuring FCP could be established.

Part III

Application in Description Logics

Chapter 6

Application To DL-Lite

Until now, we have worked in the problems of integrity checking/maintenance under the so called *closed-world assumption*. That is, we assumed that we always departed from a finite, complete and consistent initial data state I. Such assumption is typical in the world of databases, or in the UML conceptual modeling community.

However, in the other contexts such as Description Logics, the usual assumption is the so called *open-world assumption*. That is, we assume that we only have a finite but incomplete initial data state I. This incomplete data state I stands for several possible states of the domain I_1, ..., I_n (aka *models*), where the *real* state of the domain is *unknown*.

The open-world assumption carries immediately several difficulties and differences with respect to the closed-world. To begin with, the notion of consistency with regards to constraints completely varies. Whereas in the closed world assumption it is sufficient to evaluate each constraint on I to know whether I is consistent (which is straightforward, since we have the data state I and the constraints), in the open world assumption, following the traditional notion of logic consistency, we have to check whether there exists some model I_j satisfying all the constraints. Note that we are not given such model I_j, but only a subset of it: I.

The immediate idea to deal with the open world assumption might seem to try to first compute all these possible represented instances I_1, ..., I_n, however, this is, by no means, a good option. Indeed, the number of represented instances might be infinite. Moreover, some data states I_j might be infinite by itself. It is worth to mention here that, in the closed world assumption, all data states should be finite to be considered a correct data states (as we need them to be finite to be represented in some database), however, in the open world, the represented data states I_j might be infinite (as we might represent it finitely through I in some database).

Fortunately, when restricting the language of Description Logics to DL-Lite, we find a beautiful connection from the open world to the closed world that permits getting rid of all these problems. Essentially, it is known that we can reduce the problem of integrity checking from DL-Lite (under the open-world assumption), to integrity checking in a relational database (under the closed-world assumption) [20]. This connection permits immediately using all our previous machinery to deal with incremental integrity checking and maintenance of DL-Lite ontologies.

In the following, we focus on the problem of incremental integrity maintenance of DL-Lite ontologies. As we are going to see, maintaining a DL-Lite ontology satisfies some nice properties. In particular, the problem of maintaining a DL-Lite ontology is decidable, and, moreover, its repairs can be computed by means of SQL queries (thus, no chase algorithm is involved).

For doing so, we adopt the terminology of DL-Lite ontologies and thus, refer to our problem as DL-Lite consistent updating.

This also permits us to stress the following fact about of DL-Lite ontologies: the notion of ontology update has no standard yet, and thus, several proposals for such notion exists. For instance, when a user asks for the deletion of some fact $Man(John)$, it is still being discussed if the system should only remove $Man(John)$, or it should also remove anything implying that fact (such as $Husband(John)$), or facts implied by it (such as $Person(John)$).

We are going to see that, using our method of augmenting an schema through event predicates, we can compute consistent updates in DL-Lite under different update semantics. In particular, we are going to show how to build a non-recursive datalog program to compute the repairs of some data update under different semantics. Since these programs are non recursive, they can be immediately implemented through SQL queries. A careful reader should note that the datalog translation we propose consists in, essentially, the closed-world constraints stated in [20] augmented by event predicates and rewritten as repair-generating dependencies.

In the following, we first motivate our problem. Then, we bring some preliminaries about DL-Lite and datalog. We continue with a discussion on different update semantics for DL-Lite. Afterwards, we bring two different datalog programs to compute two different kind of updates in DL-Lite ontologies. Finally, we bring some experiments we did with an SQL implementation of such method, and discuss some conclusions.

6.1 Motivation and Main Results

Our goal here is to study effective techniques to perform (consistent) updates over *DL-Lite* ontologies. In particular, we focus on *DL-Lite*$_A$, which is the most expressive member of the *DL-Lite* family of Description Logics (DLs) [20, 21]. *DL-Lite*$_A$ includes virtually all constructs of the OWL 2 QL profile of the W3C OWL 2 standard. In addition, it includes the most typical cardinality restrictions on the participation in roles of UML class diagrams, i.e., any combination of mandatory participation and functional participation.

The crucial characteristic of *DL-Lite*$_A$ ontologies is that they enable the so-called *ontology-based data access* by virtue of first-order rewritability of query answering, that is, every (union of) conjunctive query over a *DL-Lite*$_A$ ontology can be rewritten into a first-order query to be evaluated over the ABox only (i.e., the individual data) considered as a database. This property, on the one hand, gives us a very low worst-case computational complexity bound w.r.t. data, namely AC^0 data complexity. On the other hand, it gives us a very effective practical technique to deal with ontologies that include very large ABoxes (i.e., a lot of individual data): perform the rewriting; transform the first-order query into SQL, or SPARQL, depending on how data are stored; and perform the resulting query exploiting a data management engine to take advantage of all optimizations available for these standard languages.

When we come to updates over ontologies, several approaches are available in the literature [35, 51, 67, 73]. In particular, we focus our attention to the so-called *instance-level* update: we add and delete (or erase) facts about individuals only. Namely, we change the ABox, while we keep the TBox unchanged. This is the most common form of update in practice, since it is essentially concerned with keeping the intensional part of the ontology fixed, while changing freely the individual data (indeed, the ABox changes are typically frequent whereas the TBox typically evolves slowly). Even in this specific kind of updates, there are sophisticated semantic issues to consider in general. One crucial issue is that, in practice, we need the result of the update to be still in the same language as the original ontology, in order to keep using the same system [73]. The most promising approaches that enjoy this property are the so-called *formula-based* approaches [47, 56, 57, 116], in which the update is seen as a change of the ontology axioms. Again, several forms of *formula-based* instance-level updates have been considered [23, 71, 72, 108]. Interestingly, however, for the DLs in the *DL-Lite* family, virtually all proposals in the literature reduce to two main approaches: the one in which we simply act on the ABox assertions explicitly stated in the ontology, and another one in which we act also on the ABox assertions that are not present but logically entailed through the use of the TBox. Notice that, while the first approach is syntax-dependent (i.e., updating logically equivalent ontologies that are stated through different assertions may give rise to logically different resulting ABoxes), the second one is not. In both cases, the semantics have been clarified, their computational tractability established, and ad-hoc algorithms are available. Though, for both approaches, there are essentially no implemented tools yet.

In this chapter we look again at the problem of instance-level formula-based update in *DL-Lite*$_A$, and we establish a result that may turn out to be crucial to generate efficient implementations: like query answering, updating an ontology is *first-order rewritable*. That is, given an update specification, we can rewrite it into a set of addition and deletion instructions over the ABox which can be characterized as the result of a first-order query. This means that (*i*) updating a *DL-Lite*$_A$ ontology is AC^0 in data complexity, and, (*ii*) updates can be processed by widely used data management engines, e.g., based on SQL or SPARQL. We proof this by showing that every update can be reformulated into a datalog program that generates the set of insertion and deletion instructions to change the ABox while preserving its consistency w.r.t. the TBox. Since the obtained datalog program is non-recursive, it can be further translated as first-order queries over the ABox considered as a database. Exploiting this result, we implement an update component for *DL-Lite*$_A$-based systems and perform some experiments over (a *DL-Lite*$_A$ version of) the LUBM ontology [62] with increasing ABox sizes, showing that the approach works in practice.

As far as we know, this is the first time that the first-order rewritability property for *DL-Lite*$_A$ ontology updating is defined, proved, and empirically evaluated. It is important to mention here that some previous work has been done in the context of RDF triplestores [4, 5], but only for the more restricted case of RDFS (with class disjunctions), which is a proper subset of the expressiveness of *DL-Lite*$_A$, the language we deal with in this paper.

6.2 Preliminaries

In this section, we first present the notion of Description Logic (DL) ontology, then we provide the definition of the specific DL considered in this work, and finally we summarize some datalog basic concepts and notation.

6.2.1 Description Logic Ontologies

Let $\mathcal{S}$ be a signature of symbols for individual (object and value) constants, and atomic elements, i.e., concepts, value-domains, attributes, and roles. If $\mathcal{L}$ is a DL, then an $\mathcal{L}$-ontology $\mathcal{O}$ over $\mathcal{S}$ is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{T}$, called *TBox*, is a finite set of intensional assertions over $\mathcal{S}$ expressed in $\mathcal{L}$, and $\mathcal{A}$, called *ABox*, is a finite set of instance assertions, i.e., assertions on individuals, over $\mathcal{S}$ expressed in $\mathcal{L}$. Different DLs allow for different kinds of concept, attribute, and role expressions, and different kinds of TBox and ABox assertions over such expressions. In this paper we assume that ABox assertions are always *atomic*, i.e., they correspond to ground atoms, and therefore we omit to refer to $\mathcal{L}$ when we talk about ABox assertions.

The semantics of a DL ontology is given in terms of interpretations. An interpretation is a *model* of an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ if it satisfies all assertions in $\mathcal{T} \cup \mathcal{A}$, where the notion of satisfaction depends on the constructs allowed by the specific DL in which $\mathcal{O}$ is expressed. We denote the set of models of $\mathcal{O}$ with $Mod(\mathcal{O})$.

Let $\mathcal{T}$ be a TBox in $\mathcal{L}$, and let $\mathcal{A}$ be an ABox. We say that $\mathcal{A}$ is $\mathcal{T}$*-consistent* if $\langle \mathcal{T}, \mathcal{A} \rangle$ is satisfiable, i.e., if $Mod(\langle \mathcal{T}, \mathcal{A} \rangle) \neq \emptyset$, $\mathcal{T}$-inconsistent otherwise. The $\mathcal{T}$*-closure* of $\mathcal{A}$ with respect to $\mathcal{T}$, denoted $\mathrm{cl}_{\mathcal{T}}(\mathcal{A})$, is the set of all atomic ABox assertions that are formed with individuals in $\mathcal{A}$, and are logically implied by $\langle \mathcal{T}, \mathcal{A} \rangle$. Note that if $\langle \mathcal{T}, \mathcal{A} \rangle$ is an $\mathcal{L}$-ontology, then $\langle \mathcal{T}, \mathrm{cl}_{\mathcal{T}}(\mathcal{A}) \rangle$ is an $\mathcal{L}$-ontology as well, and is logically equivalent to $\langle \mathcal{T}, \mathcal{A} \rangle$, i.e., $Mod(\langle \mathcal{T}, \mathcal{A} \rangle) = Mod(\langle \mathcal{T}, \mathrm{cl}_{\mathcal{T}}(\mathcal{A}) \rangle)$. $\mathcal{A}$ is said to be $\mathcal{T}$-*closed* if $\mathrm{cl}_{\mathcal{T}}(\mathcal{A}) = \mathcal{A}$.

6.2.2 The Description Logic DL-Lite$_A$

The *DL-Lite* family [20] is a family of low-complexity DLs particularly suited for dealing with ontologies with very large ABoxes. It constitutes the basis of OWL 2 QL, a tractable profile of OWL 2, the official ontology specification language of the World Wide Web Consortium (W3C)[1].

We now present the DL *DL-Lite*$_A$, which is one of the most expressive logics in the family. *DL-Lite*$_A$ distinguishes concepts from *value-domains*, which denote sets of (data) values, and roles from *attributes*, which denote binary relations between objects and values. Concepts, roles, attributes, and value-domains in this DL are formed according to the following syntax:

$$\begin{array}{llll}
B \longrightarrow A \mid \exists Q \mid \delta(U) & \quad & E \longrightarrow \rho(U) \\
C \longrightarrow B \mid \neg B & & T \longrightarrow \top_D \mid T_1 \mid \cdots \mid T_n \\
Q \longrightarrow P \mid P^- & & R \longrightarrow Q \mid \neg Q \\
V \longrightarrow U \mid \neg U & &
\end{array}$$

where A, P, and U are symbols in $\mathcal{S}$ denoting respectively an atomic concept name, an atomic role name and an attribute name, $T_1, \ldots, T_n$ are n pairwise disjoint unbounded value-domains, $\top_D$ denotes the union of all domain values. Furthermore, P^- denotes the inverse of P, $\exists Q$ denotes the objects related to by the role Q, $\neg$ denotes negation, $\delta(U)$ denotes the *domain* of U, i.e., the set of objects that U relates to values, and $\rho(U)$ denotes the *range* of U, i.e., the set of values related to objects by U.

A *DL-Lite*$_A$ TBox $\mathcal{T}$ contains intensional assertions of the form:

$B \sqsubseteq C$	(*concept inclusion*)	$E \sqsubseteq T$	(*value-domain inclusion*)
$Q \sqsubseteq R$	(*role inclusion*)	$U \sqsubseteq V$	(*attribute inclusion*)
(funct Q)	(*role functionality*)	(funct U)	(*attribute functionality*)

A concept inclusion assertion expresses that a (basic) concept B is subsumed by a (general) concept C. Analogously for the other types of inclusion assertions. Inclusion assertions that do not contain (resp. contain) the symbols '$\neg$' in the right-hand side are called *positive inclusions* (resp. *negative inclusions*). Role and attribute functionality assertions are used to impose that roles and attributes are actually functions respectively from objects to objects and from objects to domain values.

[1]http://www.w3.org/TR/2008/WD-owl2-profiles-20081008/

Finally, a *DL-Lite* TBox $\mathcal{T}$ satisfies the following condition: each role (resp., attribute) that occurs (in either direct or inverse direction) in a functional assertion, is not specialized in $\mathcal{T}$, i.e., it does not appear in the right-hand side of assertions of the form $Q \sqsubseteq Q'$ (resp., $U \sqsubseteq U'$).

A *DL-Lite*$_A$ ABox $\mathcal{A}$ is a finite set of assertions of the form $A(a)$, $P(a, b)$, and $U(a, v)$, where A, P, and U are as above, a and b are object constants in $\mathcal{S}$, and v is a value constant in $\mathcal{S}$.

We refer to [99] for the semantics of a *DL-Lite*$_A$ ontology. Here, we present an example of one such ontology.

Example 1. We consider a slightly modified version of the LUBM ontology [62] about the university domain. We know that a Person can be either a Professor or a Student, where every Student takes (takesCourse role) at least one Course, and every Professor can be either a FullProfessor or an AssociateProfessor. Finally, we know that john is a FullProfessor and that bob is a Student. The corresponding ontology $\mathcal{O}$ is:

$$\begin{array}{lll}
\mathcal{T} = \{ & \mathsf{Student} \sqsubseteq \mathsf{Person} & \mathsf{Professor} \sqsubseteq \mathsf{Person} \\
& \mathsf{FullProfessor} \sqsubseteq \mathsf{Professor} & \mathsf{AssociateProfessor} \sqsubseteq \mathsf{Professor} \\
& \mathsf{Student} \sqsubseteq \neg\mathsf{Professor} & \mathsf{FullProfessor} \sqsubseteq \neg\mathsf{AssociateProfessor} \\
& \mathsf{Student} \sqsubseteq \exists\mathsf{takesCourse} & \exists\mathsf{takesCourse}^- \sqsubseteq \mathsf{Course}\ \} \\
\mathcal{A} = \{ & \mathsf{FullProfessor(john)},\ \mathsf{Student(bob)}\ \} &
\end{array}$$

□

A notable characteristic of *DL-Lite*$_A$ is that both satisfiability checking and conjunctive query answering are First-Order (FO) rewritable. Intuitively, FO-rewritability of satisfiability (resp., query answering) captures the property that we can reduce satisfiability checking (resp., query answering) to evaluating a FO query over the ABox $\mathcal{A}$ considered as a relational database. We remark that FO-rewritability of a reasoning problem that involves the ABox of an ontology (such as satisfiability or query answering) is tightly related to low data complexity of the problem. Indeed, since the evaluation of a First-Order Logic query (i.e., an SQL query without aggregation) over an ABox is in AC^0 in data complexity [2], the FO-rewritability of a problem has as the immediate consequence that the problem is in AC^0 in data complexity.

6.2.3 Datalog Concepts and Notation

A *term* T is either a *variable* or a *constant*. An *atom* is formed by a n-ary *predicate* p together with n terms, i.e., $p(T_1, ..., T_n)$. We may write $p(\overline{T})$ for short. If all the terms $\overline{T}$ of an atom are constants, we call the atom to be *ground*. A *literal* is either an atom $p(\overline{T})$, a negated atom $\neg p(\overline{T})$, or an inequality $T_i \neq T_j$.

A predicate p is said to be *derived* (or *intensional*) if the evaluation of an atom $p(\overline{T})$ depends on some derivation rules, otherwise, it is said to be *base* (or *extensional*). A *derivation rule* is a rule of the form $p(\overline{T_p}) \leftarrow \phi(\overline{T})$, where $p(\overline{T_p})$ is an atom called the *head* of the rule, and $\phi(\overline{T})$ is a conjunction of literals called the *body*. All derivation rules must be *safe*, i.e., every variable appearing in the head or in a negated or inequality literal of the body should also appear in a positive literal of the body. Additionally, all the predicates must be stratified, i.e., it should be possible to partition the set of predicates P into several pairwise disjoint *strata* $P_1 \cup ... \cup P_m$ s.t. for each predicate $p \in P_i$, each predicate appearing in the derivation rules of p should belong to a stratum P_j with $j < i$, if it appears in a negated literal, or, $j \leq i$, if it only appears in positive literals.

Finally, a *datalog program* is a set of derivation rules together with a set of *facts*, where a fact is a ground atom of a non-derived predicate.

6.3 Formula-Based Approach for Updating DL Ontologies

In the following, we first present the intuitions on ontology update, then we define two distinct formula-based update semantics, and we argue that, for the case of *DL-Lite*$_A$, these two semantics capture virtually all other formula-based update semantics proposed so far. Then, we show that the *careful semantics*, a different formula-based update semantics proposed in the literature, is not uniquely defined in the case of *DL-Lite*$_A$, contradicting a result stated in [23], which makes this update semantics inappropriate in our approach due to its inherent nondeterminism.

6.3.1 Update Semantics for DL-Lite$_A$

In the formula-based approaches to the update, the objects of change are sets of formulae. That is, the result of the change is explicitly defined in terms of a formula, by resorting to some minimality criterion with respect to the formula expressing the original ontology.

Thus, an *update* is a set $\mathcal{U}$ of operations of two types: insertion operations, denoted by i(α), and deletion operations denoted by d(α), where α is an ABox assertion. Intuitively, updating a consistent ontology with an insertion operation i$(A(o))$, where $A(o)$ is a concept ABox assertion, means changing the extensional level of the ontology in such a way that the ontology resulting from the update is still consistent and entails the fact $A(o)$. Conversely, updating a consistent ontology with a deletion operation $\mathsf{d}(A(o))$, means changing the extensional level of the ontology in such a way that the ontology resulting from the update is still consistent and does not entail the fact $A(o)$.

After adding new facts into an ontology, one may find that the revised ontology becomes inconsistent. A strategy to overcome such a situation is to remove part of the original ABox to the aim of preserving consistency. Similarly, if the goal is to update the ontology by deleting a fact, we might need to retract several facts from the original ABox that entailed it. When applying these modifications to the original ABox, one should respect the *minimal change principle*, a widely accepted principle of the knowledge base evolution literature [43, 52, 66]. This principle states that the ontology resulting from the update should be as *close* as possible to the original one. In updating an ontology at the instance level following the formula-based approach, the goal becomes the preservation of the facts contained in the original ABox. In what follows we formalize this idea.

Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, an update $\mathcal{U}$, and an ABox $\mathcal{A}'$, we say that $\mathcal{A}'$ *accomplishes the update* of $\mathcal{O}$ with $\mathcal{U}$ if it satisfies all the insertions/deletions in $\mathcal{U}$ minimally. To formalize this notion, we first need to introduce the set $\mathcal{A}^+_{\mathcal{U}}$, which denotes the set of ABox assertions appearing in $\mathcal{U}$ in insertion operations, and the set $\mathcal{A}^-_{\mathcal{U}}$, which denotes the set of ABox assertions appearing in $\mathcal{U}$ in deletion operations.

Definition 22. ABox Update

Let $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ be an ontology, $\mathcal{U}$ an update, and $\mathcal{A}'$ be an ABox. $\mathcal{A}'$ *accomplishes the update* of $\mathcal{O}$ with $\mathcal{U}$ if $\mathcal{A}' = \mathcal{A}'' \cup \mathcal{A}^+_{\mathcal{U}}$ for some maximal subset $\mathcal{A}''$ of $\mathcal{A}$ s.t. $\mathcal{A}'' \cup \mathcal{A}^+_{\mathcal{U}}$ is $\mathcal{T}$-consistent and $\langle \mathcal{T}, \mathcal{A}' \rangle \not\models \beta$ for each $\beta \in \mathcal{A}^-_{\mathcal{U}}$.

It easy to see that, by definition, if such ABox $\mathcal{A}'$ exists, it also satisfies $\langle \mathcal{T}, \mathcal{A}' \rangle \models \alpha$ for each $\alpha \in \mathcal{A}^+_{\mathcal{U}}$ since $\mathcal{A}^+_{\mathcal{U}} \subseteq \mathcal{A}'$. In order to ensure its existence, note that $\mathcal{U}$ has to respect both of the following conditions:

i) $Mod(\langle \mathcal{T}, \mathcal{A}^+_{\mathcal{U}} \rangle) \neq \emptyset$, which means that the set of facts we are adding is consistent with the TBox of the ontology.

ii) $\mathcal{A}^-_{\mathcal{U}} \cap \mathsf{cl}_{\mathcal{T}}(\mathcal{A}^+_{\mathcal{U}}) = \emptyset$, which means that the update is not asking for deleting and inserting the same knowledge at the same time.

Given a TBox $\mathcal{T}$ and an update $\mathcal{U}$, we say that $\mathcal{U}$ is *coherent* with $\mathcal{T}$ if $\mathcal{U}$ respects both the above conditions with respect to a TBox $\mathcal{T}$.

Given a consistent ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ and an update $\mathcal{U}$ coherent with $\mathcal{T}$, there might be more than one ABox accomplishing the update of $\mathcal{O}$ with $\mathcal{U}$. This fact leads to different update semantics, each one addressing this issue by means of a different criterium, like the *Cross Product Approach* [47], the *When In Doubt Throw It Out* principle [56, 71, 72, 116], allowing the user to choose the update [108], or even nondeterminism [23]. Fortunately, when the TBox of the ontology is expressed in *DL-Lite*$_A$, the ABox accomplishing the update is uniquely defined [23]. Hence, the application of all the above approaches leads to the same result, which can be defined as follows:

Definition 23. DL ABox (foundational) Update

Let $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a consistent *DL-Lite*$_A$ ontology and $\mathcal{U}$ be an update coherent with $\mathcal{T}$. The result of updating $\mathcal{O}$ with $\mathcal{U}$, denoted by $\mathcal{O} \circ \mathcal{U}$, is the ontology $\langle \mathcal{T}, \mathcal{A}' \rangle$, where $\mathcal{A}'$ is the ABox accomplishing the update of $\mathcal{O}$ with $\mathcal{U}$.

When dealing with ontology updating, there is a fundamental philosophical aspect that has to be considered: one has to decide if the formulae explicitly given in our ontology provide a justification for our knowledge (foundational semantics) or if they are just used as a finite representation of our knowledge (coherence semantics) [52, 55]. Depending on this point of view, one may or may not need to preserve a fact that is entailed in the ontology despite not being explicitly asserted. The choice depends on the particular application and personal preferences (we refer to [55] for more details).

Clearly, the update semantics given in Definition 23 embraces the foundational theory. Depending on the specific scenario, and the particular application at hand, this semantics might be considered inappropriate. This motivates the definition of the following update semantics [23, 71] for *DL-Lite*$_A$ ontologies based on the coherence theory, in which the objects of the update is not the original ABox, but its deductive closure with respect to the TBox.

Definition 24. DL ABox (coherent) Update

Let $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a consistent *DL-Lite*$_A$ ontology and let $\mathcal{U}$ be an update coherent with $\mathcal{T}$. The result of updating $\mathcal{O}$ with $\mathcal{U}$ according to the coherence semantics, denoted by $\mathcal{O} \bullet \mathcal{U}$, is the ontology $\langle \mathcal{T}, \mathcal{A}' \rangle$, where $\mathcal{A}'$ is the ABox accomplishing the update of $\langle \mathcal{T}, \mathrm{cl}_{\mathcal{T}}(\mathcal{A}) \rangle$ with $\mathcal{U}$.

6.3.2 Careful Semantics in DL-Lite$_A$

An alternative formula-based update semantics based on the coherence theory is the *Careful semantics* [23] which was proposed with the aim of preventing *unexpected information*. Formally, an ontology updated according to the careful semantics should not entail a role constraint ϕ (i.e., a rule of the form $\exists x(R(o,x)) \wedge (x \neq c_1) \wedge \cdots \wedge (x \neq c_n))$, unless ϕ is entailed by the original ABox, or the update itself. In practice, the careful update semantics encompasses deleting more ABox assertions so that the final ontology does not entail any new role constraint ϕ. However, although the careful update semantics was thought to be uniquely defined [23, Theorem 16], it can bring to several solutions as we show in the following example.

Example 2. Consider the *DL-Lite$_A$* ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ where:

$$\begin{array}{llll}
\mathcal{T} = \{ & A \sqsubseteq \exists R_A, & R_A \sqsubseteq R, & \exists R_A^- \sqsubseteq \neg \exists R_B^-, \\
 & B \sqsubseteq \exists R_B, & R_B \sqsubseteq R, & \exists R_A^- \sqsubseteq \neg \exists R_C^-, \\
 & C \sqsubseteq \exists R_C, & R_C \sqsubseteq R, & \exists R_B^- \sqsubseteq \neg \exists R_C^-, \\
 & D \sqsubseteq \exists R_D, & R_D \sqsubseteq R, & \exists R_C^- \sqsubseteq \neg \exists R_D^- \} \\
\mathcal{A} = \{ & A(o), B(o) \ \} & &
\end{array}$$

and the update $\mathcal{U} = \{\mathsf{i}(C(o)), \mathsf{i}(D(o))\}$. It is easy to see that the ABox $\mathcal{A}' = \mathcal{A} \cup \mathcal{A}^+_{\mathcal{U}}$ is $\mathcal{T}$-consistent and that it accomplishes the update of $\mathcal{O}$ with $\mathcal{U}$. Moreover, $\langle \mathcal{T}, \mathcal{A}' \rangle \models \varphi$, where $\varphi = \exists x(R(o,x)) \wedge (x \neq c_1 \wedge (x \neq c_2)))$ (since the negative inclusions in $\mathcal{T}$ imply that in every model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A}' \rangle$ there are three distinct individuals d_a, d_b, d_c such that $\langle o, d_a \rangle \in R_A^{\mathcal{I}}$, $\langle o, d_b \rangle \in R_B^{\mathcal{I}}$, $\langle o, d_c \rangle \in R_C^{\mathcal{I}}$). However, since neither $\langle \mathcal{T}, \mathcal{A} \rangle \models \varphi$ nor $\langle \mathcal{T}, \mathcal{A}^+_{\mathcal{U}} \rangle \models \varphi$, we have that $\mathcal{A}'$ does not accomplish the update of $\mathcal{O}$ with $\mathcal{U}$ carefully. Conversely, both the ABoxes $\{A(o)\} \cup \mathcal{A}^+_{\mathcal{U}}$ and $\{B(o)\} \cup \mathcal{A}^+_{\mathcal{U}}$ accomplish the update of $\mathcal{O}$ with $\mathcal{U}$ carefully. This is because the only role-constraining formula $\exists x(R(o,x)) \wedge (x \neq c_1))$ that both entail with $\mathcal{T}$, is also entailed by $\langle \mathcal{T}, \mathcal{A}^+_{\mathcal{U}} \rangle$. Hence, we have more than one ABox that accomplishes the update of $\mathcal{O}$ with $\mathcal{U}$ carefully. □

6.4 Foundational-Semantic Updates through Datalog

Now, our intention is, given a *DL-Lite$_A$* ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, and some update $\mathcal{U}$, to define a datalog program $\mathcal{D}$ that permits querying whether $\mathcal{U}$ is coherent with $\mathcal{T}$ and, in such a case, allows for generating a set of insertion/deletion instructions that should be applied to $\mathcal{A}$ to accomplish $\mathcal{U}$ according to Definition 23 (foundational-semantic updates).

For ease of presentation, from now on we assume that the TBox $\mathcal{T}$ does not contain inclusions involving attributes and value-domains. However, all the results presented in the next two sections can be easily extended to TBoxes containing such kinds of axioms.

Formally, the datalog program $\mathcal{D}$ contains a derived predicate *incoherent_update*, together with a pair of derived predicates *ins_a/del_a* for each concept/role A such that:

– *incoherent_update()* is true iff $\mathcal{U}$ is not coherent with $\mathcal{T}$.

and, in case *incoherent_update()* is false,

– *ins_a($\overline{o}$)* is true iff the assertion $A(\overline{o})$ was not in $\mathcal{A}$, but $A(\overline{o}) \in \langle\mathcal{T},\mathcal{A}\rangle \circ \mathcal{U}$. That is, *ins_a* captures the assertions of $\mathcal{A}$ that should be inserted into $\mathcal{A}$ to accomplish the (foundational-semantic) update $\mathcal{U}$.

– *del_a($\overline{o}$)* is true iff the assertion $A(\overline{o})$ was in $\mathcal{A}$, but $A(\overline{o}) \notin \langle\mathcal{T},\mathcal{A}\rangle \circ \mathcal{U}$. That is, *del_a* captures the assertions of $\mathcal{A}$ that should be deleted from $\mathcal{A}$ to accomplish the (foundational-semantic) update $\mathcal{U}$.

Briefly, the main idea of the translation is to map each ABox assertion in $\mathcal{A}$, and each operation in $\mathcal{U}$ into different datalog facts. Then, we map each assertion in the closure of $\mathcal{T}$ into several datalog derivation rules that define the *incoherent_update*, *ins_a($\overline{X}$)*, *del_a($\overline{X}$)* predicates. In the following, we formally describe how to obtain such a datalog program $\mathcal{D}$. Then, we prove that the set of instructions generated in $\mathcal{D}$ are sound and complete to obtain $\langle\mathcal{T},\mathcal{A}\rangle \circ \mathcal{U}$.

6.4.1 Translation Rules

Translation of $\mathcal{A}$ and $\mathcal{U}$

All the assertions in $\mathcal{A}$ and operations in $\mathcal{U}$ are translated as different facts in $\mathcal{D}$. In particular:

Each assertion $A(\overline{o}) \in \mathcal{A}$ is translated as the fact *a($\overline{o}$)*.
Each operation $\mathrm{i}(A(\overline{o})) \in \mathcal{U}$ is translated as the fact *ins_a_request($\overline{o}$)*.
Each operation $\mathrm{d}(A(\overline{o})) \in \mathcal{U}$ is translated as the fact *del_a_request($\overline{o}$)*.

Intuitively, *ins_a_request($\overline{o}$)/del_a_request($\overline{o}$)* means that the ontology has received the request to insert/delete the ABox assertion $A(\overline{o})$. Since according to the Definition 23 all the insertions/deletions requested should be applied, we define the datalog rules:

```
ins_a(X) :- ins_a_request(X), not a(X).
del_a(X) :- del_a_request(X), a(X).
incoherent_update() :- ins_a_request(X), del_a_request(X).
```

for each atomic concept A. Note that *incoherent_update* becomes true in case we request for the insertion and deletion of the same axiom. Similarly, we define the rules *ins_p(X, Y)*/*del_p(X,Y)* for each atomic role P.

Translation of $cl(\mathcal{T})$

We translate positive and negative/functional axioms in the closure of $\mathcal{T}$ differently. In particular, for each positive inclusion axiom $B \sqsubseteq A$ in the closure of $\mathcal{T}$, where A is an atomic concept, we define the rules:

```
del_b(X) :- b(X), del_a_request(X).
incoherent_update() :- ins_b_request(X), del_a_request(X).
```

Intuitively, when we request for deleting $A(o)$, we have to delete any other ABox assertion $B(o)$ that entails $A(o)$. Note that it cannot be accomplished if there is a request for inserting $B(o)$, so, this case makes *incoherent_update* true. We define similar rules when the left-hand side of the axiom is of the form $\exists P$, and also for role inclusion axioms.

Note that we translate the closure of $\mathcal{T}$, instead of $\mathcal{T}$ itself, to be able to capture deletions that are propagated along the concept/role hierarchy. E.g. if in our example we have $\mathcal{U}$ = d(Person(john)), the translated datalog program $\mathcal{D}$ generates the deletion of FullProfessor(john) because of the translation of the assertion FullProfessor $\sqsubseteq$ Person appearing in $cl(\mathcal{T})$:

```
del_fullprof(X) :- fullprof(X), del_person_request(X).
```

Differently, for each negative inclusion axiom $B \sqsubseteq \neg A$ in $cl(\mathcal{T})$, we define the rules:

```
del_b(X) :- b(X), ins_a_request(X).
del_a(X) :- ins_b_request(X), a(X).
incoherent_update() :- ins_a_request(X), ins_b_request(X).
```

Intuitively, if we insert $A(o)$ when we have $B(o)$ in the ABox, we have to delete $B(o)$. In the case where the requested update tries to insert both things, we reach a contradiction and thus, *incoherent_update* becomes true. We define similar rules for role negative inclusions, negative inclusions involving the $\exists$ constructor, and functional axioms. In this last case, we require using the inequality built-in predicate to check whether the requested role assertion insertion is going to violate the functional axiom. E.g., given a functional axiom defined over R, we define:

```
del_r(X,Y) :- r(X,Y), ins_r_request(X,Z), Y<>Z.
incoherent_update() :- ins_r_request(X,Y),ins_r_request(X,Z),Y<>Z
    .
```

Again, note that since we translate the closure of $\mathcal{T}$, the rules are able to capture deletions due to inconsistencies generated by propagation. E.g. if in our previous example we have the update $\mathcal{U} = \text{i(AssociateProfessor(bob))}$, $\mathcal{D}$ generates the deletion of Student(bob) because of the first rule obtained when translating the assertion Student $\sqsubseteq \neg$AssociateProfessor appearing in $cl(\mathcal{T})$:

```
del_student(X) :- student(X), ins_assocprof_request(X).
del_assocprof(X) :- assocprof(X), ins_student_request(X).
```

6.4.2 Datalog Program Soundness and Completeness

The update generated by the datalog program $\mathcal{D}$ is sound in the sense that, for every axiom $A(\bar{o})$ that should be inserted/deleted according to $\mathcal{D}$, $A(\bar{o})$ should be truly inserted/deleted according to the foundational-semantic update. Formally:

Property 33. Datalog program soundness (foundational semantics)

Given a consistent ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, and an update $\mathcal{U}$, the datalog program $\mathcal{D}$ obtained through the translation defined in Section 6.4.1, satisfies that: if *incoherent_update()* is true in $\mathcal{D}$, $\mathcal{U}$ is incoherent with $\mathcal{T}$, otherwise, for each concept/role A, if *ins_a($\bar{o}$)* is true in $\mathcal{D}$, then, $A(\bar{o}) \in \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U} \setminus \mathcal{A}$, and if *del_a($\bar{o}$)* is true in $\mathcal{D}$, then, $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$.

Proof. (Sketch) If *incoherent_update()* is true, it can only be because of a rule generated when translating the update $\mathcal{U}$, the positive axioms of $cl(\mathcal{T})$, or the negative/-functional axioms of $cl(\mathcal{T})$. The rules generated in the first two cases are true only if $\mathcal{A}^-_{\mathcal{U}} \cap \mathcal{A}^+_{\mathcal{U}} \neq \emptyset$ and $\mathcal{A}^-_{\mathcal{U}} \cap cl_{\mathcal{T}}(\mathcal{A}^+_{\mathcal{U}}) \neq \emptyset$, respectively. The rules of the third case are true only if $Mod(\langle \mathcal{T}, \mathcal{A}^+_{\mathcal{U}} \rangle) = \emptyset$. Thus, if *incoherent_update()* is true, $\mathcal{U}$ is incoherent with $\mathcal{T}$.

If *ins_a($\bar{o}$)* is true, it is because of a rule generated when translating $\mathcal{U}$, which can only be true if $A(\bar{o}) \notin \mathcal{A}$, and $A(\bar{o}) \in \mathcal{A}^+_{\mathcal{U}}$, thus $A(\bar{o}) \in \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U} \setminus \mathcal{A}$.

If *del_a($\bar{o}$)* is true, it can only be because of (1) a rule generated when translating $\mathcal{U}$, where in such case we have $A(\bar{o}) \in \mathcal{A}$, and $A(\bar{o}) \in \mathcal{A}^-_{\mathcal{U}}$, thus $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$; or (2) a rule generated when translating a positive axiom in $\mathcal{T}$, where in such case we have that $A(\bar{o}) \in \mathcal{A}$ and that for some $B(\bar{o}) \in \mathcal{A}^-_{\mathcal{U}}$, $A(\bar{o}) \models_{\mathcal{T}} B(\bar{o})$, thus, $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$; or (3) a rule generated when translating a negative/functional axiom in $cl(\mathcal{T})$ where in such case we have $A(\bar{o}) \in \mathcal{A}$ and $Mod(\langle \mathcal{T}, \mathcal{A}^+_{\mathcal{U}} \cup \{A(\bar{o})\} \rangle) = \emptyset$, and thus, $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$. □

Conversely, $\mathcal{D}$ is also complete in the sense that any axiom insertion/deletion of $A(\bar{o})$ that should be applied according to the foundational-semantic update is also generated in $\mathcal{D}$. Formally:

Property 34. Datalog program completeness (foundational semantics)

Given a consistent ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, and an update $\mathcal{U}$, the datalog program $\mathcal{D}$ obtained through the translation defined in Section 6.4.1, satisfies that: if $\mathcal{U}$ is incoherent with $\mathcal{T}$, then, *incoherent_update()* is true in $\mathcal{D}$, otherwise, for each concept/role A, if $A(\bar{o}) \in \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U} \setminus \mathcal{A}$, then, *ins_a($\bar{o}$)* is true in $\mathcal{D}$, and if $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$, then, *del_a($\bar{o}$)* is true in $\mathcal{D}$.

Proof. (Sketch) First, if $\mathcal{U}$ is incoherent with $\mathcal{T}$, it is immediate to verify that then, *incoherent_update()* is true in $\mathcal{D}$. So, from now on we assume that $\mathcal{U}$ is coherent with $\mathcal{T}$. Moreover, since $\mathcal{U}$ is coherent with $\mathcal{T}$, $\langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U} \setminus \mathcal{A} = \mathcal{A}^+_{\mathcal{U}} \setminus \mathcal{A}$, and by definition of $\mathcal{D}$, it easily follows that, for each concept/role A, if $A(\bar{o}) \in \mathcal{A}^+_{\mathcal{U}} \setminus \mathcal{A}$, *ins_a($\bar{o}$)* is true in $\mathcal{D}$. Finally, we prove that for every assertion deleted from $\mathcal{A}$ there is a corresponding deletion instruction in $\mathcal{D}$. To this aim, we define the following algorithm:

```
Algorithm ComputeDeletedAssertions($\mathcal{T}, \mathcal{A}, \mathcal{U}$)
Input: DL-Lite$_A$ TBox $\mathcal{T}$, ABox $\mathcal{A}$, update $\mathcal{U}$ coherent with $\mathcal{T}$
Output: ABox $\mathcal{A}_d = \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$
begin
  $\mathcal{A}_d = \emptyset$;
  for each $C(a) \in \mathcal{A}^+_{\mathcal{U}}$ do begin
    for each $D(a) \in \mathcal{A}$ such that $\mathcal{T} \models C \sqsubseteq \neg D$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{D(a)\}$;
    for each $R(a,x) \in \mathcal{A}$ such that $\mathcal{T} \models C \sqsubseteq \neg \exists R$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{R(a,x)\}$;
    for each $R(x,a) \in \mathcal{A}$ such that $\mathcal{T} \models C \sqsubseteq \neg \exists R^-$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{R(x,a)\}$
  end;
  for each $R(a,b) \in \mathcal{A}^+_{\mathcal{U}}$ do begin
    for each $S(a,b) \in \mathcal{A}$ such that $\mathcal{T} \models R \sqsubseteq \neg S$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(a,b)\}$;
    for each $S(b,a) \in \mathcal{A}$ such that $\mathcal{T} \models R \sqsubseteq \neg S^-$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(b,a)\}$;
    for each $C(a) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R \sqsubseteq \neg C$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{C(a)\}$;
    for each $C(b) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R^- \sqsubseteq \neg C$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{C(b)\}$;
    for each $S(a,x) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R \sqsubseteq \neg \exists S$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(a,x)\}$;
    for each $S(x,a) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R \sqsubseteq \neg \exists S^-$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(x,a)\}$;
    for each $S(b,x) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R^- \sqsubseteq \neg \exists S$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(b,x)\}$;
    for each $S(x,b) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R^- \sqsubseteq \neg \exists S^-$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(x,b)\}$
  end;
  for each $C(a) \in \mathcal{A}^-_{\mathcal{U}}$ do begin
    for each $D(a) \in \mathcal{A}$ such that $\mathcal{T} \models D \sqsubseteq C$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{D(a)\}$;
    for each $R(a,x) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R \sqsubseteq C$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{R(a,x)\}$;
    for each $R(x,a) \in \mathcal{A}$ such that $\mathcal{T} \models \exists R^- \sqsubseteq C$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{R(x,a)\}$
  end;
  for each $R(a,b) \in \mathcal{A}_{\mathcal{U}}^-$ do begin
    for each $S(a,b) \in \mathcal{A}$ such that $\mathcal{T} \models S \sqsubseteq R$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(a,b)\}$;
    for each $S(b,a) \in \mathcal{A}$ such that $\mathcal{T} \models S \sqsubseteq R^-$ do $\mathcal{A}_d = \mathcal{A}_d \cup \{S(b,a)\}$
  end;
  return $\mathcal{A}_d$
end
```

It can easily be shown that the ABox returned by such an algorithm is equal to $\mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \circ \mathcal{U}$. Moreover, it is easy to see that, for each concept/role $\mathcal{A}$, if $A(\overline{o})$ belongs to the ABox returned by ComputeDeletedAssertions($\mathcal{T}, \mathcal{A}, \mathcal{U}$), then *del_a($\overline{o}$)* is true in $\mathcal{D}$. □

6.5 Coherent-Semantic Updates through Datalog

The previous datalog program $\mathcal{D}$ generates the set of insertions/deletions that should be applied to an ABox $\mathcal{A}$ to accomplish an update $\mathcal{U}$ according to the foundational-semantics. Now, our purpose is to modify this datalog program to deal with the coherent-semantics as described in Definition 24.

Briefly, to accomplish the coherent-semantics, we need to generate more insertion instructions in $\mathcal{D}$. This is because in the coherent-semantics we need to keep the updated ABox as *close* as possible to the $\mathcal{T}$-*closure* of the original ABox, instead of the ABox itself. For instance, if in our previous example we apply the update $\mathcal{U} = \{$d(Student(bob))$\}$ with coherent-semantics, besides deleting the assertion Student(bob), we also need to apply the insertion Person(bob) since Person(bob) appears in $cl_{\mathcal{T}}(\mathcal{A})$.

Thus, in practice, we only need to extend our datalog program $\mathcal{D}$ to (1) additionally capture those assertions $A(\overline{o})$ entailed by assertions $B(\overline{o})$ that are requested for deletion, and (2) derive their insertion in case they do not get in conflict with the assertions in $\mathcal{A}_{\mathcal{U}}^+$. Intuitively, we do (1) by considering an additional derived predicate *ins_a_closure* for each concept/role A; then, we use this new predicate to define new derivation rules for *ins_a* in case they do not get in conflict with any axiom in $\mathcal{A}_{\mathcal{U}}^+$, thus accomplishing (2).

In the following, we first define how we obtain these new derivation rules, and then we prove that the insertion/deletion instructions generated by this extended datalog program $\mathcal{D}$ are sound and complete with respect to the coherent-semantics.

6.5.1 Translation Rules

Capturing Closure Insertions due to Deletions

For each positive inclusion axiom $B \sqsubseteq A$ in the closure of $\mathcal{T}$, where A is an atomic concept, let A_1, ..., A_m be all the atomic concepts having a positive inclusion axiom of the form $A \sqsubseteq A_i$ in the TBox closure of $\mathcal{T}$, then we define the rules:

```
ins_a_closure(X) :- del_b(X), not a(X), not ins_a_request(X), not
     del_a_request(X), not del_a1_request(X), ..., not
    del_am_request(X).
```

For example, for the assertion FullProfessor $\sqsubseteq$ Professor, we define the rules:

```
ins_prof_closure(X) :- del_fullprof(X), not prof(X), not
    ins_prof_request(X), not del_prof_request(X), not
    del_person_request(X).
```

Intuitively, when we delete a FullProfessor(o), we might need to insert Professor(o) because of the closure of the semantics. However, such *closure insertion* is not necessary if Professor(o) is already in the ABox, or if there is a request for its insertion, or if it is requested for deletion (either Professor(o) itself or its parent concepts Person(o)). We define similar rules for role positive inclusion axioms and positive inclusion axioms in which the left-hand side uses the $\exists$ constructor.

Defining New Insertions due to Closure Insertions

Once we have defined the predicates *ins_a_closure*, we use them for defining new insertions in case they do not get in conflict with the assertions in $\mathcal{A}^+_\mathcal{U}$. To do so, for each atomic concept A, let $B_1, \ldots, B_n$ be all the concepts having a negative inclusion axiom with A in the TBox closure of $\mathcal{T}$, then we define the rules:

```
ins_a(X) :- ins_a_closure(X), not ins_b1_request(X) ... not
    ins_bn_request(X).
```

Following the previous example, we would define:

```
ins_prof(X):-ins_prof_closure(X), not ins_student_request(X).
```

Intuitively, any derived *closure insertion* of Professor(o) should be applied only if it does not get in conflict with any negative inclusion axiom. Such a conflict might arise if there is a request to insert some Student(o) because of the negative inclusion assertion Student $\sqsubseteq \neg$Professor. Similarly, we define the rules for roles.

6.5.2 Datalog Program Soundness and Completeness

We finally state that the generated insertion/deletions instructions generated by the datalog program $\mathcal{D}$ is sound and complete with respect to the coherent-semantics (the proof of the following theorem can be obtained by easily extending the proofs of Theorem 33 and Theorem 34).

Property 35. Datalog program correctness (coherent semantics)

Given a consistent ontology $\langle \mathcal{T}, \mathcal{A} \rangle$, and an update $\mathcal{U}$, the datalog program $\mathcal{D}$ obtained through the translation defined in Sections 6.4.1 and 6.5.1, satisfies that: (i) *incoherent_update()* is true in $\mathcal{D}$ iff $\mathcal{U}$ is incoherent with $\mathcal{T}$; (ii) if $\mathcal{U}$ is coherent with $\mathcal{T}$, then for each concept/role $\mathcal{A}$, *ins_a($\bar{o}$)* is true in $\mathcal{D}$ iff $A(\bar{o}) \in \langle \mathcal{T}, \mathcal{A} \rangle \bullet \mathcal{U} \setminus \mathcal{A}$, and *del_a($\bar{o}$)* is true in $\mathcal{D}$ iff $A(\bar{o}) \in \mathcal{A} \setminus \langle \mathcal{T}, \mathcal{A} \rangle \bullet \mathcal{U}$.

6.6 Implementation and Experiments

To show the feasibility and scalability of our technique, we have developed a Java program that, given a closed *DL-Lite*$_A$ TBox, builds the datalog program that generates the insertion/deletion instructions for applying a coherent-semantic update. Furthermore, the program translates this datalog into standard SQL queries. Since these queries depend only on the TBox, but not on the ABox nor the requested update, all of them are created in compilation time and stored in the database as SQL views. Thus, on runtime, the user can generate the instructions by means of inserting the operations s/he wants to perform in the *ins_a_request/del_a_request* tables of the database and querying these views.

We have run the experiments using a *DL-Lite*$_A$ approximation of the LUBM benchmark, an ontology describing university concepts (e.g., teachers, departments, etc) with 75 basic concept/roles and 243 assertions. For our purposes, we have removed those axioms not expressible in *DL-Lite*$_A$, and added 20 disjointness/functional assertions to increase the complexity of the updates. Thus, our final ontology consisted of 195 axioms.

Regarding the data, we have created different ABoxes of increasing size (from 10^5 to $3.5 * 10^7$ assertions). To do so, we have modified the UBA Data Generator to create a single university, but with an increasing number of connected departments, teachers, etc. Due to this increasing number of connected objects, the updates became more complex when increasing the data size. Then, we have defined an update request by means of selecting 3 tuples to delete, and 3 tuples to insert. Such tuples were selected in a way to ensure several interactions with the TBox assertions, thus, generating several insertions/deletions.

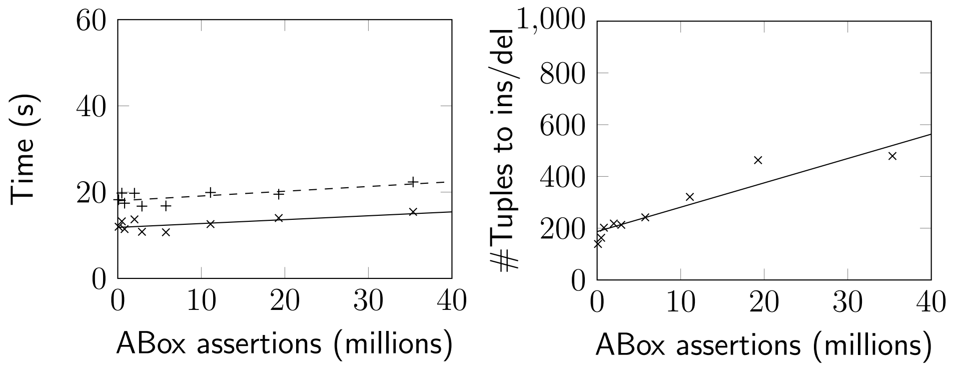


Figure 6.1: Experimental Results

In Figure 6.1 we summarize the results we have obtained using the MySQL 5.7 DBMS, running on a Windows 8.1 over an Intel Core i7-4710HQ, with 8GB of RAM [1]. In particular, we show the times to generate the instructions (x points in the first diagram), the time to generate and execute the instructions (+ points in the first diagram), and the number of instructions generated (x points in the second diagram). We also depict the different trend lines in the diagrams.

As it can be seen, our method has generated from 139 insertion/deletion instructions in 12s for the smallest ABox, to 479 instructions in 16s for the largest. Thus, although there is a constant time penalty of about 12s to generate the instructions, the time increment in function of the ABox size is small. Adding this time to the time to execute the instructions, we got a total cost near to 20s. We argue that this low time increment behavior is due to the fact that, in *DL-Lite*$_A$, an update request only causes updates *locally*, i.e., the unique tuples to insert/delete are a subset of those that are *connected* to the requested insertions/deletions. Thus, since ABoxes tends to increase its size by considering more objects, rather than infinitely augmenting the connectivity between them, increasing the ABox size barely increases the generated instructions, as can be seen in the second diagram. Hence, we argue that our approach can be effectively used in practice with large ABoxes.

6.7 Conclusions

In this chapter we have shown that the *DL-Lite* family, in particular *DL-Lite*$_A$, enjoys the first-order rewritability of instance level updates. Apart from the theoretical interest, this result gives us a practical and effective technique to perform updates over *DL-Lite* ontologies.

Although we have not considered any specific syntax to express the update, what

[1]More experiment details and results at www.essi.upc.edu/~xoriol/dllitea/

we proposed here is fully compatible with SPARQL update operators studied in [4]. There, the set of insertions and deletions are defined through unions of conjunctive queries over the current ontology. We can immediately extend our approach in the same way, producing update operators that are equivalent to the ones defined in [4] in the case of RDFS, but that deal with the more expressive *DL-Lite*$_A$ and OWL 2 QL languages.

There are several directions for future work, but maybe the most compelling one, encouraged by the practical applicability of our results, is to extend our datalog-based approach blurring the distinction between TBox and ABox assertions, in line with the use of SPARQL over OWL 2 QL ontologies.

Conclusions

Chapter 7

Conclusions And Further Research

The basic goal of this thesis was to provide a method for performing incremental integrity checking/maintenance of UML/OCL conceptual schemas. That is, we wanted to develop some technique that, given a set of UML/OCL constraints, a consistent data state, and some updates on it, we wanted to be able to (1) assess whether such updates are going to cause the violation of some constraint, and (2) which additional updates could be considered to *repair* such violation.

Bearing this in mind, we have started by studying the difficulty of checking general OCL constraints. In this study, exploiting the OCL recursion and its string operators, we have been able to reduce the problems of 0-type grammar word acceptance/non-acceptance to OCL constraint satisfaction. Thus, since such problems are known to be non decidable, it turns out that checking whether some data state satisfies a general OCL constraint is undecidable too.

To solve such situation, we have identified OCL_{FO}, the subset of OCL equivalent to relational algebra (aka domain independent first-order logics). OCL_{FO} is equivalent to RA in the sense that any OCL_{FO} constraint can be checked through a RA query, and any constraint checkable in RA can be written in OCL_{FO}. The syntax and semantics of OCL_{FO} are precisely defined through a formal grammar and basic set theory, thus, avoiding the ambiguities that are currently found in the OCL standard. Moreover, to make OCL_{FO} an easy object of study, we have shown that the full expressive power of OCL_{FO} can be achieved by 5 OCL basic operators: *allInstances*, *forAll*, *implies*, $<$, and $=$. We call such minimal set of operators as OCL_{CORE}.

Then, we have moved to the problem of incremental integrity checking of OCL constraints. Our proposed solution is based on materializing the updates as ground facts, and define some logic rules (the Event Dependency Constraints), that are able to assess whether such updates in the current data state cause a constraint violation or not. This proposal is based on a previously existing technique for constraint checking in deductive databases (the events method [82]) but extended in several directions. First, we have extended this technique to deal with aggregation operations, thus, our technique is able to deal with all OCL_{FO} constraints extended with *size*, *sum*, and *count* OCL operators. Second, we have used this extension to better treat existential variables. Briefly, we aggregate the existential variables to count the number of witnesses do they have in the current data state, thus, we can spot the violation of a constraint involving existential variables when such counts get to 0 (without requiring to check the data state again). Third, we have reduced the number of rules generated to check the constraints to a linear number (whereas the original proposal generated an exponential number of EDCs w.r.t. the size of the constraints).

Afterwards, we have modified the EDCs proposal for integrity checking to be able to perform integrity maintenance. The basic idea is to move the negated events from the EDCs to the RHS of the logic rule. We call such new rules Repair-Generating Dependencies. Then, we can compute the additional events required to repair the constraint violations by means of chasing such RGDs. The concrete chase algorithm to apply depends on the kind of RGDs involved. In the particular case of RGDs without existential variables, we have proved that we can apply the usual (disjunctive) chase; in contrast, for dealing with existential variables, the output of the traditional chase does not suit our purposes anymore since the generated labelled nulls might be replaced for incorrect values. To fix this problem, we have proposed to include built-in literals constraining the incorrect values of the generated labelled nulls, and shown that we can compute such labelled nulls incorporating the VIPs approach into the chase [50].

In order to exploit this distinction between RGDs with/without existential variables, we have identified an expressive subset of OCL_{FO} whose generated RGDs have no existential variables: OCL_{UNIV}. Intuitively, OCL_{UNIV} is obtained by removing the OCL *exists* operator in OCL_{FO}, and limiting all those other operators that might be used to emulate it.

We have also seen that the problem of maintaining OCL constraints is decidable for the case of OCL_{UNIV} (due to the absence of existential variables in its RGDs), but undecidable in the OCL_{FO} language. To deal with this issue, we have connected some decidability results appearing in [103, 107] to our RGDs. To do so, we have defined the Finite Canonical Property (FCP), showed that the previous results stated in [103, 107] ensured FCP, and that FCP ensures termination on the chase we use to repair the constraints.

Since the number of solutions obtained for repairing a set of constraints can grow exponentially, we have also studied how to prune some of them. Thus, we propose to customize the available repairs in our RGDs by moving some of their events from the RHS to the LHS. When doing so, we have seen that the negated events might cause semantic problems on our rules, in a similar way that the negated literals might cause problems in the context of logic programming. To solve such problems, we have borrowed the concepts of *well-supportedness* from the logic programming field together with the notion of *stratified negation*.

Finally, we have seen that we can apply the previous techniques in the context of DL-Lite and the open-world assumption. In virtue of the DL-Lite first-order rewritability property for the problem of integrity checking, we can pick the first-order queries that checks the consistency of a DL-Lite ontology, and use them for building our RGDs. Moreover, we have seen that such RGDs can be seen as a non-stratified datalog program, which means that they can be translated as SQL queries. Therefor, DL-Lite repairs can be computed using SQL queries, with no need of chase procedures. In other words, computing DL-Lite updates itself is first-order rewritable, which is a novel result proved in this thesis.

Further Research

There are several ways to extend this work we have done.

First of all, taking in account that it is known that the constraints that can be checked in polynomial time (w.r.t. data complexity) are those expressible in first-order logics with least fixed-points, it would be interesting to try to figure out which subset of OCL corresponds to it. Such subset would be able to deal with all OCL_{FO} but extending it with more powerful OCL operations (such as *closure*). Then, it would be interesting to extend our techniques for performing incremental integrity checking/maintenance under this new language. Essentially, it seems that the first step would be to extend the *new/old* mappings for the case of recursive derived literals.

We would also like to improve the performance of the vips-chase to incrementally maintain OCL_{FO} constraints. In this case, we should probably change our definition of *Canonical Model Set* to avoid an exponential growth of its cardinality. A preliminary idea we have is to replace the set of built-in literals restricting the values for the labelled nulls with richer constraints (e.g. combining built-in literals with disjunctions would permit us to collapse several canonical models into one).

With regarding to the open-world assumption and the ontology field, we are currently working on how to apply these techniques to build an Ontology Based Data Update framework in collaboration with researchers from *Università di Roma La Sapienza*. Other possibilities would be to try to export the current DL-Lite update approach to linear $\text{Datalog}^{\pm}$ [19].

References

[1] Eclipse ocl project. `http://wiki.eclipse.org/OCL`. Accessed: 2016-08-08. 23

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. 1995. 155

[3] Foto Afrati and Rada Chirkova. Selecting and using views to compute aggregate queries. In *Database Theory - ICDT 2005*, **3363** of *LNCS*, pages 383–397. Springer, 2005. 18

[4] Albin Ahmeti, Diego Calvanese, and Axel Polleres. Updating RDFS aboxes and tboxes in SPARQL. pages 441–456, 2014. 153, 168

[5] Albin Ahmeti, Diego Calvanese, Axel Polleres, and Vadim Savenkov. Dealing with inconsistencies due to class disjointness in SPARQL update. In *Proc. of DL 2015*, **1350** of *CEUR*, 2015. 153

[6] Harith T. Al-Jumaily, Dolores Cuadra, and Paloma Martínez. Ocl2trigger: Deriving active mechanisms for relational databases using model-driven architecture. *Journal of Systems and Software*, **81**[12]:2299–2314, 2008. 145

[7] ANSI. *The SQL 92 Standard*. ANSI, 1992. 3, 85

[8] Uwe Assmann, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, Julia Schroeter, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Dropsbox: the dresden open software toolbox. *Software & Systems Modeling*, **13**[1]:133–169, 2014. 23, 52, 83, 89, 90

[9] Hermann Balsters. Modelling database views with derived classes in the UML/OCL-framework. In *UML2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 295–309. Springer, 2003. 51

[10] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the object constraint language into first-order predicate logic. In *Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC)*, 2002. 51

[11] Gábor Bergmann. Translating OCL to graph patterns. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 670–686, 2014. 34, 90

[12] Magnus Boman, Janis A Bubenko Jr, Paul Johannesson, and Benkt Wangler. *Conceptual modelling*. Prentice-Hall, Inc., 1997. 2

[13] Achim D Brucker, Tony Clark, Carolina Dania, Geri Georg, Martin Gogolla, Frédéric Jouault, Ernest Teniente, and Burkhart Wolff. Panel discussion: Proposals for improving ocl. In *Proceedings of the MODELS 2014 OCL Workshop (OCL 2014)*, **1285**, pages 83–99. CEUR-WS. org, 2014. 23

[14] Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5. *Archive of Formal Proofs*, January 2014. `http://www.isa-afp.org/entries/Featherweight_OCL.shtml`, Formal proof development. 24

[15] Achim D. Brucker and Burkhart Wolff. *Fundamental Approaches to Software Engineering*, pages 97–100. HOL-OCL: a formal proof environment for UML/OCL. Springer, 2008. 140

[16] Jordi Cabot. *From Declarative to Imperative UML/OCL Operation Specifications*, pages 198–213. Springer, Berlin, Heidelberg, 2007. 141

[17] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying uml/ocl operation contracts. In *Integrated Formal Methods: 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, pages 40–55, Berlin, Heidelberg, 2009. Springer. 140

[18] Jordi Cabot and Ernest Teniente. Incremental integrity checking of uml/ocl conceptual schemas. *Journal of Systems and Software*, **82**[9]:1459–1478, 2009. 4, 89

[19] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, **14**:57

– 83, 2012. Special Issue on Dealing with the Messiness of the Web of Data. 172

[20] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, **39**[3]:385–429, 2007. 10, 151, 154

[21] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. *Artificial Intelligence*, **195**:335 – 360, 2013. 151

[22] Diego Calvanese, Sven Hartmann, and Ernest Teniente. Automated reasoning on conceptual schemas (dagstuhl seminar 13211). *Dagstuhl Reports*, **3**[5], 2013. 15

[23] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Dmitriy Zheleznyakov. Evolution of *DL-Lite* knowledge bases. **6496**, pages 112–128, 2010. 152, 156, 158, 159

[24] Peter Pin-Shan Chen. The entity-relationship model-toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, **1**[1]:9–36, 1976. 22

[25] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. In *Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages*, **24**. ECEASST, 2009. 51, 52

[26] Maria Claver Argudo. Safeex: Eina per a l'execució d'esquemes conceptuals en uml, 2015. 15

[27] Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972. 56

[28] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 155–166, New York, NY, USA, 1999. ACM. 18

[29] MarianoP. Consens and AlbertoO. Mendelzon. Low complexity aggregation in graphlog and datalog. In *ICDT '90*, **470** of *LNCS*, pages 379–394. Springer, 1990. 18

[30] D. Costal, , C. Gómez, A. Queralt, and E. Teniente. Drawing preconditions of operation contracts from conceptual schemas. In *Advanced Information Systems Engineering*, pages 266–280. Springer, 2008. 146

[31] Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Improving the definition of general constraints in uml. *Software & Systems Modeling*, **7**[4]:469–486, 2008. 114, 135, 136, 146

[32] Dolors Costal, Ernest Teniente, and Toni Urpí. An approach to obtain intensional translations for consistent view updating. In *Deductive and Object-Oriented Databases: 5th International Conference, DOOD'97 Montreux, Switzerland, December 8–12, 1997 Proceedings*, pages 175–192, Berlin, Heidelberg, 1997. Springer. 115

[33] Hoa Khanh Dam and Michael Winikoff. Supporting change propagation in UML models. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010. 146

[34] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: A framework to handle complex queries on large models. In *10th IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*, pages 1–12, 2016. 90

[35] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On instance-level update and erasure in description logic ontologies. *Journal of Logic and Computation*, **19**[5]:745–770, 2009. 152

[36] Giuseppe De Giacomo, Xavier Oriol, Montse Estañol, and Ernest Teniente. Linking data and bpmn processes to achieve executable models. In *29th International Conference on Advanced Information Systems Engineering, Essen, Germany, June 12–16, 2017, Proceedings (To appear)*, 2016. 14

[37] Giuseppe De Giacomo, Xavier Oriol, Riccardo Rosati, and Domenico Fabio Savo. Updating dl-lite ontologies through first-order queries. In *15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I*, pages 167–183. Springer International Publishing, 2016. 14

[38] Hendrik Decker and Lawrence Cavedon. Generalizing allowedness while retaining completeness of sldnf-resolution. In *International Workshop on Computer Science Logic*, pages 98–115. Springer, 1989. 63

[39] Birgit Demuth and Heinrich Hussmann. Using UML/OCL constraints for relational database design. In *≪UML≫99 - The Unified Modeling Language*, pages 598–613. Springer, 1999. 52

[40] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 149–158, 2008. 110, 112, 119, 121, 147

[41] Marina Egea, Carolina Dania, and Manuel Clavel. Mysql4ocl: A stored procedure-based mysql code generator for OCL. *ECEASST*, **36**, 2010. 52, 90

[42] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 99–108, 2008. 146

[43] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, **57**[2]:227 – 270, 1992. 157

[44] Ramez Elmasri and Shamkant B Navathe. *Fundamentals of database systems*. Pearson, 2014. 85

[45] David W Embley, Stephen W Liddle, and Oscar Pastor. Conceptual-model programming: a manifesto. In *Handbook of Conceptual Modeling*, pages 3–16. Springer, 2011. 4

[46] François Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Computing*, **9**[3]:425–443, 1991. 101, 128

[47] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '83, pages 352–365, New York, NY, USA, 1983. ACM. 152, 158

[48] Jean-Rémy Falleri, Xavier Blanc, Reda Bendraou, Marcos Aurélio Almeida da Silva, and Cédric Teyton. Incremental inconsistency detection with low memory overhead. *Softw., Pract. Exper.*, **44**[5]:621–641, 2014. 89

[49] Carles Farré, Guillem Rull, Ernest Teniente, and Toni Urpí. Svte: a tool to validate database schemas giving explanations. In *Proceedings of the 1st International Workshop on Testing Database Systems, DBTest 2008, Vancouver, BC, Canada, June 13, 2008*, page 9, 2008. 136, 143

[50] Carles Farré, Ernest Teniente, and Toni Urpí. Checking query containment with the CQC method. *Data & Knowledge Engineering*, **53**[2]:163–223, 2005. 99, 116, 117, 119, 120, 147, 171

[51] Giorgos Flouris, Dimitris Manakanatas, Haridimos Kondylakis, Dimitris Plexousakis, and Grigoris Antoniou. Ontology change: Classification and survey. *Knowledge Engineering Review*, **23**[2]:117–152, 2008. 152

[52] Giorgos Flouris and Dimitris Plexousakis. Handling ontology change: Survey and proposal for a future research direction. Technical report TR-362 FORTH-ICS, Institute of Computer Science, Forth. Greece, 2005. 157, 158

[53] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, **19**[1]:17 – 37, 1982. 88, 90

[54] Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. Logic foundations of the ocl modelling language. In *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, pages 657–664. Springer International Publishing, 2014. 14

[55] Peter Gärdenfors. Propositional logic based on the dynamics of belief. *J. Symb. Log.*, **50**[2]:390–394, 1985. 158

[56] Matthew L. Ginsberg and David E. Smith. Reasoning about action I: A possible worlds approach. Technical Report KSL-86-65, Knowledge Systems, AI Laboratory, 1987. 152, 158

[57] M.L. Ginsberg. Counterfactuals. *Artificial Intelligence*, **30**[1]:35–79, 1986. 152

[58] Martin Gogolla and Frank Hilken. Model validation and verification options in a contemporary UML and OCL analysis tool. In *Modellierung 2016, 2.-4. März 2016, Karlsruhe*, pages 205–220, 2016. 23, 89, 140

[59] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, **1**[1]:29–53, 1997. 60

[60] Iris Groher, Alexander Reder, and Alexander Egyed. Incremental consistency checking of dynamic constraints. In *International Conference on Fundamental Approaches to Software Engineering*, pages 203–217. Springer, 2010. 4, 89, 146

[61] R.C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Eclipse Series. Pearson Education, 2009. 89

[62] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. **3**[2–3]:158–182, 2005. 153, 155

[63] Terry Halpin. Object-role modeling (orm/niam). In *Handbook on architectures of information systems*, pages 81–103. Springer, 1998. 22

[64] Alan R Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, **19**[2]:4, 2007. 16

[65] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, **68**[1]:86 – 104, 1986. 53

[66] Hirofumi Katsuno and Alberto Mendelzon. On the difference between updating a knowledge base and revising it. In *KR proceedings*. 157

[67] Evgeny Kharlamov, Dmitriy Zheleznyakov, and Diego Calvanese. Capturing model-based ontology evolution at the instance level: The case of dl-lite. *Journal of Computer and System Sciences*, **79**[6]:835–872, 2013. 152

[68] Matthias P. Krieger, Alexander Knapp, and Burkhart Wolff. Automatic and efficient simulation of operation contracts. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, pages 53–62, 2010. 145

[69] C.F.J. Lange, M. R V Chaudron, and J. Muskens. In practice: Uml software architecture and design description. *IEEE Software*, **23**[2]:40–46, March 2006. 3

[70] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education India, 2012. 138

[71] Maurizio Lenzerini and Domenico Fabio Savo. On the evolution of the instance level of dl-lite knowledge bases. In *24th International Workshop on Description Logics*, page 290. Citeseer, 2011. 152, 158

[72] Maurizio Lenzerini and Domenico Fabio Savo. Updating inconsistent Description Logic knowledge bases. In *ECAI*, **242**, pages 516–521, 2012. 152, 158

[73] Hongkai Liu, Carsten Lutz, Maja Mili?i?, and Frank Wolter. Updating description logic aboxes. In *International Conference of Principles of Knowledge Representation and Reasoning*, pages 46–56, 2006. 152

[74] John W. Lloyd and Rodney W. Topor. Making prolog more expressive. *The Journal of Logic Programming*, **1**[3]:225–240, 1984. 56

[75] Luis Mandel and María Victoria Cengarle. On the expressive power of the object constraint language OCL. In *FM'99 — Formal Methods*, **1708** of *Lecture Notes in Computer Science*, pages 854–874. Springer Berlin Heidelberg, 1999. 23, 51

[76] Slaviša Marković and Thomas Baar. *An OCL Semantics Specified with QVT*, pages 661–675. Springer, Berlin, Heidelberg, 2006. 24

[77] Mirjana Mazuran, Elisa Quintarelli, Letizia Tanca, and Stefania Ugolini. Semi-automatic support for evolving functional dependencies. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16*, pages 293–304, 2016. 147

[78] Giansalvatore Mecca, Guillem Rull, Donatello Santoro, and Ernest Teniente. Ontology-based mappings. *Data & Knowledge Engineering*, **98**:8 – 29, 2015. Research on conceptual modeling. 112

[79] Stephen J Mellor, Tony Clark, and Takao Futagami. Model-driven development: guest editors' introduction. *IEEE software*, **20**[5]:14–18, 2003. 4

[80] Daniel P. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 42–47. AAAI Press, 1987. 90

[81] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 455–464, 2003. 145, 146

[82] Antoni Olivé. Integrity constraints checking in deductive databases. In *Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB)*, pages 513–523, 1991. 8, 11, 12, 55, 57, 68, 69, 170

[83] Antoni Olivé. Conceptual schema-centric development: A grand challenge for information systems research. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, pages 1–15, 2005. 4

[84] Antoni Olivé. *Conceptual modeling of information systems*. Springer Science & Business Media, 2007. 2, 4, 19, 133, 141

[85] Antoni Olivé and Jordi Cabot. A research agenda for conceptual schema-centric development. In *Conceptual Modelling in Information Systems Engineering*, pages 319–334. Springer, 2007. 3

[86] OMG. *Unified Modeling Language (UML) Superstructure Specification, version 2.4.1*. Object Management Group (OMG), 2011. http://www.omg.org/spec/UML/. 4, 22, 141

[87] OMG. *Object Constraint Language (UML), version 2.4*. Object Management Group (OMG), 2014. http://www.omg.org/spec/OCL/. 4, 23, 24, 32, 33, 66, 141

[88] Xavier Oriol and Ernest Teniente. Ejecución de operaciones de un esquema conceptual de forma persistente y consistente. In *Actas: Doctoral Consortium: Jornadas Sistedes*, 2014. 15

[89] Xavier Oriol and Ernest Teniente. Incremental checking of OCL constraints through SQL queries. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, pages 23–32, 2014. 15

[90] Xavier Oriol and Ernest Teniente. Incremental checking of ocl constraints with aggregates through sql. In *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, pages 199–213. Springer International Publishing, 2015. 14, 24, 34

[91] Xavier Oriol and Ernest Teniente. Simplification of uml/ocl schemas for efficient reasoning. *Journal of Systems and Software*, **128**:130 – 149, 2017. 13

[92] Xavier Oriol, Ernest Teniente, and Guillem Rull. TINTIN: a tool for incremental integrity checking of assertions in SQL server. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*, pages 632–635, 2016. 13, 14

[93] Xavier Oriol, Ernest Teniente, and Guillem Rull. Tintin: comprobación incremental de aserciones sql. In *Actas de las XXI Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2016. 15

[94] Xavier Oriol, Ernest Teniente, and Albert Tort. Fixing up non-executable operations in uml/ocl conceptual schemas. In *Conceptual Modeling: 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings*, pages 232–245. Springer International Publishing, 2014. 14

[95] Xavier Oriol, Ernest Teniente, and Albert Tort. Computing repairs for constraint violations in uml/ocl conceptual schemas. *Data & Knowledge Engineering*, **99**:39 – 58, 2015. Selected Papers from the 33rd International Conference on Conceptual Modeling (ER 2014). 13, 25, 34

[96] Xavier Oriol Hilari. Verificació i validació d'esquemes conceptuals uml/ocl amb operacions, 2012. 143

[97] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at `http://www.w3.org/TR/owl2-overview/`. 10

[98] Elena Planas, Jordi Cabot, and Cristina Gómez. Verifying action semantics specifications in UML behavioral models. In *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, pages 125–140, 2009. 146

[99] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on Data Semantics X*, pages 133–173, 2008. 155

[100] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. Resolving model inconsistencies using automated regression planning. *Software and System Modeling*, **14**[1]:461–481, 2015. 145

[101] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: finite reasoning on UML/OCL conceptual schemas. *Data & Knowledge Engineering*, **73**:1–22, 2012. 25, 52

[102] Anna Queralt and Ernest Teniente. *Reasoning on UML Conceptual Schemas with Operations*, pages 47–62. Springer, Berlin, Heidelberg, 2009. 140, 143

[103] Anna Queralt and Ernest Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *ACM Trans. Softw. Eng. Methodol.*, **21**[2]:13, 2012. 10, 19, 33, 51, 52, 56, 60, 65, 66, 99, 113, 125, 126, 147, 171

[104] Natxo Raga Llorens. Raonament sobre l'efecte dels esdeveniments estructurals en un esquema conceptual uml. 2014. 136

[105] Manuel Roldán and Francisco Durán. Dynamic validation of OCL constraints with modcl. *ECEASST*, **44**, 2011. 89, 140

[106] Guillem Rull, Carles Farré, Anna Queralt, Ernest Teniente, and Toni Urpí. Aurus: explaining the validation of UML/OCL conceptual schemas. *Software and System Modeling*, **14**[2]:953–980, 2015. 56, 120

[107] Guillem Rull Fort. Validation of mappings between data schemas. 2011. 99, 125, 126, 147, 171

[108] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management*, pages 133–140, 2002. 152, 158

[109] Ernest Teniente and Antoni Olivé. Updating knowledge bases while maintaining their consistency. *The VLDB Journal*, **4**[2]:193–241, 1995. 12, 93

[110] Vadim Tropashko and Donald Burleson. *SQL Design Patterns: Expert Guide to SQL Programming*. Rampant Techpress, 2007. 3, 85

[111] Axel Uhl, Thomas Goldschmidt, and Manuel Holzleitner. Using an OCL impact analysis algorithm for view-based textual modelling. *ECEASST*, **44**, 2011. 89

[112] Jeffrey D Ullman. Information integration using logical views. In *International Conference on Database Theory*, pages 19–40. Springer, 1997. 99, 117

[113] Dirk Van Dalen. *Logic and structure*, **3**. Springer, 1994. 123

[114] David Weber, Jakub Szymanek, and Moira C. Norrie. *UnifiedOCL: Achieving System-Wide Constraint Representations*, pages 221–229. Springer International Publishing, 2016. 90

[115] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. *Software and System Modeling*, **7**[3]:287–301, 2008. 147

[116] Marianne Winslett. *Updating Logical Databases*. 1990. 152, 158