Using Reasoning to Support ORM Conceptual Modelling and its Application in Information Systems

DISSERTATION

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Francesco Sportelli
geboren am 28. Oktober, 1986 in Castellaneta, Italy

GUTACHTER:
Prof. Dr. Till Mossakowski
Otto-von-Guericke Universität Magdeburg, Germany

Prof. Dr. Enrico Franconi
Freie Universität Bozen, Italy

Prof. Dr. Pablo Rubén Filottrani
Universidad Nacional del Sur, Argentina

Magdeburg, den 07/09/2021

Declaration

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Bolzano, December 2020

Francesco Sportelli

Zusammenfassung

Deutsch

Die konzeptionelle Modellierung ist ein kritischer Schritt bei der Software-Entwicklung. Ihr Zweck ist es, relevante Aspekte der Anwendungsdomäne in einer Sprache zu beschreiben, die für alle am Projekt beteiligten Akteure verständlich ist. Eine Möglichkeit, die konzeptuelle Modellierung zu implementieren, ist die faktenbasierte Modellierung, eine Methodik, die in der Lage ist, die konzeptuellen Modellierungs Constraints darzustellen und dabei auch die Semantik zu erfassen, um Zwischen- und Endergebnisse zu validieren.

Object-Role Modelling (ORM) ist eine faktenbasierte Sprache für die Modellierung und Abfrage von Informationen auf der konzeptionellen Ebene durch eine grafische und textuelle Sprache zur Spezifizierung von Modellen, Abfragen und Prozeduren, um die Zuordnung zu anderen Arten von Modellen wie UML und ER durchzuführen.

Konzeptionelle Modelle allein sind nicht in der Lage, die Semantik der Modelle zu überprüfen, und diese Einschränkung kann zu impliziten Konsequenzen führen, die vom Modellierer insbesondere in komplexen Diagrammen unentdeckt bleiben können; dies kann auch zu verschiedenen Formen von Inkonsistenzen oder Redundanzen im Diagramm selbst führen, die eine Verschlechterung der Qualität des Designs und/oder erhöhte Entwicklungszeiten und -kosten zur Folge haben. Dieses Problem führt zu der Notwendigkeit automatisierten Schließens, um die genannten Inkonsistenzen und Redundanzen zu überprüfen.

Das automatisierte Schließen ist ein bekanntes Verfahren, das einen logischen Prozess verwendet, bei dem eine Schlussfolgerung auf mehreren Prämissen beruht, die im Allgemeinen als wahr angenommen werden. Unter logischem Schließen verstehen wir die Ableitung von Fakten, die in unserem ORM-Diagramm nicht explizit ausgedrückt sind.

Die Anwendung des automatisierten Schließens auf die konzeptuelle Modellierungsmethodik hat einige Vorteile, da sie den Modellierer während der Modellierungsphase unterstützt, um Fehler zu vermeiden. Redundanzen oder Inkonsistenzen des Diagramms, die Ableitung neuer Constraints, die Bestätigung der Gültigkeit des Modells oder der Vorschlag einer Überarbeitung sind weitere Vorteile.

Diese Schritte sind vor allem für Kontexte, in denen riesige konzeptuelle Diagramme verwendet werden, in denen es sehr schwierig und zeitaufwändig ist, die Semantik der Diagramme manuell zu überprüfen, zeitsparend.

Die Hauptidee dieser Arbeit besteht darin, eine Methodik zur Anwendung des automatisierten Schließens auf konzeptuelle Modelldiagramme zu entwickeln, um die Semantik der Diagramme zu überprüfen, damit die Vorteile des automatisierten Schließens genutzt werden können. In dieser Arbeit wird die Methodik auf ORM angewandt, eine mächtige Sprache die eine Vielzahl von Constraints bereitstellt.

Unter all diesen Constraints gibt es ORM-Ableitungsregeln, die in der Lage sind, Wissen auszudrücken, das über die Standard-ORM-Fähigkeiten hinausgeht, was zu einer weiteren Komplexität das Schließens führt, da sie der ORM-Sprache Ausdruckskraft verleihen.

Diese Regeln ähneln in gewisser Weise den OCL-Constraints für UML oder SQL-Triggers.

Ein weiterer Beitrag dieser Arbeit besteht in der Formalisierung der ORM-Ableitungsregeln. Auf diese Weise wird es möglich sein, das automatisierte Schließen auch auf die ORM-Diagramme auszudehnen, die mit ORM-Ableitungsregeln ausgestattet sind.

Ein Beitrag mit einem eher praktischen Charakter wird durch die Implementierung eines Frameworks namens UModel gegeben, das das automatisierte Schließen über konzeptuellen Diagrammen anwendet. Obwohl der Schwerpunkt der Verwendung des Frameworks in dieser Arbeit auf ORM liegt, wurde das Framework so konzipiert, dass es mit den gängigsten konzeptuellen Modellierungssprachen wie UML und ER kompatibel ist.

ORM ist in einer offiziellen Microsoft Visual Studio-Erweiterung namens NORMA implementiert, die es dem Benutzer ermöglicht, ORM-Diagramme zu erstellen, zu modifizieren und zu exportieren, und die eine Vielzahl von Funktionen bereitstellt, die dem Modellierer bei der Verwaltung seines ORM-Diagramms helfen.

Obwohl NORMA ein leistungsstarkes Werkzeug ist, ist es nicht in der Lage, die Konsistenz der Modelle zu überprüfen, und aus diesem Grund könnte die Ausstattung mit Schlussfolgerungfunktionen ein Schritt nach vorn sein, um die Qualität der ORM-Diagramme zu erhalten.

Ein weiterer Teil dieser Arbeit ist die Erweiterung der NORMA-Funktionalitäten durch ein Plugin unter Verwendung des UModel-Frameworks, das automatische Schließen über die in NORMA geladenen ORM-Diagramme aktiviert und dann den Endbenutzern die Schlussfolgerungen anzeigt.

ORM wird auch in industriellen Unternehmen eingesetzt. Diese Unternehmen verwenden normalerweise CASE-Tools, mit denen sie die konzeptuellen Diagramme erstellen können. Diese Tools berücksichtigen nicht die Überprüfung

der Semantik der konzeptuellen Diagramme, und eine weitere Herausforderung besteht darin, das oben erwähnte automatisierte Schlussverfahren für eine Zielsoftware zu verallgemeinern.

Aus diesem Grund wurde eine Fallstudie auf der Grundlage eines realen industriellen Szenarios erstellt, um mögliche Vorteile, die sich aus der verwendeten Methodik ergeben, konkret zu messen und zu beobachten.

Abstract

English

Conceptual modelling is a critical step during software development. Its purpose is to describe relevant aspects of the application domain in a language that is understandable by all the stakeholders taking part to the project. A way to implement conceptual modelling is by using fact-based modelling, a methodology that is able to represent the conceptual modelling constraints capturing also the semantics in order to validate intermediate and final results.

Object-Role Modelling (ORM) is a fact-based language for modelling and querying information at the conceptual level by a graphical and textual language for specifying models, queries and procedures to perform the mapping to other kinds of models like UML and ER.

Conceptual models alone are not able to check the semantics of the models and this limitation may lead to implicit consequences that can go undetected by the modeller especially in complex diagrams; this may also lead to various forms of inconsistencies or redundancies in the diagram itself that give rise to the degradation of the quality of the design and/or increased development times and costs. This issue leads to the need of automated reasoning to check the mentioned inconsistencies and redundancies.

Automated reasoning is a well-known procedure that uses a logical process in which a conclusion is based on multiple premises that are generally assumed to be true. By reasoning we mean deriving facts that are not expressed in our ORM diagram explicitly. Applying automated reasoning to the conceptual modelling methodology has some benefits, since it supports the modeller during the modelling phase in order to avoid mistakes, as redundancies or inconsistencies of the diagram, deriving new constraints, confirm the validity of the model or suggest revision. These steps are a time saver especially for contexts where huge conceptual diagrams are used, where it is very difficult and time consuming to manually check the semantics of the diagrams.

The main idea of this work is to develop a methodology to apply the automated reasoning on conceptual modelling diagrams to check the semantics of the diagrams, in order to take the benefits from the automated reasoning. In this work the methodology is applied to ORM which is a powerful language providing a rich set of constraints.

Among all the constraints there are ORM Derivation rules which are able to express knowledge that is beyond the standard ORM capabilities, bringing to a further complexity of the reasoning because they add expressiveness to

the ORM language. Those rules are in a way similar to OCL constraints for UML, or SQL triggers. Another contribution of this work is to formalise ORM Derivation rules, in this way it will be possible to extend the automated reasoning even on those ORM diagrams equipped with ORM Derivation Rules.

A contribution with a more practical flavour is given by the implementation of a framework named UModel which applies the automated reasoning over conceptual diagrams. Although the usage of the framework in this work has its focus on ORM, the framework has been designed to be compatible with the most popular conceptual modelling languages such as UML and ER.

ORM is implemented in an official Microsoft Visual Studio extension named NORMA, which allows the user to create, modify and export ORM diagrams and which provides a rich set of functionalities to help the modeller to manage its ORM diagram. Despite NORMA being a powerful tool, it is not able to check the consistency of the models and for this reason equipping it with reasoning capabilities could be a step forward in order to preserve the quality of the ORM diagrams. A contribution of this work is the extension of NORMA functionalities by a plugin, using the UModel framework, that activates the automated reasoning over those ORM diagrams loaded into NORMA and then showing the inferences to the final users.

ORM is also used in the industry world companies. Those organisation usually use CASE tools that allow them to build the conceptual diagrams. These tools does not take into account checking the semantics of the conceptual diagrams and another challenge is to generalize the aforementioned automated reasoning procedure for a target software. For this reason, a case study based on real-world industrial scenario has been provided in order to concretely measure and observe possible benefits coming from the used methodology.

Contents

1

Introduction and Motivation

We briefly outline the scope and motivation of the thesis. This work is largely concerned with the formal specification of fact-base modelling structures, precisely the fact-based language ORM. For reasons such as correctness and clarity, information systems are best specified first at the conceptual level. For large database systems applications, conceptual modelling software have generally become the most important target systems onto which conceptual information structures are mapped. Designing correct conceptual and relational schemas for practical applications is a non-trivial task and the wrong design could lead to bad consequences in the development life cycle. The main goal of the thesis is to provide a concrete methodology in order to improve the conceptual modelling by automated reasoning. In this way the process of making design choices could help the modeller, and in general all the stakeholders, to build a robust software infrastructure by taking under control the semantics of the models.

The way a model is designed has a direct impact in the real world where conceptual modelling tools are used to manage complex domains. These tools also known as CASE tools (Computer-aided software engineering), are powerful systems which accelerate the development of the software, providing a set of powerful features to model a domain. They use conceptual modelling languages which are closer to the way we abstract the world in our cognition, making them ideal to model a domain. The limitation of the conceptual modelling tools is that they lack semantics check capabilities. This limitation could lead to software degradation and unexpected software behaviours, especially for large-scale environment this could be a serious issue. The core

idea of the present work is to provide a methodology that overcomes this limitation, expanding the capabilities of the conceptual modelling software such that they are not limited to check the syntax of the diagrams, but check the semantics as well. In this way it could enhance the trust of the system or suggest the revision, preventing serious issues during the next stages of the software development.

The methodology to accomplish this goal is grounded on the formalisation of ORM language that allows to activate reasoning procedures, carried out by Description Logics reasoners. The reasoning procedures are able to perform semantic checks over ORM diagrams, in this way it is possible to overcome the aforementioned limitation.

The research follows two main tracks: the theoretical aspects related to ORM and Derivation Rules formalisation; the implementation counterpart concerning the creation of a set of tools in order to support the modeller.

Outlined are the main goals of the research:

- **Goal 1**: providing an encoding for ORM in OWL;
- **Goal 2**: providing a formalisation for ORM Derivation Rules, both Subtype and FactType;
- **Goal 3**: implementation of a framework to enable automated reasoning for conceptual modelling languages and conceptual modelling software as well;
- **Goal 4:** application of the methodology in a real case scenario.

The fulfilment of the first goal is the prerequisite to achieve the other goals since it constitutes the first step of the methodology.

The second goal is an extension of the first one. Although several papers presented their own ORM formalisation, no one has taken into account the formalization of ORM derivation rules so far. Derivation rules express knowledge that is beyond normal ORM capabilities, but this feature leads to an increase of expressiveness of the ORM diagrams. For this reason, the

challenge is to identify a decidable fragment in order to extend the reasoning even to those ORM diagrams equipped with those rules.

Reaching the third goal would give to the community a tool useful to expand the features of a target application. The direct impact of this goal involves various actors, such as the modellers, the database or software engineers. The versatility of the software makes possible to enrich any conceptual modelling software of reasoning capabilities, making the impact of the research in particular relevant for the industry world.

A successful result in the last goal could demonstrate the efficiency of the methodology applied to real cases.

1.1 Summary of major contributions

The major research contributions of this thesis are hereby summarized. The whole work is a combination of theoretical aspects and their methodological counterpart, so they are grouped as follows:

- **Theoretical**
  - OWL encoding of the ORM language;
  - formalisation of ORM Derivation Rules;
  - detection of a decidable fragment for ORM and ORM Derivation Rules.
- **Methodological**
  - Building a framework embeddable into other systems;
  - implementation of the theoretical points and integration into the framework;
  - executing the workflow on a real-case study for industry.

1.2 Structural overview

Following there is a structured overview of the thesis:

1. This chapter is meant to introduce the scope of the thesis along its motivation and summarizing the main contributions.

2. The second chapter presents a list of arguments that the reader should be familiar with. Here it is explained the need of the conceptual modelling in software development and the related fact-based methodology. The ORM language is introduced. It is a dialect of the fact-based methodology which is also the main focus of this thesis. ORM Derivation Rules are also introduced, which are one of the core components of this thesis.

3. The third chapter presents the ORM formalisation used in this work. ORM constructs and ORM Derivation Rules are mapped in first-order logic.

4. The forth chapter shows some use cases where the automated reasoning is applied to the ORM diagrams in order to show its benefits. In this chapter are also provided some reasoning examples with ORM Derivation Rules.

5. The fifth chapter defines a decidable fragment for ORM where the language $\mathcal{DLR}^{\pm}$ is used to encode the ORM constraints. This decidable fragment is the one used in the scope of this thesis.

6. The six chapter present a discussion about the difference between conceptual modelling in OWL and ORM.

7. The seventh chapter speaks about the UModel framework. It is an implementation of the work presented so far. UModel is a framework that comes with a design specifically built to ease the process of integrating the automated reasoning in any conceptual modelling software.

8. The eighth chapter is about ORMIE, a plugin for NORMA used in a real-case scenario by the European Space Agency. This tool integrates the UModel framework to enable the automated reasoning over ORM diagrams loaded inside NORMA. A benchmark is also provided.

9. The ninth chapter is a collection of tools similar to ORMIE in order to provide a comparison.

2

The ORM language

This chapter introduces some concepts that the reader should be familiar with in order to read and understand the next chapters.

2.1 Conceptual modelling

Information modelling is about the representation of symbol structures that model some aspects of the real world. In Computer Science such structures are defined as databases or knowledge bases, that represent a part of information in the real world that is modelled into a system. We shall refer to the part of a real world being modelled by an information base as its universe of discourse (UofD), also known as application domain. Databases and knowledge bases are checked for consistency, and sometimes queried and updated through special-purpose languages. As with all models, the advantage of information models is that they abstract away irrelevant details, and allow more efficient examination of both the current, as well as past and projected future states of the UofD. An information model is represented by a specific language, and this language influences the kinds of details that are considered. A language provides the semantics for modelling an application, such as entity and relationship, as well as means for organizing information. Conceptual models are used in different areas, for example in Artificial intelligence, where programs require the representation of the human knowledge in order to act with intelligence. These programs may rely on conceptual models built up using knowledge representation languages such as DLs. Conceptual languages are also suitable for database design where the first step is crucial for the

construction of a conceptual schema which determines the information needs of the users, and that may be then converted to a physical implementation schema. Chen's Entity-Relationship model [43], and later semantic data models [98] were the result of efforts in this direction. In software development, the early acquisition stage it is a delicate step, which is seen to consist of a requirements model that describes the relationship of the proposed system and its environment, that is represented by a conceptual model. Moreover, the object-oriented software community has also proposed viewing software components (classes/objects) as models of real-world entities. This was evident in the features of Simula, the first object-oriented programming language, and became a cornerstone of most object-oriented techniques, including the current leader, UML [27]. One interesting aspect of conceptual modelling recurring in database development is the abstraction mechanism to support large conceptual models by abstracting details initially, and then introducing them in a step-wise way. Important abstractions are the capacity to think of objects as wholes, not just a collection of their attributes/components (aggregation); also abstracting the differences among individuals in order to be classified (classification); and abstracting the hierarchy structure of a set of conceptually related classes (generalization). The benefit of the abstraction in conceptual modelling is that the information are structured making the model easy to develop and maintain. A smart way to build up a model is to encode it in Description Logics, a family of knowledge representation languages widely used in artificial intelligence to describe and reason about the relevant concepts of an application domain. In the context of the conceptual modelling, it is possible to use DLs as a reasoning backend to take advantage of DLs properties which reveal some formal properties that may not have been recognized by the modeller. From the implementation perspective then it comes OWL, the Web Ontology Language [136]. OWL is based on Description Logics and may be coupled with a reasoner to perform the automated reasoning over the conceptual model it is representing. Unlike other languages coming from the Description Logics family, such as $\mathcal{DLR}^{+}$ [10], OWL has only binary predicates.

2.2 The fact-based modelling methodology

Fact-based modelling (FMB) is the methodology we use in this work to represent conceptual diagrams [76], [107]. FBM is used for modelling the semantics of a specific domain of interest for the purpose of developing information systems, rule systems or sharing information. The main purpose of fact based modelling is to capture as much of the semantics as possible, bridging the gap among stakeholders preferably using concrete illustrations and to remain independent of the representation for a specific implementation. Unlike Entity-Relationship (ER) modelling or object-oriented modelling, fact based modelling treats all facts as relationships (unary, binary, ternary etc.). How facts are grouped into structures (e.g. attribute-based entity types, classes, relation schemes, XML schemas) is considered a software design level, implementation issue that is irrelevant to the capturing of business semantics. Avoiding attributes in the base model enhances semantic stability and understandability. Fact based modelling facilitates natural verbalization and thus enables productive communication with all stakeholders. Fact based modelling provides the means to capture the knowledge of the domain experts in terms of "what" (i.e. the user requirements). FBM is conceptual, hence free of any software implementation bias.

FBM is based on logic and controlled natural language, whereby the resulting fact based model (the conceptual data model) captures the semantics of the domain of interest by means of fact types, together with the associated concept definitions and the integrity and derivation rules applying to populations (facts) associated with these fact types.

All facts, constraints and derivation rules are expressed in controlled natural language sentences that are intelligible to users in the business domain being modelled. In addition to textual verbalization of data models, FBM includes graphical notations for depicting data models with a rich variety of constraints. For example in Figure 2.1, we have a graphical representation of the following fact types:

```
Monument is ancient
Monument is located in Country
```

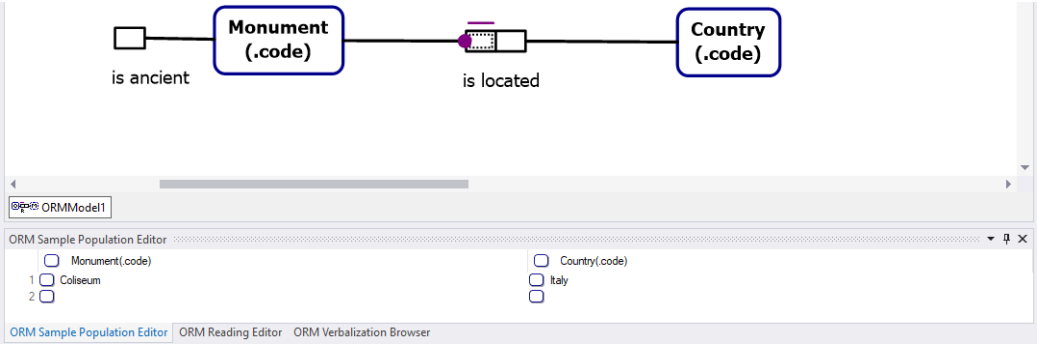


Figure 2.1: ORM example - Monument

along its instances:

```
Coliseum is ancient
Coliseum monument is located in Italy
```

Here "is ancient" denotes a unary predicate, and "is located in" a binary predicate.

In Entity-Relationship (ER) and Unified Modelling Language (UML) approaches, the unary fact type would instead typically be modelled as the attribute assignment Volcano.isActive = true. Besides facilitating more natural expression, and ease of population with multiple instances, the usage of attributes in favour of relationships promotes semantic stability (e.g. there is no need to remodel what has already been modelled if we later decide to record facts about an attribute). Fact types of higher arity (ternaries, quaternaries etc.) are also allowed. For flexibility, and to cater for foreign languages, predicates may be represented in mixfix form, where the terms for the objects being predicated over are inserted in relevant placeholder positions to form the fact sentence. For example, the fact type "Person plays Sport for Country" involves the ternary predicate reading "...plays...for...".

The FBM approach originated in Europe in the 1970s, and has since evolved into a family of closely related dialects including Object-Role Modelling (ORM), Cognition enhanced Natural Language Information Analysis Method (CogNIAM), Fully Communication oriented Information Modelling (FCO-IM) and the Developing Ontology-Grounded Methods and Applications (DOGMA)

method. In the following section we speak about ORM, which is the language we use in this work. In the next section we present the history and the foundations of the ORM language, plus some illustrative examples.

2.3 ORM

2.3.1 History

The history of ORM is grounded in 1973, when Falkenberg generalized work by Abrial and Senko on binary relationships to n-ary relationships, and excluded attributes at the conceptual level to avoid "fuzzy" distinctions and to simplify schema evolution. Later, Falkenberg proposed the fundamental ORM framework, which he called the "object-role model" [50]. This framework allowed n-ary and nested relationships, but depicted roles with arrowed lines. Nijssen adapted this framework by introducing a circle-box notation for objects and roles, and adding a linguistic orientation and design procedure to provide a modelling method called ENALIM (Evolving NAtural Language Information Model). Nijssen's team of researchers at Control Data in Belgium developed the method further, including van Assche who classified object types into lexical object types (LOTs) and non-lexical object types (NOLOTs). Today, LOTs are commonly called "entity types" and NOLOTs are called "value types". Meersman added subtyping to the approach, and made major contributions to the RIDL query language [111] with Falkenberg and Nijssen. The method was renamed "aN Information Analysis Method" (NIAM). Later, the acronym "NIAM" was given different expansions, and is now known as "Natural language Information Analysis Method" [139], [138]. In the 1980s, Nijssen and Falkenberg worked on the design procedure and moved to the University of Queensland, where the method was further enhanced by Halpin, who provided the first full formalization, including schema equivalence proofs, and made several refinements and extensions. In 1989, Halpin and Nijssen coauthored a book on the approach, followed a year later by Wintraecken's book [139]. Today several books, including major works by Halpin [79], and Bakema [15] expound on the approach. Many researchers contributed to the fact oriented approach over the years, and there is no space here to list

them all. Today various versions exist, but all adhere to the fundamental object-role framework. Habrias developed an object-oriented version called MOON (Normalized Object-Oriented Method). The Predicator Set Model (PSM), developed mainly by ter Hofstede et al. [89], includes complex object constructors. De Troyer and Meersman developed a version with constructors called Natural Object-Relationship Model (NORM). Halpin developed an extended version simply called ORM, and with Bloesch and others developed an associated query language called ConQuer [28]. Bakema et al. [15] recast all entity types as nested relationships, to produce Fully Communication Oriented NIAM, which they later modified to Fully Communication Oriented Information modelling (FCO-IM). More recently, Meersman and others adapted ORM for ontology modelling, using a framework called DOGMA. Nijssen and others extended NIAM to a version called NIAM2007. Halpin and others developed a second generation ORM (ORM 2), whose graphical notation is used in this work [45, 92].

The formalisation of ORM represents a further step to the usage of ORM both in the logicians' community and in the industry world. Formalising ORM enables automated reasoning over ORM conceptual diagrams; in this way, it is possible to detect relevant formal properties automatically, in order to check the semantics of the ORM diagrams. In a real-world context, this could be useful to support the modeller during the initial step of software development, which is the design step. The ORM formalisation started with Terry Halpin's PhD Thesis [74]. In the context of design conceptual and relational schemas, Halpin formalized the NIAM language that is the ancestor of ORM. In his thesis there is the first attempt to formalize a modelling language in order to perform reasoning tasks, so the main objective is to provide formal basis for reasoning about conceptual schemas and for making decision choices. After the spreading of ORM and its implementation in NORMA [44],[91], [125], ORM became more popular so the logicians' community took into account the possibility to formalize this very expressive language.

In 2005, Terry Halpin releases a new version of ORM, namely ORM2 [75]. From now on we refer to ORM2 as ORM.

In 2007, Jarrar formalizes ORM using $\mathcal{DLR}_{ifd}$ [102], an extension of Description Logics introduced by Calvanese in [37]. The paper shows that a formalisation in OWL $\mathcal{SHOIN}$ would be less efficient than $\mathcal{DLR}_{ifd}$ because some ORM constraints cannot be translated (predicate uniqueness, external uniqueness, set-comparison constraints between single roles and between not contiguous roles, objectification n-ary relationships). Another formalisation of ORM in $\mathcal{DLR}_{ifd}$ was done by Keet in [106]. During the same year also Bach in [101] analysed the semantics of OWL-DL and ORM, explaining how to represent OWL-DL constraints in ORM without losing semantics. In [104] Jarrar mapped ORM into the $\mathcal{DLR}_{ifd}$, which is one of the most expressive description logics. Every ORM constraint is decidable, except two rare cases. $\mathcal{DLR}_{ifd}$ was developed indeed to allow the majority of the database primitives to be represented, including n-ary relations, identities, and functional dependencies. However, the problem is that not all $\mathcal{DLR}_{ifd}$'s constraints are implemented by current reasoning engines. In this paper the mapping translates ORM into the $\mathcal{SHOIN}$ description logic, which is the logic underpinning OWL (only version 1), the standard (W3C recommendation) Ontology Web Language. $\mathcal{DLR}_{ifd}$ was developed as a compromise between expressive power and decidability. This implies that the ORM mappings into $\mathcal{SHOIN}$ are easier to implement and exploit. $\mathcal{SHOIN}$/OWL is supported in almost all reasoning engines, and it is the most popular language in ontology engineering. However, $\mathcal{SHOIN}$/OWL does not support some ORM constraints like n-ary relations and external uniqueness. The result of the paper is the encoding of 22 mapping rules out of the 29 ORM constraints.

In 2009 OWL 2 was recommended by W3C Consortium as a standard of ontology representation on the Web bringing some benefits: it is the recommended ontology web language; it is used to publish and share ontologies on the Web semantically; it is used to construct a structure to share information standards for both human and machine consumption; automatic reasoning can be done against ontologies represented in OWL 2 to check consistency and coherency of these ontologies [136].

An ORM formalisation based on OWL2 is proposed by Franconi in [62], where he introduces a new linear syntax and FOL semantics for a generalization of

ORM2 [75], called ORM2plus, allowing the specification of join paths over an arbitrary number of relations. The paper also identifies a core fragment of ORM2, called ORM2zero, that can be translated in a sound and complete way into the ExpTime-complete Description Logic $\mathcal{ALCQI}$. In [57] a provably correct encoding of a fragment of ORM2zero into a decidable fragment of OWL2 is provided and it is discussed how to extend ORM2zero in a maximal way by retaining at the same time the nice computational properties of ORM2zero.

The most recent paper related to ORM formalisation is [10] where the language $\mathcal{DLR}^+$ is used to encode a decidable fragment of ORM. $\mathcal{DLR}^+$ like $\mathcal{DLR}_{ifd}$ is an extension of $\mathcal{DLR}$, a description logic representing a natural generalization of traditional description logics towards nary relations [16]. The feature of this language is to represent n-ary relationships which are suitable for languages like ORM. The backbone of this work is based on the decidable fragment of $\mathcal{DLR}^+$, which is named $\mathcal{DLR}^{\pm}$. $\mathcal{DLR}^{\pm}$ is provided with an encoding in $\mathcal{ALCQI}$ and the languages are equisatisfiable. Since it is proved that this fragment captures a significant fragment of ORM, it is used in this work to encode some ORM constraints in order to perform a mapping into OWL and take advantage of logical reasoning. Since OWL is essentially $\mathcal{SROIQD}$ the encoding in $\mathcal{ALCQI}$ covers a fragment of OWL.

All the aforementioned formalisations do not include ORM Derivation Rules as part of the formalisation. The present work aims to cover this part of the ORM language in order to provide a full ORM formalisation.

Despite ORM Derivation Rules have not being formalised, some papers explore them in a different context. For example, in [115] a classification of the most popular rule types is presented, where the type of rules taken into account are integrity rules, derivation rules, production rules, and reaction rules. The work presents a general overview of these different types of rules, including ORM derivation rules, but the handbook is not meant to provide a formalisation for such rules, but only a general classification with the goal to highlight their functionalities in the context of rule modelling.

In [46] an approach to navigate through an ORM model by roles is described, namely role paths. A role path represents a traversal of related roles, starting

with one or more roles connected to a root object type. Each subsequent role in a path either is a role in the same fact type as the previous role, or involves a join operation to a role with the same role player. That work introduces the role paths as a foundation for both subtype and fact type derivation rules. The metamodel described in the paper is currently implemented as the basis for formal derivation rules in the NORMA tool. Users will formulate derivation rules via high level graphical and textual views that are automatically transformed into the low level role path structures. It is important to state that role paths have been implemented by object-oriented data structures in NORMA, so it is a pure object-model. In other words, this is a way to traverse an ORM diagram that can be equipped with ORM Derivation Rules as well, but it is not a formalisation as in classical logic with a syntax, a semantics and a mapping.

Other papers take into account different kind of rules such as dynamic rules [22], [23]. These dynamic rules specify an elementary transaction type by indicating which kinds of objects or facts (being added, deleted or updated) are involved. Dynamic rules may declare pre-conditions relevant to the transaction, and a post-condition stating the properties of the new state, in a way similar to SQL triggers.

The Section 2.3.3 introduces ORM Derivation Rules in order to give to the reader the appropriate knowledge to understand the rest of the work.

2.3.2 ORM constraints

Object-Role modelling (ORM) is a fact-based language for modelling and querying the information semantics of business domains in terms of the underlying facts of interest, where all facts and rules may be verbalized in language easy to understand for non-technical users. Since ORM is fact-based, it differs from UML and ER [81] [83], treating all facts as relationships of arbitrary arity (unary, binary, ternary etc.); for this feature ORM is said to be attribute-free. Avoiding attributes in the base model enhances semantic stability and natural verbalization, facilitating communication with all stakeholder taking part to model their are working on.

ORM includes graphical and textual languages for modelling and querying information at the conceptual level, as well as procedures for designing conceptual models, of transforming between different conceptual representations, forward engineering ORM schemas to implementation schemas (e.g., relational database schemas, object-oriented schemas, XML schemas, and external schemas) and reverse engineering of implementation schemas to ORM schemas.

In ORM all fact structures are expressed as relationships with a given arity and are called fact types. These may be unary (e.g., Person smokes), binary (e.g., Person is identified by a Document), ternary (e.g., Person plays Sport for a Country), and so on. The natural verbalization makes easy to bridge the gap among stakeholders since all facts and rules may be easily verbalized in sentences understandable to the domain expert that often is not an IT person. An advantage of the attribute-free feature is that no nulls occur in populations of base fact types, which must be elementary or existential. The consequence of this feature is that a attribute-free diagram usually consumes more space, but this apparent disadvantage is easy to mitigate by the usage of an ORM tool to automatically create attribute-based structures (e.g., ER, UML class, or relational schemas) as views of an ORM schema.

Running example

Now, let us suppose we want to design in ORM a system managing the people documents for a certain country. We build this ORM diagram step by step introducing the ORM constraints along with the FORML (Formal ORM Language), a controlled natural language that encodes each ORM constraint in a language easy to understand for non-technical people. This language is useful to express in natural language the semantics of ORM constraints unambiguously. We also show the semantics in first-order logic. We start stating that a person is identified by a document. In ORM, the following statement is depicted as in Figure 2.2. The entity Person is represented by the rounded rectangle where inside is specified the name of the entity, in this case Person. This represents the instances belonging to the set of people in the system. The same applies for the entity Document. The relation is depicted

by the sequence of tiny rectangle boxes representing the ORM roles; in this case we have two roles since the relation is binary. Each role is connected to the entity involved into that relation.

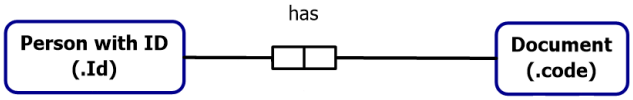


Figure 2.2: ORM diagram example - STEP 1

In FORML we have:

```
Person with ID is an entity type.
Document is an entity type.
Person with ID has Document.
```

In FOL:

$$\forall x, y.has(x, y) \rightarrow PersonWithID(x) \wedge Document(y)$$

The ORM entities (like *Person with ID*, *Document*, etc.) do not occur in the formalisation since they are declared in the ORM signature (this will be explained in Chapter 3).

We need to specify that, as it happens in the real world, a person is identified by a document which uniquely identifies that person. This is represented in Figure. 2.3 by a purple dash on a role stating that there are no duplicate instances inside that role, this means that the cardinality is set to one, in a way similar to SQL primary keys.

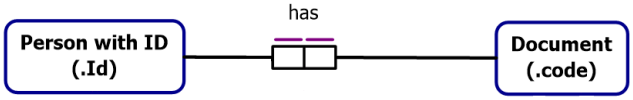


Figure 2.3: ORM diagram example - STEP 2

In FORML we have:

```
Each Person with ID has at most one Document.
For each Document, at most one Person with ID has that Document.
```

In FOL:

$\forall x, y.has(x, y) \rightarrow \exists^{\leqslant 1} z.has(x, z)$
$\forall x, y.has(x, y) \rightarrow \exists^{\leqslant 1} z.has(z, y)$

We now want to say that each person *must* be identified by a document. In other words, we want to set the mandatory participation of person to the relationship *has*. In Figure 2.4 this is depicted by the purple dot near a role.

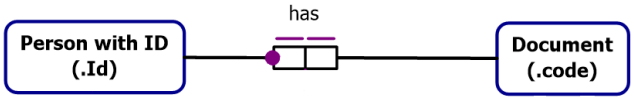


Figure 2.4: ORM diagram example - STEP 3

In FORML:

```
Each Person with ID has exactly one Document.
```

In FOL:

$\forall x.PersonWithID(x) \rightarrow \exists^{=1} y.has(x, y)$

We can add more details to the diagram. For example, we may want to make a distinction among the type of documents. In Figure 2.5 we introduce two subset of the entity Document: Visa and IDCard. Conceptually, Visa represents those documents belonging to the visitors of that country; instead, IDCard is the document for the citizen of that country. Those subset are depicted as entity types with an arrow pointing to the super type entity.

In FORML we have:

```
VISA is an entity type.
Each Visa is an instance of Document.
IDCard is an entity type.
Each IDCard is an instance of Document.
```

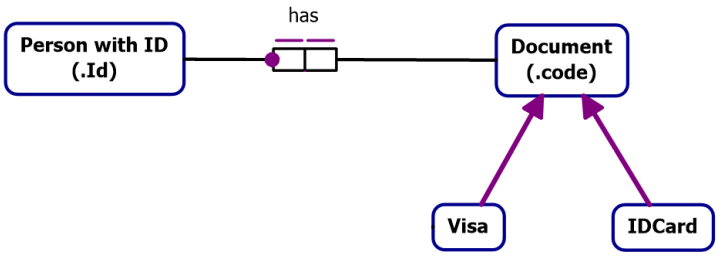


Figure 2.5: ORM diagram example - STEP 4

In FOL:

$\forall x. Visa(x) \rightarrow Document(x)$
$\forall x. IDcard(x) \rightarrow Document(x)$

In this configuration some instances inside Visa may also belong to IDCard. We need to add a constraint stating that there are not instances in common between these two entities. This is achieved by introducing the disjointness constraint depicted by a circle with a cross which is connected to the entities that are disjoint. We may want to specify that the whole set of documents is covered by the visas and idcards. This is expressed by the purple dot added inside the cross, as in Figure 2.6.

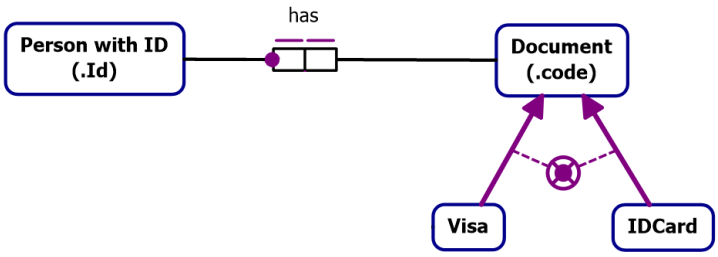


Figure 2.6: ORM diagram example - STEP 5

In FORML:

```
For each Document, exactly one of the following holds:
that Document is some Visa;
that Document is some IDCard.
```

In FOL:

$\forall x. Visa(x) \rightarrow \neg IDCard(x)$
$\forall x. Document(x) \rightarrow Visa(x) \vee IDCard(x)$

We now want to state that some people in the set of Person are visitors. Moreover, the visitor owns a visa. Please note that we have named the new entity SomeVisitor instead of Visitor, since it cannot capture exactly all the visitor; this is related to the ORM Derivation Rules and it will explained in 2.3.3.

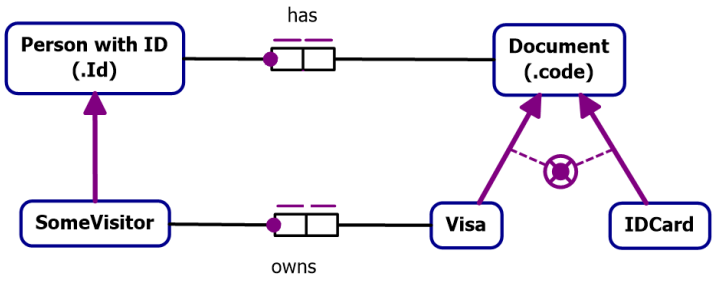


Figure 2.7: ORM diagram example - STEP 6

In FORML:

```
Each SomeVisitor is an instance of Person with ID.
SomeVisitor owns Visa.
Each SomeVisitor owns exactly one Visa.
For each Visa, at most one SomeVisitor owns that Visa.
```

In FOL:

$\forall x, y. owns(x, y) \rightarrow SomeVisitor(x) \wedge Visa(y)$
$\forall x. SomeVisitor(x) \rightarrow PersonWithID(x)$
$\forall x. SomeVisitor(x) \rightarrow \exists y. owns(x, y)$
$\forall x, y. owns(x, y) \rightarrow \exists^{\leq 1} z. owns(x, z)$
$\forall x, y. owns(x, y) \rightarrow \exists^{\leq 1} z. owns(z, y)$

We want to state that all the pairs inside the relationship *owns* are also inside the relationship *has*. We express this restriction introducing the ORM subset constraint, depicted with a $\subseteq$ inside a circle.

Finally, the complete ORM diagram is shown in Figure 2.8.

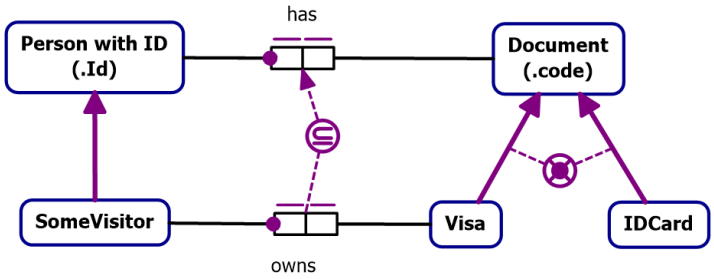


Figure 2.8: ORM diagram example - STEP 7

In FORML:

```
If some SomeVisitor owns some Visa
then some Person with ID that is that SomeVisitor
has some Document that is that Visa.
```

In FOL:

$\forall x, y.owns(x, y) \rightarrow has(x, y)$

Another example

ORM has a rich set of constraints that allow the modeller to design very expressive diagrams. An ORM schema for a book publishing domain is shown in Figure 2.9. Each book is identified by an International Standard Book Number (ISBN), each person is identified by a person number, each grade is identified by a grade number in the range 1 through 5, each gender is identified by a code ("M" for male and "F" for Female), and each year is identified by its common era (CE) number. Published Book is a derived subtype determined by the subtype definition shown at the bottom of the figure. Review Assignment

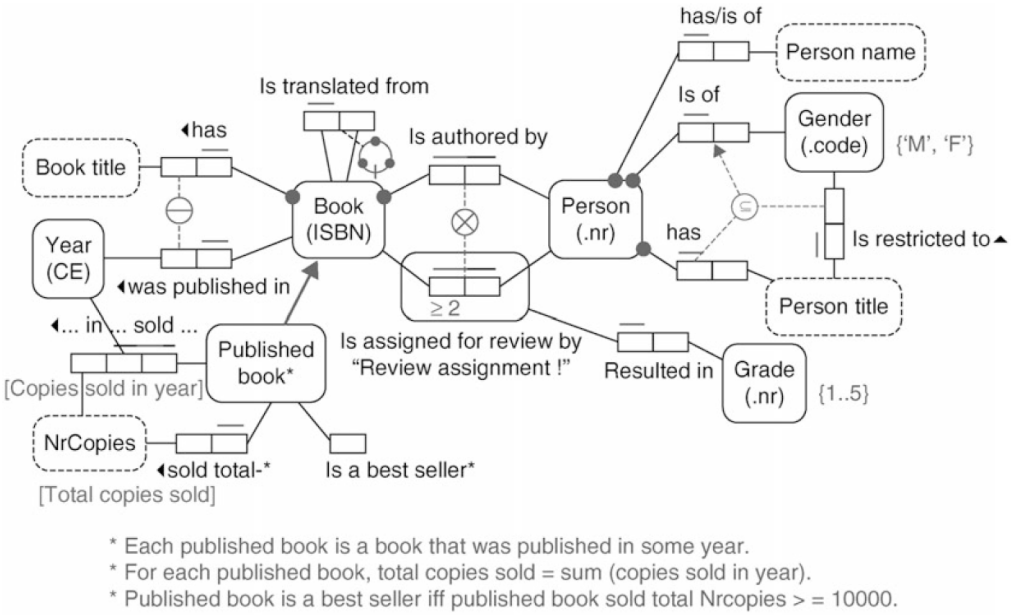


Figure 2.9: ORM Example - Book publishing domain

objectifies the relationship Book is assigned for review by Person, and is independent since an instance of it may exist without playing any other role (one can known about a review assignment before knowing what grade will result from that assignment). The internal uniqueness constraints (depicted as bars) and mandatory role constraints (solid dots) verbalize as follows: Each Book is translated from at most one Book; Each Book has exactly one Book Title; Each Book was published in at most 1 Year; For each Published Book and Year, that Published Book in that Year sold at most one NrCopies; Each Published Book sold at most one total NrCopies; It is possible that the same Book is authored by more than one Person and that more than one Book is authored by the same Person; Each Book is authored by some Person; It is possible that the same Book is assigned for review by more than one Person and that more than one Book is assigned for review by the same Person; Each Review Assignment resulted in at most one Grade; Each Person has exactly one Person Name; Each Person has at most one Gender; Each Person has at most one Person Title; Each Person Title is restricted to at most one Gender. The external uniqueness constraint (circled bar) indicates that the combination of BookTitle and Year applies to at most one Book.

The acyclic ring constraint (circle with three dots and a bar) on the book translation predicate indicates that no book can be a translation of itself or any of its ancestor translation sources. The exclusion constraint (circled cross) indicates that no book can be assigned for review by one of its authors. The frequency constraint indicates that each book that is assigned for review is assigned for review by at least two persons. The subset constraint (circled subset symbol) means that if a person has a title that is restricted to some gender, then the person must be of that gender. The first argument of this subset constraint is a person/gender role pair projected from a join path that performs a conceptual join on PersonTitle. The last two lines at the bottom of the schema declare two derivation rules, one specified in attribute style using role names and the other in relational style using predicate readings.

The list of all ORM constraints

As we have seen in the previous examples, ORM's graphical language has a rich notation that makes it easy to detect and express constraints. This graphical notation has been defined in [79] and [73]. Figure 2.10 lists the main graphical symbols of the ORM notation [90], numbered for easy reference.

An entity type (e.g., Person) is depicted as a named, soft rectangle (symbol 1), or alternatively an ellipse or hard rectangle. Value type (e.g., Person Name) shapes have dashed lines (symbol 2). Each entity type has a reference scheme, indicating how each instance may be mapped via predicates to a combination of one or more values. Injective (1:1 into) reference schemes mapping entities (e.g., countries) to single values (e.g., country codes) may be abbreviated as in symbol 3 by displaying the reference mode in parentheses, e.g., Country (.code). The reference mode indicates how values relate to the entities. Values are constants with a known denotation, so require no reference scheme. Relationships used for preferred reference are called existential facts (e.g., there exists a country that has the country code "IT"). The other relationships are elementary facts (e.g., The country with country code "IT" has a population of 60.000.000). The exclamation mark in symbol 4 declares that an object type is independent (instances may exist without participating in any elementary facts). Object types displayed in multiple

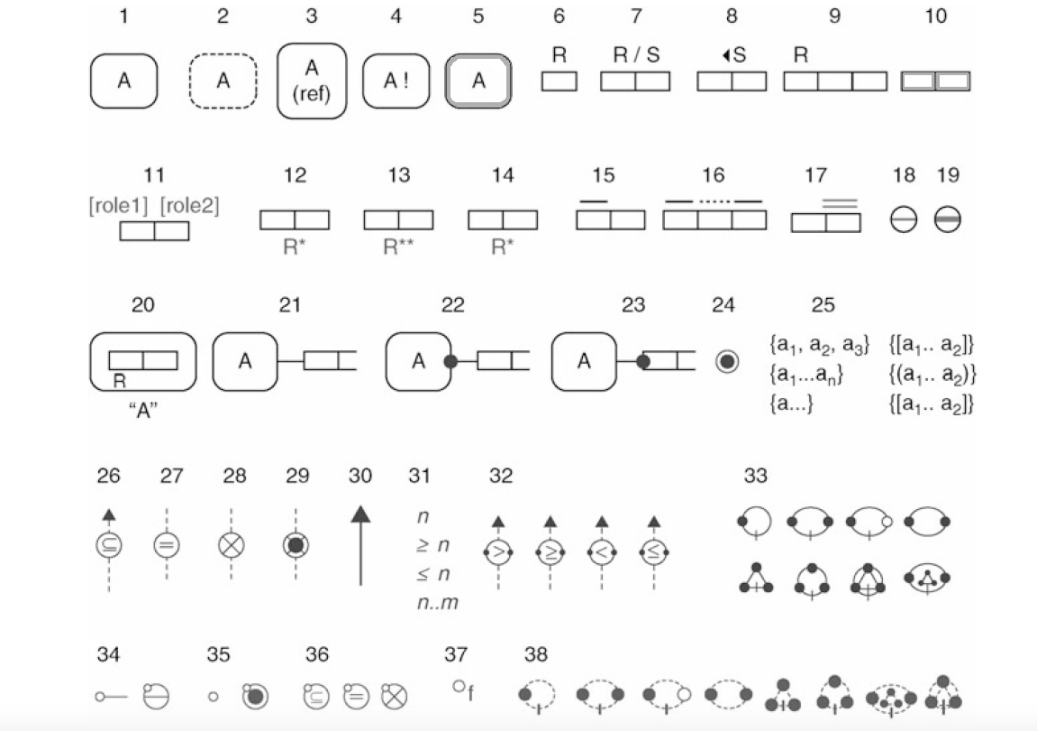


Figure 2.10: List of ORM constraints

place are shadowed (symbol 5). A fact type results from applying a logical predicate to a sequence of one or more object types. Each predicate comprises a named sequence of one or more roles (parts played in the relationship). A predicate is sentence with object holes, one for each role, with each role depicted as a box and played by exactly one object type. Symbol 6 shows a unary predicate (e.g., ... smokes), symbols 7 and 8 depict binary predicates (e.g., ... loves ...), and symbol 9 shows a ternary predicate. Predicates of higher arity (number of roles) are allowed. Each predicate has at least one predicate reading. ORM uses mixfix predicates, so objects may be placed at any position in the predicate (e.g., the fact type Person introduced Person to Person involves the predicate "... introduced ... to ... "). Mixfix predicates allow natural verbalization of nary relationships, as well as binary relationships where the verb is not in the infix position (e.g., in Japanese, verbs come at the end). By default, forward readings traverse the predicate from left to right (if displayed horizontally) or top to bottom (if displayed vertically).

Other reading directions may be indicated by an arrow-tip (symbol 8). For binary predicates, forward and inverse readings may be separated by a slash (symbol 7). Duplicate predicate shapes are shadowed (symbol 10). Roles may be given role names, displayed in square brackets (symbol 11). An asterisk indicates that the fact type is derived from one or more other fact types (symbol 12). By default, the populations of derived fact types are not stored, but are instead computed on demand. ORM also allows to declare that the population of a derived fact type is stored, so that it is always immediately available. If the fact type is derived and stored, a double asterisk is used (symbol 13). Fact types that are semi-derived are marked "C" (symbol 14). Internal uniqueness constraints, depicted as bars over one or more roles in a predicate, declare that instances for that role (combination) in the fact type population must be unique (e.g., symbols 15, 16). For example, a uniqueness constraint on the first role of Person was born in Country verbalizes as: Each person was born in at most one Country. If the constrained roles are not contiguous, a dotted line separates the constrained roles (symbol 16). A predicate may have many uniqueness constraints, at most one of which may be declared preferred by a double-bar (symbol 17). An external uniqueness constraint shown as a circled uniqueness bar (symbol 18) may be applied to two or more roles from different predicates by connecting to them with dotted lines. This indicates that instances of the role combination in the join of those predicates are unique. For example, if a state is identified by combining its state code and country, an external uniqueness constraint is added to the roles played by State code and Country in: State has State code; State is in Country. Preferred external uniqueness constraints are depicted by a circled double-bar (symbol 19). To talk about a relationship, one may objectify it (i.e., make an object out of it) so that it can play roles. Graphically, the objectified predicate (a.k.a. nested predicate) is enclosed in a soft rectangle, with its name in quotes (symbol 20). Roles are connected to their players by a line segment (symbol 21). A mandatory role constraint declares that every instance in the population of the role's object type must play that role. This is shown as a large dot placed at the object type end (symbol 22) or the role end (symbol 23). An inclusive-or (disjunctive mandatory) constraint applied to two or more roles indicates that all instances of the

object type population must play at least one of those roles. This is shown by connecting the roles by dotted lines to a circled dot (symbol 24). To restrict the population of an object type or role, the relevant values may be listed in braces (symbol 25). An ordered range may be declared separating end values by ".". For continuous ranges, a square/round bracket indicates an end value is included/excluded. For example, "(0.10)"denotes the positive real numbers up to 10. These constraints are called value constraints. Symbols 26–28 denote set comparison constraints,which apply only between compatible role sequences. A dotted arrow with a circled subset symbol depicts a subset constraint, restricting the population of the first sequence to be a subset of the second (symbol 26). A dotted line with a circled "=" symbol depicts an equality constraint, indicating the populations must be equal (symbol 27). A circled "X" (symbol 28) depicts an exclusion constraint, indicating the populations are mutually exclusive. Exclusion and equality constraints may be applied between two or more sequences. Combining an inclusive or and exclusion constraint yields an exclusive-or constraint (symbol 29). A solid arrow (symbol 30) from one object type to another indicates that the first is a (proper) subtype of the other (e.g., Woman is a subtype of Person). Mandatory (circled dot) and exclusion (circled "X") constraints may be displayed between subtypes, but are implied by other constraints if the subtypes have formal definitions. Symbol 31 shows four kinds of frequency constraint. Applied to a role sequence, these indicate that instances that play those roles must do so exactly n times, at least n times, at most n times, or at least n and at most m times. Symbol 32 shows four varieties of value-comparison constraint. The arrow shows the direction in which to apply the circled operator between two instances of the same type (e.g., For each Employee, hiredate > birthdate). Symbol 33 shows the main kinds of ring constraint that may apply to a pair of compatible roles. Read left to right and top row first, these indicate that the binary relation formed by the role population must respectively be irreflexive, asymmetric, antisymmetric, reflexive, intransitive, acyclic, intransitive and acyclic, or intransitive and asymmetric. The previous constraints are alethic (necessary, so can't be violated) and are colored violet. ORM 2 also supports deontic rules (obligatory, but can be violated). These are colored blue, and either add an "o" for obligatory, or soften lines to dashed lines. Displayed

here are the deontic symbols for uniqueness (symbol 34), mandatory (symbol 35), set-comparison (symbol 36), frequency (symbol 37) and ring (symbol 38) constraints.

2.3.3 Derivation Rules

ORM Derivation Rules are special ORM construct that are able to express knowledge that goes beyond standard ORM capabilities. In a way similar to SQL triggers or OCL constraints, ORM Derivation Rules define which instances may appear in the population of subtypes and fact types. We recall that an instance of an ORM model maps each object-type and fact-type in the model to a population, where the population of an object-type is a set of objects (values or entities) and the population of a fact-type is a set of tuples of objects. ORM Derivation Rules are used to derive new facts from other facts. A fact that is not derived is an asserted fact (also known as a primitive or base fact). The population of an asserted fact is made by asserted instances; the population of a derived fact is made by derived instances.

In ORM, subtypes may be asserted, derived, or semiderived. A subtype is asserted if and only if, for each state of the fact base, only asserted instances may appear in its population. A subtype is derived if and only if, for each state of the fact base, only derived instances may appear in its population. A subtype is semiderived if and only if some of its instances may be simply asserted and some other instances may be derived. Graphically, derived subtypes are displayed with an asterisk "*" after their name, and semiderived subtypes are displayed with a plus superscript "+" after their name.

Consider the ORM schema in Figure 2.11. Here the subtypes are simply asserted, since there is no way to derive which persons are smokers and which persons are males.

In contrast, the ORM schema in 2.12 includes smokes and gender fact types that enable us to derive which persons are members of which subtypes. In this case, we must declare the subtypes to be derived and provide appropriate derivation rules for the subtype definitions. Otherwise, it would be possible to have incorrect models. For example, we could declare an instance of Smoker

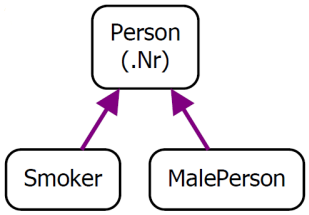


Figure 2.11: ORM schema with asserted subtypes

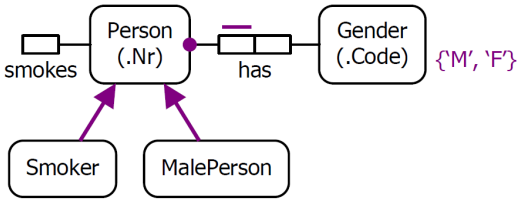


Figure 2.12: ORM schema with asserted subtypes that should be derived

without having that person play the smokes role; or we could declare an instance of MalePerson for a person with gender code "F".

The ORM schema in Figure 2.12 marks the subtypes as derived, and expresses their derivation rules in the syntax of FORML (Formal ORM Language), a formal, textual language for ORM that is currently under development. Here pseudo-reserved words are displayed in bold. Currently, NORMA does not support the option of choosing "a" or "an" as an alternative reading for the existential quantifier, which it always renders as "some".

While one can declare that a subtype is derived simply by entering some text in the DerivationNote property for the subtype in the ORM Model Browser, derivation notes are treated by NORMA simply as informal comments, so no

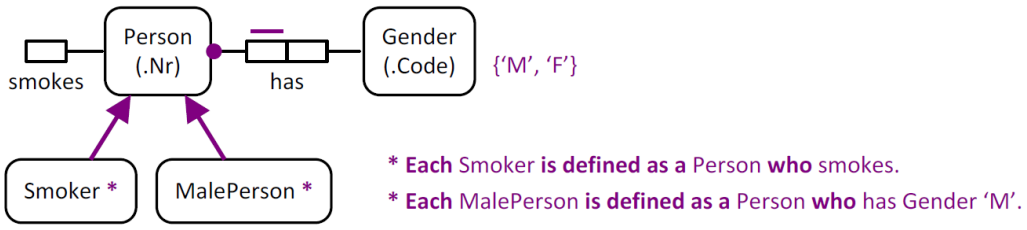


Figure 2.13: ORM schema with derived subtypes and derivation rules

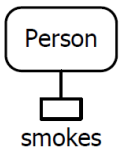


Figure 2.14: The derivation path for the Smoker subtype in Figure 2.13

code will be generated from them to enforce the derivation rules. Currently, the only way to formally declare a derivation rule in NORMA is to specify a derivation path for it in the ORM Model Browser.

A subtype's derivation path is the path through the ORM schema, including any operators and conditions that apply, that corresponds to the subtype derivation rule. The path always starts at an object type, known as the root object type for the path. For subtype derivation paths, the root object type will always be a supertype of that subtype. With complex paths, you might need to traverse through the same role more than once, in which case we need to distinguish different occurrences of the same role. When this is not the case, the term "role" is often used informally for "role occurrence".

For the Smoker subtype in Figure 2.13, the derivation path is the path starting at Person (the root object type for the path) and ending at the role in the smokes predicate. You can visualize it as shown in Figure 2.14. More complex subtype definitions may have derivation paths that look like a tree, with multiple branches stemming from the root object type or from object type occurrences later in the path.

We now extend the example in Figure. 2.8 by ORM Derivation Rule.

The conceptual diagram is not expressive enough to encode further information, e.g., that all the visitors are exactly those who are identified by a visa or to capture all the people that have no documents. Observe, that there is no constraint stating that all Visa holders are visitors; for this reason we named *Some*Visitor the entity which is a subtype of Person with ID entity. How can we capture in the schema exactly all the visitors? We need to use a derivation rule stating the needed exact definition. We now add a new entity called VisitorWithVisa as a subtype of PersonWithID with an attached derivation rule as shown in Figure 2.15.

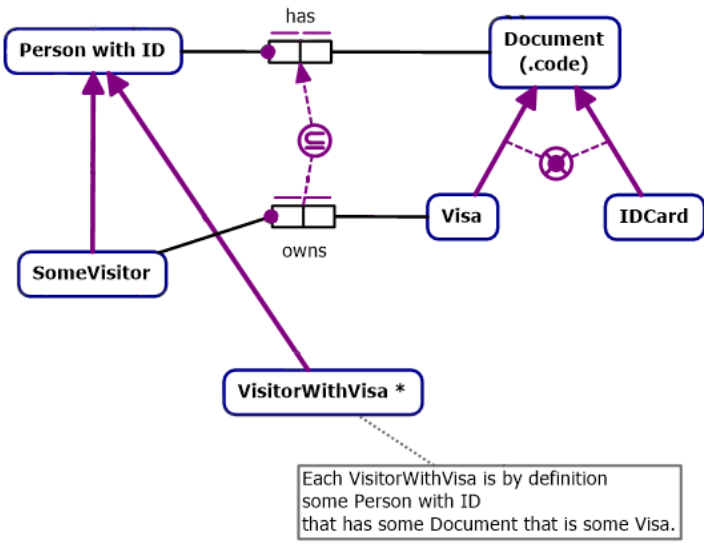


Figure 2.15: ORM example with ORM Derivation Rule

As we have seen before for other ORM constraints, ORM Derivation Rules can also be expressed in FORML:

```
*Each VisitorWithVisa is by definition some Person with ID
that has some Document that is some Visa.
```

In FOL:

$$\forall x.\, VisitorWithVisa(x) \leftrightarrow PersonWithID(x) \wedge \exists y.has(x,y) \wedge Visa(y)$$

In this way all the instances inside VisitorWithVisa are constrained to posses a Visa document.

2.4 NORMA

NORMA is implemented as a plug-in to Microsoft Visual Studio. Most of NORMA is open-source, and a public domain version is freely downloadable [110]. Fig. 2.16 summarizes the main components of the tool. Users may declare ORM object types and fact types textually using the Fact Editor, or

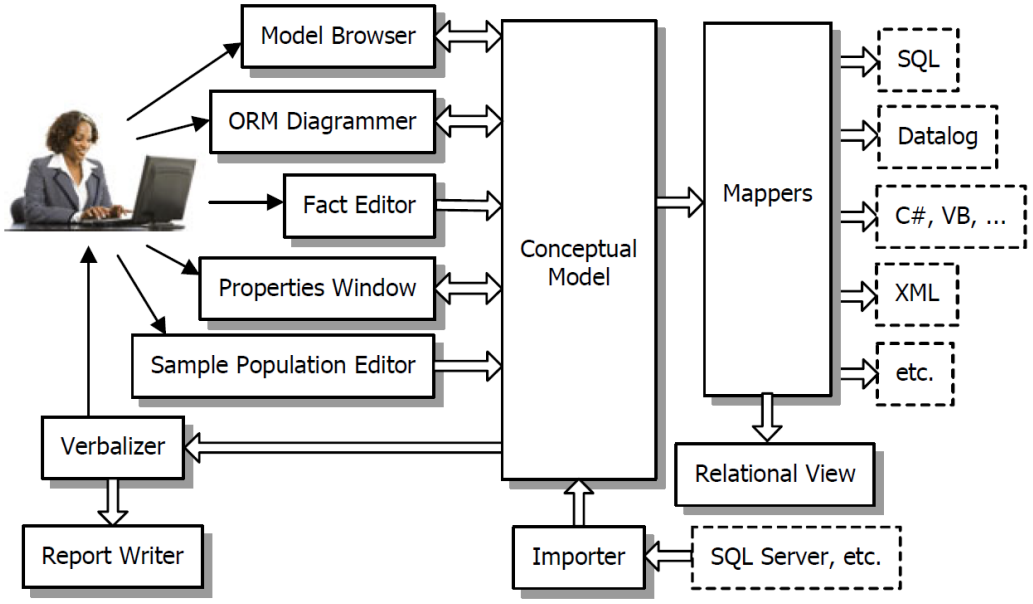


Figure 2.16: NORMA overview

drag new elements off the toolbox. New model components are added to the conceptual model and displayed with graphical shapes on one or more ORM diagrams. The Model Browser tool window also provides a hierarchical view of all model components. Sample object and fact instances may be entered in tabular format in the Sample Population Editor.

Currently, ORM constraints and ORM Derivation Rules must be entered in the ORM diagrammer or the Properties Window. These constraints are automatically verbalized in FORML (Formal ORM Language), a controlled natural language that is understandable even by non-technical people. The Model Browser is also able to handle derivation rules for both fact types and subtypes with verbalisation. Using mappers, ORM schemas may be automatically transformed into various implementation targets, including relational database schemas for popular database management systems (SQL Server, Oracle, DB2, MySQL, PostgreSQL), datalog, .NET languages (C#, VB, etc.), and XML schemas. A Relational View extension displays the relational schemas in diagram form. The semantics underlying relational columns can be exposed by selecting them and automatically verbalizing the ORM fact types

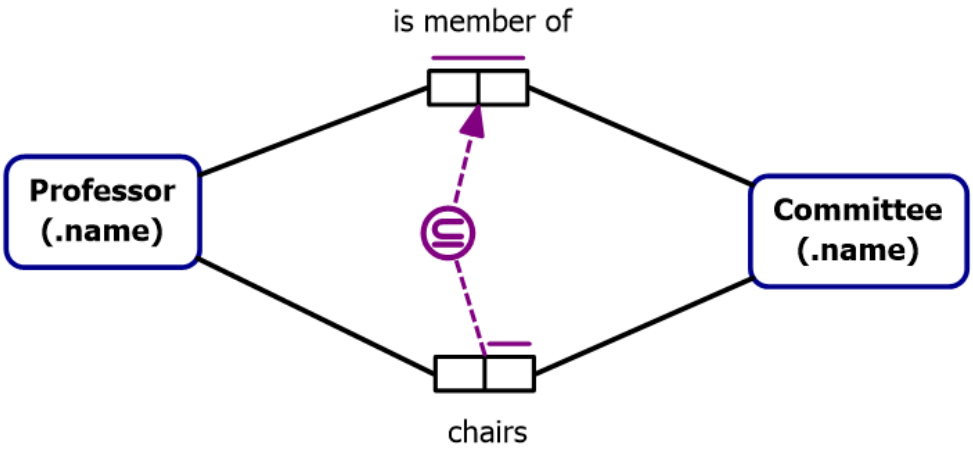


Figure 2.17: ORM diagram done with NORMA

from which they were generated. An import facility can import ORM models created in some other ORM tools, and can reverse engineer relational schemas in SQL Server into ORM schemas. Other components facilitate navigation and abstraction. For example, multiple concurrent windows viewing the same model allow shapes to be copied between diagrams, the ORM Diagram Spy and hyperlinks in the Verbalization Browser allow rapid navigation through a model, and the ORM Context Window

One of the most useful features in NORMA is the automated verbalization [90], [108], [45]. Considering the ORM diagram in Figure 2.17, we have the verbalised counterpart in Figure 2.18.

In the ORM diagram we have two binary fact types: Professor is member of Committee and Professor chairs Committee. Entity types are shown as named, soft rectangles with their reference mode in parenthesis. Logical predicates are depicted as a named sequence of role boxes connected to the object types whose instances play those roles. The bar over each predicate depicts a spanning uniqueness constraint, indicating that the fact types are m:n, and can be populated with sets of fact instances, but not bags. The circled subsetsymbol connected by dashed lines to role pairs depicts a subset constraint. When the constraint shape is selected, NORMA displays role numbers to highlight the role sequence arguments to the constraint.

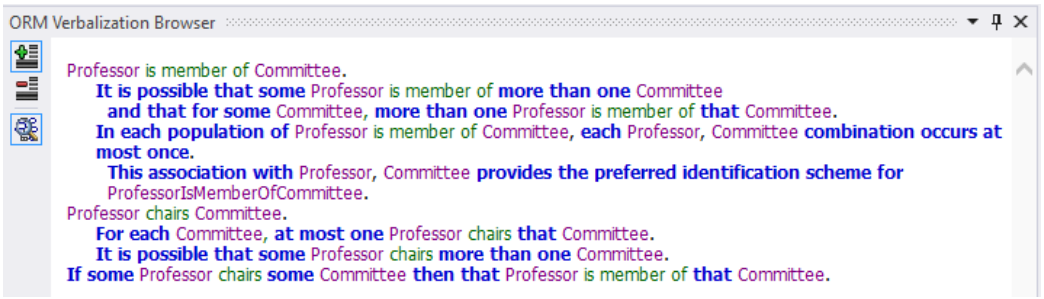


Figure 2.18: ORM verbalisation

The verbalisation is displayed in the ORM Verbalization Browser, one of the main NORMA components. Here each part of the ORM diagram is displayed in a controlled natural language. This feature is useful to bridge the gap between stakeholders working on the same project, especially those who are not IT experts.

A feature of NORMA that is especially useful to modelers is its live error checking capability. Modelers are notified immediately of errors that violate a metarule that has been implemented in the underlying ORM metamodel. Fig. 2.19 shows an example where the subset constraint is marked with red fill because it is inconsistent with other constraints present. In this case, the committee role of being chaired is declared to be mandatory (as shown by the solid dot on the role connection), while the committee role of including a member is declared to be optional. But the subset constraint implies that if a committee has a chair then it must have that person as a member. So it is impossible for the two fact types to be populated in this situation. NORMA not only detects the error but suggests three possible ways to fix the problem.

As we can see in Figure 2.20, complex diagrams are usually managed in NORMA by multiple pages. The user can browse and organize the diagram by means of pages, that generally represent a precise sub-domain of the whole universe of discourse.

The mapper component allows to represent the ORM diagram in different ways, for example one useful function is to visualize and export the ORM conceptual diagram into a database relational view as shown in Figure 2.21.

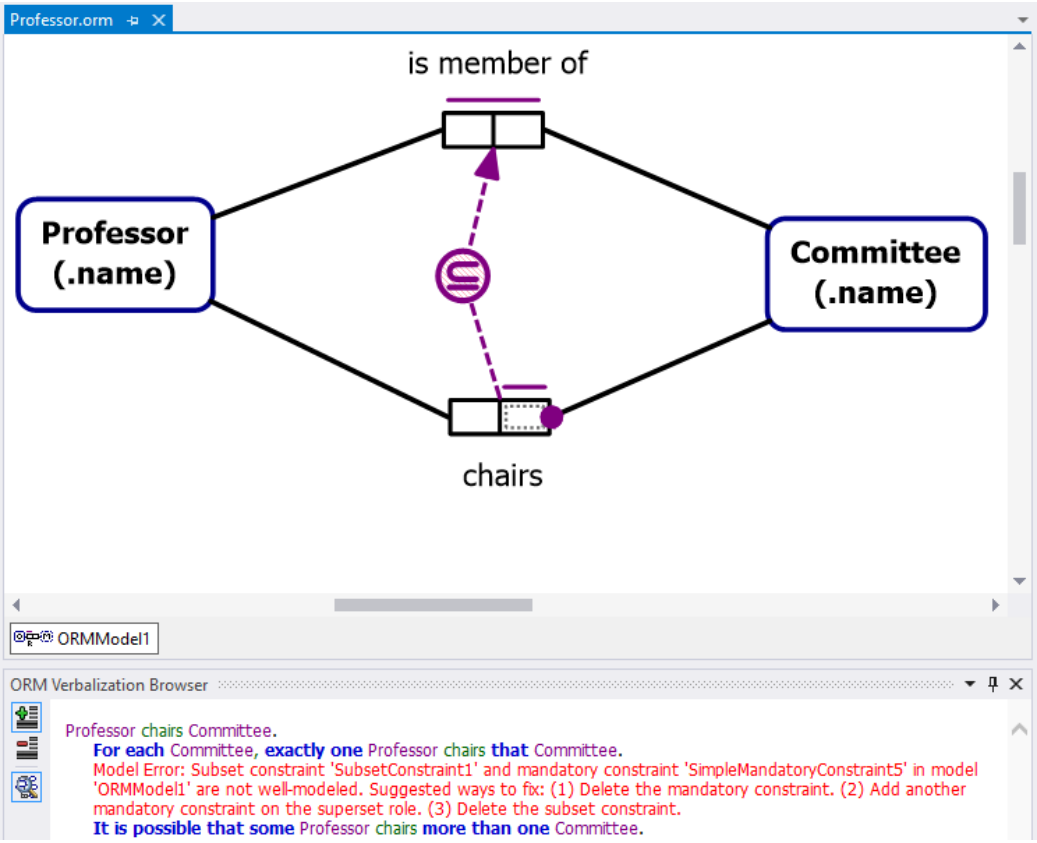


Figure 2.19: ORM live error check

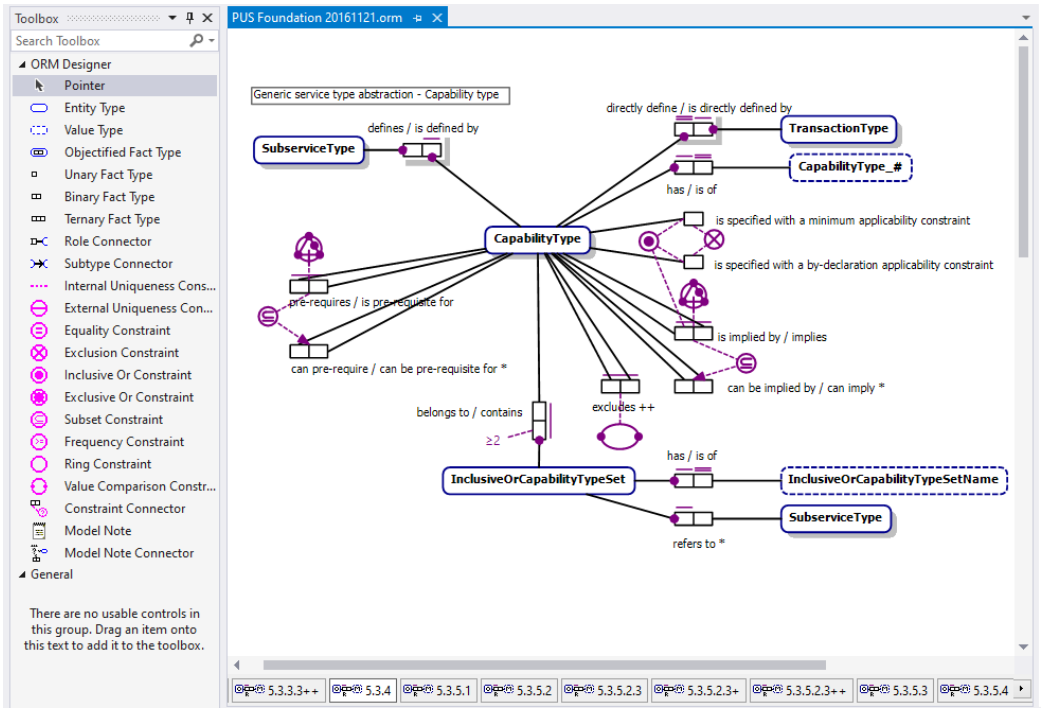


Figure 2.20: NORMA pages

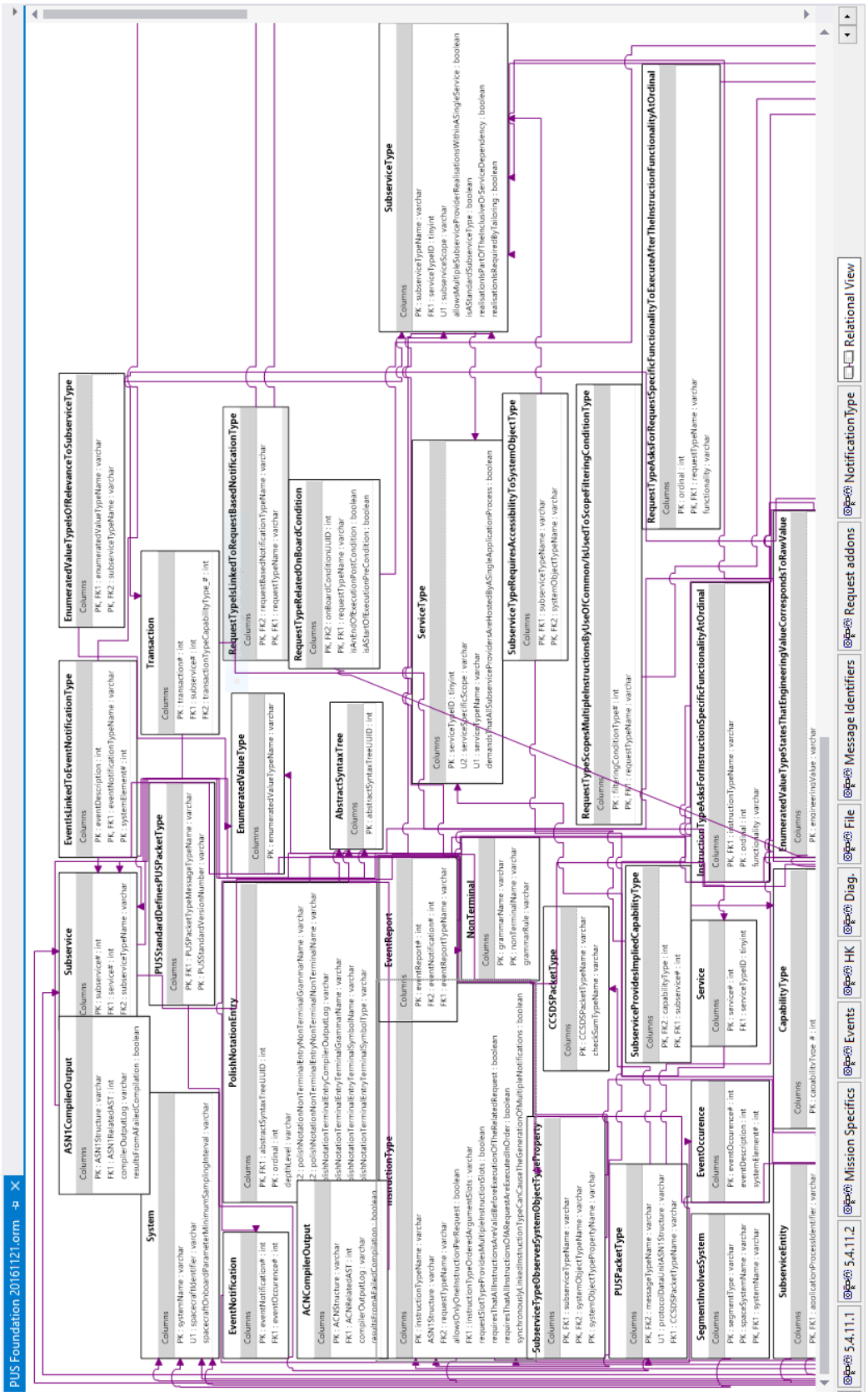


Figure 2.21: NORMA relational view

3

ORM Foundations

Object-Role modelling (ORM) is a rigorous approach to modelling and querying at the conceptual level the information semantics of arbitrary domains. In this work is defined an abstract syntax for ORM conceptual models together with its formal semantics. An ORM conceptual model comprises an ORM conceptual schema plus a population of (object and fact) instances. In addition to type and constraint declarations, an ORM schema may include derivation rules. The semantics of an ORM conceptual model is defined by transforming the model to first-order logic axioms, whose models denote the legal abstract information structures of the conceptual specification.

3.1 Formal syntax and semantics

The latest version of ORM (ORM 2) is thoroughly described in [79], and in [77] and its companion [78] providing an up-to-date coverage of the latest enhancements to ORM and its conceptual schema design procedure. This document refers to the ORM 2 version of ORM.

In this section we define the normative syntax and the semantics of ORM Conceptual Models. The work in this section does not define an interchange format for ORM, but its ultimate goal is to provide a self-contained document defining the non-ambiguous semantics of all the ORM basic constructs. The syntax defines the ORM language, and it is given by specifying a signature (the ORM Conceptual Model Signature), and then the set of well formed formulae which can be built using the signature. An ORM Conceptual Model

is any set of well-formed formulae from a given ORM signature, satisfying all the constraints. An ORM Conceptual Model has an "abstract" syntax since it is in one-to-one correspondence with the concrete graphical syntax; this is shown in the section 3.2.1 by a set of examples. The semantics is a standard denotational semantics. The semantics of an ORM Conceptual Model is given with a transformation of the ORM conceptual model to first-order logic formulas: the finite first-order structures satisfying the obtained first-order logical theory are in one-to-one correspondence with the legal populations of the ORM conceptual model. After having introduced ORM Conceptual Model Signatures and the corresponding First-Order Logic Signatures, we describe the abstract syntax and semantics of ORM's main conceptual model constructs, namely declarations, constraints, and derivation rules.

3.1.1 Naming Conventions

An ORM conceptual model is formally composed, following precise syntactic rules, by declarations and constraints, built from terms of different syntactic categories (object type names, predicate names, role names, predicate role names, data elements) taken from a signature. The ORM graphical notation depicts a fact type as a left-to-right top-to-bottom ordered sequence of role boxes, each of which is attached to exactly one object type shape. A fact type is bijectively associated to a canonical predicate. Fact types (resp. object types) appearing in the ORM graphical notation as distinct are associated in the signature to distinct predicate names (resp. object type names). A role is uniquely identified in ORM graphical notation by the fact type in which it appears together with its relative position (left-to-right top-to-bottom) within the fact type; each role is given in the signature the role name obtained by concatenating the canonical predicate name it belongs to and the relative position within it. In this document we assume that the signature of an ORM conceptual model (e.g., the choice of the canonical predicate associated to a fact type) has been specified without ambiguity, following maybe linguistic conventions or other design methodologies. This document does not focus on the pre-logical or linguistic means necessary in order to get the formal signature. Predicate readings, as normally introduced in the ORM

methodology, are used simply for readability and compactness of display on the graphical notation of ORM schema diagrams, and are not necessarily identifying, since distinct fact types may have the same predicate reading. On the other hand, distinct predicate names in the signature identify necessarily distinct fact types. Alternate predicate readings for the same fact type (e.g., useful to verbalise differently a predicate with different role orderings) are all obviously denoting the same fact type, and therefore will be given the same predicate name in the signature.

3.1.2 ORM Conceptual Model Signature

An ORM conceptual model signature is composed by the elements $\langle \mathcal{T}, \mathcal{V}, \mathcal{P}, \mathcal{R}, \mathcal{D}, \beta, \mathcal{F}, \alpha \rangle$ denoting the following:

$\mathcal{T}$	a finite set of domain object type names
$\mathcal{V}$	a set $\mathcal{V} \subseteq \mathcal{T}$ of domain value type names
$\mathcal{P}$	a finite set of predicate names
$\mathcal{R}$	a finite set of role names
$\mathcal{D}$	a finite set of domain values
β	a function $\beta : \mathcal{V} \rightarrow 2^D$, the domain value type extension
$\mathcal{F}$	a finite set of value function names
α	a total function $\alpha : \mathcal{P} \cup \mathcal{F} \rightarrow \mathbb{N}^+$ specifying the predicate or the function arity

Table 3.1: ORM Conceptual Model Signature

We adopt the following syntactic conventions in the ORM conceptual model:

- the letters *h,i, j, k,l,m,n* denote positive integer numbers;
- the letters *p,q* denote integer numbers;
- T denotes a domain object type name $\in \mathcal{T}$;
- V denotes a domain value type name $\in \mathcal{V}$;
- P denotes a predicate name $\in \mathcal{P}$;
- r denotes a role name $\in \mathcal{R}$;

- d denotes a domain value $\in \mathcal{D}$;
- f denotes a value function name $\in \mathcal{F}$;
- P.i denotes the i-th role identifier of P, with $1 \leqslant i \leqslant \alpha(P)$
- ?v denotes a variable symbol within a derivation rule, with v $\in$ STRINGS;

3.1.3 First-order logic signature

The First-Order Logic (FOL) signature of an ORM conceptual model reuses the same symbols from $\mathcal{T}$, $\mathcal{V}$, $\mathcal{P}$, $\mathcal{D}$ $\mathcal{F}$ of the ORM conceptual model signature, and it is composed by the elements $\langle \mathcal{T}, \mathcal{V}, \mathcal{P}, \mathcal{L}, \mathcal{D}, \mathcal{G}, \mathcal{F} \rangle$ denoting the following:

$\mathcal{T}$	a finite set of unary predicate symbols
$\mathcal{V}$	a set $\mathcal{V} \subseteq \mathcal{T}$ of data values
$\mathcal{P}$	a finite set of predicate symbols, each P$\in$ $\mathcal{P}$ with arity $\alpha(P)$
$\mathcal{L}$	a family of injective and well-founded objectification functions, one for each P $\in \mathcal{P}$ and with arity $\alpha(P)$
$\mathcal{D}$	a finite set of constant symbols
$\mathcal{G}$	A family of domain value to data value injective functions γ_v for each v $\in$ $\mathcal{V}$ such that for each d $\in \beta(\mathcal{V})$ and T $\in$ $\mathcal{T}$, it holds that V(d) and $\forall x.\mathrm{T}(x) \rightarrow x \neq \gamma_v(d)$
$\mathcal{F}$	A family of functions over data values, namely with domain and range over the range of the function γ

Table 3.2: First-Order Conceptual Model Signature

We adopt the following syntactic conventions in the FOL formulas:

- the precedence of Boolean operators is:$\neg$ $\wedge$ $\vee$ $\rightarrow$;
- l_p denotes a function $\in \mathcal{L}$ associated to the predicate P$\in \mathcal{P}$ and with arity $\alpha(P)$;
- γ_v denotes a function $\in \mathcal{G}$ associated to the domain value type V.

3.1.4 First-order logic ORM Conceptual Model

The following extensions to FOL are used the specify the semantics of ORM Conceptual Models.

- The translation of derivation rules is given in first-order logic extended with lambda expressions. In the specification as a first order logic formula of the semantics of a derivation rule containing a PATH expression, a PATH expression corresponds to an open first order formula with one free variable. Such an open formula is built inductively from the parse tree of the PATH expression using its grammar specification in a way similar to Montague grammars. The composition among steps in the induction makes use of lambda expressions and their application using variable bindings and substitutions: if ϕ is a formula with a free variable x, and t is a term, an application of the lambda calculus β-reduction rule $((\lambda x.\phi)(t)) \rightarrow (\phi_{[x/t]})$ replaces the occurrences of the bound variable x within the body ϕ of the lambda expression with the term t.

- The translation of ring constraints is given in first-order logic extended with the transitive closure operator * over binary predicates. First-order logic extended with the transitive closure operator is strictly more expressive than first-order logic. The transitive closure of a binary predicate P can be expressed in first-order logic enriched with least fixpoints as follows: $\forall x^{'}y^{'}.P^{*}x^{'}y^{'} \leftrightarrow \mathbf{lfp}_{Q,xy}(Pxy \vee (\exists z.Qxz \wedge Pzy))x^{'}y^{'}$.

- The translation of identification constraints is given in first-order logic extended with well-founded binary relations. A binary relation R is well-founded, well-founded(R), if its interpretation contains no countable infinite descending chains: that is, in the interpretation of R there is no infinite sequence $a_0, a_1, a_2, \ldots$ of non necessarily distinct elements such that $R\ a_n a_{n+1}$ for every natural number n, i.e, there is no infinite ascending chain.

- According to the definition above, objectification functions should be well-founded. A function is well-founded if the binary relation F = $\{\langle x, y\rangle \mid f(x) = y\}$ is well founded.

The First-Order Logic (FOL) Conceptual Model of an ORM conceptual model is a FOL theory composed by the theory Φ obtained by applying the transformations specified in the table below, with an additional closure theory Θ. The closure theory Θ is needed in order to give the right semantics to the identification and objectification constraints, and it includes:

- top-level disjointness axioms of the form $\forall x.\mathrm{T}_1(x) \rightarrow \neg \mathrm{T}_2(x)$ for any pair of object types such there is no object type T such that:
  $\Phi \models (\forall x.\mathrm{T}_1(x) \rightarrow \mathrm{T}(x)) \wedge (\forall x.\mathrm{T}_2(x) \rightarrow \mathrm{T}(x))$;
- a well-foundedness axiom involving all the special binary predicate symbols $\mathrm{P}_j^{ID\text{-}s_i}$ introduced by identification and objectification constraints in the conceptual model:
  well-founded$(\mathrm{P}_1^{ID\text{-}s_1} \cup \cdots \cup \mathrm{P}_{m_1}^{ID\text{-}s_i} \cup \cdots \cup \mathrm{P}_1^{ID\text{-}s_k} \cup \cdots \cup \mathrm{P}_{m_k}^{ID\text{-}s_k})$.

Table 3.3: ORM syntax and semantics

FactType(P ($\mathrm{T}_1 \ldots \mathrm{T}_{\alpha(\mathrm{P})}$)) $\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow T_1(x_1) \wedge \cdots \wedge T_{\alpha(P)}(x_{\alpha(P)})$	P does not appear as an AlternatePredicate
AlternatePredicate(P, $\mathrm{P_a}$ ($\mathrm{P}_{\cdot \mathrm{i}_1}$ … $\mathrm{P}_{\cdot \mathrm{i}_{\alpha(\mathrm{P})}}$)) (MACRO)Replace all occurrences of $\mathrm{P_a}$.j in the ORM conceptual model with $\mathrm{P}_{\cdot \mathrm{i_j}}$, and then all occurrences of $\mathrm{P_a}$ with P	$\mathrm{P} \neq \mathrm{P_a}$ $\alpha(\mathrm{P}) = \alpha(\mathrm{P_a})$ $\{\mathrm{i}_1 \ldots \mathrm{i}_{\alpha(\mathrm{P})}\} = \{1 \ldots \alpha(\mathrm{P})\}$
RoleNaming(P.i r) (MACRO)Replace all occurrences of the role name r in the ORM conceptual model with the role identifier P.i	
Mandatory(T $\mathrm{P}_1.\mathrm{i}_1$ … $\mathrm{P_m}.\mathrm{i_m}$) $\forall x.T(x) \rightarrow \exists y_1 \ldots y_{\alpha(P_1)}.\left(P_1(y_1 \ldots y_{\alpha(P_1)}) \wedge x = y_{i_1}\right)$ $\vee \cdots \vee \exists y_1 \ldots y_{\alpha(P_m)}.\left(P_m(y_1 \ldots y_{\alpha(P_m)}) \wedge x = y_{i_m}\right)$	for $\mathrm{j} \neq \mathrm{k}$ and $\mathrm{j,k} \leqslant \mathrm{m}$: $\mathrm{P_j} \neq \mathrm{P_k}$
Unique($\mathrm{P.i}_1$ … $\mathrm{P.i_m}$) (MACRO) Frequency($\mathrm{P.i}_1$ … $\mathrm{P.i_m}$ (1, 1))	for $\mathrm{j} \neq \mathrm{k}$ and $\mathrm{j,k} \leqslant \mathrm{m}$: $\mathrm{i_j} \neq \mathrm{i_k}$
Identification(T $\mathrm{P.i_{m+1}}$ ($\mathrm{P.i}_1$ … $\mathrm{P.i_m}$))	for $\mathrm{j} \neq \mathrm{k}$ and $\mathrm{j,k} \leqslant \mathrm{m+1}$: $\mathrm{i_j} \neq \mathrm{i_k}$

(MACRO)
Unique ($P.i_{m+1}$)
Mandatory ($T(P.i_{m+1})$)
Unique ($P.i_1 \ldots P.i_m$)
and each of the binary predicates defined below is well-founded:
FactTypeRule($B_1(P.i_{m+1} \hookrightarrow (P.i_1 \times ?x))(?x)$)
...
FactTypeRule($B_m(P.i_{m+1} \hookrightarrow (P.i_m \times ?x))(?x)$)

ExternalUnique($P.i_1 \ldots P.i_m$) for $j \neq k$ and $j,k \leqslant m+1$: $\alpha(P)=2$, $P_j \neq P_k$, if $i_j=1$ then $l_j=2$ else $l_j=1$; and P fresh predicate name of arity m+1

(MACRO)
FactTypeRule($P(P_1.i_1 \hookrightarrow (P_1.l_1 \times ?x)) \ldots (P_m.i_m \hookrightarrow (P_{m.lm} \times ?x))(?x)$)
Unique($P.1 \ldots P.m$)

ExternalIdentification($T\ (P_1.i_1 \ldots P_m.i_m)$) for $j \neq k$ and $j,k \leqslant m+1$: $\alpha(P_j)=2$, $P_j \neq P_k$, if $i_j=1$ then $l_j=2$ else $l_j=1$; and P fresh predicate name of arity m+1

(MACRO)
FactTypeRule($P(P_1.i_1 \hookrightarrow (P_1.l_1 \times ?x)) \ldots (P_m.i_m \hookrightarrow (P_m.l_m \times ?x))(?x)$)
Unique(P.m+1)
Mandatory($T\ (P.m+1)$)
Unique($P.1 \ldots P.m$)
and each of the binary predicates defined below is well-founded:
FactTypeRule($B_1(P.m+1 \hookrightarrow (P.1 \times ?x))(?x)$)
...
FactTypeRule($B_m(P.m+1 \hookrightarrow (P.m \times ?x))(?x)$)

Frequency($P.i_1 \ \ldots \ P.i_m\ \underline{F}$) for $j \neq k$ and $j,k \leqslant m$: $i_j \neq i_k$. $p,q \geqslant 1$ (1) $\underline{F} \equiv (p..)$ (2) $\underline{F} \equiv (..q)$ (3) $\underline{F} \equiv (p..q)$ (4) $\underline{F} \equiv (p)$

(1) $\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow \exists^{\geqslant p} y_1 \ldots y_{\alpha(P)}.P(y_1 \ldots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \cdots \wedge x_{i_m} = y_{i_m}$

$(2)\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow \exists^{\leqslant q} y_1 \ldots y_{\alpha(P)}.P(y_1 \ldots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \cdots \wedge x_{i_m} = y_{i_m}$
$(3)\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow \exists^{\geqslant p} y_1 \ldots y_{\alpha(P)}.P(y_1 \ldots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \cdots \wedge x_{i_m} = y_{i_m}$
$\wedge \forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow \exists^{\leqslant q} y_1 \ldots y_{\alpha(P)}.P(y_1 \ldots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \cdots \wedge x_{i_m} = y_{i_m}$
$(4)\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow \exists^{=p} y_1 \ldots y_{\alpha(P)}.P(y_1 \ldots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \cdots \wedge x_{i_m} = y_{i_m}$

ExternalFrequency($P_1.i_1 \ldots P_m.i_m$ $\underline{F}$) — for $j \neq k$ and $j,k \leqslant m$: $\alpha(P_j)=2$, $P_j \neq P_k$, if $i_j=1$ then $l_j=2$ else $l_j=1$; and P fresh predicate name $p,q \geqslant 1$
(1) $\underline{F} \equiv$ (p..)
(2) $\underline{F} \equiv$ (..q)
(3) $\underline{F} \equiv$ (p..q)
(4) $\underline{F} \equiv$ (p)

(MACRO)
FactTypeRule($P(P_1.i_1 \hookrightarrow (P_1.l_1 \times ?x)) \ldots (P_m.i_m \hookrightarrow (P_m.l_m \times ?x))(?x)$)
Frequency(P.1…P.m $\underline{F}$)

Subtype((T_1 … T_m) T)
$(\forall x.T_1(x) \rightarrow T(x)) \wedge \cdots \wedge (\forall x.T_m(x) \rightarrow T(x))$

ExclusiveSubtypes((T_1 … T_m) T)
$(\forall x.T_1(x) \rightarrow T(x) \wedge \neg T_2(x) \wedge \cdots \wedge \neg T_m(x)) \wedge \cdots \wedge (\forall x.T_{m-1}(x) \rightarrow T(x) \wedge \neg T_m(x)) \wedge (\forall x.T_m(x) \rightarrow T(x))$

ExhaustiveSubtypes((T_1 … T_m) T)

$\forall x.T(x) \rightarrow T_1(x) \vee \cdots \vee T_m(x)$

Subset(($P_1.i_1$ $P_2.h_1$) … ($P_1.i_m$ $P_2.h_m$)) — $P_1 \neq P_2$ and for $j \neq k$ and $j,k \leqslant m$: $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$

$\forall x_1 \ldots x_{\alpha(P_1)}.P_1(x_1 \ldots x_{\alpha(P_1)}) \rightarrow \exists y_1 \ldots y_{\alpha(P_2)}.P_2(y_1 \ldots y_{\alpha(P_2)}) \wedge x_{i_1} = y_{h_1} \wedge \cdots \wedge x_{i_m} = y_{h_m}$

Exclusive(($P_1.i_1$ $P_2.h_1$) … ($P_1.i_m$ $P_2.h_m$)) — $P_1 \neq P_2$ and for $j \neq k$ and $j,k \leqslant m$: $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$

$\forall x_1 \ldots x_{\alpha(P_1)}.P_1(x_1 \ldots x_{\alpha(P_1)}) \rightarrow \exists y_1 \ldots y_{\alpha(P_2)}.\neg P_2(y_1 \ldots y_{\alpha(P_2)}) \wedge x_{i_1} = y_{h_1} \wedge \cdots \wedge x_{i_m} = y_{h_m}$

Equal(($P_1.i_1$ $P_2.h_1$) … ($P_1.i_m$ $P_2.h_m$)) — $P_1 \neq P_2$ and for $j \neq k$ and $j,k \leqslant m$: $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$

(MACRO) Subset((P$_1$.i$_1$ P$_2$.h$_1$) ... (P$_1$.i$_m$ P$_2$.h$_m$)) Subset((P$_2$.h$_1$ P$_1$.i$_1$) ... (P$_2$.h$_m$ P$_1$.i$_m$))	
Objectifies(T P) $\forall x_1 \ldots x_n . P(x_1 \ldots x_n) \leftrightarrow T\,(\ell_P\,(x_1 \ldots x_n))$	
TypeCardinality(T (p,q)) $\exists^{p \ldots q} x.\, T(x)$	p,q $\geqslant 0$ and q possibly ∞
RoleCardinality(P.i (p,q)) $\forall x_1 \ldots x_n .\ \exists^{p \ldots q} y.\, P(x_1 \ldots x_n) \wedge x_i = y$	p,q $\geqslant 0$ and q possibly ∞
ValuesOf(V (d$_1$... d$_m$)) $\forall x . V(x) \rightarrow (x = d_1) \vee \cdots \vee (x = d_m)$	
ValuesOf(P.i (d$_1$ ... d$_m$)) $\forall x_1 \ldots x_n . P(x_1 \ldots x_{\alpha(P)}) \rightarrow (x_i = d_i) \vee \cdots \vee (x_i = \mathrm{d}_m)$	
$\lesseqgtr(P.i_1 P.i_2)$ $\forall x_1 \ldots x_{\alpha(P)} y_1 \ldots y_{\alpha(P)} . P(x_1 \ldots x_{\alpha(P)}) \wedge P(_{y_1} \ldots y_{\alpha(P)}) \rightarrow \gamma_{V_1}(x_{i_1}) \lesseqgtr \gamma_{V_2}(x_{i_2})$	$\lesseqgtr \in \{<, \leqslant, >, \geqslant, \neq, =\}$
RingConstraint(P.i P.j) MACRO JointPath(P$_b$ (P.i P.j))	i $\neq$ j and P$_b$ fresh predicate name with α(P$_b$)=2
Locally Reflexive $\forall x_1 x_2 .\ P_b(x_1, x_2) \rightarrow P_b(x_1, x_1)$	
Purely Reflexive $\forall x_1 x_2 .\ P_b(x_1, x_2) \rightarrow x_1 = x_2$	
Irreflexive $\forall x_1 x_2 .\ \neg P_b(x, x)$	
Symmetric $\forall x_1 x_2 .\ P_b(x_1, x_2) \rightarrow P_b(x_2, x_1)$	
Asymmetric $\forall x_1 x_2 .\ P_b(x_1, x_2) \rightarrow \neg P_b(x_2, x_1)$	
Antisymmetric $\forall x_1 x_2 .\ P_b(x_1, x_2) \wedge x_1 \neq x_2 \rightarrow \neg P_b(x_2, x_1)$	
Transitive $\forall x_1 x_2 y_1 y_2 .\ P_b(x_1, x_2) \wedge P_b(y_1, y_2) \wedge x_2 = y_1 \rightarrow P_b(x_1, x_2)$	
Intransitive $\forall x_1 x_2 y_1 y_2 .\ P_b(x_1, x_2) \wedge P_b(y_1, y_2) \wedge x_2 = y_1 \rightarrow \neg P_b(x_1, x_2)$	
Strongly Intransitive $\forall x_1 x_2 y_1 y_2 .\ P_b(x_1, x_2) \wedge P_b^*(y_1, y_2) \wedge x_2 = y_1 \rightarrow \neg P_b(x_1, x_2)$	
Acyclic	

$\forall x_1 x_2.\, \neg P_b^*(x,x)$

DERIVATION RULES

SubTypeRule(T PATH)
$\forall x.\, T(x) \leftrightarrow \exists var_1 \ldots var_m.\mathrm{PATH}^{\rightarrowtail} x$
$var_1 \ldots \mathrm{var_m}$ includes all the ?VAR variable symbols in $\mathrm{PATH}^{\rightarrowtail}$

FactTypeRule(P $\mathrm{PATH_1} \ldots \mathrm{PATH}_{\alpha_{(P)}}$)
$\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \leftrightarrow \exists var_1 \ldots var_m.\mathrm{PATH}_1^{\rightarrowtail} x_1 \wedge \cdots \wedge \mathrm{PATH}_{\alpha(P)}^{\rightarrowtail} x_{\alpha(P)}$
$\mathrm{var_1} \ldots \mathrm{var_m}$ includes all the ?VAR variable symbols in $\mathrm{PATH}^{\rightarrowtail}$

JoinPath(P ($\mathrm{P_1.i_1}$ $\mathrm{P_1.j_1}$) … ($\mathrm{P_m.i_m}$ $\mathrm{P_m.j_m}$)) — α(P)=2 and for k⩽m: $\mathrm{i_k} \neq \mathrm{j_k}$

(MACRO)
FactTypeRule($P(P_1.i_1 \hookrightarrow (P_1.\mathrm{j_1} \times (P_2.i_2 \hookrightarrow (P_2.\mathrm{j_2} \times (\ldots (P_m.i_m \hookrightarrow (P_m.\mathrm{j_m} \times ?x)))))))$)

PATH: T | — $\lesseqgtr \doteq < \leqslant > \geqslant \neq =$
$\mathrm{P_u}$ | — $\alpha(\mathrm{P_u})$=1
$\mathrm{P.i} \hookrightarrow [\mathrm{P.i_1} \times \mathrm{PATH_1}] \ldots [\mathrm{P.i_m} \times \mathrm{PATH_m}]$| — for j≠k and j,k⩽m: $\mathrm{i_j} \neq \mathrm{i_k}$
$\mathrm{PATH_1} \wedge \mathrm{PATH_2}$ |
$\mathrm{PATH_1} \vee \mathrm{PATH_2}$ |
$\mathrm{PATH_1} \setminus \mathrm{PATH_2}$ |
$\{\mathrm{d_1} \ldots \mathrm{d_n}\}$ |
?VAR |
V$\lesseqgtr$TERM |
TERM: d |
?VAR |
$\mathrm{f(TERM_1} \ldots \mathrm{TERM}_{\alpha \mathrm{F(f)}})$

$$
\begin{array}{rcl}
\mathrm{T} & \mapsto & \lambda x.\mathrm{T(x)} \\
\mathrm{P_u} & \mapsto & \lambda x.\mathrm{P_u(x)} \\
\mathrm{P.i} \hookrightarrow [\mathrm{P.i_1} \times \mathrm{PATH_1}] \ldots [\mathrm{P.i_m} \times \mathrm{PATH_m}] & \mapsto & \lambda x.\exists x_1 \ldots x_{\alpha(P).} \mathrm{P(x_1} \ldots \mathrm{x}_{\alpha \mathrm{P}}) \wedge \\
 & & \mathrm{PATH}_1^{\rightarrowtail} x_{i_1} \wedge \cdots \wedge \mathrm{PATH}_m^{\rightarrowtail} x_{i_m} \wedge x = x_i \\
\mathrm{PATH_1} \wedge \mathrm{PATH_2} & \mapsto & \lambda x.\mathrm{PATH}_1^{\rightarrowtail} \wedge \mathrm{PATH}_2^{\rightarrowtail} x \\
\mathrm{PATH_1} \vee \mathrm{PATH_2} & \mapsto & \lambda x.\mathrm{PATH}_1^{\rightarrowtail} \vee \mathrm{PATH}_2^{\rightarrowtail} x \\
\mathrm{PATH_1} \setminus \mathrm{PATH_2} & \mapsto & \lambda x.\mathrm{PATH}_1^{\rightarrowtail} \wedge \neg \mathrm{PATH}_2^{\rightarrowtail} x \\
\{\mathrm{d_1} \ldots \mathrm{d_n}\} & \mapsto & \lambda x.x = \mathrm{d_1} \vee \cdots \vee \mathrm{x} = \mathrm{d_n} \\
?\mathrm{VAR} & \mapsto & \lambda x.x = ?\mathrm{VAR} \\
\mathrm{V} \lesseqgtr \mathrm{TERM} & \mapsto & \lambda x.\mathrm{V}x \wedge \gamma_V(x) \lesseqgtr \gamma_V(\mathrm{TERM}^{\hookrightarrow}) \\
\mathrm{d} & \hookrightarrow & \mathrm{d} \\
?\mathrm{VAR} & \hookrightarrow & ?\mathrm{VAR} \\
\mathrm{f(TERM_1} \ldots \mathrm{TERM}_{\alpha \mathrm{F(f)}}) & \hookrightarrow & \mathrm{f(TERM_1}^{\hookrightarrow} \ldots \mathrm{TERM}_{\alpha \mathrm{F(f)}}{}^{\hookrightarrow})
\end{array}
$$

3.2 ORM syntax and semantics by examples

3.2.1 ORM constraints

We now present a structured overview for each ORM construct with some graphical notation examples and the corresponding syntax and semantics.

Table 3.4: ORM constraints examples

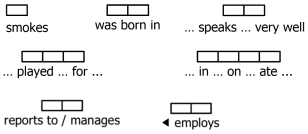
Construct and Examples
Signature: Entity type name

Normative Abstract Syntax of Examples
Entity Type Name: Country

Construct and Examples
Signature: Value type name

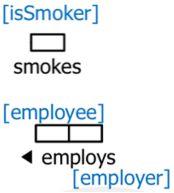
Normative Abstract Syntax of Examples
Value Type name: CountryCode

Construct and Examples
Signature: Predicate name



Normative Abstract Syntax of Examples
Signature:
Unary predicate name: smokes
Binary predicate names: wasBornIn, ?speaks?veryWell,
reportsTo, employs Ternary predicate name: ?played?for?
Quaternary predicate name: ?in?on?ate?
Alternate predicate name: AlternatePredicate(reportsTo, manages (2 1))

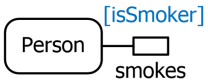
Construct and Examples
Signature: Role name



Normative Abstract Syntax of Examples

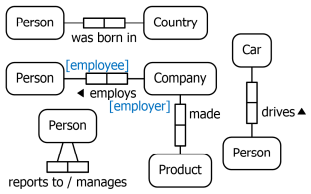
Signature:
Role identifier for the unary predicate smokes:
smokes.1
Role identifiers for the binary predicate employs:
employs.1, employs.2
Role names:
RoleNaming(smokes.1, smokes.isSmoker)
RoleNaming(employs.1, employs.employer)
RoleNaming(employs.2, employs.employee)

Construct and Examples
Unary Fact Type



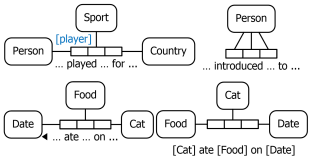
Normative Abstract Syntax of Examples
FactType(smokes (Person))
Normative Semantics of Examples
$\forall x.\text{smoker}(x) \leftrightarrow \text{Person}(x)$

Construct and Examples
Binary Fact Type



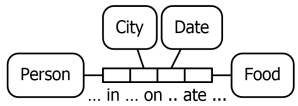
Normative Abstract Syntax of Examples
FactType(wasBornIn (Person Country))
FactType(employs (Company Person))
FactType(made (Company Product))
FactType(drives (Person Car))
FactType(reportsTo (Person Person))
Normative Semantics of Examples
$\forall x, y.\text{wasBornIn}(x, y) \rightarrow \text{Person}(x) \wedge \text{Country}(y)$
$\forall x, y.\text{employs}(x, y) \rightarrow \text{Company}(x) \wedge \text{Person}(y)$
$\forall x, y.\text{made}(x, y) \rightarrow \text{Company}(x) \wedge \text{Product}(y)$
$\forall x, y.\text{drives}(x, y) \rightarrow \text{Person}(x) \wedge \text{Car}(y)$
$\forall x, y.\text{reportsTo}(x, y) \rightarrow \text{Person}(x) \wedge \text{Person}(y)$

Construct and Examples
Ternary Fact Type



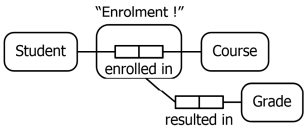
Normative Abstract Syntax of Examples
FactType(?played?for? (Person Sport Country))
FactType(?introduced?to? (Person Person Person))
FactType(?ate?on? (Cat Food Date))
Normative Semantics of Examples
$\forall x, y, z.?\text{played?for?}(x, y, z) \rightarrow \text{Person}(x) \wedge \text{Sport}(y) \wedge \text{Country}(z)$
$\forall x, y, z.?\text{introduced?to?}(x, y, z) \rightarrow \text{Person}(x) \wedge \text{Person}(y) \wedge \text{Person}(z)$
$\forall x, y, z.?\text{ate?on?}(x, y, z) \rightarrow \text{Cat}(x) \wedge \text{Food}(y) \wedge \text{Date}(z)$

Construct and Examples
Quaternary Fact Type



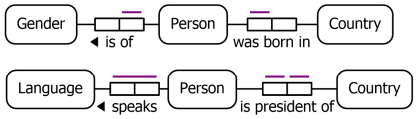
Normative Abstract Syntax of Examples
FactType(?in?on?ate? (Person City Date Food))
Normative Semantics of Examples
$\forall x, y, z, k.?\text{in?on?ate?}(x, y, z, k) \rightarrow \text{Person}(x) \wedge \text{City}(y) \wedge \text{Date}(z) \wedge \text{Food}(k)$

Construct and Examples
Objectification



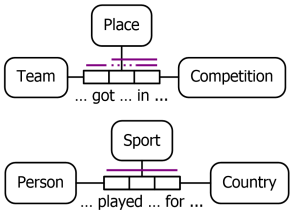
Normative Abstract Syntax of Examples
FactType(enrolledIn (Student Course))
Objectifies(Enrolment enrolledIn)
FactType(resultedIn (Enrolment Grade))
Normative Semantics of Examples
$\forall x, y.\text{enrolledIn}(x, y) \rightarrow \text{Student}(x) \wedge \text{Course}(y)$
$\forall x, y.\text{enrolledIn}(x, y) \leftrightarrow \text{Enrollment}(l_{enrolledIn(x,y)})$
$\forall x, y.\text{enrolledIn}(x, y) \rightarrow \text{Enrollment}(x) \wedge \text{Grade}(y)$

Construct and Examples
Uniqueness on Binary Fact Type



Normative Abstract Syntax of Examples
Unique(isOf.1)
Unique(wasBornIn.1)
Unique(speaks.1 speaks.2)
Unique(isPresidentOf.1)
Unique(isPresidentOf.2)
Normative Semantics of Examples
$\forall x_1, x_2 \text{isOf}(x_1, x_2) \rightarrow \exists^{=1} y.\text{isOf}(x_1 y)$
$\forall x_1, x_2 \text{wasBornIn}(x_1, x_2) \rightarrow \exists^{=1} y.\text{wasBornIn}(x_1 y)$
$\forall x_1, x_2 \text{speaks}(x_1, x_2) \rightarrow \text{speaks}(x_1, x_2)$
$\forall x_1, x_2 \text{isPresidentOf}(x_1, x_2) \rightarrow \exists^{=1} y.\text{isPresidentOf}(x_1 y)$
$\forall x_1, x_2 \text{isPresidentOf}(x_1, x_2) \rightarrow \exists^{=1} y.\text{isPresidentOf}(y x_2)$

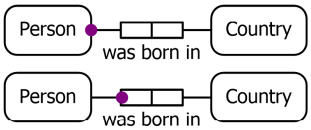
Construct and Examples
Uniqueness on Ternaries



Normative Abstract Syntax of Examples

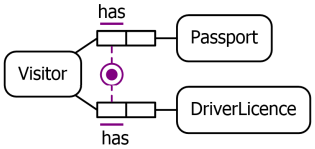
Unique(?got?in?.1 ?got?in?.3)
Unique(?got?in?.2 ?got?in?.3)
Unique(?played?for?.1 ?played?for?.2 ?played?for?.3)
Normative Semantics of Examples
$\forall x_1, x_2, x_3.?\text{got?in?}(x_1, x_2, x_3) \rightarrow \exists^{=1} y.?\text{got?in?}(x_1, y, x_3)$
$\forall x_1, x_2, x_3.?\text{got?in?}(x_1, x_2, x_3) \rightarrow \exists^{=1} y.?\text{got?in?}(y, x_2, x_3)$
$\forall x_1, x_2, x_3.?\text{played?for?}(x_1, x_2, x_3) \rightarrow \exists^{=1} y.?\text{played?for?}(x_1, x_2, x_3)$

Construct and Examples
Simple Mandatory Role



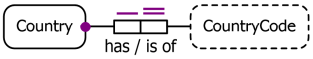
Normative Abstract Syntax of Examples
Mandatory(Person wasBornIn.1)
Normative Semantics of Examples
$\forall x.\text{Person}(x) \rightarrow \exists y.\text{wasBornIn}(x, y)$

Construct and Examples
Inclusive-or



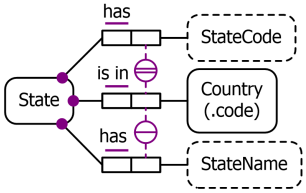
Normative Abstract Syntax of Examples
Mandatory(Visitor hasPassport.1 hasDriverLicence.1)
Normative Semantics of Examples
$\forall x.\text{Visitor}(x) \rightarrow (\exists y.\text{hasPassport}(x, y)) \vee (\exists y.\text{hasDriverLicense}(x, y)))$

Construct and Examples
Preferred internal Uniqueness



Normative Abstract Syntax of Examples
Identification(Country has.1 (has.2))
Normative Semantics of Examples
$\forall x_1, x_2.\text{has}(x_1, x_2) \rightarrow \exists^{=1} y.\text{has}(x_1, y)$
$\forall x.\text{Country}(x) \rightarrow \exists y.\text{has}(x, y)$
$\forall x_1, x_2.\text{has}(x_1, x_2) \rightarrow \exists^{=1} y.\text{has}(y, x_2)$
well-founded(has)

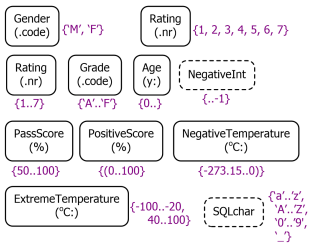
Construct and Examples
External Uniqueness



Normative Abstract Syntax of Examples
ExternalIdentification(State (hasStateCode.2 isIn.2))
ExternalUnique(hasStateName.2 isIn.2)
Normative Semantics of Examples

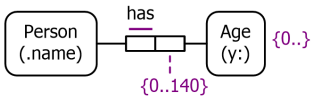
$\forall x_1, x_2, x_3.\text{JoinPath1}(x_1, x_2, x_3) \leftrightarrow \exists y.\text{hasStateCode}(x_3, x_1) \wedge \text{isIn}(x_3, x_2)$
$\forall x_1, x_2, x_3.\text{JoinPath1}(x_1, x_2, x_3) \rightarrow \exists^{=1} y.\text{JoinPath1}(x_1, x_2, y)$
$\forall x_1, x_2, x_3.\text{JoinPath1}(x_1, x_2, x_3) \rightarrow \exists^{=1} y1, y2.\text{JoinPath1}(y_1, y_2, x_3)$
$\forall x \text{State}(x) \rightarrow \exists y_1, y_2 \text{K1}(y_1, y_2, x)$
well-founded(hasStateCode $\cup$ isIn)
$\forall x_1, x_2, x_3.\text{JoinPath2}(x_1, x_2, x_3) \leftrightarrow \text{hasStateName}(x_3, x_1) \wedge \text{isIn}(x_3, x_2)$
$\forall x_1, x_2, x_3.\text{JoinPath2}(x_1, x_2, x_3) \rightarrow \exists^{=1} y.\text{JoinPath2}(x_1, x_2, y)$

Construct and Examples
Object Type Value



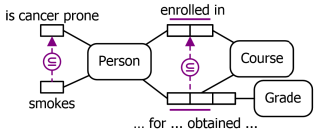
Normative Abstract Syntax of Examples
ValuesOf(GenderCode (M F))
Normative Semantics of Examples
$\forall x.\text{GenderCode}(x) \rightarrow x = M \vee x = F$

Construct and Examples
Role value



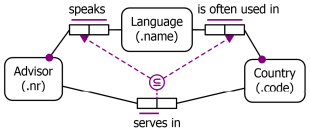
Normative Abstract Syntax of Examples
ValuesOf(has.2 (0 ... 140))
Normative Semantics of Examples
$\forall x_1, x_2.\text{has}(x_1, x_2) \rightarrow x_2 = 0 \vee \cdots \vee x_2 = 140$

Construct and Examples
Subset



Normative Abstract Syntax of Examples
Subset((smokes.1 isCancerProne.1))
Subset((?for?obtained?.1 enrolledIn.1) (?for?obtained?.2 enrolledIn.2))
Normative Semantics of Examples
$\forall x.\text{smokes}(x) \rightarrow \text{isCancerPro}(x)$
$\forall x_1, x_2, x_3.\text{?for?obtained?}(x_1, x_2, x_3) \rightarrow \text{enrolledIn}(x_1, x_2)$

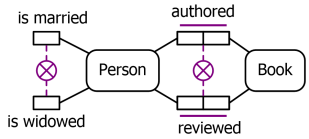
Construct and Examples
Join Subset



Normative Abstract Syntax of Examples
JoinPath(P (speaks.1 speaks.2) (isOftenUsedIn.1 isOftenUsedIn.2))
Subset((servesIn.1 P.1)(servesIn.2 P.2))

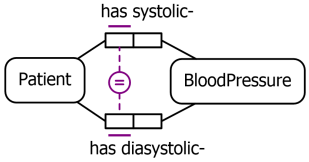
Normative Semantics of Examples
$\forall x_1, x_2.\mathrm{P}(x_1, x_2) \leftrightarrow \exists y.\mathrm{speaks}(x_1, y) \wedge \mathrm{isOftenUsedIn}(y, x_2)$
$\forall x, y.\mathrm{servesIn}(x, y) \rightarrow \mathrm{P}(x, y)$

Construct and Examples
Exclusion



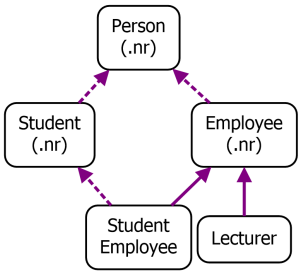
Normative Abstract Syntax of Examples
Exclusive((isWidowed.1 isMarried.1))
Exclusive((reviewed.1 authored.1) (reviewed.2 authored.2))
Normative Semantics of Examples
$\forall x.\mathrm{isWidowed}(x) \rightarrow \neg\mathrm{isMarried}(x)$
$\forall x, y.\mathrm{reviewed}(x, y) \rightarrow \neg\mathrm{authored}(x, y)$

Construct and Examples
Equality



Normative Abstract Syntax of Examples
Equal((hasSystolic.1 hasDiasystolic.1))
Normative Semantics of Examples
$\forall x, y.\mathrm{hasSystolic}(x, y) \rightarrow \exists z.\mathrm{hasDiasystolic}(x, z) \wedge$
$\forall x, y.\mathrm{hasDiasystolic}(x, y) \rightarrow \exists z.\mathrm{hasSystolic}(x, z)$

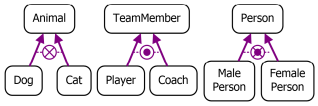
Construct and Examples
Subtyping



Normative Abstract Syntax of Examples
Subtype(Lecturer Employee)
Subtype(Employee Person)
Subtype(Student Person)
Subtype(StudentEmployee Student)
Subtype(StudentEmployee Employee)
Normative Semantics of Examples
$\forall x.\mathrm{Lecturer}(x) \rightarrow \mathrm{Employee}(x)$
$\forall x.\mathrm{Employee}(x) \rightarrow \mathrm{Person}(x)$
$\forall x.\mathrm{Student}(x) \rightarrow \mathrm{Person}(x)$
$\forall x.\mathrm{StudentEmployee}(x) \rightarrow \mathrm{Student}(x)$
$\forall x.\mathrm{StudentEmployee}(x) \rightarrow \mathrm{Employee}(x)$

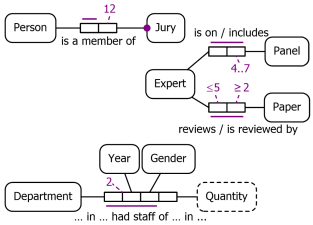
Construct and Examples

Subtyping constraints



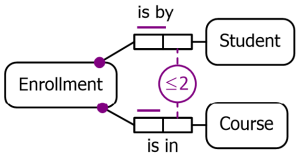
Normative Abstract Syntax of Examples
ExclusiveSubtypes((Dog Cat) Animal)
ExhaustiveSubtypes((Player Coach) TeamMember)
ExclusiveSubtypes((MalePerson FemalePerson) Person)
ExhaustiveSubtypes((MalePerson FemalePerson) Person)
Normative Semantics of Examples
$(\forall x.\text{Dog}(x) \rightarrow \text{Animal}(x) \wedge \neg\text{Cat}(x)) \wedge (\forall x.\text{Cat}(x) \rightarrow \text{Animal}(x))$
$(\forall x.\text{Player}(x) \rightarrow \text{TeamMember}(x)) \wedge (\forall\text{Coach}(x) \rightarrow \text{TeamMember}(x)) \wedge (\forall x.\text{TeamMember}(x) \rightarrow \text{Coach}(x) \vee \text{Player}(x))$
$(\forall x.\text{MalePerson}(x) \rightarrow \text{Person}(x) \wedge \neg\text{FemalePerson}(x)) \wedge (\forall x.\text{FemalePerson}(x) \rightarrow \text{Person}(x))$
$(\forall x.\text{MalePerson}(x) \rightarrow \text{Person}(x)) \wedge (\forall x.\text{FemalePerson(x)} \rightarrow \text{Person}(x)) \wedge (\forall x.\text{Person}(x) \rightarrow \text{FemalePerson}(x) \vee \text{MalePerson}(x))$

Construct and Examples
Internal Frequency



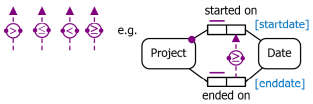
Normative Abstract Syntax of Examples
Frequency(isAMemberOf.2 (12))
Frequency(isOn.2 (4, 7))
Frequency(reviews.1 (..5))
Frequency(reviews.2 (2..))
Frequency(?in?hadStaffOf?in?.1
?in?hadStaffOf?in?.2 (2))
Normative Semantics of Examples
$\forall x_1, x_2.\text{isAMemberOf}(x_1, x_2) \rightarrow \exists^{=12} y.\text{isAMemberOf}(x_1, y)$
$\forall x_1, x_2.\text{isOn}(x_1, x_2) \rightarrow \exists^{\geqslant 4, \leqslant 7} y.\text{isOn}(x_1, y)$
$\forall x_1, x_2.\text{reviews}(x_1, x_2) \rightarrow \exists^{\leqslant 5} y.\text{reviews}(y, x_2)$
$\forall x_1, x_2.\text{reviews}(x_1, x_2) \rightarrow \exists^{\geqslant 2} y.\text{reviews}(x_1, y)$

Construct and Examples
External frequency



Normative Abstract Syntax of Examples
ExternalFrequency(isBy.2 isIn.2 (..2))
Normative Semantics of Examples
$\forall x_1, x_2, x_3.\text{JoinPath}(x_1, x_2, x_3) \leftrightarrow \text{isBy}(x_3, x_1) \wedge \text{isIn}(x_3, x_2))$
$\forall x_1, x_2, x_3.\text{JoinPath}(x_1, x_2, x_3) \rightarrow \exists^{\leqslant 2} y_1, y_2.\text{JoinPath}(y_1, y_2, x_3)$

Construct and Examples
Value-comparison

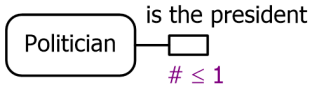


Normative Abstract Syntax of Examples
$\geqslant$(endedOn.2 startedOn.2)
Normative Semantics of Examples
$\forall x_1, x_2, x_3.\text{JoinPath}(x_1, x_2.x_3) \leftrightarrow \text{startedOn}(x_3, x_1) \wedge \text{endedOn}(x_3, x_2))$
$\forall x_1, x_2, x_3, y_1, y_2, y_3.\text{P}(x_1, x_2, x_3) \wedge \text{P}(y_1, y_2, y_3) \leftrightarrow \gamma_{Date}(x_2) \geqslant \gamma_{Date}(y_1)$

Construct and Examples
Object Cardinality

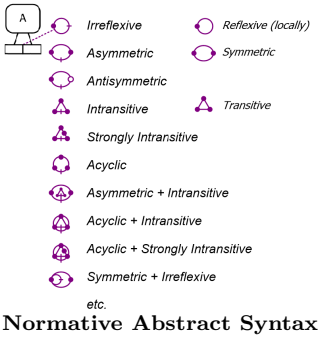
Normative Abstract Syntax of Examples
TypeCardinality(President (0, 1))
Normative Semantics of Examples
$\exists^{\leqslant 1} x.\text{President}(x)$

Construct and Examples
Role Cardinality



Normative Abstract Syntax of Examples
RoleCardinality(isThePresidentOf (0, 1))
Normative Semantics of Examples
$\exists^{\leqslant 1} x.\text{isThePresidentOf}(x)$

Construct and Examples
Ring Constraint

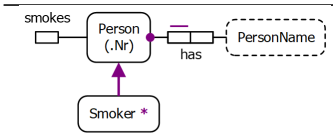


Normative Abstract Syntax of Examples
LocallyReflexive(P.1 P.2) (etc.)
Normative Semantics of Examples
$\forall x_1, x_2.\text{P}(x_1, x_2) \rightarrow \text{P}(x_1, x_1)$

3.2.2 Derivation Rules

We now present the structure of ORM Derivation Rules by examples with the graphical notation and the corresponding syntax and semantics.

Table 3.5: ORM Derivation Rules examples



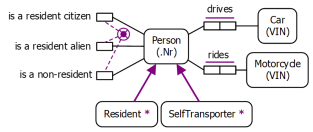
* **Each** Smoker **is a** Person **who** smokes.

Normative Abstract Syntax of Examples

SubTypeRule(Smoker (Person $\wedge$ smokes))

Normative Abstract Semantics of Examples

$\forall x.\text{Smoker}(x) \leftrightarrow \text{Person}(x) \wedge \text{smokes}(x)$



* **Each** Resident **is a** Person **who** is a resident citizen **or** is a resident alien.

* **Each** SelfTransporter **is a** Person **who** drives **a** Car **or** rides **a** Motorcycle.

Normative Abstract Syntax of Examples

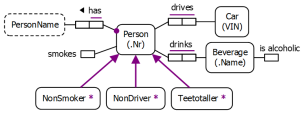
SubTypeRule(Resident
(Person $\wedge$ (isAResidentCitizen $\wedge$
isAResidentAlien))

SubTypeRule(SelfTransporter
(Person $\wedge$
((drives.1 $\hookrightarrow$ [drives.2 $\times$ Car]) $\vee$
(rides.1 $\hookrightarrow$ [rides.2 $\times$ Motorcycle])))

Normative Abstract Semantics of Examples

$\forall x.\text{Resident}(x) \leftrightarrow$
$\text{Person}(x) \wedge (\text{isAResidentCitizen}(x) \vee$
$\text{isAResidentAlien}(x))$

$\forall x.\text{SelfTransporter}(x) \leftrightarrow$
$(\text{Person}(x) \wedge$
$((\exists y.\text{drives}(x, y) \wedge \text{Car}(y)) \vee$
$(\exists y.\text{rides}(x, y) \wedge \text{Motorcycle}(y)))$



* **Each** NonSmoker **is a** Person **where it is not true that that** Person smokes.

* **Each** NonDriver **is a** Person **who** drives **no** Car.

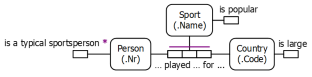
* **Each** Teetotaller **is a** Person **who** drinks **no** Beverage **that** is alcoholic.

Normative Abstract Syntax of Examples

SubTypeRule(NonSmoker (Person \ smokes))
SubTypeRule(NonDriver
(Person \ (drives.1 $\hookrightarrow$ [drives.2 $\times$ Car]))
SubTypeRule(TeeTotaller
(Person \
(drinks.1 $\hookrightarrow$ [drinks.2 $\times$
(Beverage $\wedge$ isAlcoholic)]))

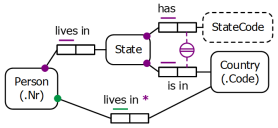
Normative Abstract Semantics of Examples

$\forall x.\text{NonSmoker}(x) \leftrightarrow \text{Person}(x) \wedge \neg\text{smokes}(x)$
$\forall x.\text{NonDriver}(x) \leftrightarrow$
  $\text{Person}(x) \wedge \neg(\exists y.\text{drives}(x, y) \wedge \text{Car}(y))$
$\forall x.\text{TeeTotaller}(x) \leftrightarrow$
  $(\text{Person}(x) \wedge$
    $(\exists y.\text{drinks}(x, y) \wedge$
    $\text{Beverage}(y) \wedge \text{isAlcoholic}(y)))$



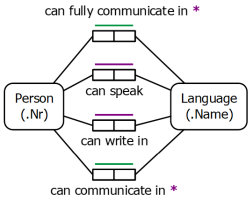
* Person is a typical sportsperson **iff**
that Person played **a** Sport **that** is popular for **a** Country **that** is large.

Normative Abstract Syntax of Examples
FactTypeRule(isATypicalSportsPerson
  (Person $\wedge$?played?for?.1 $\hookrightarrow$
    [?played?for?.2 $\times$ (Sport $\wedge$ isPopular)]
    [?played?for?.3 $\times$ (Country $\wedge$ isLarge)]))
Normative Abstract Semantics of Examples
$\forall x.\text{isATypicalSportsPerson}(x) \leftrightarrow$
  $(\text{Person}(x) \wedge$
    $\exists y, z.?\text{played?for?}(x, y, z) \wedge \text{Sport}(y) \wedge \text{isPopular}(y) \wedge$
    $\text{Country}(z) \wedge \text{isLarge}(z))$



* Person lives in Country **iff**
that Person lives in **a** State **that** is in **that** Country.

Normative Abstract Syntax of Examples
FactTypeRule(livesInCountry
  (Person \ livesInState.1 $\hookrightarrow$
    [livesInState.2 $\times$ (State $\wedge$ isIn.1 $\hookrightarrow$
      [isIn.2 $\times$ (Country $\wedge$?x)])])
  (Country $\wedge$?x))
Normative Abstract Semantics of Examples
$\forall x, y.\text{livesInCountry}(x, y) \leftrightarrow$
$(\text{Person}(x) \wedge$
  $\exists z.\text{livesInState}(x, z) \wedge \text{State}(z) \wedge$
  $\text{isIn}(z, y) \wedge \text{Country}(y))$



* Person can fully communicate in Language **iff**
that Person can speak **that** Language
and can write in **that** Language.

* Person can communicate in Language **iff**
that Person can speak **that** Language
or can write in **that** Language.

Normative Abstract Syntax of Examples

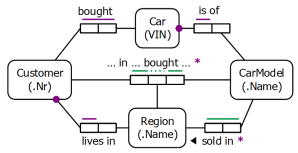
FactTypeRule(canFullyCommunicateIn
(Person ∧
(canSpeak.1 ↪[canSpeak.2 × (Language ∧ ?x)]) ∧
(canWrite.1 ↪[canwrite.2 × (Language ∧ ?x)]))
(Language ∧ ?x))

FactTypeRule(canCommunicateIn
(Person ∧
(canSpeak.1 ↪[canSpeak.2 × (Language ∧ ?x)]) ∨
(canWrite.1 ↪[canwrite.2 × (Language ∧ ?x)]))
(Language ∧ ?x))

Normative Abstract Semantics of Examples

$\forall x, y.\text{canFullyCommunicateIn}(x, y) \leftrightarrow$
$(\text{Person}(x) \wedge$
$\text{canSpeak}(x, y) \wedge$
$\text{canWrite}(x, y) \wedge$
$\text{Language}(y))$

$\forall x, y.\text{canCommunicateIn}(x, y) \leftrightarrow$
$(\text{Person}(x) \wedge$
$(\text{canSpeak}(x, y) \vee$
$\text{canWrite}(x, y)) \wedge$
$\text{Language}(y))$



* CarModel sold in Region **iff**
some Customer lives in **that** Region
and bought **a** Car **that** is of **that** CarModel.

* Customer in Region bought CarModel **iff**
that Customer lives in **that** Region
and bought **a** Car **that** is of **that** CarModel.

Normative Abstract Syntax of Examples

FactTypeRule(soldIn
(CarModel ∧ ?x)
(Region ∧
(livesIn.2 ↪[livesIn.1 × Customer ∧
(bought.1 ↪[bought.2 × Car ∧
(isOf.1 ↪[isOf.2 × (CarModel ∧?x)])])])))

FactTypeRule(?in?bought?
(Customer ∧
(livesIn.1 ↪ [livesIn.2 × (Region ∧ ?x)]) ∧
(bought.1 ↪ [bought.2 × (Car ∧
(isOf.1 ↪ [isOf.2 × CarModel ∧ ?y]))]))
(Region ∧ ?x)
(CarModel ∧ ?y))

Normative Abstract Semantics of Examples

$$\begin{array}{l}\forall x, y.\text{soldIn}(x, y) \leftrightarrow \\ \quad \text{CarModel}(x) \wedge \\ \quad \text{Region}(y) \wedge \\ \qquad \exists z.\text{livesIn}(z, y) \wedge \text{Customer}(z) \wedge \\ \qquad\quad \exists k.\text{bought}(z, k) \wedge \text{Car}(k) \\ \qquad\qquad \text{isOf}(k, x)\end{array}$$

$$\begin{array}{l}\forall x, y.?\text{in}?\text{bought}?(x, y, z) \leftrightarrow \\ \quad (\text{Customer}(x) \wedge \\ \quad \text{livesIn}(x, y) \wedge \text{Region}(y) \wedge \\ \quad \exists k.\text{bought}(x, k) \wedge \text{Car}(k) \wedge \\ \qquad \text{isOf}(k, z) \wedge \text{CarModel}(z))\end{array}$$

3.2.3 Example with ORM Derivation Rules

We can now apply this formalisation to the example about documents from the Section 2.3.2. For each ORM constraint we show the syntax and the corresponding semantics in first-order logic.

FactType(has (PersonWithID Document))
$\forall x, y.has(x, y) \rightarrow PersonWithID(x) \wedge Document(y)$
FactType(has (SomeVisitor Visa))
$\forall x, y.owns(x, y) \rightarrow SomeVisitor(x) \wedge Visa(y)$
Subtype((SomeVisitor) PersonWithID)
$\forall x.SomeVisitor(x) \rightarrow PersonWithID(x)$
Subtype((Visa IDcard) Document)
$\forall x.Visa(x) \rightarrow Document(x)$
$\forall x.IDcard(x) \rightarrow Document(x)$
Mandatory(PersonWithID has.1)
$\forall x.PersonWithID(x) \rightarrow \exists y.has(x, y)$
Mandatory(SomeVisitor owns.1)
$\forall x.SomeVisitor(x) \rightarrow \exists y.owns(x, y)$
Unique (has.1)
$\forall x, y.has(x, y) \rightarrow \exists^{\leqslant 1} z.has(x, z)$
Unique (has.2)
$\forall x, y.has(x, y) \rightarrow \exists^{\leqslant 1} z.has(z, y)$
Unique (owns.1)
$\forall x, y.owns(x, y) \rightarrow \exists^{\leqslant 1} z.owns(x, z)$
Unique (owns.2)

$\forall x, y.owns(x, y) \rightarrow \exists^{\leqslant 1} z.owns(z, y)$
ExclusiveSubtypes ((Visa IDCard) Document)
$\forall x.Visa(x) \rightarrow \neg IDCard(x)$
Exhaustive ((Visa IDCard) Document)
$\forall x.Document(x) \rightarrow Visa(x) \vee IDCard(x)$
Subset ((owns.1 has.1) (owns.2 has.2))
$\forall x, y.owns(x, y) \rightarrow has(x, y)$

Additionally, the ORM diagram has been extended in Section 2.3.3 with a Derivation Rule on the VisitorWithVisa entity:

SubtypeRule(VisitorWithVisa (PersonWithID $\wedge$
(has.1 $\hookrightarrow$ [has.2 $\times$ (Document $\wedge$ Visa)]))
$\forall x.VisitorWithVisa(x) \leftrightarrow PersonWithID(x) \wedge \exists y.has(x, y) \wedge Visa(y)$

4

ORM Reasoning

The purpose of this chapter is to show the benefits coming from the application of automated reasoning to ORM models. A set of examples are presented for different scenarios where the modeller may design wrong models, or the inferred knowledge coming from the application of the reasoning may reveal unexpected software behaviours. In this way, the inferred knowledge is helpful to the modeller in order to enhance the control over the ORM models semantics. The first section is a set of examples where the reasoning is applied to some ORM models; the second section extends the reasoning on ORM diagrams equipped with ORM Derivation Rules.

4.1 Reasoning with ORM

We show the reasoning for the ORM conceptual diagram in Figure 4.1. As we have seen in Section 2.3.2, this diagram represents a domain about people and their documents. The conceptual diagram captures all the necessary entities (Person with ID, Visitor, Document, etc.) together with their relationships (owns, has) and additional constraints (such as cardinalities, subtyping, uniqueness, etc.), thus providing a quite precise idea of the specific domain, where each Person with ID has a document which can be either visa or id card. A Person with ID can be a citizen or visitor.

What can be the outcome of the reasoning and why? The system could automatically complete the diagram in the way depicted in Figure 4.2. The uniqueness constraints are place on the fact type *owns*, since this fact type is a subset relation of the fact type *has*. This means that all pairs inside *owns*

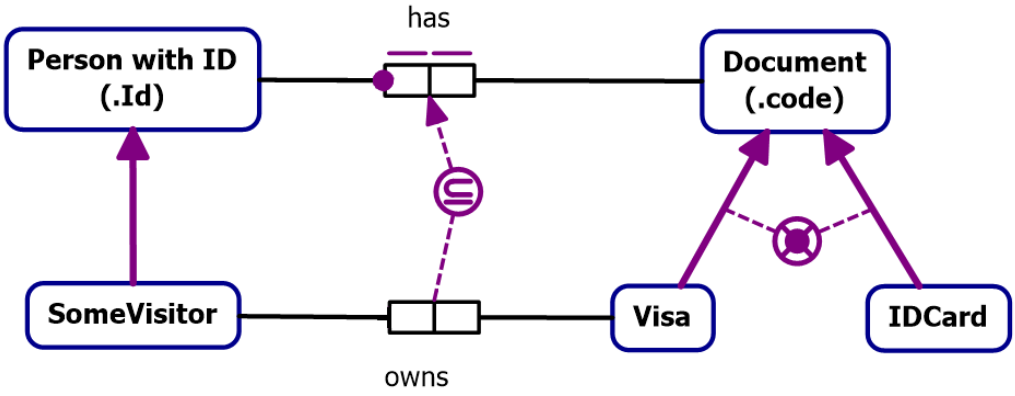


Figure 4.1: Document example

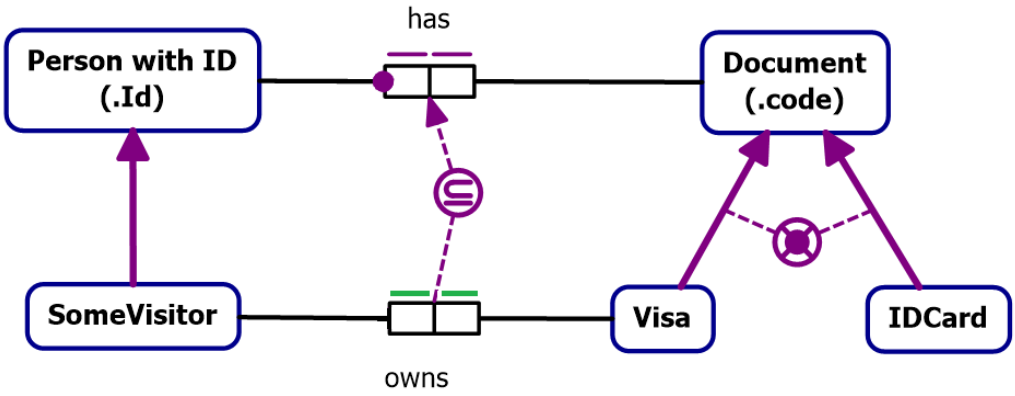


Figure 4.2: Document example: uniqueness inferred

are also included inside *has*; since all the pairs inside has have the uniqueness keys and this is inherited by the fact type owns, it is necessarily true that each SomeVisitor owns at most one Visa and for each Visa, at most one SomeVisitor owns that Visa.

Now, let us suppose the modeller decides to state that each visa document is also an IDCard, as in Figure 4.3. The Visa entity is inconsistent as shown in Figure 4.4, i.e., does not have any instance, since the disjointness constraint in the IsA link states that there is no element in common between Visa and IDCard. The empty set denoted by the visa entity is the only set which can be at the same time disjoint to and a subset of another set. Since the Document entity is formed by the union of the Visa and IDCard, and the Visa

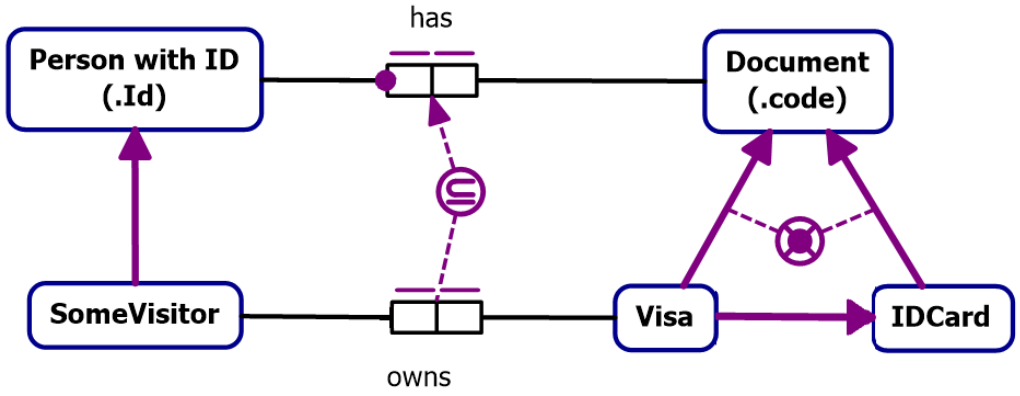


Figure 4.3: Document example: IsA

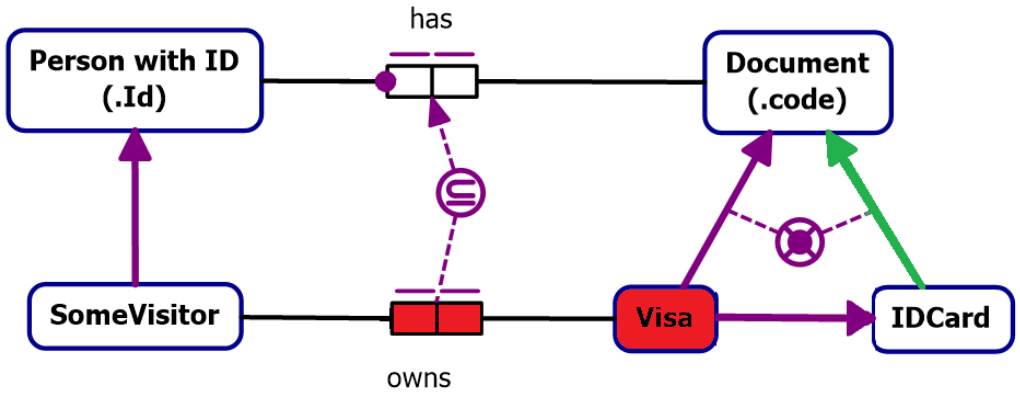


Figure 4.4: Document example: inconsistencies

entity is inconsistent, the IDCard entity becomes equivalent to the Document entity. Others inferences are triggered by the empty set. Since there is no visa, there is no pair in the owns relationship as well (i.e., it is inconsistent): the diagram states that any second argument of the owns relationship should be of the visa type. The SomeVisitor entity is not inconsistent, since it may be populated by people which do not necessarily own a visa at all (this is possible, since there is no mandatory participation constraint).

Indeed, let us now add a cardinality constraint, stating that each SomeVisitor must own a Visa document (i.e., a mandatory participation constraint by the purple dot). The change results in the diagram of Figure 4.5. Now the system deduces that the Somevisitor entity is inconsistent as well (see Figure 4.6).

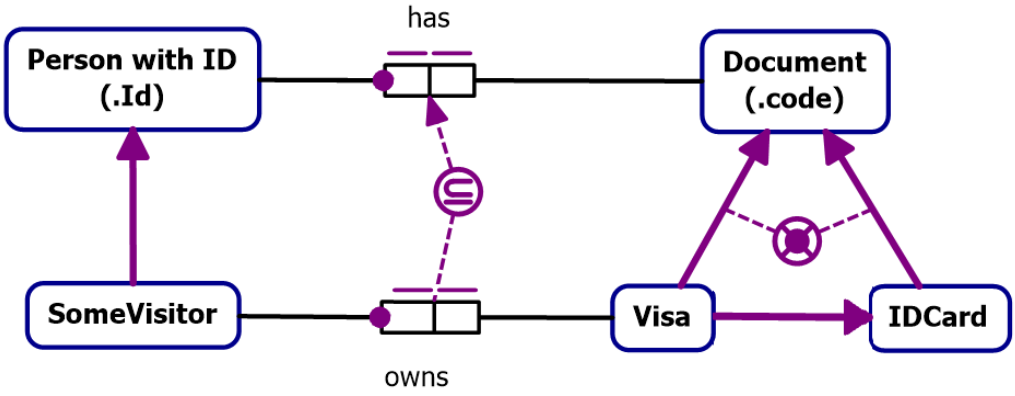


Figure 4.5: Document example: mandatory constraint

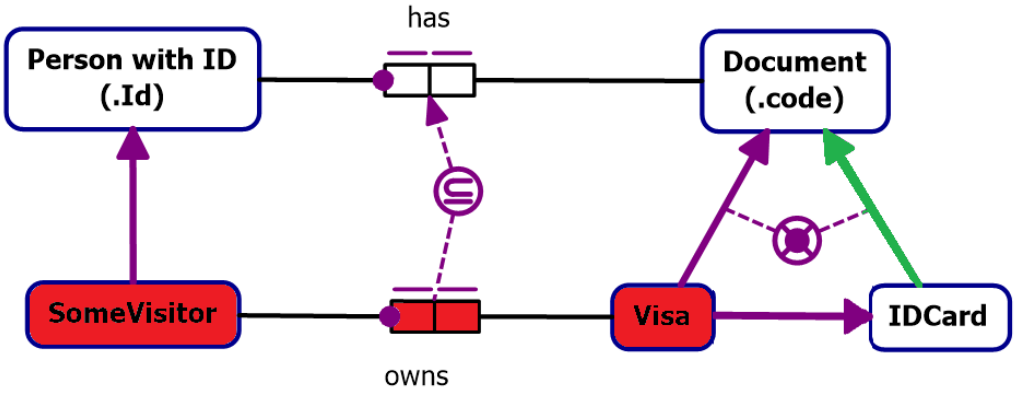


Figure 4.6: Document example: deductions with the mandatory constraint

Despite the tiny size of the ORM diagrams in these examples, some trivial mistakes can trigger a chain of an unexpected behaviours that may degrade the quality of the diagram. This may affect the quality of the software or database which usually is obtained from the conceptual model. In the real-world, huge diagrams are often used (as shown in the real-world scenario presented in Chapter 7), so the benefits coming from the reasoning may be crucial in scenarios where huge data are managed and multiple iterations are performed.

4.2 Reasoning with ORM Derivation Rules

In this section we analyse ORM Derivation Rules. They are special ORM constraints able to express knowledge that is beyond standard ORM capabilities, in a way similar to OCL constraints for UML or SQL triggers. ORM Derivation Rules are expressed in the same controlled natural language as standard ORM, namely FORML [90]. In this way it is very simple for non-IT users to bridge the gap to the technical stakeholders working in the same domain. In the ORM graphical notation, ORM Derivation Rules are depicted by an asterisk (*) near the involved object types or fact types, plus a text box where the sentence in controlled natural language is put in. There are two main categories of rules: Subtype and Fact Types Derivation Rules. Subtype Derivation Rules are placed on object types that are a specialisation of another entity types (IsA relationship); Fact Type Derivation Rules define fact types by means of better specified argument types. A derivation rule is a collection of restrictions defined along one or more paths in the ORM diagram, in order to precisely define the constraint for an object type or fact type to be valid inside the domain.

The following examples are meant to show the usage of a reasoner engine over those diagrams equipped with ORM Derivation Rules. In this way the automated reasoning is also applied to the rules with the consequence of providing additional inferences for the whole ORM diagram.

```
Each VisitorWithVisa is by definition some Person that has some Document
that is some Visa.
```

$$\forall x.\, VisitorWithVisa(x) \leftrightarrow Person(x) \wedge \exists y.has(x,y) \wedge\ Visa(y)$$

The derivation rule *defines* exactly what a visitor with VISA is, by means of an *if-and-only-if* statement.

Given the ORM conceptual schema with the derivation rule of Fig. 4.7, it is obvious that the entity *SomeVisitor* should turn out to be a subtype of *VisitorWithVisa*, as shown in green in Fig. 4.8. This inference can be automatically computed by a logic prover using the semantic translation of

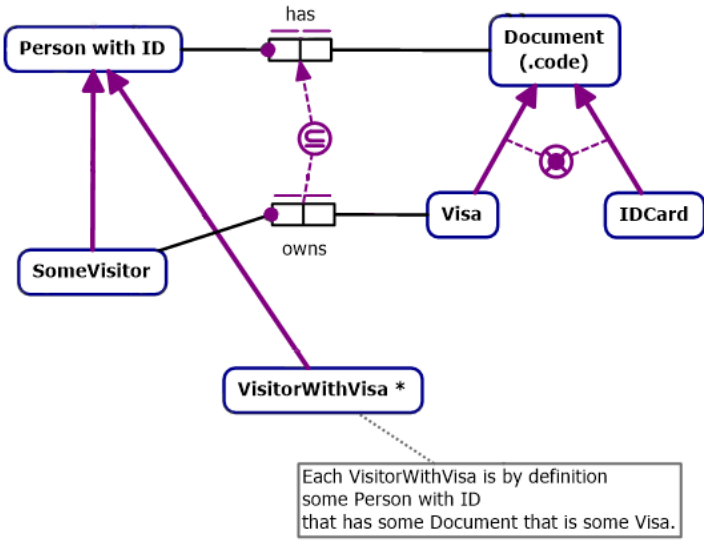


Figure 4.7: Adding the entity type *VisitorWithVisa*.

the ORM conceptual schema. As a matter of fact, the tools implementing an inference engine for ORM are all based on description logics provers (a comparison of ORMiE with these tools is provided in Chapter 9), exploiting

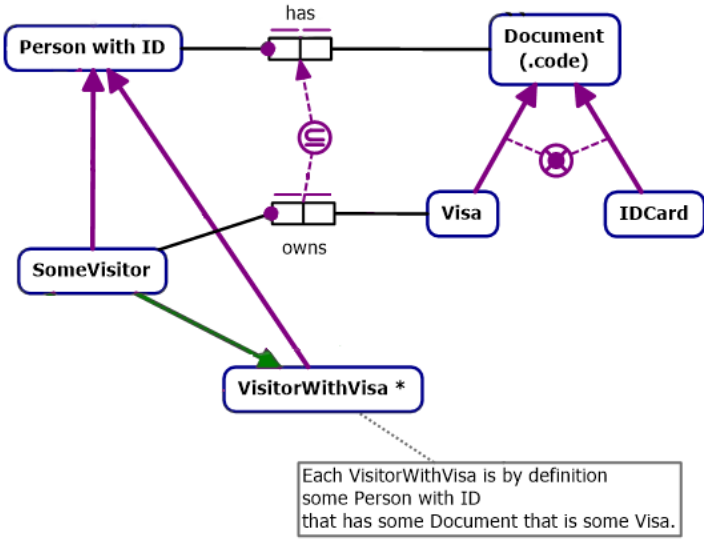


Figure 4.8: Reasoning after adding the entity type *VisitorWithVisa*.

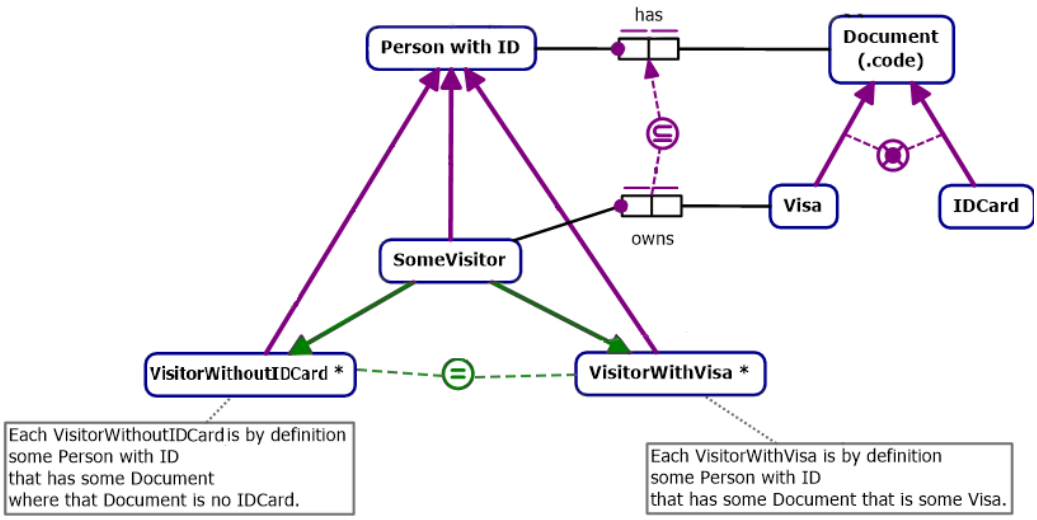


Figure 4.9: Reasoning after adding the entity type *VisitorWithoutIDCard*.

the conversion of the first-order semantics to some computable fragment of description logic.

Note that there is also an alternative way to express the same derivation rule, since documents can be either VISAs or IDCards but not both. Indeed, if we add the entity *VisitorWithoutIDCard* with the following derivation rule:

```
Each VisitorWithoutIDCard is by definition some Person that has some
Document where that Document is no IDCard.
```

$$\forall x.\, VisitorWithoutIDCard(x) \leftrightarrow Person(x) \wedge \exists y.has(x,y) \wedge \neg IDCard(y),$$

it turns out that the new entity behaves exactly like the previous one, and indeed it can be derived that they are equivalent (see the new inferred links in green in Fig. 4.9).

Now, let's suppose that all the people with an IDCard are citizens and viceversa, and stateless people are exactly those without any document:

```
Each Citizen is by definition some Person that has some Document that
is
some IDCard.
```

```
Each Stateless with ID is by definition some Person that
has no Document that is some IDCard
and has no Document that is some VISA.
```

$$\forall x.Citizen(x) \leftrightarrow Person(x) \wedge \exists y.has(x,y) \wedge IDCard(y)$$

$$\forall x.StatelessWithID(x) \leftrightarrow Person(x) \wedge \neg\exists y.(has(x,y) \wedge IDCard(y)) \wedge \neg\exists z.(has(x,z) \wedge Visa(z))$$

Note the complete outcome of the reasoning process in green and red in Fig. 4.11 among others, valid inferences according to this formalisation of the domain are that persons are partitioned between citizens and visitors with VISA, and that there can't be any stateless person. Indeed, stateless persons are defined not to hold any VISA nor IDCard, but persons are required to have exactly one document. The whole schema makes the *StatelessWithId* entity inconsistent (in red). Clearly, if persons were not obliged to have

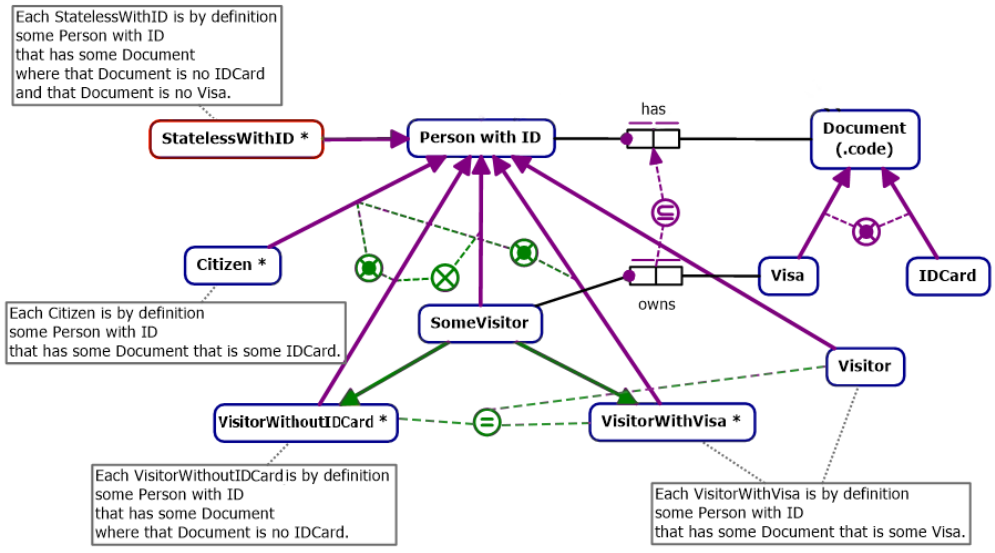


Figure 4.10: Reasoning after adding the entity types *Citizen* and *StatelessWithID*.

exactly one document (i.e., via the mandatory participation constraint), the reasoning process would not derive the inconsistency of the Stateless entity.

In order to clarify the distinction between the people who possess a document and the others, in Figure 4.11 we rename the entity previously called Person as *Person with ID*, and we introduce a more generic entity named *Person*. We also introduce a new entity called Stateless associated to the same derivation rule specified for StatelessWithID. The outcome of the reasoning depicted in green tells us that the newly defined Stateless entity type, unlike the previous one, is now consistent, and disjoint from Person with ID. The reason behind this is the absence of the mandatory constraint on the entity Person, which states that it is not mandatory for a person to have a document.

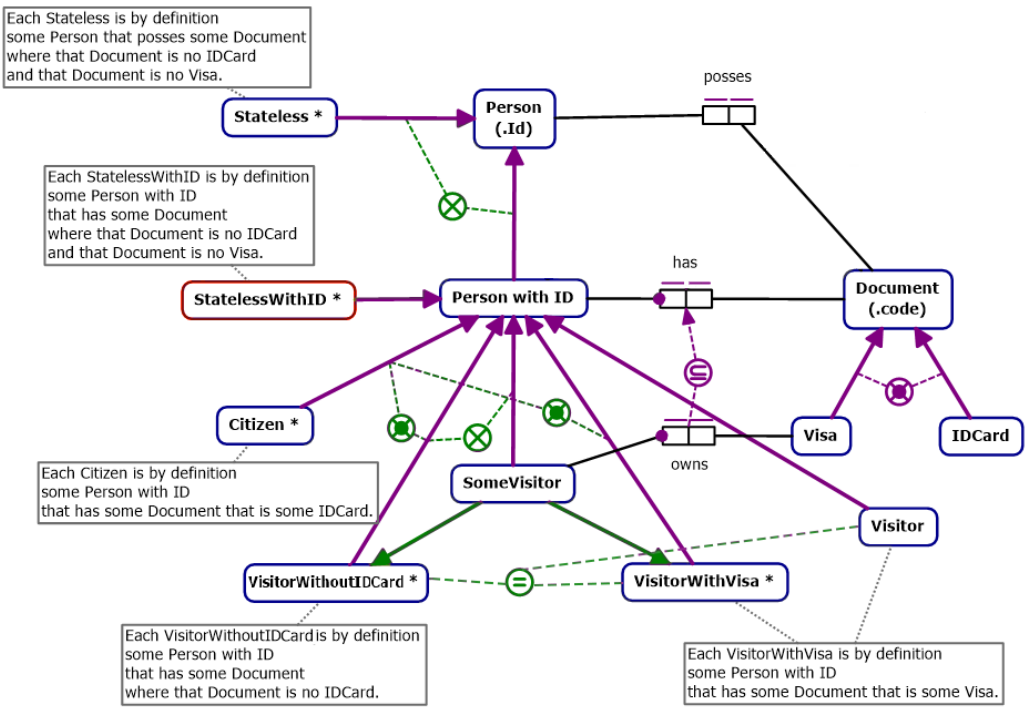


Figure 4.11: Fact type Derivation Rule reasoning.

5

Encoding ORM in $\mathcal{DLR}^{\pm}$

This chapter defines a decidable fragment for ORM in order to activate reasoning algorithms on those ORM diagram expressed in a fragment named $\mathcal{ORM}^{\pm}$. We start from the usage of $\mathcal{DLR}^{+}$ which is a language from the Description Logics family, suitable for representing conceptual modelling languages such as UML, ER and ORM, since it supports n-ary relationships and some relevant constructs that are easy to map in this language. A decidable version of $\mathcal{DLR}^{+}$ is $\mathcal{DLR}^{\pm}$ which is used to capture the most relevant ORM constraints and to define the $\mathcal{ORM}^{\pm}$ decidable fragment.

5.1 The $\mathcal{DLR}^{+}$ language

The content of this section is taken by the following publication [10], where my contribution is related to the creation of an API system which includes an implementation of the $\mathcal{DLR}^{\pm}$ language and the encoding in $\mathcal{ALCQI}$. This API system is currently part of the conceptual modelling framework described in Chapter 7 which provides an encoding for those ORM diagrams expressed in the $\mathcal{ORM}^{\pm}$ fragment. This work has also been integrated in the ORMiE tool which is described in Chapter 8.

$\mathcal{DLR}^{+}$ is an extension of the n-ary propositionally closed description logic $\mathcal{DLR}$ to deal with attribute-labelled tuples (generalising the positional notation), projections of relations, and global and local objectification of relations, able to express inclusion, functional, key, and external uniqueness dependencies. The logic is equipped with both TBox and ABox axioms. We show how

a simple syntactic restriction on the appearance of projections sharing common attributes in a $\mathcal{DLR}^{+}$ knowledge base makes reasoning in the language decidable with the same computational complexity as $\mathcal{DLR}$. The obtained $\mathcal{DLR}^{\pm}$ n-ary description logic is able to encode more thoroughly conceptual data models such as EER, UML, and ORM.

We introduce the description logic (DL) $\mathcal{DLR}^{+}$ extending the n-ary DL $\mathcal{DLR}$ [40], in order to capture database oriented constraints. While $\mathcal{DLR}$ is a rather expressive logic, tailored for conceptual modelling and ontology design, generalising many aspects of classical description logics and OWL, it lacks a number of expressive means relevant for database applications that can be added without increasing the complexity of reasoning—when used in a carefully controlled way. The added expressivity is motivated by the increasing use of description logics as an abstract conceptual layer (an *ontology*) over relational databases. For example, the $\mathcal{DLR}$ family of description logics is used to formalise and perform reasoning in the ORM conceptual modelling language for database design (adopted by Microsoft in Visual Studio) [127].

We remind that a $\mathcal{DLR}$ knowledge base, as defined in [40], can express axioms with (i) propositional combinations of concepts and (compatible) n-ary relations – as opposed to just binary roles as in classical description logics and OWL, (ii) concepts as unary projections of n-ary relations – generalising the existential operator over binary roles in classical description logics and OWL, and (iii) relations with a selected typed component.
As an example of $\mathcal{DLR}$, in a knowledge base where `Pilot` and `RacingCar` are concepts and `DrivesCar`, `DrivesMotorbike`, `DrivesVehicle` are binary relations, the following statements:

$$\texttt{Pilot} \sqsubseteq \exists[1]\sigma_{2:\texttt{RacingCar}}\texttt{DrivesCar}$$

$$\texttt{DrivesCar} \sqcup \texttt{DrivesMotorbike} \sqsubseteq \texttt{DrivesVehicle}$$

assert that a pilot drives a racing car and that driving a car or a motorbike implies driving a vehicle.

The language we propose here, $\mathcal{DLR}^{+}$, extends $\mathcal{DLR}$ in the following ways.

- While $\mathcal{DLR}$ instances of n-ary relations are n-tuples of objects—whose components are identified by their position in the tuple — instances of

relations in $\mathcal{DLR}^+$ are *attribute-labelled tuples* of objects, i.e., tuples where each component is identified by an attribute and not by its position in the tuple For example, the relation `Employee` may have the signature:

$$\texttt{Employee}(\texttt{firstname}, \texttt{lastname}, \texttt{dept}, \texttt{deptAddr}),$$

and an instance of `Employee` could be the tuple:

$\langle \texttt{firstname} : \texttt{John}, \texttt{lastname} : \texttt{Doe}, \texttt{dept} : \texttt{Purchase}, \texttt{deptAddr} : \texttt{London} \rangle$.

- Attributes can be *renamed*, for example to recover the positional attributes:

$$\texttt{firstname}, \texttt{lastname}, \texttt{dept}, \texttt{deptAddr} \rightleftarrows 1, 2, 3, 4.$$

- *Relation projections* allow the formation of new relations by projecting a given relation on some of its attributes. For example, if `Person` is a relation with signature `Person(name, surname)`, it could be related to `Employee` as follows::

$$\pi[\texttt{firstname}, \texttt{lastname}]\texttt{Employee} \sqsubseteq \texttt{Person},$$
$$\texttt{firstname}, \texttt{lastname} \rightleftarrows \texttt{name}, \texttt{surname}.$$

- The *objectification* of a relation (also known as *reification*) is a concept whose instances are unique *object identifiers* of the tuples instantiating the relation. Those identifiers could be unique only within an objectified relation (*local objectification*), or they could be uniquely identifying tuples independently on the relation they are instance of (*global objectification*). For example, the concept `EmployeeC` could be the *global* objectification of the relation `Employee`, assuming that there is a global 1-to-1 correspondence between pairs of values of the attributes `firstname`, `lastname` and `EmployeeC` instances:

$$\texttt{EmployeeC} \equiv \textcircled{\circ}\, \exists[\texttt{firstname}, \texttt{lastname}]\texttt{Employee}.$$

Consider the relations with the following signatures:

$$\texttt{DrivesCar}(\texttt{name}, \texttt{surname}, \texttt{car}), \quad \texttt{OwnsCar}(\texttt{name}, \texttt{surname}, \texttt{car}),$$

and assume that anybody driving a car also owns it: $\texttt{DrivesCar} \sqsubseteq \texttt{OwnsCar}$. The *locally* objectified events of driving and owning, defined as

$$\texttt{CarDrivingEvent} \equiv \textcircled{\cdot}\,\texttt{DrivesCar}, \quad \texttt{CarOwningEvent} \equiv \textcircled{\cdot}\,\texttt{OwnsCar},$$

do not imply that a car driving event by a person is the owning event by the same person and the same car: $\texttt{CarDrivingEvent} \not\sqsubseteq \texttt{CarOwningEvent}$. Indeed, they are even disjoint: $\texttt{CarDrivingEvent} \sqcap \texttt{CarOwningEvent} \sqsubseteq \bot$.

It turns out that $\mathcal{DLR}^{+}$ is an expressive description logic able to assert relevant constraints typical of relational databases. In Section 5.1.3 we will consider *inclusion dependencies*, *functional and key dependencies*, *external uniqueness* and *identification* axioms. For example, $\mathcal{DLR}^{+}$ can express the fact that the attributes `firstname`, `lastname` play the role of a multi-attribute key for the relation `Employee`:

$$\pi[\texttt{firstname}, \texttt{lastname}]\texttt{Employee} \sqsubseteq \pi^{\leqslant 1}[\texttt{firstname}, \texttt{lastname}]\texttt{Employee},$$

and that the attribute `deptAddr` functionally depends on the attribute `dept` within the relation `Employee`:

$$\exists[\texttt{dept}]\texttt{Employee} \sqsubseteq \exists^{\leqslant 1}[\texttt{dept}]\,(\pi[\texttt{dept}, \texttt{deptAddr}]\texttt{Employee})\,.$$

While $\mathcal{DLR}^{+}$ turns out to be undecidable, we show how a simple syntactic condition on the appearance of projections sharing common attributes in a knowledge base makes the language decidable. The result of this restriction is a new language called $\mathcal{DLR}^{\pm}$. We prove that $\mathcal{DLR}^{\pm}$, while preserving most of the $\mathcal{DLR}^{+}$ expressivity, has a reasoning problem whose complexity does not increase w.r.t. the computational complexity of the basic $\mathcal{DLR}$ language.

5.1.1 Syntax

We start by introducing the syntax of $\mathcal{DLR}^{+}$. A $\mathcal{DLR}^{+}$ *signature* is a tuple $\mathcal{L} = (\mathcal{C}, \mathcal{R}, \mathcal{O}, \mathcal{U}, \tau)$ where $\mathcal{C}$, $\mathcal{R}$, $\mathcal{O}$ and $\mathcal{U}$ are finite, mutually disjoint sets of *concept names*, *relation names*, *individual names*, and *attributes*, respectively, and τ is a *relation signature* function, associating a set of attributes to each relation name $\tau(RN) = \{U_1, \ldots, U_n\} \subseteq \mathcal{U}$, with $n \geqslant 2$.

$$
\begin{array}{rcl}
C & \rightarrow & CN \mid \neg C \mid C_1 \sqcap C_2 \mid \exists^{\geqslant q}[U_i]R \mid \circledcirc R \mid \odot RN \\
R & \rightarrow & RN \mid R_1 \backslash R_2 \mid R_1 \sqcap R_2 \mid R_1 \sqcup R_2 \mid \sigma_{U_i:C}R \\
R & \rightarrow & \pi^{\leqslant q}[U_1, \ldots, U_k]R \mid \pi^{\geqslant q}[U_1, \ldots, U_k]R \\
\varphi & \rightarrow & C_1 \sqsubseteq C_2 \mid R_1 \sqsubseteq R_2 \mid CN(o) \mid RN(U_1{:}o_1, \ldots, U_n{:}o_n) \mid o_1 = o_2 \mid o_1 \neq o_2 \\
\vartheta & \rightarrow & U_1 \rightleftarrows U_2
\end{array}
$$

Figure 5.1: The syntax of $\mathcal{DLR}^+$.

$$
\begin{array}{rcll}
\tau(R_1 \backslash R_2) & = & \tau(R_1) & \\
\tau(R_1 \sqcap R_2) & = & \tau(R_1) & \text{if } \tau(R_1) = \tau(R_2) \\
\tau(R_1 \sqcup R_2) & = & \tau(R_1) & \text{if } \tau(R_1) = \tau(R_2) \\
\tau(\sigma_{U_i:C}R) & = & \tau(R) & \text{if } U_i \in \tau(R) \\
\tau(\pi^{\lessgtr q}[U_1, \ldots, U_k]R) & = & \{U_1, \ldots, U_k\} & \text{if } \{U_1, \ldots, U_k\} \subset \tau(R) \\
\multicolumn{2}{c}{\text{undefined}} & & \text{otherwise}
\end{array}
$$

Figure 5.2: The signature of $\mathcal{DLR}^+$ relations.

The syntax of concepts C, relations R, formulas φ, and attribute renaming axioms ϑ is given in Figure 5.1, where $CN \in \mathcal{C}$, $RN \in \mathcal{R}$, $U \in \mathcal{U}$, $o \in \mathcal{O}$, q is a positive integer and $2 \leqslant k < \text{ARITY}(R)$. The *arity* of a relation R is the number of the attributes in its signature; i.e., $\text{ARITY}(R) = |\, \tau(R) \,|$, with the relation signature function τ extended to complex relations as in Figure 5.2. Note that it is possible that the same attribute appears in the signature of different relations.

As mentioned in the introduction, the $\mathcal{DLR}^+$ constructors added to $\mathcal{DLR}$ are the *local* and *global objectification* ($\odot RN$ and $\circledcirc R$, respectively); *relation projections* with the possibility to count the projected tuples ($\pi^{\lessgtr q}[U_1, \ldots, U_k]R$), and *renaming axioms* over attributes ($U_1 \rightleftarrows U_2$). Note that local objectification ($\odot R$) can be applied to relation names, while global objectification ($\circledcirc RN$) can be applied to arbitrary relation expressions. We use the standard abbreviations:

$$
\bot = C \sqcap \neg C, \quad \top = \neg\bot, \quad C_1 \sqcup C_2 = \neg(\neg C_1 \sqcap \neg C_2), \quad \exists[U_i]R = \exists^{\geqslant 1}[U_i]R,
$$
$$
\exists^{\leqslant q}[U_i]R = \neg(\exists^{\geqslant q+1}[U_i]R), \quad \pi[U_1, \ldots, U_k]R = \pi^{\geqslant 1}[U_1, \ldots, U_k]R.
$$

A $\mathcal{DLR}^+$ *TBox* $\mathcal{T}$ is a finite set of *concept inclusion* axioms of the form $C_1 \sqsubseteq C_2$ and *relation inclusion* axioms of the form $R_1 \sqsubseteq R_2$. We use $X_1 \equiv X_2$ as a shortcut for the two axioms $X_1 \sqsubseteq X_2$ and $X_2 \sqsubseteq X_1$. A $\mathcal{DLR}^+$ *ABox* $\mathcal{A}$ is a finite set of *concept instance* axioms of the form $CN(o)$, *relation instance* axioms of the form $RN(U_1{:}o_1, \ldots, U_n{:}o_n)$, and *same/distinct individual* axioms of the form $o_1 = o_2$ and $o_1 \neq o_2$, with $o_i \in \mathcal{O}$. Restricting ABox axioms to concept and relation names only does not affect the expressivity of $\mathcal{DLR}^+$ due to the availability of unrestricted TBox axioms. A $\mathcal{DLR}^+$ *renaming schema* $\Re$ is a finite set of renaming axioms of the form $U_1 \rightleftarrows U_2$. We use the shortcut $U_1 \ldots U_n \rightleftarrows U'_1 \ldots U'_n$ to group many renaming axioms with the meaning that $U_i \rightleftarrows U'_i$ for all $i = 1, \ldots, n$. A $\mathcal{DLR}^+$ knowledge base (KB) $\mathcal{KB} = (\mathcal{T}, \mathcal{A}, \Re)$ is composed by a TBox $\mathcal{T}$, an ABox $\mathcal{A}$, and a renaming schema $\Re$.

The renaming operator $\rightleftarrows$ is an equivalence relation over the attributes $\mathcal{U}$, $(\rightleftarrows, \mathcal{U})$. The partitioning of $\mathcal{U}$ into equivalence classes induced by a renaming schema is meant to represent the alternative ways to name attributes in the knowledge base. A unique *canonical representative* for each equivalence class is chosen to replace all the attributes in the class throughout the knowledge base. From now on we assume that a knowledge base is consistently rewritten by substituting each attribute with its canonical representative. After this rewriting, the renaming schema does not play any role in the knowledge base. We allow only *arity-preserving* renaming schemas, i.e., there is no equivalence class containing two attributes from the same relation signature.

As shown in the introduction, the renaming schema is useful to reconcile the named attribute perspective and the positional perspective on relations. It is also important to enforce *union compatibility* among relations involved in relation inclusion axioms, and among relations involved in $\sqcap$- and $\sqcup$-set expressions. Two relations are *union compatible* (w.r.t. a renaming schema) if they have the same signature (up to the attribute renaming induced by the renaming schema). Indeed, as it will be clear from the semantics, a relation inclusion axiom involving non union compatible relations would always be false, and a $\sqcap$- and $\sqcup$-set expression involving non union compatible relations would always be empty.

$$
\begin{aligned}
(\neg C)^{\mathcal{I}} &= \top^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(\exists^{\geqslant q}[U_i]R)^{\mathcal{I}} &= \{d \in \Delta \mid |\{t \in R^{\mathcal{I}} \mid t[U_i] = d\}| \geqslant q\} \\
(\circledcirc R)^{\mathcal{I}} &= \{d \in \Delta \mid d = \imath(t) \wedge t \in R^{\mathcal{I}}\} \\
(\odot RN)^{\mathcal{I}} &= \{d \in \Delta \mid d = \ell_{RN}(t) \wedge t \in RN^{\mathcal{I}}\} \\
(R_1 \backslash R_2)^{\mathcal{I}} &= R_1^{\mathcal{I}} \backslash R_2^{\mathcal{I}} \\
(R_1 \sqcap R_2)^{\mathcal{I}} &= R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}} \\
(R_1 \sqcup R_2)^{\mathcal{I}} &= R_1^{\mathcal{I}} \cup R_2^{\mathcal{I}} \\
(\sigma_{U_i:C} R)^{\mathcal{I}} &= \{t \in R^{\mathcal{I}} \mid t[U_i] \in C^{\mathcal{I}}\} \\
(\pi^{\lesseqgtr q}[U_1, \ldots, U_k]R)^{\mathcal{I}} &= \{\langle U_1 : d_1, \ldots, U_k : d_k\rangle \in T_\Delta(\{U_1, \ldots, U_k\}) \mid \\
&\qquad 1 \leqslant |\{t \in R^{\mathcal{I}} \mid t[U_1] = d_1, \ldots, t[U_k] = d_k\}| \lesseqgtr q\}
\end{aligned}
$$

Figure 5.3: The semantics of $\mathcal{DLR}^+$ expressions.

5.1.2 Semantics

The semantics of $\mathcal{DLR}^+$ uses the notion of *labelled tuples* over a countable potentially infinite domain Δ. Given a set of labels $\mathcal{X} \subseteq \mathcal{U}$, an *$\mathcal{X}$-labelled tuple over Δ* (or *tuple* for short) is a *total* function $t\colon \mathcal{X} \to \Delta$. For $U \in \mathcal{X}$, we write $t[U]$ to refer to the domain element $d \in \Delta$ labelled by U. Given $d_1, \ldots, d_n \in \Delta$, the expression $\langle U_1 : d_1, \ldots, U_n : d_n\rangle$ stands for the tuple t defined on the set of labels $\{U_1, \ldots, U_n\}$ such that $t[U_i] = d_i$, for $1 \leqslant i \leqslant n$. The *projection* of the tuple t over the attributes $U_1, \ldots, U_k$ is the function t restricted to be undefined for the labels not in $U_1, \ldots, U_k$, and it is denoted by $t[U_1, \ldots, U_k]$. The relation signature function τ is extended to labelled tuples to obtain the set of labels on which a tuple is defined. $T_\Delta(\mathcal{X})$ denotes the set of all $\mathcal{X}$-labelled tuples over Δ, for $\mathcal{X} \subseteq \mathcal{U}$, and we overload this notation by denoting with $T_\Delta(\mathcal{U})$ the set of all possible tuples with labels within the whole set of attributes $\mathcal{U}$.

A $\mathcal{DLR}^+$ *interpretation* is a tuple $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}}, \imath)$ consisting of a nonempty countable potentially infinite *domain* Δ specific to $\mathcal{I}$, an *interpretation function* $\cdot^{\mathcal{I}}$, and an *objectification function* $\imath$. The objectification function is an injective function associating a *unique* domain element to each tuple, $\imath : T_\Delta(\mathcal{U}) \to \Delta$,

called in the following *tuple identifier*. Note that, since the range of the objectification function is Δ, tuple identifiers can, in turn, instantiate classes or components of labelled tuples. The interpretation function $\cdot^{\mathcal{I}}$ assigns a domain element to each individual, $o^{\mathcal{I}} \in \Delta$, a set of domain elements to each concept name, $CN^{\mathcal{I}} \subseteq \Delta$, and a set of $\tau(RN)$-labelled tuples over Δ to each relation name RN, $RN^{\mathcal{I}} \subseteq T_{\Delta}(\tau(RN))$. Also note that the semantics does not enforce the *unique name assumption* (UNA) for the individuals (requiring that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ if $a \neq b$), but we can syntactically impose it using the *distinct individuals* axioms in the ABox. The interpretation function $\cdot^{\mathcal{I}}$ is unambiguously extended over concept and relation expressions as specified in Figure 5.3.

As for the semantics of a $\mathcal{DLR}^{+}$ KB, the interpretation $\mathcal{I}$ satisfies the concept inclusion axiom $C_1 \sqsubseteq C_2$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, and the relation inclusion axiom $R_1 \sqsubseteq R_2$ if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$. It satisfies the concept instance axiom $CN(o)$ if $o^{\mathcal{I}} \in CN^{\mathcal{I}}$, the relation instance axiom $RN(U_1{:}o_1, \ldots, U_n{:}o_n)$ if $\langle U_1 \colon o_1^{\mathcal{I}}, \ldots, U_n \colon o_n^{\mathcal{I}} \rangle \in RN^{\mathcal{I}}$, the axiom $o = \langle U_1{:}o_1, \ldots, U_n{:}o_n \rangle$ if $o^{\mathcal{I}} = \iota(\langle U_1{:}o_1^{\mathcal{I}}, \ldots, U_n{:}o_n^{\mathcal{I}} \rangle)$, and the axioms $o_1 = o_2$ and $o_1 \neq o_2$ if $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$, and $o_1^{\mathcal{I}} \neq o_2^{\mathcal{I}}$, respectively. $\mathcal{I}$ is a *model* of the knowledge base $(\mathcal{T}, \mathcal{A})$ if it satisfies all the axioms in the TBox $\mathcal{T}$ and in the ABox $\mathcal{A}$.

In the following we provide an example of the expressivity of $\mathcal{DLR}^{+}$.

Example 5.1.1. Consider the relation names R_1, R_2 with $\tau(R_1) = \{W_1, W_2, W_3, W_4\}$, $\tau(R_2) = \{V_1, V_2, V_3, V_4, V_5\}$, and a knowledge base with the renaming axiom $W_1W_2W_3 \rightleftarrows V_3V_4V_5$ and a TBox $\mathcal{T}_{\mathsf{exa}}$:

$$\pi[W_1, W_2]R_1 \sqsubseteq \pi^{\leqslant 1}[W_1, W_2]R_1 \tag{5.1}$$

$$\pi[V_3, V_4]R_2 \sqsubseteq \pi^{\leqslant 1}[V_3, V_4](\pi[V_3, V_4, V_5]R_2) \tag{5.2}$$

$$\pi[W_1, W_2, W_3]R_1 \sqsubseteq \pi[V_3, V_4, V_5]R_2. \tag{5.3}$$

The axiom equation (5.1) expresses that W_1, W_2 form a multi-attribute key for R_1; equation (5.2) introduces a functional dependency in the relation R_2 where the attribute V_5 is functionally dependent from attributes V_3, V_4, and equation (5.3) states an inclusion between two projections of the relation names R_1, R_2 based on the renaming schema axiom. □

KB satisfiability refers to the problem of deciding the existence of a model of a given knowledge base; *concept satisfiability* (resp. *relation satisfiability*) is the problem of deciding whether there is a model of the knowledge base with a non-empty interpretation of a given concept (resp. relation). A knowledge base *entails* (or *logically implies*) an axiom if all models of the knowledge base are also models of the axiom. All the decision problems in $\mathcal{DLR}^+$ can be all reduced to KB satisfiability, as stated in the following:

Lemma 1. In $\mathcal{DLR}^+$, concept and relation satisfiability and entailment are reducible to KB satisfiability.

5.1.3 Expressiveness

$\mathcal{DLR}^+$ is an expressive description logic able to assert relevant constraints in the context of relational databases, such as *inclusion dependencies* (namely inclusion axioms among arbitrary projections of relations), *equijoins*, *functional dependency* axioms, *key* and *foreign key* axioms, *external uniqueness* axioms, *identification* axioms, and *path functional dependencies*.

An *equijoin* among two relations with disjoint signatures is the set of all combinations of tuples in the relations that are equal on their selected attribute names. Let R_1, R_2 be relations with signatures $\tau(R_1) = \{U, U_1, \ldots, U_{n_1}\}$ and $\tau(R_2) = \{V, V_1, \ldots, V_{n_2}\}$; their equijoin over U and V is the relation $R = R_1 \underset{U=V}{\bowtie} R_2$ with signature $\tau(R) = \tau(R_1) \cup \tau(R_2) \backslash \{V\}$, which is expressed by the $\mathcal{DLR}^+$ axioms:

$$\pi[U, U_1, \ldots, U_{n_1}]R \equiv \sigma_{U:(\exists[U]R_1 \sqcap \exists[V]R_2)} R_1$$
$$\pi[V, V_1, \ldots, V_{n_2}]R \equiv \sigma_{V:(\exists[U]R_1 \sqcap \exists[V]R_2)} R_2$$
$$U \rightleftarrows V \ .$$

A *functional dependency* axiom $(R\!:\!U_1 \ldots U_j \rightarrow U)$ (also called *internal uniqueness* axiom [93]) states that the values of the attributes $U_1 \ldots U_j$ uniquely determine the value of the attribute U in the relation R. Formally, the interpretation $\mathcal{I}$ satisfies this functional dependency axiom if, for all tuples $s, t \in R^{\mathcal{I}}$, $s[U_1] = t[U_1], \ldots, s[U_j] = t[U_j]$ imply $s[U] = t[U]$. Functional dependencies

can be expressed in $\mathcal{DLR}^{+}$, assuming that $\{U_1, \ldots, U_j, U\} \sqsubseteq \tau(R)$, with the axiom:

$$\pi[U_1, \ldots, U_j]R \sqsubseteq \pi^{\leqslant 1}[U_1, \ldots, U_j](\pi[U_1, \ldots, U_j, U]R).$$

A special case of a functional dependency are *key* axioms ($R : U_1 \ldots U_j \rightarrow R$), which state that the values of the key attributes $U_1 \ldots U_j$ of a relation R uniquely identify tuples in R. A key axiom can be expressed in $\mathcal{DLR}^{+}$, assuming that $\{U_1 \ldots U_j\} \sqsubseteq \tau(R)$, with the axiom:

$$\pi[U_1, \ldots, U_j]R \sqsubseteq \pi^{\leqslant 1}[U_1, \ldots, U_j]R.$$

A *foreign key* is the obvious result of an inclusion dependency together with a key constraint involving the foreign key attributes.

The *external uniqueness* axiom ($[U^1]R_1 \downarrow \ldots \downarrow [U^h]R_h$) states that the join R of the relations $R_1, \ldots, R_h$ via the attributes $U^1, \ldots, U^h$ has the joined attribute functionally dependent on all the others [93]. This can be expressed in $\mathcal{DLR}^{+}$ with the axioms:

$$R \equiv R_1 \underset{U^1=U^2}{\bowtie} \cdots \underset{U^{h-1}=U^h}{\bowtie} R_h$$
$$R : U_1^1, \ldots, U_{n_1}^1, \ldots, U_1^h, \ldots, U_{n_h}^h \rightarrow U^1$$

where $\tau(R_i) = \{U^i, U_1^i, \ldots, U_{n_i}^i\}, 1 \leqslant i \leqslant h$, and R is a new relation name with $\tau(R) = \{U^1, U_1^1, \ldots, U_{n_1}^1, \ldots, U_1^h, \ldots, U_{n_h}^h\}$.

Identification axioms as defined in $\mathcal{DLR}_{ifd}$ [37] (an extension of $\mathcal{DLR}$ with functional dependencies and identification axioms) are a variant of external uniqueness axioms, constraining only the elements of a concept C; they can be expressed in $\mathcal{DLR}^{+}$ with the axiom:

$$[U^1]\sigma_{U_1:C}R_1 \downarrow \ldots \downarrow [U^h]\sigma_{U_h:C}R_h.$$

Path functional dependencies—as defined in the description logics family $\mathcal{CFD}$ [132]—can be expressed in $\mathcal{DLR}^{+}$ as identification axioms involving joined sequences of functional binary relations. $\mathcal{DLR}^{+}$ also captures the *tree-based identification constraints (**tid**)* introduced in [39] to express functional dependencies in *DL-Lite*$_{RDFS,tid}$.

The DL $\mathcal{DLR}_{ifd}$ [37] extends $\mathcal{DLR}$ with functional dependencies and identification axioms, and is therefore included in $\mathcal{DLR}^+$. $\mathcal{DLR}^+$ can express complex inclusion and functional dependencies, for which it is well known that reasoning is undecidable [42].

The rich set of constructors in $\mathcal{DLR}^+$ allows us to extend the known mappings in description logics of popular conceptual database models, and to provide the foundations for their reasoning tasks. The EER mapping as introduced in [6] can be extended to deal with multi-attribute keys (by using identification axioms) and named roles in relations; the ORM mapping as introduced in [62,127] can be extended to deal with arbitrary subset and exclusive relation constructs (by using inclusions among global objectifications of projections of relations), arbitrary internal and external uniqueness constraints, arbitrary frequency constraints (by using projections), local objectification, named roles in relations, and fact type readings (by using renaming axioms); the UML mapping as introduced in [26] can be fixed to deal properly with association classes (by using local objectification) and named roles in associations.

Aside from conceptual modelling, $\mathcal{DLR}^+$ could be studied in relation to other tasks relevant for database scenarios, such as query answering [40], constraint checking with respect to a partially closed world (i.e., with DBoxes [118]), inconsistent database repairing, etc. In this paper, we focus just on the basic consistency and entailment reasoning tasks.

5.1.4 The $\mathcal{DLR}^{\pm}$ decidable fragment

Since a $\mathcal{DLR}^+$ knowledge base can express inclusions and functional dependencies, the entailment problem is undecidable [42]. Thus, in this section we present $\mathcal{DLR}^{\pm}$, a decidable syntactic fragment of $\mathcal{DLR}^+$ limiting the coexistence of relation projections in a knowledge base.

Given a $\mathcal{DLR}^+$ knowledge base $\mathcal{KB} = (\mathcal{T}, \mathcal{A}, \Re)$, we define the *projection signature of* $\mathcal{KB}$ as the set $\mathscr{T}$ containing the signatures $\tau(RN)$ of all relations $RN \in \mathcal{R}$, the singleton sets associated with each attribute name $U \in \mathcal{U}$, and the relation signatures that appear explicitly in projection constructs in some axiom from $\mathcal{T}$, together with their implicit occurrences due to the renaming

schema. Formally, $\mathscr{T}$ is the smallest set such that (i) $\tau(RN) \in \mathscr{T}$ for all $RN \in \mathcal{R}$; (ii) $\{U\} \in \mathscr{T}$ for all $U \in \mathcal{U}$; and (iii) $\{U_1, \ldots, U_k\} \in \mathscr{T}$ for all $\pi^{\leqslant q}[V_1, \ldots, V_k]R$ appearing as sub-formulas in $\mathcal{T}$ and $V_i \in [U_i]_{\Re}$ for $1 \leqslant i \leqslant k$.

The *projection signature graph* of $\mathcal{KB}$ is the directed acyclic graph corresponding to the Hasse diagram of $\mathscr{T}$ ordered by the proper subset relation $\supset$, whose sinks are the attribute singletons $\{U\}$. We call this graph $(\supset, \mathscr{T})$. Given a set of attributes $\tau = \{U_1, \ldots, U_k\} \subseteq \mathcal{U}$, the *projection signature graph dominated by* τ, denoted as $\mathscr{T}_\tau$, is the sub-graph of $(\supset, \mathscr{T})$ with τ as root and containing all the nodes reachable from τ. Given two sets of attributes $\tau_1, \tau_2 \subseteq \mathcal{U}$, $\textsc{path}_{\mathscr{T}}(\tau_1, \tau_2)$ denotes the set of paths in $(\supset, \mathscr{T})$ between τ_1 and τ_2. Note that, $\textsc{path}_{\mathscr{T}}(\tau_1, \tau_2) = \varnothing$ both when a path does not exist and when $\tau_1 \subseteq \tau_2$. The notation $\textsc{child}_{\mathscr{T}}(\tau_1, \tau_2)$ means that τ_2 is a child (i.e., a direct descendant) of τ_1 in $(\supset, \mathscr{T})$. We now introduce $\mathcal{DLR}^{\pm}$ as follows.

Definition 2. A $\mathcal{DLR}^{\pm}$ *knowledge base* is a $\mathcal{DLR}^{+}$ knowledge base that satisfies the following syntactic conditions:

1. the projection signature graph $(\supset, \mathscr{T})$ is a multitree: i.e., for every node $\tau \in \mathscr{T}$, the graph $\mathscr{T}_\tau$ is a tree; and
2. for every projection construct $\pi^{\leqslant q}[U_1, \ldots, U_k]R$ and every concept expression of the form $\exists^{\geqslant q}[U_1]R$ appearing in $\mathcal{T}$, if $q > 1$ then the length of the path $\textsc{path}_{\mathscr{T}}(\tau(R), \{U_1, \ldots, U_k\})$ is 1.

The first condition in $\mathcal{DLR}^{\pm}$ restricts $\mathcal{DLR}^{+}$ in the way that multiple projections of relations may appear in a knowledge base: intuitively, there cannot be different projections sharing a common attribute. Moreover, observe that in $\mathcal{DLR}^{\pm}$ $\textsc{path}_{\mathscr{T}}$ is necessarily functional, due to the multitree restriction. By relaxing the first condition the language becomes undecidable, as we mentioned at the beginning of this Section. The second condition is also necessary in our proof of decidability of $\mathcal{DLR}^{\pm}$; however, we do not know whether this condition could be relaxed while preserving decidability.

Figure 5.4 shows that the projection signature graph of the knowledge base from Example 5.1.1 is indeed a multitree. Note that in the figure we have collapsed equivalent attributes in a unique equivalence class, according to the

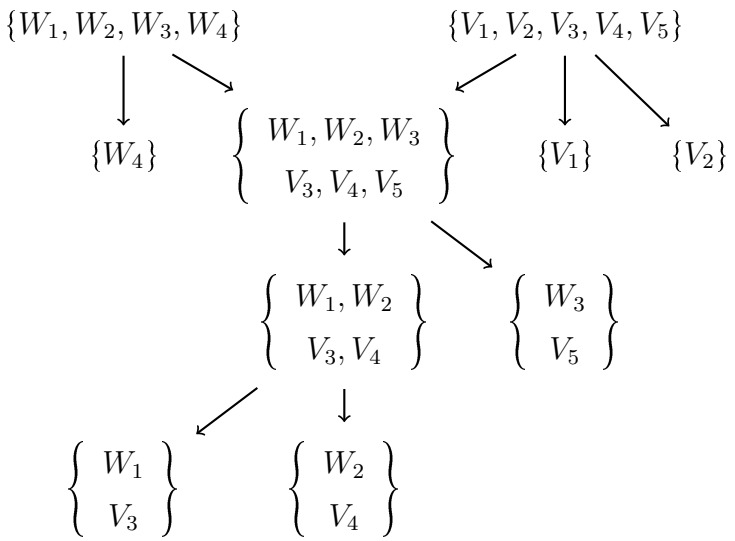


Figure 5.4: The projection signature graph of Example 5.1.1.

renaming schema. Furthermore, since all its projection constructs have $q = 1$, this knowledge base belongs to $\mathcal{DLR}^{\pm}$.

$\mathcal{DLR}$ is included in $\mathcal{DLR}^{\pm}$, since the projection signature graph of any $\mathcal{DLR}$ knowledge base is always a degenerate multitree with maximum depth equal to 1. Not all the database constraints as introduced in Section 5.1.3 can be directly expressed in $\mathcal{DLR}^{\pm}$. While functional dependency and key axioms can be expressed directly in $\mathcal{DLR}^{\pm}$, equijoins, external uniqueness axioms, and identification axioms introduce projections of a relation which share common attributes, thus violating the multitree restriction. For example, the axioms for capturing an equijoin between two relations, R_1, R_2 would generate a projection signature graph with the signatures of R_1, R_2 as projections of the signature of the join relation R sharing the attribute on which the join is performed, thus violating condition 1.

However, in $\mathcal{DLR}^{\pm}$ it is still possible to reason over both external uniqueness and identification axioms by encoding them into a set of saturated ABoxes (as originally proposed in [37]) and check whether there is a saturation that satisfies the constraints. Therefore, we can conclude that $\mathcal{DLR}_{ifd}$ extended with unary functional dependencies is included in $\mathcal{DLR}^{\pm}$, provided that

projections of relations in the knowledge base form a multitree projection signature graph. Since (unary) functional dependencies are expressed via the inclusions of projections of relations, by constraining the projection signature graph to be a multitree, the possibility to build combinations of functional dependencies as the ones in [37] leading to undecidability is ruled out.

Note that the *non-conflicting keys* sufficient condition guaranteeing the decidability of inclusion dependencies and keys of [109] is in conflict with our more restrictive requirement: indeed [109] allow for overlapping projections, but the considered datalog language is not comparable to $\mathcal{DLR}^{+}$. In general, description logic based languages, such as $\mathcal{DLR}^{+}$, and datalog based languages, such as the language proposed in [109], are incomparable in terms of expressiveness, due to the inability of description logics to distinguish tree and non-tree models in the TBox. Note that, unlike the typical restrictions of datalog-like languages, there is no problem in stating arbitrary cyclic dependencies in relation inclusion axioms involving projections on the left and right hand sides.

Concerning the ability of $\mathcal{DLR}^{\pm}$ to capture conceptual data models, only the mapping of ORM schemas is affected by the $\mathcal{DLR}^{\pm}$ restrictions: $\mathcal{DLR}^{\pm}$ is able to correctly express an ORM schema if the projections involved in the schema satisfy the $\mathcal{DLR}^{\pm}$ multitree restriction.

5.1.5 Mapping $\mathcal{DLR}^{\pm}$ to $\mathcal{ALCQI}$

This section shows that reasoning in $\mathcal{DLR}^{\pm}$ is an ExpTime-complete problem. The lower bound is clear by observing that $\mathcal{ALC}$ is a sublanguage of $\mathcal{DLR}^{\pm}$ [36]. More challenging is the upper bound obtained by providing a mapping from $\mathcal{DLR}^{\pm}$ knowledge bases to $\mathcal{ALCQI}$ knowledge bases. $\mathcal{ALCQI}$ is a Description Logics which extends $\mathcal{ALC}$ with qualified number restrictions $\exists^{\geqslant q} R.\, C$, and inverse roles R^{-} (see [16] for more details).

We briefly recall the syntax of $\mathcal{ALCQI}$ as shown in Fig. 5.5:

where A stands for atomic concepts and R for atomic roles. We adapt and extend the mapping presented for $\mathcal{DLR}$ in [40], with the modifications proposed by [97] to deal with ABoxes without the unique name assumption.

$$
\begin{array}{rcl}
C & \rightarrow & \bot \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C \mid \forall R.C \mid \leqslant n R.C \mid \geqslant n R.C \\
R & \rightarrow & P \mid P^-
\end{array}
$$

Figure 5.5: $\mathcal{ALCQI}$ syntax.

$$
\begin{array}{rcl}
(\neg C)^\dagger & = & \neg C^\dagger \\
(C_1 \sqcap C_2)^\dagger & = & C_1^\dagger \sqcap C_2^\dagger \\
(\exists^{\geqslant q}[U_i]R)^\dagger & = & \begin{cases} \exists^{\geqslant q}\big(\textsc{path}_{\mathscr{T}}(\tau(R), \{U_i\})^\dagger\big)^- . R^\dagger, & \text{if } \textsc{path}_{\mathscr{T}}(\tau(R), \{U_i\}) \neq \varnothing \\ \bot, & \text{otherwise} \end{cases} \\
(\circledcirc R)^\dagger & = & R^\dagger \\
(\odot RN)^\dagger & = & A^l_{RN} \\
 & & \\
(R_1 \backslash R_2)^\dagger & = & R_1^\dagger \sqcap \neg R_2^\dagger \\
(R_1 \sqcap R_2)^\dagger & = & R_1^\dagger \sqcap R_2^\dagger \\
(R_1 \sqcup R_2)^\dagger & = & \begin{cases} R_1^\dagger \sqcup R_2^\dagger, & \text{if } \tau(R_1) = \tau(R_2) \\ \bot, & \text{otherwise} \end{cases} \\
(\sigma_{U_i:C} R)^\dagger & = & \begin{cases} R^\dagger \sqcap \forall \textsc{path}_{\mathscr{T}}(\tau(R), \{U_i\})^\dagger . C^\dagger, & \text{if } \textsc{path}_{\mathscr{T}}(\tau(R), \{U_i\}) \neq \varnothing \\ \bot, & \text{otherwise} \end{cases} \\
(\pi^{\lesseqgtr q}[U_1, \ldots, U_k]R)^\dagger & = & \begin{cases} \exists^{\geqslant 1, \lesseqgtr q}\left(\textsc{path}_{\mathscr{T}}(\tau(R), \{U_1, \ldots, U_k\})^\dagger\right)^- . R^\dagger, \\ \qquad\qquad \text{if } \textsc{path}_{\mathscr{T}}(\tau(R), \{U_1, \ldots, U_k\}) \neq \varnothing \\ \bot, \qquad\qquad \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.6: The mapping to $\mathcal{ALCQI}$ for concept and relation expressions.

We recall that the renaming schema $\Re$ does not play any role since we assumed that a $\mathcal{DLR}^\pm$ knowledge base is rewritten by choosing a single canonical representative for each equivalence class of attributes induced by $\Re$. Thus, we consider $\mathcal{DLR}^\pm$ knowledge bases as pairs of TBox and ABox axioms.

We first introduce a mapping function $\cdot^\dagger$ from $\mathcal{DLR}^\pm$ concepts and relations to $\mathcal{ALCQI}$ concepts. The function $\cdot^\dagger$ maps each concept name *CN* and each relation name *RN* appearing in the $\mathcal{DLR}^\pm$ KB to the $\mathcal{ALCQI}$ concept names *CN* and A_{RN}, respectively. The latter can be informally understood as the "global" reification of *RN*. For each relation name *RN*, the $\mathcal{ALCQI}$

signature also includes a concept name A^{l}_{RN} and a role name Q_{RN} to capture local objectification. The mapping $\cdot^{\dagger}$ is extended to concept and relation expressions as illustrated in Figure 5.6, where the notation $\exists^{\geqslant 1, \leqslant q} R.\, C$ is a shortcut for the conjunction $\exists R.\, C \sqcap \exists^{\leqslant q} R.\, C$.

The mapping crucially uses the projection signature graph to map projections and selections, by accessing paths in the projection signature graph $(\supset, \mathscr{T})$ associated to the $\mathcal{DLR}^{\pm}$ KB. If there is a path $\text{PATH}_{\mathscr{T}}(\tau, \tau') = \tau, \tau_1, \ldots, \tau_n, \tau'$ from τ to τ' in $\mathscr{T}$, then the $\mathcal{ALCQI}$ signature contains role names $Q_{\tau'}, Q_{\tau_i}$, for $i = 1, \ldots, n$, and the following role chain expression is generated by the mapping:

$$\text{PATH}_{\mathscr{T}}(\tau, \tau')^{\dagger} = Q_{\tau_1} \circ \ldots \circ Q_{\tau_n} \circ Q_{\tau'},$$

In particular, the mapping uses the following notation: the inverse role chain $(R_1 \circ \ldots \circ R_n)^{-}$, for R_i a role name, stands for the chain $R_n^{-} \circ \ldots \circ R_1^{-}$, with R_i^{-} an inverse role, the expression $\exists^{\leqslant 1} R_1 \circ \ldots \circ R_n.\, C$ stands for the $\mathcal{ALCQI}$ concept expression $\exists^{\leqslant 1} R_1. \ldots . \exists^{\leqslant 1} R_n.\, C$ and $\forall R_1 \circ \ldots \circ R_n.\, C$ for the $\mathcal{ALCQI}$ concept expression $\forall R_1. \ldots . \forall R_n.\, C$. Thus, since $\mathcal{DLR}^{\pm}$ restricts to $q = 1$ the cardinalities on any path of length strictly greater than 1 (see condition 2 in Def. 2), the above notation shows that we remain within the $\mathcal{ALCQI}$ syntax when the mapping applies to cardinalities. If, e.g., we need to map the $\mathcal{DLR}^{\pm}$ cardinality constraint $\exists^{\leqslant q}[U_i]R$ with $q > 1$, then, to stay within the $\mathcal{ALCQI}$ syntax, U_i must not be mentioned in any other projection in such a way that $\text{PATH}_{\mathscr{T}}(\tau(R), \{U_i\}) = 1$. Finally, notice that the mapping introduces a concept name $A^{\tau_i}_{RN}$ for each projected signature τ_i in the projection signature graph dominated by $\tau(RN)$, i.e., $\tau_i \in \mathscr{T}_{\tau(RN)}$, informally to capture the global reifications of the various projections of RN in the given KB. We also use the shortcut A_{RN} which stands for $A^{\tau(RN)}_{RN}$.

Intuitively, each node in the projection signature graph associated to a $\mathcal{DLR}^{\pm}$ KB denotes a relation projection and the mapping reifies each of these projections. The target $\mathcal{ALCQI}$ signature resulting from mapping the $\mathcal{DLR}^{\pm}$ KB of Example 5.1.1 is partially presented in Fig. 5.7, together with the projection signature graph (showed in Fig. 5.4). Each node of the graph is labelled with the corresponding global reification concept $(A^{\tau_j}_{R_i})$, for each $R_i \in \mathcal{R}$ and each projected signature τ_j in the projection signature graph

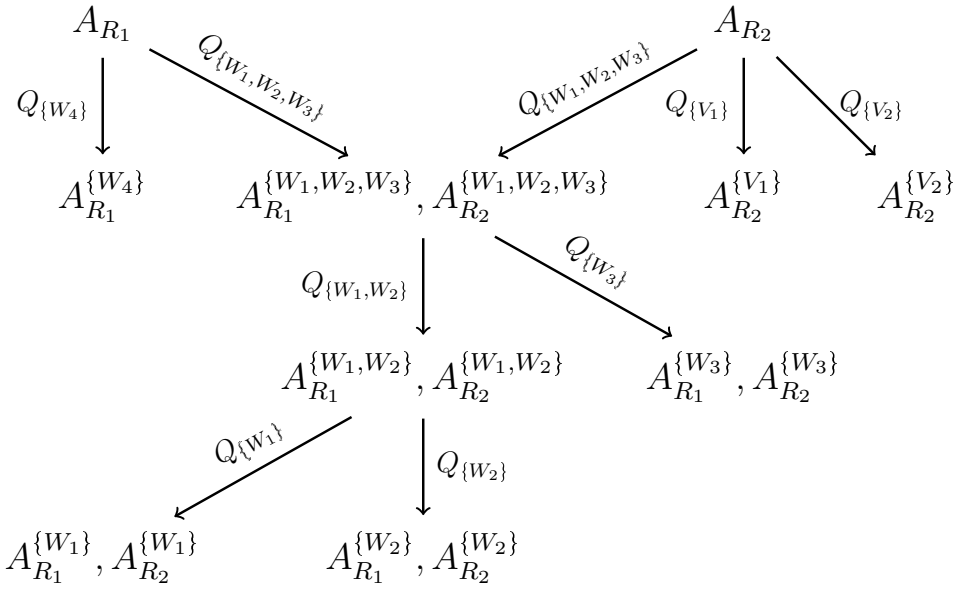


Figure 5.7: The $\mathcal{ALCQI}$ signature generated by $\mathcal{T}_{\mathsf{exa}}$.

dominated by $\tau(R_i)$, while the edges are labelled by the roles (Q_{τ_i}) needed for the reification.

To better clarify the need for the path function in the mapping, notice that each $\mathcal{DLR}^{\pm}$ relation is reified according to the decomposition dictated by the projection signature graph it dominates. Thus, to access, e.g., an attribute U_j of a $\mathcal{DLR}^{\pm}$ relation R_i it is necessary to follow the path through the projections that use the attribute. Such a path, from the node denoting the whole signature of the relation, $\tau(R_i)$, to the node denoting the attribute U_j is returned by the $\textsc{path}_{\mathscr{T}}(\tau(R_i), U_j)$ function. For example, considering the example in Fig. 5.7, to access the attribute W_1 of the relation R_2 in the expression $(\sigma_{W_1:C}R_2)$, the mapping of the path $\textsc{path}_{\mathscr{T}}(\tau(R_2), \{W_1\})^{\dagger}$ is equal to the role chain $Q_{\{W_1,W_2,W_3\}} \circ Q_{\{W_1,W_2\}} \circ Q_{\{W_1\}}$, so that $(\sigma_{W_1:C}R_2)^{\dagger} = A_{R_2} \sqcap \forall Q_{\{W_1,W_2,W_3\}}.\, \forall Q_{\{W_1,W_2\}}.\, \forall Q_{\{W_1\}}.\, C$. Similar considerations can be done when mapping cardinalities over relation projections.

We now present in details the mapping of a $\mathcal{DLR}^{\pm}$ KB into a KB in $\mathcal{ALCQI}$. Let $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ be a $\mathcal{DLR}^{\pm}$ KB with signature $(\mathcal{C}, \mathcal{R}, \mathcal{O}, \mathcal{U}, \tau)$. The mapping $\gamma(\mathcal{KB})$ is assumed to be unsatisfiable (i.e., it contains the axiom $\top \sqsubseteq \bot$) if

the ABox contains a relation assertion $RN(t)$ with $\tau(RN) \neq \tau(t)$, for some relation $RN \in \mathcal{R}$ and some tuple t. Otherwise, $\gamma(\mathcal{KB}) = (\gamma(\mathcal{T}), \gamma(\mathcal{A}))$ defines an $\mathcal{ALCQI}$ KB as follows:

$$\begin{aligned}
\gamma(\mathcal{T}) = \gamma_{dsj} \;\cup \bigcup_{RN \in \mathcal{R}} \gamma_{rel}(RN) \;\cup \bigcup_{RN \in \mathcal{R}} \gamma_{lobj}(RN) \;\cup \\
\bigcup_{C_1 \sqsubseteq C_2 \in \mathcal{KB}} C_1^{\dagger} \sqsubseteq C_2^{\dagger} \;\cup \bigcup_{R_1 \sqsubseteq R_2 \in \mathcal{KB}} R_1^{\dagger} \sqsubseteq R_2^{\dagger} \\
\gamma_{dsj} = \; \{ A^{\tau_i}_{RN_1} \sqsubseteq \neg A^{\tau_j}_{RN_2} \mid RN_1, RN_2 \in \mathcal{R} \} \\
\gamma_{rel}(RN) = \bigcup_{\tau_i \in \mathscr{T}_{\tau(RN)}} \bigcup_{\text{CHILD}_{\mathscr{T}}(\tau_i, \tau_j)} \{ A^{\tau_i}_{RN} \sqsubseteq \exists Q_{\tau_j}.\, A^{\tau_j}_{RN},\; \exists^{\geqslant 2} Q_{\tau_j}.\, \top \sqsubseteq \bot \} \\
\gamma_{lobj}(RN) = \{ A_{RN} \sqsubseteq \exists Q_{RN}.\, A^{l}_{RN},\; \exists^{\geqslant 2} Q_{RN}.\, \top \sqsubseteq \bot, \\
A^{l}_{RN} \sqsubseteq \exists Q^{-}_{RN}.\, A_{RN},\; \exists^{\geqslant 2} Q^{-}_{RN}.\, \top \sqsubseteq \bot \}.
\end{aligned}$$

Intuitively, γ_{dsj} ensures that relations with different signatures are disjoint, thus, e.g., enforcing the union compatibility. The axioms in γ_{rel} introduce classical reification axioms for each relation and its relevant projections. The axioms in γ_{lobj} make sure that local objectifications are correctly captured wrt the global ones since each role Q_{RN} defines a bijection.

To translate the ABox, we first map each individual $o \in \mathcal{O}$ in the $\mathcal{DLR}^{\pm}$ ABox $\mathcal{A}$ to an $\mathcal{ALCQI}$ individual o. Each tuple in relation instance axioms occurring in $\mathcal{A}$ is mapped via an injective function ξ to a distinct individual. That is, $\xi : T_{\mathcal{O}}(\mathcal{U}) \rightarrow \mathcal{O}_{\mathcal{ALCQI}}$, with $\mathcal{O}_{\mathcal{ALCQI}} = \mathcal{O} \cup \mathcal{O}^t$ being the set of individual names in $\gamma(\mathcal{KB})$, $\mathcal{O} \cap \mathcal{O}^t = \varnothing$ and

$$\xi(t) \;=\; \begin{cases} o \in \mathcal{O}, & \text{if } t = \langle U{:}o \rangle \\ o \in \mathcal{O}^t, & \text{otherwise.} \end{cases}$$

Following [97], the mapping $\gamma(\mathcal{A})$ in Fig. 5.8 introduces a new concept name Q_o for each individual $o \in \mathcal{O}$ and a new concept name Q_t for each relation instance t occurring in $\mathcal{A}$, with each Q_t restricted as follows:

$$Q_t \sqsubseteq \exists^{\leqslant 1}\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_1\})^\dagger\big)^- .\\ \exists\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_2\})^\dagger\big).Q_{o_2} \sqcap \ldots \sqcap \exists\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_n\})^\dagger\big).Q_{o_n}$$

Intuitively, equation (5.6) and equation (5.7) reify each relation instance axiom occurring in $\mathcal{A}$ using the projection signature of the involved tuple itself. The formulas equation (5.8)-equation (5.9) together with the axioms for concepts Q_t guarantee that there is exactly one $\mathcal{ALCQI}$ individual reifying a given tuple in a relation instance axiom. Clearly, the size of $\gamma(\mathcal{KB})$ is polynomial in the size of $\mathcal{KB}$ under the same coding of the numerical parameters.

We are now able to state our main technical result.

Theorem 3. A $\mathcal{DLR}^{\pm}$ knowledge base $\mathcal{KB}$ is satisfiable iff the $\mathcal{ALCQI}$ knowledge base $\gamma(\mathcal{KB})$ is satisfiable.

Proof (in [12, p. 13]).

As a direct consequence of this theorem and the fact that $\mathcal{DLR}$ is a sublanguage of $\mathcal{DLR}^{\pm}$, we obtain the following corollary.

Corollary 4. Reasoning in $\mathcal{DLR}^{\pm}$ is ExpTime-complete.

$$\begin{aligned}
\gamma(\mathcal{A}) = &\{CN^\dagger(o) \mid CN(o) \in \mathcal{A}\} \cup && (5.4)\\
&\{o_1 \neq o_2 \mid o_1 \neq o_2 \in \mathcal{A}\} \cup \{o_1 = o_2 \mid o_1 = o_2 \in \mathcal{A}\} \cup && (5.5)\\
&\{A^{\tau_i}_{RN}(\xi(t[\tau_i])) \mid RN(t) \in \mathcal{A} \text{ and } \tau_i \in \mathscr{T}_{\tau(RN)}\} \cup && (5.6)\\
&\{Q_{\tau_j}\big(\xi(t[\tau_i]), \xi(t[\tau_j])\big) \mid RN(t) \in \mathcal{A}, \tau_i \in \mathscr{T}_{\tau(RN)} \text{ and } \text{CHILD}_{\mathscr{J}}(\tau_i, \tau_j)\} \cup && (5.7)\\
&\{Q_o(o) \mid o \in \mathcal{O}\} \cup && (5.8)\\
&\{Q_t(o_1) \mid t = \langle U_1{:}o_1, \ldots, U_n{:}o_n\rangle \text{ occurs in } \mathcal{A}\} \cup && (5.9)\\
&\{Q_t \sqsubseteq \exists^{\leqslant 1}\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_1\})^\dagger\big)^- . && (5.10)\\
&\exists\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_2\})^\dagger\big).Q_{o_2} \sqcap \ldots \sqcap \exists\big(\text{PATH}_{\mathscr{J}}(\tau(t), \{U_n\})^\dagger\big).Q_{o_n}\} && (5.11)
\end{aligned}$$

Figure 5.8: The mapping $\gamma(\mathcal{A})$

5.2 ORM encoding in $\mathcal{DLR}^{\pm}$

In this section we provide the encoding of an ORM fragment called $\mathcal{ORM}^{\pm}$, into $\mathcal{DLR}^{\pm}$, the decidable fragment of $\mathcal{DLR}^{+}$.

The mapping we present is based on the $\mathcal{DLR}^{+}$ restriction involving the concept of equivalence classes, as seen in Section 5.1.1. In an ORM diagram, each role is renamed in order to obtain a unique canonical representative for each equivalence class. So, in the $\mathcal{ORM}^{\pm}$ fragment we consider only those ORM diagrams where each attribute within the same relation signature has a distinct equivalence class. This operation is done by a pre-processing procedure that verifies if the ORM schema does not come with equivalence classes containing attributes from the same relation signature. After this procedure, the renaming schema does not play any role in the knowledge base.

Thus, we only consider ORM schemas rewritable in a $\mathcal{DLR}^{\pm}$ knowledge base with a single canonical representative for each equivalence class of attributes; if the pre-processing procedure fails because the constraint is violated, the ORM schema will not be encoded in $\mathcal{DLR}^{\pm}$.

As a consequence of this restriction and from Definition 2, it follows that not all ORM constraints can be expressed in $\mathcal{DLR}^{\pm}$ since this language limits the coexistence of relation projections in a $\mathcal{DLR}^{\pm}$ knowledge base, in this way the projections can not share common attributes. Thus, ORM constraints with arbitrary join paths are excluded from the mapping because they lead to the undecidability. For this reason, External Id, External Unique and Identification are not expressible in $\mathcal{DLR}^{\pm}$. Other ORM constraints excluded from the mapping are Type Cardinality, Role Cardinality, Values Of and Ring because they are not known in $\mathcal{DLR}^{\pm}$.

Table 5.1: $\mathcal{ORM}^{\pm}$ encoding in $\mathcal{DLR}^{\pm}$

FactType(P T_1 ... $\mathrm{T}_{\alpha(\mathrm{P})}$))		P does not appear as an AlternatePredicate
$\mathcal{DLR}^{\pm}$	$P \sqsubseteq (\sigma_{P.1:\mathrm{T}_1} P) \sqcap \cdots \sqcap (\sigma_{P.\alpha(P):\mathrm{T}_{\alpha(P)}} P)$	

Mandatory(T P_1.i_1 ... P_m.i_m)	for j≠k and j,k⩽m: $P_j \neq P_k$
$\mathcal{DLR}^{\pm}$ $\quad T \sqsubseteq \exists[P_1.i_1]P_1 \sqcup \cdots \sqcup \exists[P_m.i_m]P_m$	
Frequency(P.i_1 ... P.i_m $\underline{F}$)	for j≠k and j,k⩽m: $i_j \neq i_k$. p,q⩾1 (1) $\underline{F} \equiv$ (p..) (2) $\underline{F} \equiv$ (..q) (3) $\underline{F} \equiv$ (p..q) (4) $\underline{F} \equiv$ (p) (q=p)
$\mathcal{DLR}^{\pm}$ $\quad \pi[P_{\cdot i_1} \ldots P_{\cdot i_m}]P \sqsubseteq \pi^{(\geqslant p, \leqslant q)}[P_{\cdot i_1} \ldots P_{\cdot i_m}]P$	
Subtype((T$_1$ ... T$_m$) T)	
$\mathcal{DLR}^{\pm}$ $\quad T_1 \sqsubseteq T, \ldots, T_m \sqsubseteq T$	
ExclusiveSubtypes((T$_1$ ... T$_m$) T)	
$\mathcal{DLR}^{\pm}$ $\quad T_1 \sqsubseteq T \sqcap \neg T_2 \sqcap \cdots \sqcap \neg T_m, \ldots, T_{m-1} \sqsubseteq T \sqcap \neg T_m, T_m \sqsubseteq T$	
ExhaustiveSubtypes((T$_1$ ... T$_m$) T)	
$\mathcal{DLR}^{\pm}$ $\quad T \sqsubseteq T_1 \sqcup \cdots \sqcup T_m$	
Subset((P$_1$.i_1 P$_2$.h_1) ... (P$_1$.i_m P$_2$.h_m))	$P_1 \neq P_2$ and for j≠k and j,k⩽m **:** $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$
$\mathcal{DLR}^{\pm}$	
$\pi[P_{1\cdot i_1} \ldots P_{1\cdot i_m}]P_1 \sqsubseteq \pi[P_{2\cdot h_1} \ldots P_{2\cdot h_m}]P_2$	
with $P_1.i_1 \ldots P_1.i_m \rightleftarrows P_2.h_1 \ldots P_2.h_m$	if m < α(P$_1$) and m < α(P$_2$)
$P_1 \sqsubseteq \pi[P_{2\cdot h_1} \ldots P_{2\cdot h_m}]P_2$	
with $P_1.i_1 \ldots P_1.i_m \rightleftarrows P_2.h_1 \ldots P_2.h_m$	if m = α(P$_1$) and m < α(P$_2$)
$\pi[P_{1\cdot i_1} \ldots P_{1\cdot i_m}]P_1 \sqsubseteq P_2$	
with $P_1.i_1 \ldots P_1.i_m \rightleftarrows P_2.h_1 \ldots P_2.h_m$	if m < α(P$_1$) and m = α(P$_2$)
$P_1 \sqsubseteq P_2$	
with $P_1.i_1 \ldots P_1.i_m \rightleftarrows P_2.h_1 \ldots P_2.h_m$	if m = α(P$_1$) and m = α(P$_2$)

Exclusive((P$_1$.i$_1$ P$_2$.h$_1$) ... (P$_1$.i$_m$ P$_2$.h$_m$))	P$_1$≠P$_2$ and for j≠k and j,k⩽m **:** P$_1$.i$_j$ ≠ P$_1$.i$_k$ and P$_2$.h$_j$ ≠ P$_2$.h$_k$
$\mathcal{DLR}^{\pm}$	
$\circledcirc\pi[P_{1\cdot i_1} \dots P_{1\cdot i_m}]P_1 \sqsubseteq \neg \circledcirc \pi[P_{2\cdot h_1} \dots P_{2\cdot h_m}]P_2$	
with $P_1.i_1 \dots P_1.i_m \rightleftarrows P_2.h_1 \dots P_2.h_m$	if m < α(P$_1$) and m < α(P$_2$)
$\circledcirc P_1 \sqsubseteq \neg \circledcirc \pi[P_{2\cdot h_1} \dots P_{2\cdot h_m}]P_2$	
with $P_1.i_1 \dots P_1.i_m \rightleftarrows P_2.h_1 \dots P_2.h_m$	if m = α(P$_1$) and m < α(P$_2$)
$\circledcirc\pi[P_{1\cdot i_1} \dots P_{1\cdot i_m}]P_1 \sqsubseteq \neg \circledcirc P_2$	
with $P_1.i_1 \dots P_1.i_m \rightleftarrows P_2.h_1 \dots P_2.h_m$	if m < α(P$_1$) and m = α(P$_2$)
$\circledcirc P_1 \sqsubseteq \neg \circledcirc P_2$	
with $P_1.i_1 \dots P_1.i_m \rightleftarrows P_2.h_1 \dots P_2.h_m$	if m = α(P$_1$) and m = α(P$_2$)
Objectifies(T P)	
$\mathcal{DLR}^{\pm}$ $\quad \mathrm{T} \equiv \circledcirc P$	

To prove the correctness of the $\mathcal{ORM}^{\pm}$ mapping wrt FOL, we show the equivalence between the ORM constraints expressed in FOL and the same ORM constraints expressed in $\mathcal{DLR}^{\pm}$.

Lemma 5. $\mathcal{DLR}^{+}$ constraints expressed in set semantics as in Figure 5.3 can also be expressed in FOL as shown in Figure 5.9.
Note that we translate only the constraints which are relevant for the encoding.

$$
\begin{aligned}
(C)^f &= \lambda x.C^f(x) \\
(\neg C)^f &= \lambda x.\neg C^f(x) \\
(C_1 \sqcap C_2)^f &= \lambda x.C_1^f(x) \wedge C_2^f(x) \\
(\exists^{\leqslant q}[U_i]R)^f &= \lambda x.\exists^{\leqslant q}\overline{y}.R^f(\overline{y}) \wedge \mathrm{y_i} = x \\
(\bigodot RN)^f &= \lambda x.\exists\overline{y}.RN^f(\overline{y}) \wedge l_{\mathrm{RN}}(\overline{y}) = x \\
(\sigma_{U_i:C}R)^f &= \lambda\overline{y}.R^f(\overline{y}) \wedge C^f(y_{\mathrm{i}}) \\
(\pi^{\leqslant q}[U_1,\ldots,U_k]R)^f &= \lambda y_1 \ldots y_k.\exists^{\leqslant q}\overline{z}.R^f(\overline{z}) \wedge z_1 = y_1 \wedge \cdots \wedge z_k = y_k
\end{aligned}
$$

Figure 5.9: The semantics of $\mathcal{DLR}^{+}$ in FOL.

We define the mapping function ^{f} which translates $\mathcal{DLR}^{\pm}$ concepts and relations to first-order logic formulas extended with lambda expressions. The function ^{f} maps each concept name C and each relation name R appearing in the $\mathcal{DLR}^{\pm}$ KB to FOL formulas. The translation is defined as the set of $\mathcal{DLR}^{\pm}$ inclusion axioms defined in Table 5.1, which are in the form of $R_1 \sqsubseteq R_2$ for relations and in the form of $C_1 \sqsubseteq C_2$ for concepts. These axioms are mapped in FOL as follows:

$$(C_1 \sqsubseteq C_2)^f = \forall C_1^f(x) \rightarrow C_2^f(x) \tag{5.12}$$

$$(R_1 \sqsubseteq R_2)^f = \forall\overline{y}.R_1^f(\overline{y}) \rightarrow R_2^f(\overline{y}) \tag{5.13}$$

Figure 5.10: $\mathcal{DLR}^{+}$ inclusion axioms in FOL

We now prove the equivalence between the ORM constructs expressed in $\mathcal{DLR}^{\pm}$ and the same constructs expressed in FOL that are part of the $\mathcal{ORM}^{\pm}$ fragment.

Proof of lemma 5.

1. **FactType** $P \sqsubseteq (\sigma_{P.1:\mathrm{T}_1} P) \sqcap \cdots \sqcap (\sigma_{P.\alpha(P):\mathrm{T}_{\alpha(P)}} P)$
   The left-hand side is trivial since it represents a predicate in FOL: $\overline{x}.P(\overline{x})$
   The right-hand side is composed by a sequence of conjunctive $\mathcal{DLR}^{+}$ selections. Applying the reduction we obtain:
   $P(\overline{x}) \wedge T_1(x_1) \wedge \cdots \wedge P(\overline{x}) \wedge T_{\alpha(P)}(x_{\alpha(P)})$.
   Applying 5.13 we obtain:
   $\forall \overline{x}.P(\overline{x}) \rightarrow P(\overline{x}) \wedge T_1(x_1) \wedge \cdots \wedge P(\overline{x}) \wedge T_{\alpha(P)}(x_{\alpha(P)})$.
   We can remove $P(\overline{x})$ from the right-hand side since it is redundant. Using the extended notation, we finally obtain a FOL formula equivalent to the one defined in Table 3.3 for the ORM Fact Type:
   $\forall x_1 \ldots x_{\alpha(P)}.P(x_1 \ldots x_{\alpha(P)}) \rightarrow T_1(x_1) \wedge \cdots \wedge T_{\alpha(P)}(x_{\alpha(P)})$.

2. **Mandatory** $\mathrm{T} \sqsubseteq \exists[P_1.i_1]P_1 \sqcup \cdots \sqcup \exists[P_m.i_m]P_m$
   The left-hand side is trivial since it represents a unary predicate in FOL: $T(x)$. The right-hand side is composed by a sequence of $\mathcal{DLR}^{+}$ disjunctive concepts. Applying the reduction we obtain:
   $(\exists \overline{y} P(\overline{y}) \wedge y_{i_1} = x) \vee \cdots \vee (\exists \overline{y} P(\overline{y}) \wedge y_{i_m} = x)$.
   Applying 5.12 we obtain:
   $\forall x.T(x) \rightarrow \exists \overline{y} P_1(\overline{y}) \wedge y_{i_1} = x \vee \cdots \vee \exists \overline{y} P_m(\overline{y}) \wedge y_{i_m} = x$.
   Using the extended notation, we finally obtain a FOL formula equivalent to the one defined in Table 3.3 for the ORM Mandatory constraint:
   $\forall x.T(x) \rightarrow \exists y_1 \ldots y_{\alpha(P_1)}.\left(P_1(y_1 \ldots y_{\alpha(P_1)}) \wedge x = y_{i_1}\right)$
   $\vee \cdots \vee \exists y_1 \ldots y_{\alpha(P_m)}.\left(P_m(y_1 \ldots y_{\alpha(P_m)}) \wedge x = y_{i_m}\right)$.

3. **Frequency** $\pi[P_{.i_1} \ldots P_{.i_m}]P \sqsubseteq \pi^{(\geqslant p, \leqslant q)}[P_{.i_1} \ldots P_{.i_m}]P$
   Both sides use the $\mathcal{DLR}^{+}$ projection over a set of attributes and the right-hand side defines the cardinality. The corresponding lambda expression of the left-hand side is:
   $\lambda y_1 \ldots y_m.\exists \overline{z}.\mathrm{P}(\overline{z}) \wedge z_1 = y_1 \wedge \cdots \wedge z_k = y_m$.
   The same applies for the hand-right side which only differs because the cardinality:

$\lambda y_1 \dots y_m.\exists^{(\geqslant p,\leqslant q)}\overline{z}.\mathrm{P}(\overline{z}) \wedge z_1 = y_1 \wedge \dots \wedge z_k = y_m.$
Applying the reduction and 5.12 we obtain:
$\forall x_1 \dots x_{\alpha(P)}.P(x_1 \dots x_{\alpha(P)}) \rightarrow \exists^{(\geqslant p,\leqslant q)} y_1 \dots y_{\alpha(P)}.P(y_1 \dots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \dots \wedge x_{i_m} = y_{i_m}$
which is equivalent to the FOL formula defined for the ORM Frequency constraint. Please note that the translation is the same for every cardinality defined in Table 3.3 .

4. **Subtype** $\mathrm{T}_1 \sqsubseteq \mathrm{T}, \dots, \mathrm{T}_m \sqsubseteq \mathrm{T}$
This is trivial since each ORM Object Type can be directly expressed in a $\mathcal{DLR}^{+}$ concept like $T(x_m)$. Applying 5.12 we obtain the same FOL formula for the ORM Subtype:
$(\forall x.T_1(x) \rightarrow T(x)) \wedge \dots \wedge (\forall x.T_m(x) \rightarrow T(x))$

5. **ExclusiveSubtypes**
$\mathrm{T}_1 \sqsubseteq \mathrm{T} \sqcap \neg\mathrm{T}_2 \sqcap \dots \sqcap \neg\mathrm{T}_m, \dots, \mathrm{T}_{m-1} \sqsubseteq \mathrm{T} \sqcap \neg\mathrm{T}_m, \mathrm{T}_m \sqsubseteq \mathrm{T}$
Similar to previous case.

6. **ExhaustiveSubtypes** $\mathrm{T} \sqsubseteq \mathrm{T}_1 \sqcup \dots \sqcup \mathrm{T}_m$
Similar to previous case.

7. **Subset** $\pi[P_{1\cdot i_1} \dots P_{1\cdot i_m}]P_1 \sqsubseteq \pi[P_{2\cdot h_1} \dots P_{2\cdot h_m}]P_2$
This constraint involves two predicates: P_1 and P_2. According to the arity of these predicates there are 4 different combination as shown in Table 5.1. We consider the following case: $\mathrm{m} = \alpha(\mathrm{P}_1)$ and $\mathrm{m} < \alpha(\mathrm{P}_2)$.
As we have seen before, the left-hand side is trivial: $P_1(x_1, \dots x_{\alpha(P)})$.
The right-hand side is a partition and in FOL the equivalent is:
$\exists y_1 \dots y_{\alpha(P)}.P(y_1 \dots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \dots \wedge x_{i_m} = y_{i_m}.$
Finally we apply 5.13 along the previously defined renaming and we obtain:
$\forall x_1 \dots x_{\alpha(P)}.P(x_1 \dots x_{\alpha(P)}) \rightarrow \exists y_1 \dots y_{\alpha(P)}.P(y_1 \dots y_{\alpha(P)}) \wedge x_{i_1} = y_{i_1} \wedge \dots \wedge x_{i_m} = y_{i_m}$
which is equivalent to the FOL formula for the ORM Subset. The translation is similar for the remaining 3 cases.

8. **Exclusive** $\pi[P_{1\cdot i_1} \dots P_{1\cdot i_m}]P_1 \sqsubseteq \neg \odot \pi[P_{2\cdot h_1} \dots P_{2\cdot h_m}]P_2$
   This is similar to the previous case, the only difference is the negation on the right-hand side of the axiom.

9. **Objectifies** $\mathrm{T} \equiv \odot P$
   This is trivial since both sides are directly expressible in FOL. The left-hand side is an object type and the right-hand side is objectified. Applying 5.12 we obtain:
   $\forall x_1 \dots x_n . T(x) \leftrightarrow \ell_P\,(x_1 \dots x_n) = x$

□

Given lemma 5.2, we can now prove the following theorem:

Theorem 6. Entailment in $\mathcal{ORM}^{\pm}$ is decidable since any ORM conceptual diagram expressed in the $\mathcal{ORM}^{\pm}$ fragment can be encoded into a logically equivalent $\mathcal{DLR}^{\pm}$ TBox.

As a direct consequence of the theorem 6, we obtain the following corollary:

Corollary 7. Since $\mathcal{DLR}^{\pm}$ has a direct encoding in $\mathcal{ALCQI}$ as shown in 5.6, we conclude that exists a direct mapping from $\mathcal{ORM}^{\pm}$ to $\mathcal{ALCQI}$. Since $\mathcal{ALCQI}$ can be expressed in OWL, it follows that it is also possible to express $\mathcal{ORM}^{\pm}$ into OWL.

We show the complexity of entailment in $\mathcal{ORM}^{\pm}$:

1. It is known from [36] that $\mathcal{ALC}$ is ExpTime-complete; $\mathcal{ALC}$ can be expressed in $\mathcal{ORM}^{\pm}$ (trivial adaptation of proof with EER in [7]); it follows that $\mathcal{ORM}^{\pm}$ is ExpTime-hard (lower bound).

2. It is known from corollary 4 that $\mathcal{DLR}^{\pm}$ is ExpTime-complete; since $\mathcal{ORM}^{\pm}$ can be expressed in $\mathcal{DLR}^{\pm}$ as in lemma 5.2, it follows that $\mathcal{ORM}^{\pm}$ is in ExpTime (upper bound).

From 1 and 2, $\mathcal{ORM}^{\pm}$ is ExpTime-complete.

Corollary 8. Reasoning in $\mathcal{ORM}^{\pm}$ is ExpTime-complete.

These results have been used to build an implementation of $\mathcal{ORM}^{\pm}$ and its integration in a conceptual modelling framework, as described in Chapter 7.

In the last years, alternative proposals addressed the issue of encoding ORM conceptual diagrams into Description Logics knowledge bases. These proposals [18, 95, 101–103, 106] have already been discussed in detail in [60], where it has been found that some of these proposals lack a complete formal approach (as in [95, 103]) and some encodings are shown not to be correct (as [102, 106]). These proposals have in common the usage of the $\mathcal{DLR}_{ifd}$ language to encode the ORM diagrams. As we have seen in [10], $\mathcal{DLR}^{\pm}$ extends the n-ary description logics $\mathcal{DLR}$ and $\mathcal{DLR}_{ifd}$ without increasing the computational complexity of the basic $\mathcal{DLR}$ language. Choosing $\mathcal{DLR}^{\pm}$ to encode ORM allows to perform reasoning on more expressive ORM diagrams equipped with some constraints which are not expressible in $\mathcal{DLR}_{ifd}$, for example $\mathcal{DLR}^{\pm}$ is able to express projections between relations. Moreover, the proposal described in this work follows a completely formal approach which is based on the ORM formalisation from Chapter 2, where the syntax and the semantics of each ORM constraint has been unambiguously defined. After defining the ORM foundations, a DL language ($\mathcal{DLR}^{\pm}$) has been involved in the process to capture a relevant ORM fragment, namely $\mathcal{ORM}^{\pm}$. Some proposals also come with a tool which implements the encoding, as [95, 101, 102] for DogmaModeler [114] (which is compared to ORMiE in Chapter 9). Other proposals [122, 124, 137] have also been discussed in another work [55], where [122, 124] uses the numeric model instead of DL languages, as in the aforementioned proposals. In particular, [55] encodes a fragment of ORM using the PTIME Description Logic $\mathcal{CFDI}_{nc}^{\forall-}$. Similar to our proposal, the captured ORM fragment excludes covering and disjunctive mandatory constraints; it includes limited cardinalities, subsumption and disjointness between relationships and the definition of join paths; ORM Derivation Rules are not taken into account. To the best of our knowledge, the formalisation provided here is the only one including ORM Derivation Rules.

6

Modelling with ORM versus OWL

6.1 Introduction

In this chapter we analyse the difference between conceptual modelling in ORM and conceptual modelling directly in OWL.

The way we have tackled this issue so far in this document has been by considering OWL a sort of "assembler" into which encode ORM conceptual models. Indeed, we have considered the main advantages of the OWL modelling language, namely to have efficient implemented reasoners with universal APIs for a decidable fragment of description logics. Since we found a polynomial provably sound and complete encoding of arbitrary ORM conceptual models (in the restricted decidable fragment of ORM) into OWL knowledge bases, we could then reason correctly with ORM by exploiting the OWL reasoners which are acting hidden in the background. We can of course also access directly to the OWL knowledge base produced by the encoding of the ORM model, and we could use it directly.

But now we want to analyse how these OWL knowledge bases are meaningful taken autonomously, or, even better, whether people could build conceptual models in OWL following the patterns suggested by the ORM conceptual models.

Unluckily, the answer of this analysis is negative, namely it turns out that conceptual modelling directly in OWL, when following typical ORM of fact-based modelling principles, is unpractical and it would generate weird OWL knowledge bases.

This is mainly because OWL has a class-based flavour, as opposed to ORM which has a fact-based flavour. We have argued elsewhere in this document about the advantages of fact-based modelling. A class-based modelling is mainly different because it considers classes (i.e., entity types) as first class citizens, by focussing on many different aspects of their modelling (but neglecting completely the aspect of defining a reference schema for them, which is crucial for information systems design), while relations (i.e., fact types) are second class citizens, for which only typing constraints and ring constraints can be asserted - not to mention that OWL can only have binary relations.

This difference is basically the same difference between ORM and UML class diagrams. The absence of reference schemes makes these modelling language "object-centred" languages, since at the instance level there will be plenty of object identifiers as opposed to meaningful values identifying entities or values of properties. Of course, UML modelling has been indeed conceived as the conceptual modelling language for object oriented approaches (e.g., Java or C++), so we can not complain here about the qualities of these O-O modelling languages. Still, it has been argued that a fact-based approach could be also very appropriate for modelling O-O scenarios, but we do not want to argue that here.

We can observe that the same observation can be made about other modelling languages which have a rigorous semantics, such as Datalog, which also lack the ability to express reference schemes, while, however, allowing for n-ary relations.

Indeed, any logic-based conceptual modelling approach, while it can be related with ORM, via some possibly sound or possibly complete encoding (i.e., mapping), can be considered a valid approach by its own range of applications. What is important, is that for any of these alternative approaches, we can, or better we should, find the appropriate invertible mappings between pairs of them, so that we can easily move concepts and data across the various frameworks.

In fact, we argue that, in the same way in the research presented in this document we have found a (sound and complete) mapping from ORM to

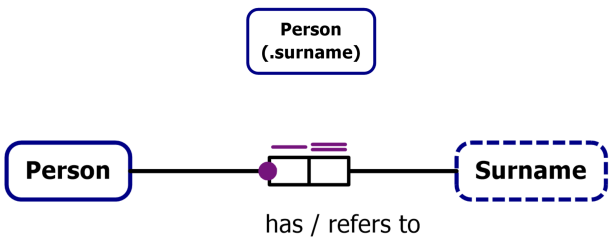
OWL, we should work in the future on finding mappings from OWL, or UML, or Datalog, or ArchiMate, or some other relevant approach to conceptual modelling. Those mappings most likely will be incomplete and will capture only some (relevant) fragment of the languages, but the could be immensely useful, in particular when the would map autonomous systems each one in its own modelling language, so that it suffices to map the alphabets of the different systems, but not the constraints; with this limitations, we believe that most of those mappings could be soundly and completely done.

In the following section we show with an abstract simple example, how the relation between ORM and OWL modelling develops for the basic constructs. This will emphasise and explain the differences we have summarised above.

6.2 ORM vs OWL via an example

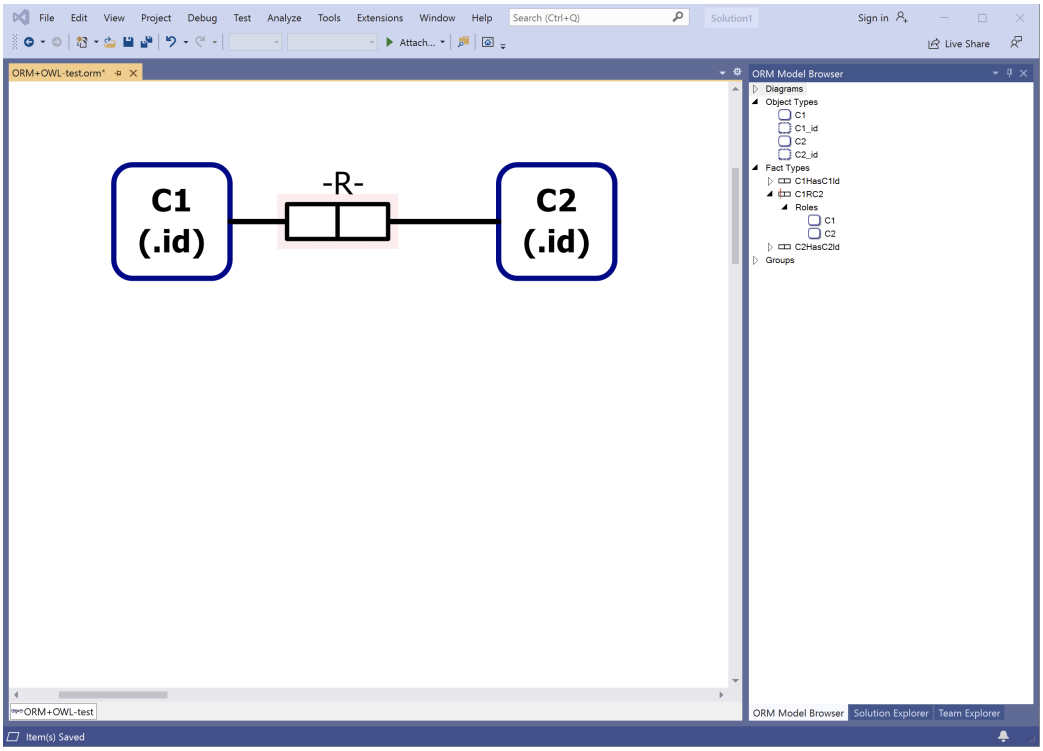
Let's start with the simplest ORM construct, namely the fact type.

Note that, throughout our running example, we will not consider the reference schemes, since they would be mapped as per their defining expansion, which, as we know, is rather clumsy to have in the explicit form, since it looses its original direct meaning:



In OWL we have no alternative than to have the reference scheme explicit, with all the complications, as we will see, necessary to express the encoding of the basic constructs involved in the explicit form.

Let's start our example with a plain binary predicate `R` typed by the entity types `C1` and `C2`, which themselves have a reference scheme with the attribute `id`:



The basic fact type ORM construct is expressed in OWL as follows:

```
# FactType(C1-R-C2 (C1 C2))
SubClassOf(PRED-C1-R-C2-{1,2}
           ObjectSomeValuesFrom(Q-{1} PRED-C1-R-C2-{1}))
SubClassOf(PRED-C1-R-C2-{1,2}
           ObjectSomeValuesFrom(Q-{2} PRED-C1-R-C2-{2}))

SubClassOf(PRED-C1-R-C2-{1,2}
           ObjectAllValuesFrom(Q-{1} TYPE-C1))
SubClassOf(PRED-C1-R-C2-{1,2}
           ObjectAllValuesFrom(Q-{2} TYPE-C2))
```

with the proviso that the object properties, representing the two roles, are functional:

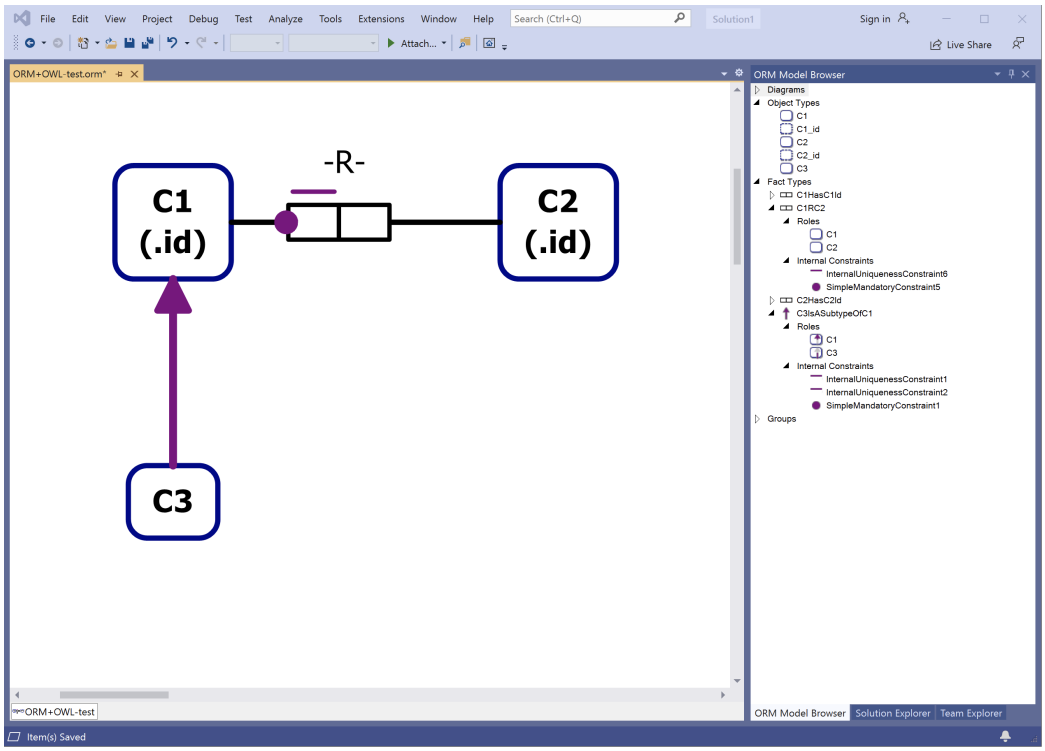
```
FunctionalObjectProperty(Q-{1})
FunctionalObjectProperty(Q-{2})
```

What does this really mean?

Since OWL is a class centred language, the fact type is expressed as the class `PRED-C1-R-C2-{1,2}`; this class is related via the object properties `Q-{1}` and `Q-{2}` to the classes representing the two roles `PRED-C1-R-C2-{1}` and `PRED-C1-R-C2-{2}`: this defines the so-called reified version of the fact type, namely the repreentation of the fact type in a class oriented language. Moreover, the actual typing is express by the universal quantifications from the fact type class to the two entity types `C1` and `C2` via the object properties `Q-{1}` and `Q-{2}` respectively.

It is immediately clear that this encoding is not directly intuitive if one would like to write it directly in OWL.

Let's consider now the addition of three more basic constructs, such as mandatory, uniqueness, and subtype:



This is expressed in OWL with the following additional axioms:

```
# Mandatory(C1 C1-R-C2.1)
```

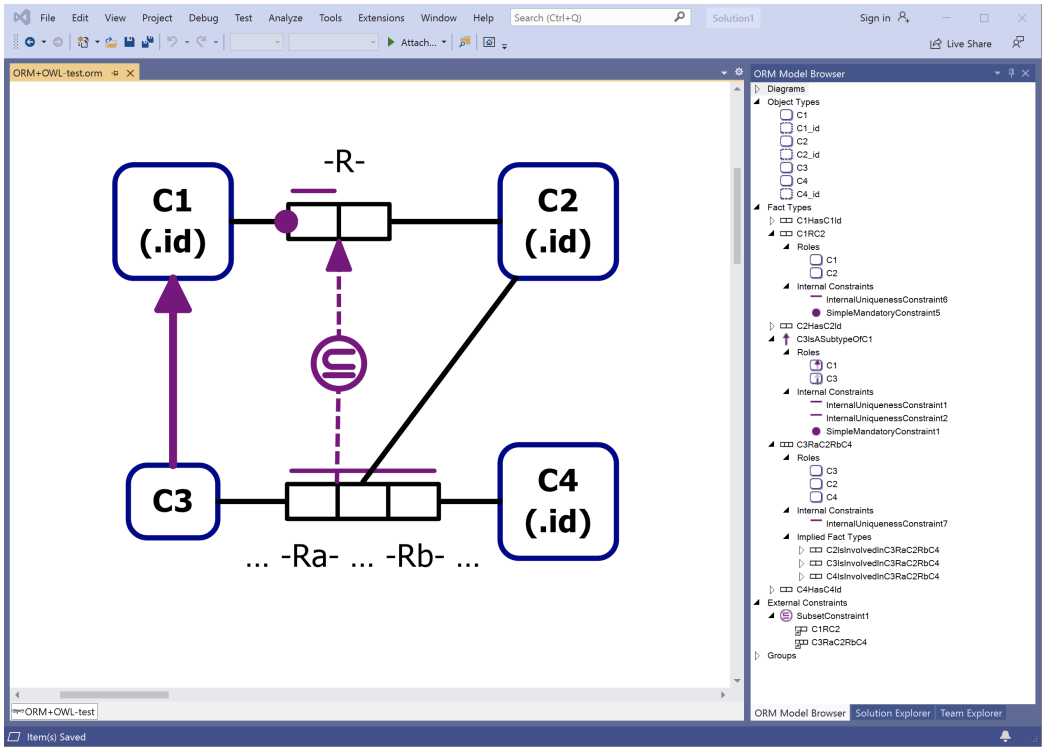
```
SubClassOf(TYPE-C1
           ObjectSomeValuesFrom(ObjectInverseOf(Q-{1})
                                PRED-C1-R-C2-{1,2}))

# Unique(C1-R-C2.1)
SubClassOf(ObjectSomeValuesFrom(ObjectInverseOf(Q-{1})
                                PRED-C1-R-C2-{1,2})
           ObjectExactCardinality(1 ObjectInverseOf(Q-{1})
                                    PRED-C1-R-C2-{1,2}))

# Subtype(C3 C1)
SubClassOf(TYPE-C3 TYPE-C1)
```

While the subtype constraint OWL axiom is rather intuitive, the mandatory and uniqueness constraints are not.

The OWL expression for the mandatory constraint states that each element of `C1` is in the projection over the first role of `PRED-C1-R-C2-{1,2}`, i.e., the class representation of the fact type.

The OWL expression for the uniqueness constraint is more subtle: it says that the projection over the first role of `PRED-C1-R-C2-{1,2}`, i.e., the class representation of the fact type., is among the elements that appear exactly once in tht projection. This is the way we can express in OWL the absence of duplicates.

Continuing with our example, let's complete it with a predicate with arity more than two, and with a subset constraint - elements which are quite common in ORM conceptual models:



Below you can find the additional axioms needed in the representation of the above model in OWL. Let's consider first the fact type declaration.

```
# FactType(C3-Ra-C2-Rb-C4 (C3 C2 C4))
FunctionalObjectProperty(Q-{1})
FunctionalObjectProperty(Q-{2})
FunctionalObjectProperty(Q-{3})

DisjointClasses(PRED-C1-R-C2-{1,2} PRED-C3-Ra-C2-Rb-C4-{1,2,3})

SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectSomeValuesFrom(Q-{1} PRED-C3-Ra-C2-Rb-C4-{1}))
SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectSomeValuesFrom(Q-{2} PRED-C3-Ra-C2-Rb-C4-{2}))
SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectSomeValuesFrom(Q-{3} PRED-C3-Ra-C2-Rb-C4-{3}))
```

```
SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectAllValuesFrom(Q-{1}
                               ObjectAllValuesFrom(Q-{1,2}
                                                   TYPE-C3)))

SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectAllValuesFrom(Q-{2}
                               ObjectAllValuesFrom(Q-{1,2}
                                                   TYPE-C2)))

SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectAllValuesFrom(Q-{3} TYPE-C4))

SubClassOf(PRED-C3-Ra-C2-Rb-C4-{1,2,3}
           ObjectSomeValuesFrom(Q-{1,2}
                                PRED-C3-Ra-C2-Rb-C4-{1,2}))
```

The OWL representation of the ternary fact type mirrors the way we represented the binary fact type, but with an additional element. Indeed, the last axiom above serves the purpose of introducing the projection (as a class) over the first two roles of the ternary fact type, via an additional funtional object property `Q-{1,2}` connecting the reified fact type to its projection.

Let's consider now the subset constraint:

```
# Subset((C3-Ra-C2-Rb-C4.1 C1-R-C2.1)(C3-Ra-C2-Rb-C4.2 C1-R-C2.2))
SubClassOf(ObjectSomeValuesFrom(ObjectInverseOf(Q-{1,2})
                                PRED-C3-Ra-C2-Rb-C4-{1,2})
           PRED-C1-R-C2-{1,2})
```

It says that the projection (as a class) over the first two roles of the ternary fact type is a subset of the class representing the binary relation `R`.

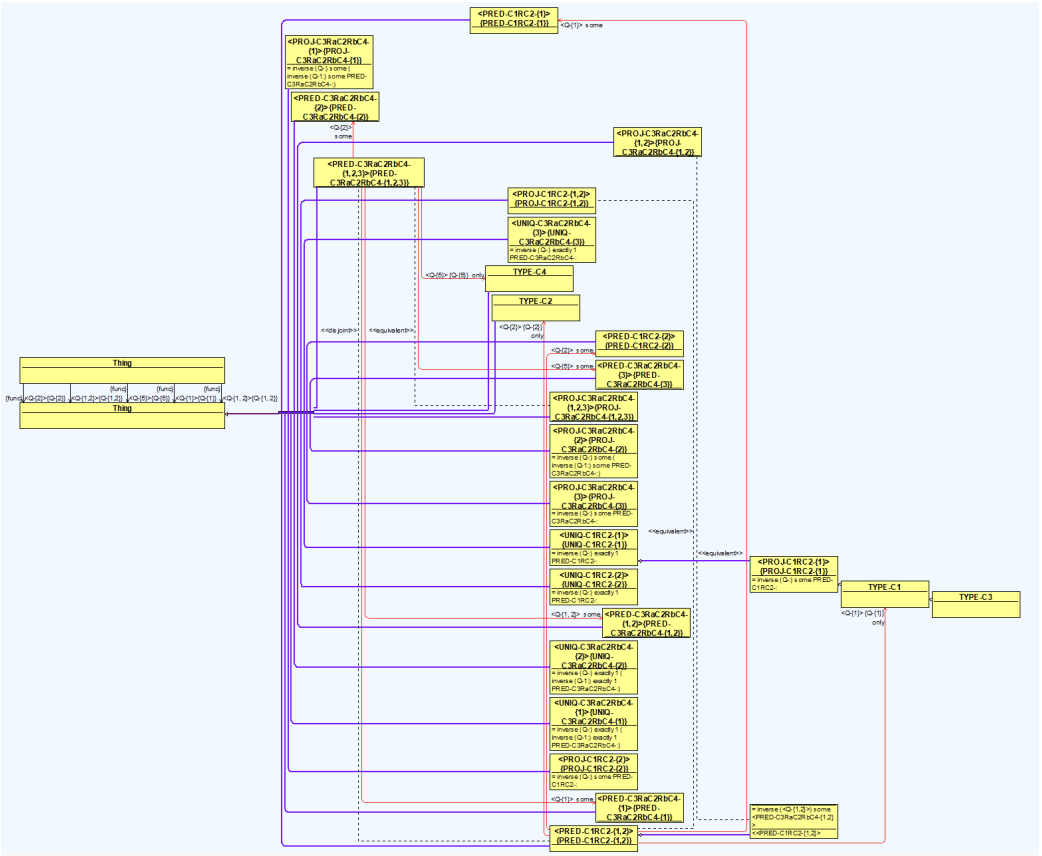
As we mentioned in the introduction, we can observe that:

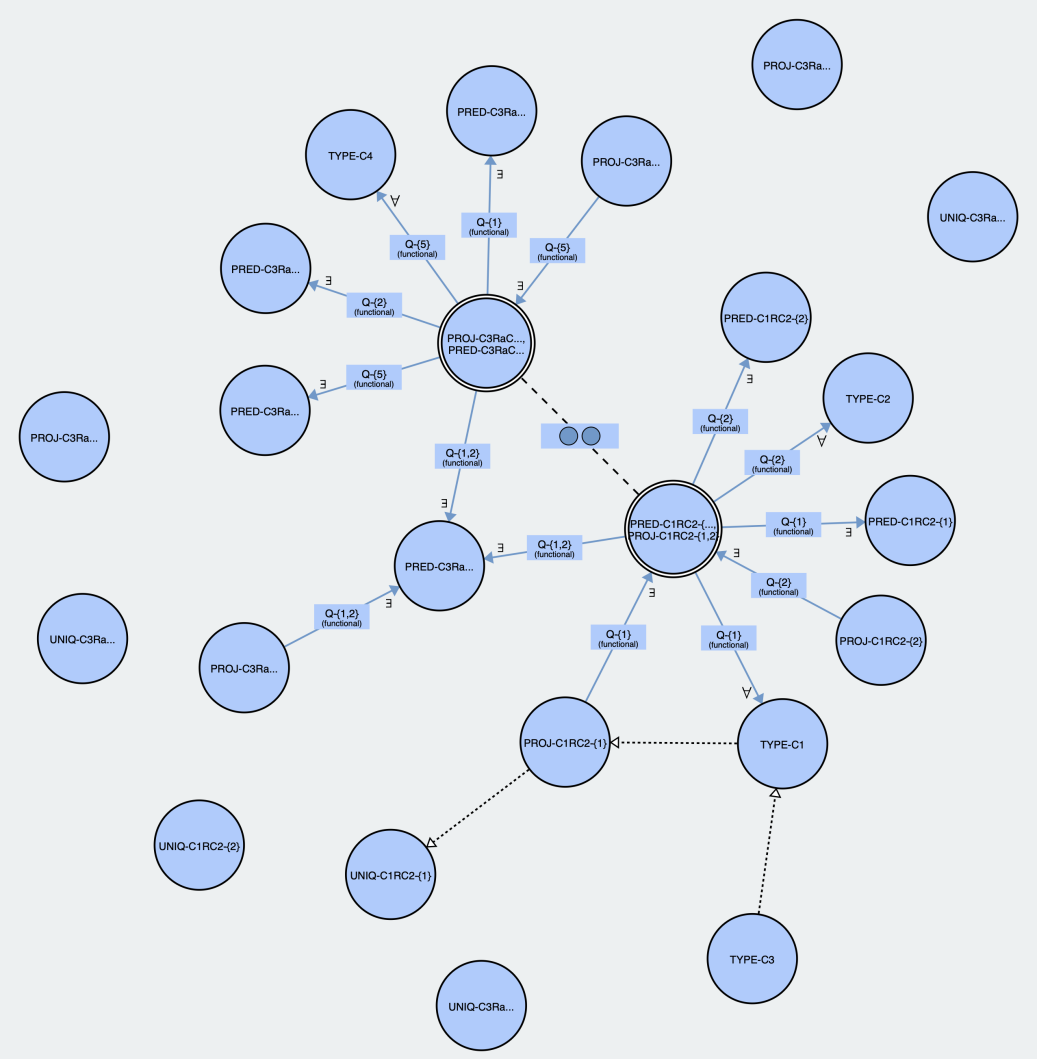
1. since there is no native reference scheme construct, this should be expressed in OWL as a combination of a reified binary relation, a mandatory constraint, and two uniqueness constraints;

2. all the fact types and their projections (in our example, unary and binary) are represented in OWL with classes, interconnected by functional object properties.

We claim that this kind of modelling is counter-intuitive for a typical OWL modeller, and definitely quite prone to errors.

To enforce our argument, we show below how the OWL knowledge base of our example would be graphically depicted using the two most prominent graphing tools for OWL: *OWLGrEd* (`http://owlgred.lumii.lv`) and *VOWL* (`http://vowl.visualdataweb.org`). These graphs have lost completely the intuitive meaning of the graphical form of ORM. x




PROJ-C3Ra...
PRED-C3Ra...
TYPE-C4
PROJ-C3Ra...
UNIQ-C3Ra...
Q-{1} (functional)
Q-{5} (functional)
Q-{5} (functional)
PRED-C3Ra...
Q-{2} (functional)
PROJ-C3RaC..., PRED-C3RaC...
PRED-C1RC2-{2}
Q-{5} (functional)
PRED-C3Ra...
TYPE-C2
PROJ-C3Ra...
Q-{1,2} (functional)
Q-{2} (functional)
Q-{2} (functional)
PRED-C1RC2-{...} PROJ-C1RC2-{1,2}
Q-{1} (functional)
PRED-C1RC2-{1}
Q-{1,2} (functional)
PRED-C3Ra...
Q-{2} (functional)
UNIQ-C3Ra...
Q-{1,2} (functional)
PROJ-C3Ra...
Q-{1} (functional)
Q-{1} (functional)
PROJ-C1RC2-{2}
PROJ-C1RC2-{1}
TYPE-C1
UNIQ-C1RC2-{2}
UNIQ-C1RC2-{1}
TYPE-C3
UNIQ-C3Ra...

7 UModel

In this chapter we present UModel [49], a framework to activate automated reasoning over conceptual modelling languages.

In this work, UModel will be used to implement the ORM formalisation presented previously. UModel is not limited to ORM because it has been created to be a general purpose system designed to be used with any conceptual language, after which it may be integrated into a conceptual modelling CASE tool. The main idea behind this system is to provide a generalised solution for the conceptual modelling community to deal with the problem of applying reasoning to any CASE tool with any conceptual modelling language. We are going to take a deep look at this framework, exploring each module in detail providing a practical usage examples as well.

7.1 System description

The main goal of UModel is to enable automated reasoning on conceptual diagrams which are used in CASE tools to model domains. CASE tools are widely used to model software or databases since they are quite powerful and rich of features, but they are not able to check the semantics over the conceptual diagrams. UModel has been designed to be compatible with those tools that adopt the conceptual modelling languages such as ORM, UML and ER.

Enriching a CASE tool with reasoning capabilities gives the modeller more control over the semantics of the diagram and an added value to the tool.

Due to the fact that the modeller is a human being, he cannot automatically control the semantics of the conceptual diagrams because the model size may be unbearable and the task of checking the semantics will be repetitive and time consuming. Managing this situation manually involves finding, tracing and fixing errors, which it could be very difficult to accomplish and time consuming. We must also consider that in the real world industry the diagrams may be very large and different stakeholders take part in the same project, so the need of the automated reasoning is essential in this context. The absence of semantic checks on conceptual diagrams could lead to software degradation and unexpected software behaviours. In order to overcome this problem, it is highly recommended to equip a CASE tool with reasoning capabilities.

It is also important to make a distinction between the final users and the developers: the final users are in other words the modellers, the people who use a conceptual modelling software to model a domain; the developers, on the other hand, are the ones that use the API system provided by UModel to integrate the system into the target CASE tool. It is also possible that these two kind of users may be the same.

The benefits of using UModel are related to the inferred knowledge coming from the automated reasoning; this may suggest revision or confirm the consistency of the conceptual diagram. The automated reasoning can discover inconsistencies, redundancies or any other formal properties that can be useful to detect unexpected behaviours during the modelling step. Another benefit of using automated reasoning is to prevent all the negative consequences coming from bad modelling, which may require further iterations during the development in order to rebuild the diagrams. Performing these iterations may be time consuming.

At the moment the UModel system fully implements the ORM language and a completed integration with the NORMA tool. The system is written in Java and ported in .NET because the cross-platform feature of Java and the popularity of .NET framework, so it works smoothly on any Java and C# application. No other portings have been provided yet.

7.1.1 Specs

UModel is written in Java. The language choice is motivated by the cross-platform capability of Java and also because it is a popular language, thereby easy to integrate into any system. To extend the range of compatibility, we also built a version of UModel in C# to be used in .NET. The porting has been made using the IKVM tool. The version of Java used is Java 8 and the graphical interface is built using Java Swing components. As for the OWLAPI the system uses the OWLAPI 4.1.4 and the reasoner is Fact++. After several tests, this combination has been elected as the most compatible with the bast performance. We also tried different reasoners, such as JFact, but they lacked performance and had some unresolved bugs and limited reasoning capabilities. Despite Fact++ is not in the same stack as Java language, since it is not a Java library but native code written in C++ which needs to be properly configured in the environment, it is selected as the best choice due to its performance.

7.2 Workflow

The main idea behind this system is depicted in a workflow generally used to apply automated reasoning over conceptual modelling languages, as described in Figure 7.1.

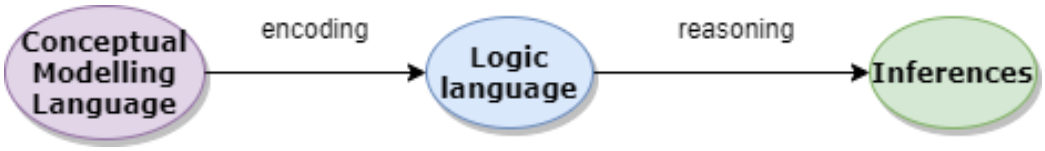


Figure 7.1: The workflow main idea

First of all we recall the goal: performing the automated reasoning over a conceptual diagram expressed in a conceptual modelling language, in order to obtain relevant inferences coming from the reasoning procedure. The entry point is always a conceptual diagram expressed in a conceptual modelling language (e.g., UML, ER, ORM); then, we use a procedure to encode the diagram expressed in that language, into a logical language. In this way, we

can enable the automated reasoning procedure since we take advantage of the logic language properties and reasoning algorithms; in the last step, the inferences are shown as results of the automated reasoning procedure.

In this specific context, we decline this general approach to UModel using ORM, as in Figure 7.2.

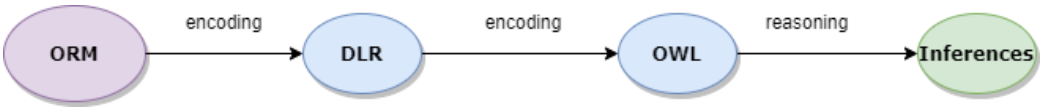


Figure 7.2: The workflow in UModel with the ORM language

This workflow slightly differs from the previous one because ORM is first translated into $\mathcal{DLR}^{\pm}$ language [10],[9],[8].

$\mathcal{DLR}^{\pm}$ acts as a middleware language between the conceptual modelling languages and its encoding in OWL. The reason behind this approach is that the family $\mathcal{DLR}$ has been specifically designed for this class of modelling languages and it can also deal with n-ary relationships that frequently occur in conceptual diagrams.

An architectural benefit comes from this choice: the $\mathcal{DLR}^{\pm}$ module encodes any conceptual diagram into OWL independently, so that the core procedure that encodes a conceptual language into OWL is always the same and completely transparent to the developer. In this way, the developer just needs to create the conceptual diagram using the API provided, without caring about any internal implementation.

7.3 Architecture and components

In this section we explain in detail each single component of the UModel framework. Components are described following the execution order, starting from the model definition until the last step showing the inferences. In order to understand the framework design, UML class diagrams are used to show the structure of the system components. In Figure 7.3 the architecture of the system is summarized.

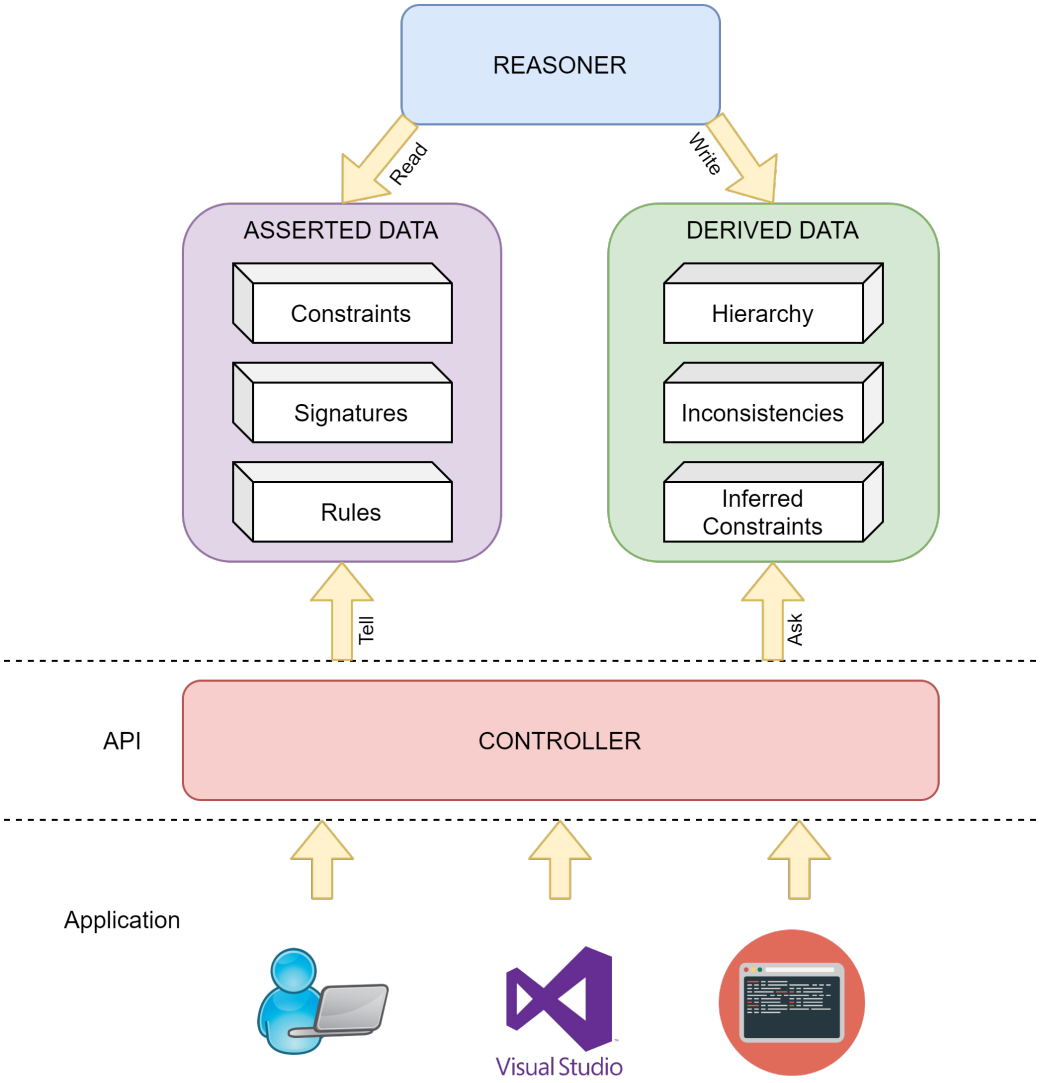


Figure 7.3: UModel architecture

The whole process must be read as a precise sequence of steps. Briefly, the starting point involves representing ORM constraints in Java data structures to represent an ORM diagram. After this step, there is the $\mathcal{DLR}^{\pm}$ encoding which is the backbone of the next steps: the OWL generation and the reasoning. After the reasoning is completed the inferences are encoded in object-oriented data structures, this makes them easy and fast to query.

Optionally, an interface is shown to graphically represent the outcome of the reasoning.

7.3.1 Creation of the conceptual model

A conceptual model consists of a set of constraints. The main class representing the model is named UModel and the main class for the constraints is called UModelElement, as we can see in Figure 7.4. Both classes are abstract, this means that they are made to be extended by classes with a specific language implementation, like ORM. This architectural choice has been taken to make UModel modular and extensible for any conceptual modelling language. Since we are dealing with ORM, we want to create an ORM conceptual model by means of ORM constraints, so we need to specify both classes as shown in Figure 7.4.

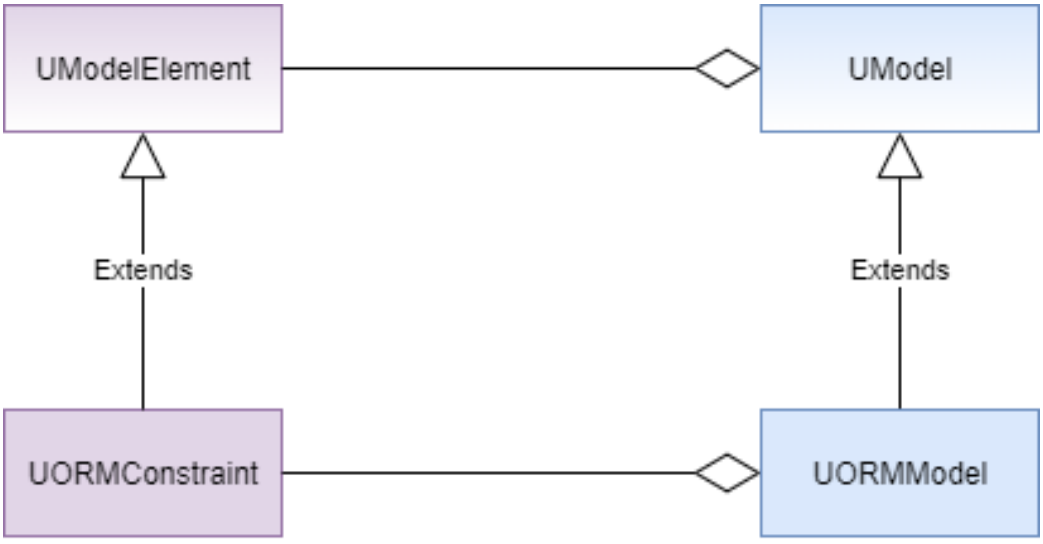


Figure 7.4: The core components in UModel

The class UORMModel extends the class UModel; the class UORMConstraint extends the UModelElement one. The UModelElement class is abstract since it represents a generic ORM constraint. For this reason, each ORM constraint extends the UModelElement class with its own structure.

UModel encodes the ORM signature as follows:

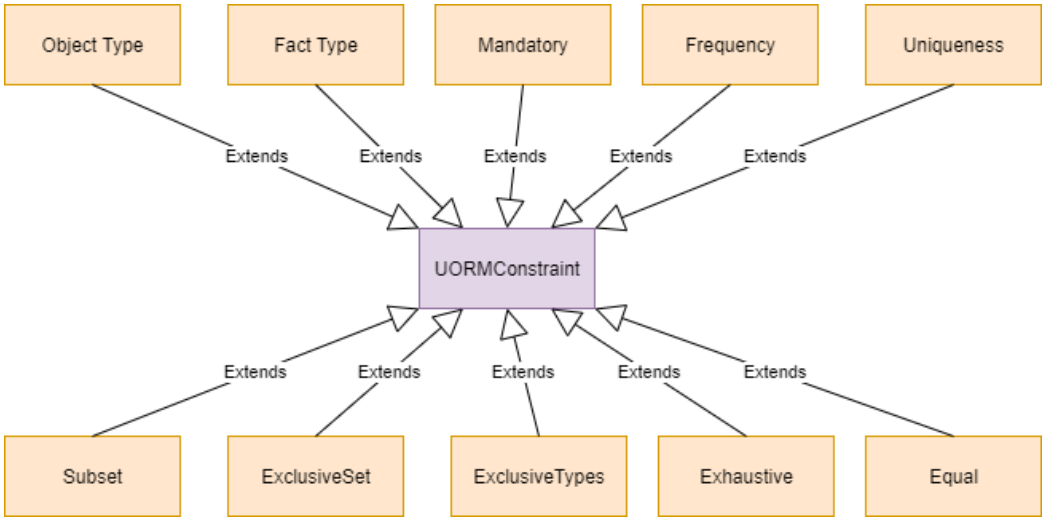


Figure 7.5: The ORM constraints in UModel

1. ORMEntityType: a structure representing the ORM Object Type, it is defined by the name of the entity type representing a specific domain;

2. ORMFactType: defines a relationships in ORM a relationship consisting of a predicate name and an arity. A fact type is composed by a sequence of roles.

3. ORMRole: the roles that are embedded in the ORMFactType data structure. As with ORM, each role has an index which is the position inside the predicate, and a name composed by the predicate followed by the dot and the index.

And the ORM constraints:

1. Mandatory, specifically the simple Mandatory which is placed on an ORM role of a predicate;

2. Uniqueness, specifically the simple Uniqueness which is placed on an ORM role of a predicate;

3. Exclusion, the exclusion ORM constraint applicable to entity types and fact types

4. Subtype, for IsA relationships between entity types
5. Subset, for IsA relationships between fact types
6. Exhaustive, for the covering among entity types and fact types
7. Frequency, to set the cardinality

The reader should note that despite object types, fact type and roles belongs conceptually to the ORM signature as we have seen in Chapter 2, in the implementation context it is more suitable to derive them from the abstract class UModelElement, since they will be added as part of the ORM model, exactly as the other constraints.

In this step the developer builds the ORM model adding a set of constraints using the API system shipped with UModel. A complete example is provided in Section 7.3.6.

7.3.2 $\mathcal{DLR}^{\pm}$ encoding

The purpose of this step is to encode ORM models in $\mathcal{DLR}^{\pm}$, in order to perform the OWL mapping in the next step. The ORM fragment covered by UModel is $\mathcal{ORM}^{\pm}$ and as we have seen in Chapter 5 this fragment can be expressed in $\mathcal{DLR}^{\pm}$. As shown in Figure 7.6, the translation is made into two steps: from ORM to $\mathcal{DLR}^{\pm}$ and from $\mathcal{DLR}^{\pm}$ to OWL. In this section we discuss the first step.

More generally, the role of the $\mathcal{DLR}^{\pm}$ language in the UModel workflow is acting like a "proxy" between any conceptual modelling language and OWL.

The benefit of this approach is avoiding further reimplementation of the $\mathcal{DLR}^{\pm}$ logic and algorithms over and over again for each language; moreover, this procedure is transparent to the developer because it is automatically executed by the system before starting the reasoner.

In order to understand how this procedure works, we have to recall Section 5.1.4, where a $\mathcal{DLR}^{\pm}$ knowledge base is represented such as in the multitree in Figure 7.7.

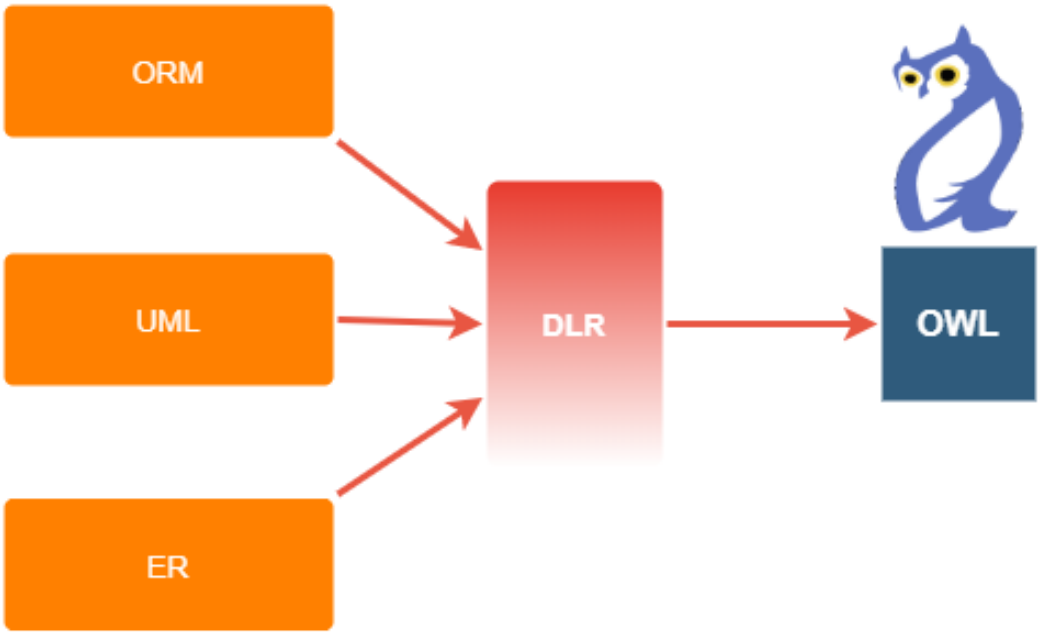


Figure 7.6: $\mathcal{DLR}^{\pm}$ as an intermediate language

In $\mathcal{DLR}^{\pm}$ each node is composed by a sequence of attributes. In ORM a node represents the sequence of roles belonging to a fact type. Nodes can share some attributes (roles for ORM) when the renaming is applied (for example, when an ORM subset constraints is involved in two fact types). Leaf nodes (also called as "singleton") encode a single attribute in the tree. Intermediate nodes (also called as "partition") are created when a projection of a relationship is asserted in the knowledge base. In ORM this happens when some constraints involve a subset of roles in a fact type (i.e., uniqueness spanning over one or more roles).

If we consider the ORM diagram in Figure 7.8, the corresponding projection signature graph is shown in Figure 7.9.

This small model has only two fact types: *origin* and *morigin*. In the model there is a subset constraint stating that all the pairs inside *morigin* are also included in *origin*. This means that these two fact types share the same roles. Because the presence of this subset constraint, the $\mathcal{DLR}^{+}$ renaming function is activated collapsing the corresponding tree nodes of *origin* and *morigin* into a single one.

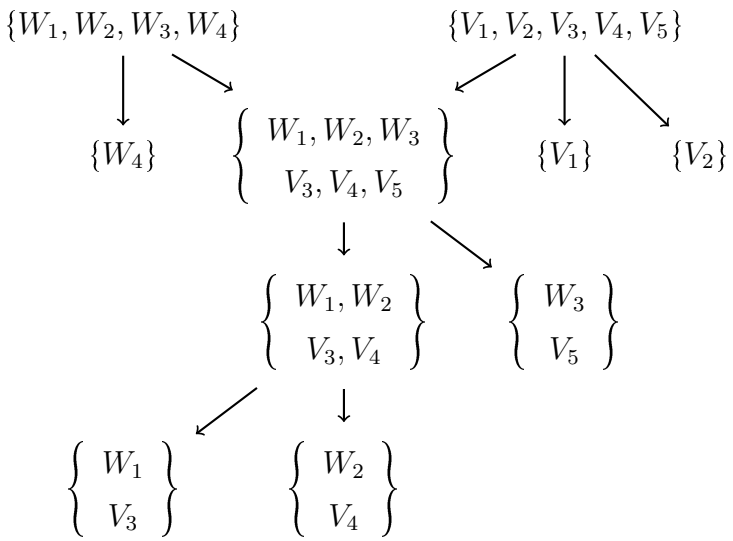


Figure 7.7: The projection signature graph of Example 5.1.1.

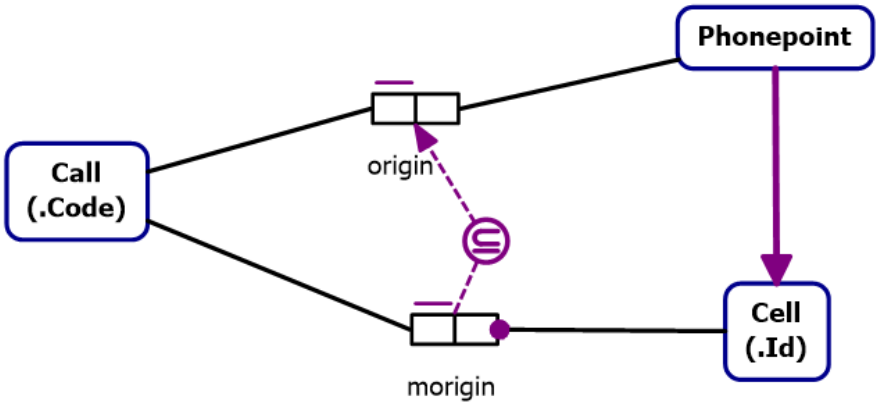


Figure 7.8: ORM schema to be encoded via API

The framework UModel implements the $\mathcal{DLR}^{\pm}$ multitree with an *hashmap* data structure:

- key: the sequence of roles in the node

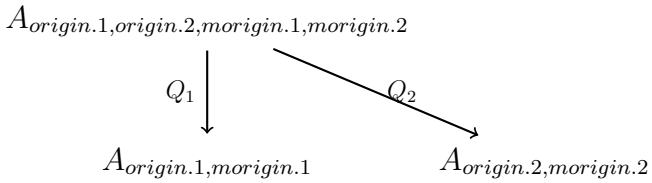


Figure 7.9: The projection signature graph of Example 5.1.1.

- value: MultiTreeNode data structure containing information about that node

Each entry encodes the node information as shown in Table 7.1.

Table 7.1: $\mathcal{DLR}^{\pm}$ implemented as an HashMap

Entry	**Key**	**Value**
	Fact Type roles	MultiTreeNode

The sequence of roles uniquely identifies a fact type, a partition of the projection signature graph and a single role. Each sequence of roles uniquely identifies an entry in the hashmap data structure.

The *MultiTreeNode* is a data structure containing information about the node:

```
protected MultiTreeNode(BitSet bitset, MultiTreeNodeType
    type, String rn) {
this.bitset = bitset;
this.type = type;
this.rn = rn;
}
```

1. BitSet - The BitSet native Java class is used to store the indexes of roles to efficiently compare them performing bitwise operations among multitree nodes. This is particularly useful when permutations are needed to calculate the $\mathcal{DLR}^{\pm}$ functions as *dsj* and *rel*. BitSet class is also used to traverse a path between two nodes, as in the $\mathcal{DLR}^{\pm}$ Path function.

2. MultiTreeNodeType is an ENUM type which assigns tags to the nodes. This comes handy when we need to detect if a node is root, a partition or a singleton.

```
public enum MultiTreeNodeType {
ROOT, PARTITION, SINGLETON
}
```

3. RN - This is the fact type name belonging to an entry.

Considering the example in Figure 7.8, the corresponding multitree is shown in Table 7.2.

Table 7.2: Multitree entries for the ORM diagram 7.8

Entry	**Key**	**Value**
0	morigin.1	10 | SINGLETON | morigin
1	origin.1,origin.2	11 | ROOT | origin
2	morigin.2	01 | SINGLETON | morigin
3	origin.1	10 | SINGLETON | origin
4	morigin.1,morigin.2	11 | ROOT | morigin
5	origin.2	01 | SINGLETON | origin

Observing Table 7.2, we notice that in the knowledge base there are no partition nodes because the maximum fact types arity in the model is two, so splitting them means creating exactly two singleton nodes. The two root nodes have the same bitset because of the renaming; the same applies for each role. This situation is generated by the ORM subset constraint which states that all the pairs in *morigin* are also in *origin*. Bitsets are set considering the position of the indexes for each role. For example, morigin.1 has the first bitset set and the second unset; the opposite is for morigin.2, where the second bitset is set and the first one unset.

At the end of this step, UModel has encoded a data structure filled with all the necessary information that will be used to perform the OWL encoding in the next step.

7.3.3 OWL generation

The purpose of this step is to encode the ORM model into OWL since we already collected (in the previous step) the multitree data, producing an OWL ontology to be used in the next step in order to perform the automated reasoning.

In the previous step, we have encoded the ORM model into the fragment $\mathcal{ORM}^{\pm}$ which is expressed in $\mathcal{DLR}^{\pm}$ as previously shown (Section 5.1). Recalling the corollary 7 which states that an ORM model expressed in $\mathcal{ORM}^{\pm}$ can be encoded in $\mathcal{ALCQI}$ (and therefore in OWL), we now show in Table 7.3 how the ORM constraints in $\mathcal{ORM}^{\pm}$ are encoded in $\mathcal{ALCQI}$. We recall the usage of the † function (as in Figure 5.6) which maps each concept name *CN* and each relation name *RN* appearing in the $\mathcal{DLR}^{\pm}$ KB (since we are using $\mathcal{ORM}^{\pm}$) to the $\mathcal{ALCQI}$ concept names and relation names. This encoding is implemented in UModel using OWLAPI.

Table 7.3: $\mathcal{ORM}^{\pm}$ constraints encoded in $\mathcal{ALCQI}$

FactType(P $\mathrm{T}_1 \ldots \mathrm{T}_{\alpha(\mathrm{P})}$))	P does not appear as an AlternatePredicate
OWL $\mathrm{P}^{\dagger} \sqsubseteq \mathrm{P}^{\dagger} \sqcap \forall \mathrm{PATH}_{\mathscr{J}}(\tau(\mathrm{P}), \mathrm{P}_{\cdot 1})^{\dagger}.\mathrm{T}_1^{\dagger} \sqcap \cdots \sqcap \mathrm{P}^{\dagger} \sqcap \forall \mathrm{PATH}_{\mathscr{J}}(\tau(\mathrm{P}), \mathrm{P}_{\cdot \alpha(\mathrm{P})})^{\dagger}.\mathrm{T}_{\alpha(\mathrm{P})}^{\dagger}$	
Subtype(($\mathrm{T}_1 \ldots \mathrm{T}_{\mathrm{m}}$) T) OWL $\mathrm{T}_1^{\dagger} \sqsubseteq \mathrm{T}^{\dagger}, \ldots, \mathrm{T}_m^{\dagger} \sqsubseteq \mathrm{T}^{\dagger}$	
ExclusiveSubtypes(($\mathrm{T}_1 \ldots \mathrm{T}_{\mathrm{m}}$) T) OWL $\mathrm{T}_1^{\dagger} \sqsubseteq \mathrm{T}^{\dagger} \sqcap \neg\, \mathrm{T}_2^{\dagger} \sqcap \cdots \sqcap \neg\, \mathrm{T}_m^{\dagger}, \ldots, \mathrm{T}_{m-1}^{\dagger} \sqsubseteq \mathrm{T}^{\dagger} \sqcap \neg\, \mathrm{T}_m^{\dagger}, \ldots, \mathrm{T}_m^{\dagger} \sqsubseteq \mathrm{T}\dagger$	
ExhaustiveSubtypes(($\mathrm{T}_1 \ldots \mathrm{T}_{\mathrm{m}}$) T) OWL $\mathrm{T}^{\dagger} \sqsubseteq \mathrm{T}_1^{\dagger} \sqcup \cdots \sqcup \mathrm{T}_m^{\dagger}$	
Mandatory(T $\mathrm{P}_1.\mathrm{i}_1 \ldots \mathrm{P}_m.\mathrm{i}_m$)	for j≠k and j,k⩽m: $\mathrm{P_j} \neq \mathrm{P_k}$
OWL $\mathrm{T}^{\dagger} \sqsubseteq \exists(\mathrm{PATH}_{\mathscr{J}}(\tau(\mathrm{P}_1), \mathrm{P}_1.\mathrm{i}_1)).{}^{-}\mathrm{P}_1^{\dagger} \sqcup \cdots \sqcup \exists(\mathrm{PATH}_{\mathscr{J}}(\tau(P_m), \mathrm{P_m}.\mathrm{i}_m)).{}^{-}\mathrm{P}_m^{\dagger}$	
Unique($\mathrm{P.i}_1 \ldots \mathrm{P.i}_m$)	for j≠k and j,k⩽m: $\mathrm{i}_j \neq \mathrm{i}_k$
OWL $\exists(\mathrm{PATH}_{\mathscr{J}}(\tau(P), \mathrm{P.i}_1 \ldots \mathrm{P.i}_m))^{-}.\mathrm{P}^{\dagger} \sqsubseteq \exists^{=1}(\mathrm{PATH}_{\mathscr{J}}(\tau(P), \mathrm{P.i}_1 \ldots \mathrm{P.i}_m))^{-}.\mathrm{P}^{\dagger}$	

Subset((P$_1$.i$_1$ P$_2$.h$_1$) ... (P$_1$.i$_m$ P$_2$.h$_m$))	P$_1$≠P$_2$ and for j≠k and j,k⩽m **:** P$_1$.i$_j$ ≠ P$_1$.i$_k$ and P$_2$.h$_j$ ≠ P$_2$.h$_k$
OWL	
$\exists(\text{PATH}_{\mathscr{J}}(\tau(P_1), \text{P}_1.\text{i}_1 \dots \text{P}_1.\text{i}_m)))^-.\text{P}_1^\dagger \sqsubseteq \exists(\text{PATH}_{\mathscr{J}}(\tau(P_2), \text{P}_2.\text{j}_1 \dots \text{P}_2.\text{j}_m))^-.\text{P}_2^\dagger$	
$\text{P}_1^\dagger \sqsubseteq \exists(\text{PATH}_{\mathscr{J}}(\tau(P_2), \text{P}_2.\text{j}_1 \dots \text{P}_2.\text{j}_m))^-.\text{P}_2^\dagger$	
$\exists(\text{PATH}_{\mathscr{J}}(\tau(P_1), \text{P}_1.\text{i}_1 \dots \text{P}_1.\text{i}_m))^-.\text{P}_1^\dagger \sqsubseteq \text{P}_2^\dagger$	
$\text{P}_1^\dagger \sqsubseteq \text{P}_2^\dagger$	

Exclusive((P$_1$.i$_1$ P$_2$.h$_1$) ... (P$_1$.i$_m$ P$_2$.h$_m$))	P$_1$≠P$_2$ and for j≠k and j,k⩽m **:** P$_1$.i$_j$ ≠ P$_1$.i$_k$ and P$_2$.h$_j$ ≠ P$_2$.h$_k$
OWL	
$\exists(\text{PATH}_{\mathscr{J}}(\tau(P_1), \text{P}_1.\text{i}_1 \dots \text{P}_1.\text{i}_m)))^-.\text{P}_1^\dagger \sqsubseteq \neg\exists(\text{PATH}_{\mathscr{J}}(\tau(P_2), \text{P}_2.\text{j}_1 \dots \text{P}_2.\text{j}_m))^-.\text{P}_2^\dagger$	
$\text{P}_1^\dagger \sqsubseteq \neg\exists(\text{PATH}_{\mathscr{J}}(\tau(P_2), \text{P}_2.\text{j}_1 \dots \text{P}_2.\text{j}_m))^-.\text{P}_2^\dagger$	
$\exists(\text{PATH}_{\mathscr{J}}(\tau(P_1), \text{P}_1.\text{i}_1 \dots \text{P}_1.\text{i}_m))^-.\text{P}_1^\dagger \sqsubseteq \neg\text{P}_2^\dagger$	
$\text{P}_1^\dagger \sqsubseteq \neg\text{P}_2^\dagger$	

Objectifies(T P)
OWL $\text{T}^\dagger \equiv \text{P}^\dagger$

The encoding in $\mathcal{ALCQI}$ also includes a set of axioms related to the multitree data structure. Moreover, other axioms have been added in the generated ontology in order to improve the system performance and the readability of the ontology from the user perspective. Below is provided a complete example illustrating the structure of an OWL ontology generated from an ORM Model, plus details about the aforementioned axioms.

An OWL ontology consists of a set of OWL declarations and axioms. We are going to show step by step how the ontology is built using the running example in Figure 7.8. Declarations are generated for each entity type and fact type; additional declarations are also generated for multitree nodes and edges.

- Entity Type - (OWL Class)
- Fact Type - Root nodes in the multitree - (OWL Class)
- Possible Partition nodes in the multitree - (OWL Class)
- Role - Singleton nodes in the multitree - (OWL Class)
- Multitree edges - (OWL property)

The generated ontology is displayed in the OWL "Functional Syntax Document Format" [116] with prefix "http://www.ormie.org/".

In order to distinguish Entity Types from Fact Types, some tags are appended to the prefix. In the example below we have the TYPE tag for the object types Call, Cell and Phonepoint and the tag PRED for the fact types origin and morigin. As for the multitree nodes, the tag PROJ indicates a node of the multitree (the tag PROJ stands for "projections", referring to the set of attributes in the project signature graph) and Q is for the edges. Each OWL class with the tag PROJ is based on the pre-computation of the multitree performed in the previous step. Among the declared OWL classes there are some with the tag UNIQ to optimize the reasoning calculations to detect possible uniqueness constraints (this will be explained in the next section).

```
### ENTITY TYPES
Declaration(Class(<http://www.ormie.org/#TYPE-Call>))
Declaration(Class(<http://www.ormie.org/#TYPE-Cell>))
Declaration(Class(<http://www.ormie.org/#TYPE-Phonepoint>))
### FACT TYPES
Declaration(Class(<http://www.ormie.org/#PRED-morigin-{1,2}>))
Declaration(Class(<http://www.ormie.org/#PRED-morigin-{1}>))
Declaration(Class(<http://www.ormie.org/#PRED-morigin-{2}>))
Declaration(Class(<http://www.ormie.org/#PRED-origin-{1,2}>))
Declaration(Class(<http://www.ormie.org/#PRED-origin-{1}>))
Declaration(Class(<http://www.ormie.org/#PRED-origin-{2}>))
### UNIQUE NODES
Declaration(Class(<http://www.ormie.org/#UNIQ-morigin-{1}>))
Declaration(Class(<http://www.ormie.org/#UNIQ-morigin-{2}>))
Declaration(Class(<http://www.ormie.org/#UNIQ-origin-{1}>))
Declaration(Class(<http://www.ormie.org/#UNIQ-origin-{2}>))
### MULTITREE TREE NODES
Declaration(Class(<http://www.ormie.org/#PROJ-morigin-{1,2}>))
Declaration(Class(<http://www.ormie.org/#PROJ-morigin-{1}>))
Declaration(Class(<http://www.ormie.org/#PROJ-morigin-{2}>))
Declaration(Class(<http://www.ormie.org/#PROJ-origin-{1,2}>))
Declaration(Class(<http://www.ormie.org/#PROJ-origin-{1}>))
Declaration(Class(<http://www.ormie.org/#PROJ-origin-{2}>))
```

```
### MULTITREE EDGES (PROPERTIES)
Declaration(ObjectProperty(<http://www.ormie.org/#Q-{1}>))
Declaration(ObjectProperty(<http://www.ormie.org/#Q-{2}>))
```

ORM constraints are generated following the Table 7.3. The axioms related to the example in Figure 7.8 are:

```
# FACTTYPE ORIGIN
SubClassOf(<http://www.ormie.org/#PRED-origin-{1,2}> ObjectAllValuesFrom(<
    http://www.ormie.org/#Q-{1}> <http://www.ormie.org/#TYPE-Call>))
SubClassOf(<http://www.ormie.org/#PRED-origin-{1,2}> ObjectAllValuesFrom(<
    http://www.ormie.org/#Q-{2}> <http://www.ormie.org/#TYPE-Phonepoint>))

# FACTTYPE MORIGIN
SubClassOf(<http://www.ormie.org/#PRED-morigin-{1,2}> ObjectAllValuesFrom(<
    http://www.ormie.org/#Q-{1}> <http://www.ormie.org/#TYPE-Call>))
SubClassOf(<http://www.ormie.org/#PRED-morigin-{1,2}> ObjectAllValuesFrom(<
    http://www.ormie.org/#Q-{2}> <http://www.ormie.org/#TYPE-Cell>))

# MANDATORY
SubClassOf(<http://www.ormie.org/#TYPE-Cell> <http://www.ormie.org/#PROJ-
    morigin-{2}>)

# ISA
SubClassOf(<http://www.ormie.org/#TYPE-Phonepoint> <http://www.ormie.org/#
    TYPE-Cell>)

# UNIQUENESS
SubClassOf(<http://www.ormie.org/#PROJ-morigin-{1}> <http://www.ormie.org/#
    UNIQ-morigin-{1}>)
SubClassOf(<http://www.ormie.org/#PROJ-origin-{1}> <http://www.ormie.org/#
    UNIQ-origin-{1}>)

# SUBSET
SubClassOf(<http://www.ormie.org/#PRED-morigin-{1,2}> <http://www.ormie.org
    /#PRED-origin-{1,2}>)
```

As stated before, additional axioms are needed to complete the $\mathcal{ALCQI}$ encoding.

- $\mathcal{DLR}^{\pm}$ functions: dsj, rel and obj
- $\mathcal{DLR}^{\pm}$ multitree
- "dummy" nodes for query optimization

A $\mathcal{DLR}^{\pm}$ ontology is also made by the axioms related to the functions *dsj*, *rel* and *obj* as in the $\mathcal{ALCQI}$ mapping in Section 5.1.5. The *dsj* function ensures that relations with different signatures are disjoint. The axioms generated by *rel* introduce classical reification axioms for each relation and its relevant

projections. The axioms in *obj* make sure that each local objectification differs from the global one.

Since the considered example has a subset constraint stating that all the pairs in *morigin* are also in *origin*, this implies that the two relationships are not disjoint and in the multitree they share the same attributes. In this case, there are no generated DSJ axioms but only REL ones.

```
# REL ORIGIN
SubClassOf(<http://www.ormie.org/#PRED-origin-{1,2}> ObjectSomeValuesFrom(<
    http://www.ormie.org/#Q-{1}> <http://www.ormie.org/#PRED-origin-{1}>))
SubClassOf(<http://www.ormie.org/#PRED-origin-{1,2}> ObjectSomeValuesFrom(<
    http://www.ormie.org/#Q-{2}> <http://www.ormie.org/#PRED-origin-{2}>))

# REL MORIGIN
SubClassOf(<http://www.ormie.org/#PRED-morigin-{1,2}> ObjectSomeValuesFrom(<
    http://www.ormie.org/#Q-{1}> <http://www.ormie.org/#PRED-morigin-{1}>))
SubClassOf(<http://www.ormie.org/#PRED-morigin-{1,2}> ObjectSomeValuesFrom(<
    http://www.ormie.org/#Q-{2}> <http://www.ormie.org/#PRED-morigin-{2}>))
```

In order to complete the $\mathcal{DLR}^{\pm}$ encoding, the axioms related to the multitree must be added to the ontology. Relying on the hashmap data structure used before to encode the $\mathcal{DLR}^{\pm}$ multitree, the Algorithm 1 shows the procedure to encode a $\mathcal{DLR}^{\pm}$ multitree into OWL.

Algorithm 1 $\mathcal{DLR}^{\pm}$ tree OWL mapping

```
private void generateDLRTreeForOWL()
    {
        //Variables declaration
        BitSet singletonBitset;
        String rn;
        ArrayList<String> listOfRoles;
        BitSet predicateBitset;
        ArrayList<BitSet> path;
        OWLClass proj;
        OWLClassExpression expression;
        OWLClassExpression predicate;

        //Let's iterate the multitree hashmap
        Iterator<Entry<ArrayList<String>, MultiTreeNode>> iterator = dlr.getMultiTree().
             entrySet().iterator();
        while(iterator.hasNext() )
        {
            Entry<ArrayList<String>, MultiTreeNode> entry = iterator.next();
            //the current tree node is a SINGLETON
            if( (entry.getValue().getType()==MultiTreeNodeType.SINGLETON) )
            {
                singletonBitset = entry.getValue().getBitset();
                rn = entry.getValue().getRn();
                listOfRoles = factTypesMap.get(rn).getRoles();
                predicateBitset = dlr.getMultiTree().get(listOfRoles).getBitset();
                path = dlr.path(predicateBitset, singletonBitset);

                //Defining the role data
                String key = entry.getKey().get(0);
                String idRole = key.substring(key.length()-1);
                predicate = factory.getOWLClass("PRED-" + rn + "-" +(factTypesMap.get(rn
                     ).getFullArity()), prefix);
                expression = dlr.getOWLSomeValuesFromComposition(path, predicate,"empty"
                     , 1);
                proj = factory.getOWLClass("PROJ-" + rn + "-{" + (idRole) + "}", prefix)
                     ;
                //writing the axiom in the ontology
                manager.addAxiom(ontology, factory.getOWLEquivalentClassesAxiom(proj,
                     expression));
            }

            //the current tree node is a PARTITION
            else if( (entry.getValue().getType()==MultiTreeNodeType.PARTITION) )
            {
                singletonBitset = entry.getValue().getBitset();
                rn = entry.getValue().getRn();
                listOfRoles = factTypesMap.get(rn).getRoles();
                predicateBitset = dlr.getMultiTree().get(listOfRoles).getBitset();
                path = dlr.path(predicateBitset, singletonBitset);
                predicate = factory.getOWLClass("PRED-" + rn + "-" +(singletonBitset.
                     toString().replaceAll("\\s+","")), prefix);
                expression = dlr.getOWLSomeValuesFromComposition(path, predicate,"empty"
                     , 1);
                proj = factory.getOWLClass("PROJ-" + rn + "-" + (singletonBitset.
                     toString().replaceAll("\\s+","")), prefix);
                //writing the axiom in the ontology
                manager.addAxiom(ontology, factory.getOWLEquivalentClassesAxiom(proj,
                     expression));
            }

            //the current tree node is the ROOT
            else if((entry.getValue().getType()==MultiTreeNodeType.ROOT)){
                rn = entry.getValue().getRn();
                predicate = factory.getOWLClass("PRED-" + rn + "-" +(factTypesMap.get(rn
                     ).getFullArity()), prefix);
                proj = factory.getOWLClass("PROJ-" + rn + "-" + (factTypesMap.get(rn).
                     getFullArity()), prefix);
                //writing the axiom in the ontology
                manager.addAxiom(ontology, factory.getOWLEquivalentClassesAxiom(proj,
                     predicate));
            }
        }
    }
```

The algorithm iterates over each entry of the multitree data structure in order to detect the node type (ROOT, PARTITION, SINGLETON). According to this parameter an axiom is generated to represent that node into the ontology. If the node is a partition or a singleton it involves an edge in the multitree, so according to the $\mathcal{DLR}^{\pm}$ mapping an OWL expression is created to compute the path. The $\mathcal{DLR}^{\pm}$ path function takes as input two sets of nodes and returns the path between them. Speaking in OWL terms, this is represented as a role chain composed by OWL properties where each property corresponds to the edge in the multitree. In order to optimize the reasoner tasks, an equivalence axiom is added to the ontology where the OWL class with the PROJ tag representing the node is equivalent to the OWL class expression. In this way the reasoner has to read only the class with the tag PROJ, instead of recomputing from scratch the OWL class expression. The computational cost of this operation is constant in time and done once during the ontology generation.

The axioms generated by Algorithm 1 are:

```
# TREE ROOT NODE origin
EquivalentClasses(<http://www.ormie.org/#PRED-origin-{1,2}> <http://www.
    ormie.org/#PROJ-origin-{1,2}>)

# TREE ROOT NODE morigin
EquivalentClasses(<http://www.ormie.org/#PRED-morigin-{1,2}> <http://www.
    ormie.org/#PROJ-morigin-{1,2}>)

# TREE SINGLETON NODE morigin.1
EquivalentClasses(<http://www.ormie.org/#PROJ-morigin-{1}>
    ObjectSomeValuesFrom(ObjectInverseOf(<http://www.ormie.org/#Q-{1}>) <
    http://www.ormie.org/#PRED-morigin-{1,2}>))

# TREE SINGLETON NODE morigin.2
EquivalentClasses(<http://www.ormie.org/#PROJ-morigin-{2}>
    ObjectSomeValuesFrom(ObjectInverseOf(<http://www.ormie.org/#Q-{2}>) <
    http://www.ormie.org/#PRED-morigin-{1,2}>))

# TREE SINGLETON NODE origin.1
EquivalentClasses(<http://www.ormie.org/#PROJ-origin-{1}>
    ObjectSomeValuesFrom(ObjectInverseOf(<http://www.ormie.org/#Q-{1}>) <
    http://www.ormie.org/#PRED-origin-{1,2}>))

# TREE SINGLETON NODE origin.2
EquivalentClasses(<http://www.ormie.org/#PROJ-origin-{2}>
    ObjectSomeValuesFrom(ObjectInverseOf(<http://www.ormie.org/#Q-{2}>) <
    http://www.ormie.org/#PRED-origin-{1,2}>))
```

The last set of axioms have the purpose to optimize the queries when the reasoner discovers possible inferred uniqueness constraints. In order to do this, additional axioms are provided in the following form:

```
# DUMMY NODE UNIQUENESS
EquivalentClasses(<http://www.ormie.org/#UNIQ-origin-{1}>
    ObjectExactCardinality(1 ObjectInverseOf(<http://www.ormie.org/#Q-{1}>)
    <http://www.ormie.org/#PRED-origin-{1,2}>))
EquivalentClasses(<http://www.ormie.org/#UNIQ-origin-{2}>
    ObjectExactCardinality(1 ObjectInverseOf(<http://www.ormie.org/#Q-{2}>)
    <http://www.ormie.org/#PRED-origin-{1,2}>))
EquivalentClasses(<http://www.ormie.org/#UNIQ-morigin-{1}>
    ObjectExactCardinality(1 ObjectInverseOf(<http://www.ormie.org/#Q-{1}>)
    <http://www.ormie.org/#PRED-morigin-{1,2}>))
EquivalentClasses(<http://www.ormie.org/#UNIQ-morigin-{2}>
    ObjectExactCardinality(1 ObjectInverseOf(<http://www.ormie.org/#Q-{2}>)
    <http://www.ormie.org/#PRED-morigin-{1,2}>))
```

The number of these axioms is equivalent to the total number of roles in the given ORM diagram. In this example we have 4 roles in total (origin.1, origin.2, morigin.1, morigin.2). The left side of each axiom has a dummy node marked as UNIQ, equivalent to the OWL class expression indicating the cardinality of the uniqueness.

The purpose of adding these axioms is to avoid recomputing the path function several times. Computing once the OWL class expression related to cardinality and the already computed node in the multitree (the one marked with PROJ), makes it easier for the reasoner to answer the query. When the time for verification comes, if the query axiom is entailed or not, the system already has the two OWL classes to test the axiom, instead of recalculating for each of them the paths in the multitree. In the next Section we provide the OWL mapping where we show this set of axioms.

This optimization is done in constant time during the ontology generation; the query itself is executed a number of times equal to the number of roles in the ORM diagram, so the query has a linear complexity.

7.3.4 Automated reasoning

After the ontology is generated, the reasoner performs a set of calculations in order to detect possible inferences. The reasoner used by UModel is FaCT++[47], made by the University of Manchester and published under

LGPL. FaCT++ is written in C++ and it is a based tableaux reasoner for OWL2-DL, quite popular for being highly optimised to compute ontologies in the $\mathcal{SROIQ}$ fragment and integrated also in the OWLAPI. Considering that the ontology is based on the $\mathcal{DLR}^{\pm}$ mapping the expressivity of ontologies generated by UModel will be in the $\mathcal{ALCQI}$ fragment.

Query formulation is slightly different from standard OWL ontologies where each relationship is restricted by OWL to be binary; this restriction simplifies the language, but it makes much more complicated to deal with n-ary relationships. Since the OWL encoding is based on $\mathcal{DLR}^{\pm}$ and ORM diagrams may deal with n-ary relationships as well, the ontology will be populated by the reified concepts. $\mathcal{DLR}^{\pm}$ mapping produces ontologies with several axioms and classes generated by the multitree, the $\mathcal{DLR}^{\pm}$ functions and also the dummy classes and axioms mentioned previously; on one hand the presence of these axioms is mandatory to maintain the consistency of the whole ontology, on the other hand they may generate several inferences that are not interesting for the final user because the focus is only the conceptual modelling perspective. The nature of a $\mathcal{DLR}^{\pm}$ ontology combined with the need to apply the automated reasoning over conceptual diagrams leads to the use of a set of procedures to filter only the inferences that are considered relevant for the conceptual modelling user's perspective.

We summarize the queries executed by the reasoner:

1. object type hierarchy
2. fact type hierarchy
3. unsat object types
4. unsat fact types
5. inferred IsA between object types
6. inferred IsA between fact types
7. disjointness among object types
8. disjointness among fact types

9. equivalence among object types
10. equivalence among fact types
11. inferred mandatory
12. inferred uniqueness

In Table 7.4 we show the queries that UModel asks to the Fact++ reasoner. Since we are using the $\mathcal{DLR}^{\pm}$ encoding, we recall the usage of the † function which maps each concept name *CN* and each relation name *RN* appearing in the $\mathcal{DLR}^{\pm}$ KB to the $\mathcal{ALCQI}$ concept names and relation names.

Table 7.4: UModel - Queries

Subtype($T_1 \dots T_m$) T KB $\stackrel{?}{\models} (T_1^\dagger \sqsubseteq T) \dots (T_m^\dagger \sqsubseteq T)$	
ExclusiveSubtypes($T_1 \dots T_m$) T KB $\stackrel{?}{\models} T_i^\dagger \sqsubseteq \neg\ T_j^\dagger$	for $1 \leqslant i,j \leqslant m$: $i \neq j$
Mandatory(T $P_1.i_1 \dots P_m.i_m$) KB $\stackrel{?}{\models} T^\dagger \sqsubseteq$ PROJ-P_j-i_j	for $j \neq k$ and $j,k \leqslant m$: $P_j \neq P_k$
Unique($P.i_1 \dots P.i_m$) KB $\stackrel{?}{\models}$ PROJ-P-i $\sqsubseteq$ UNIQ-P-i	for $j \neq k$ and $j,k \leqslant m$: $i_j \neq i_k$
Subset(($P_1.i_1$ $P_2.h_1$) $\dots$ ($P_1.i_m$ $P_2.h_m$)) KB $\stackrel{?}{\models}$ PROJ-P_1-$\{P.i_1 \dots P.i_m\} \sqsubseteq$ PROJ-P_2-$\{P.h_1 \dots P.h_m\}$	$P_1 \neq P_2$ and for $j \neq k$ and $j,k \leqslant m$ **:** $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$
Exclusive(($P_1.i_1$ $P_2.h_1$) $\dots$ ($P_1.i_m$ $P_2.h_m$))	$P_1 \neq P_2$ and for $j \neq k$ and $j,k \leqslant m$ **:** $P_1.i_j \neq P_1.i_k$ and $P_2.h_j \neq P_2.h_k$

$$\text{KB} \overset{?}{\models} \text{PROJ-P}_1\text{-}\{\text{P.i}_1 \dots \text{P.i}_m\} \sqsubseteq \neg\text{PROJ-P}_2\text{-}\{\text{P.h}_1 \dots \text{P.h}_m\}$$

The queries may involve the OWL classes with the PROJ prefix which are the ones representing the multitree $\mathcal{DLR}^{\pm}$ nodes. Some queries are simplified for better performance: for example, the uniqueness constraint query is limited to the single role and not for the spanning roles, in order to avoid all possible permutations. The same applies to the mandatory constraint where only the simple mandatory is considered (i.e., a single role). Calculating an inferred exhaustive constraint means to check, for each single entity type in the diagram, all possible subsets of entity types making the covering, resulting in a huge amount of calculations significantly slowing down the reasoning execution. For this reason, the reasoning procedure does not compute derived exhaustive constraints.

OWLAPI comes with a native function to precompute the class hierarchy, but this produces several unwanted OWL classes (from the final user perspective) because of the multitree axioms and the set of additional axioms added for performance reasons (e.g., the ones with the UNIQ tag). In any case, multitree axioms are needed for the ontology and the automated reasoning, but since they are not interesting for the final user they do not have to be displayed in the output, although they take part to the ontology composition and so the reasoning. The purpose of hierarchy is to reflect the conceptual model hierarchy with possible inferred IsA or Subset ORM constraints. To solve this problem, two temporary ontologies are built during the ontology generation step. They are made by sets of IsA and Subset constraints given in the input ORM model. In this way, the hierarchies for object types and fact types are preserved and memorized in constant time. These two hierarchies are distinct and they will be computed separately generating two temporary ontologies that will share data with the current ontology that has all the needed information to retrieve possible inferred constraints. With this combination the position of each element in the tree is preserved and the inferred elements in the hierarchy (e.g., an inferred IsA) are positioned in the right place.

The need of this technique is explained in Figure 7.10, where the ontology corresponding to the ORM diagram in Figure 7.8 is visualized inside Protégé. We know that Phonepoint is a subclass of Cell, but in the ontology we can see that Cell is a subclass of PROJ-morigin2. This is correct from the pure ontology perspective according to the $\mathcal{ALCQI}$ mapping, but this output is not suitable for the conceptual modelling perspective and it may be confusing for the final user. For this reason, while the ontology remains the same, we encode the hierarchies in additional data structures that will be used later in order to populate the graphical interface.

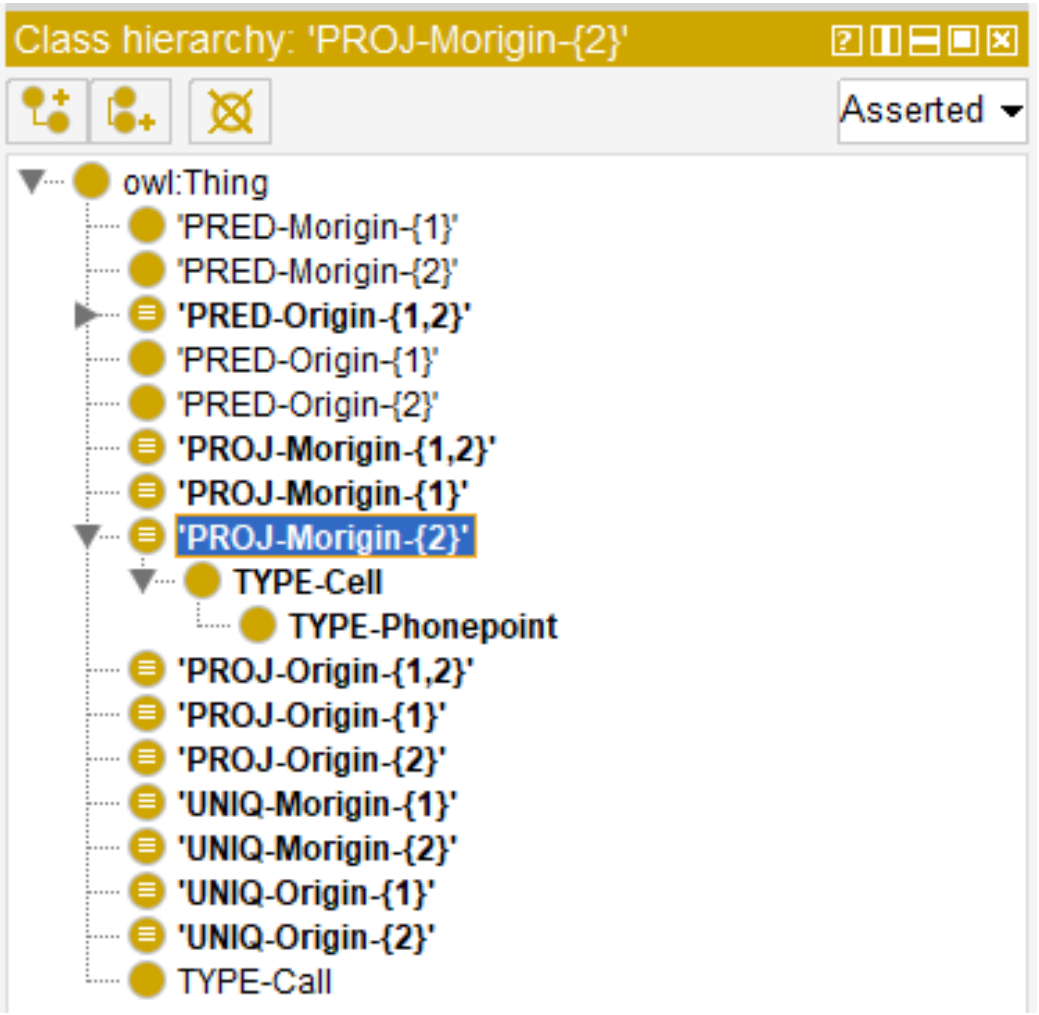


Figure 7.10: Ontology related to Figure 7.8 in Protégé

The whole procedure is shown in Algorithm 2.

Algorithm 2 Hierarchy building for object types

```
private void buildHierarchyForObjectTypes(OWLReasoner reasonerType, OWLClass clazz, Set<
    OWLClass> visited, SortedNode node) {
        // A convenient way to traverse each node once
        if (!visited.contains(clazz)) {
            visited.add(clazz);

            //Asserted ISA - Get the direct children of the current node
            NodeSet<OWLClass> subClasses = reasonerType.getSubClasses(clazz, true);
            for (OWLClass child : subClasses.getFlattened()) {
                if(child.toString().contains("TYPE-"))
                {
                    SortedNode currentChild = new SortedNode(owlToString(child.toString
                        ()));
                    node.add(currentChild);
                    //Recursive procedure using the temporary ontology
                    buildHierarchyForObjectTypes(reasonerType, child, visited,
                        currentChild);
                }
            }

            //Inferred ISA
            NodeSet<OWLClass> subClassesDerived = reasoner.getSubClasses(clazz, true);
            for (OWLClass child : subClassesDerived.getFlattened()) {
                if(child.toString().contains("TYPE-") && reasoner.isSatisfiable(child) )
                {
                    //Test the ISA axiom to verify if it is inferred
                    OWLAxiom subAxiomTest = UORMModel.factory.getOWLSubClassOfAxiom(
                        child, clazz);
                    if(!ontology.containsAxiom(subAxiomTest) && subClassesDerived.
                        containsEntity(child) && !clazz.isTopEntity())
                    {
                        SortedNode currentChild = new SortedNode(owlToString(child.
                            toString()));
                        node.add(currentChild);
                    }
                }
            }
        }

        //Compute Unsat
        if(!reasoner.isSatisfiable(clazz))
        {
            String unsatName = owlToString(clazz.toString());
            udmodel.getUnsatNodes().add(node);
        }

        //Compute  EQUIV - DISJ - MAND
        else
        {
            computeEquivForTypes(clazz, node);
            computeDisjointForTypes(clazz, node);
            computeMandatory(clazz);
        }
    }
```

This procedure builds the hierarchy recursively. Moreover, after all the descendants of the current node (which is an OWL Class) have been computed, a sequence of algorithms is performed to detect additional inferences (equivalence, disjointness, etc.). This is a convenient way to take advantage of this iteration in order to avoid re-scanning the whole OWL classes to find the inferences. In this procedure are also involved the multitree nodes (the

ones with the PROJ tag), according to the queries in Table 7.4. Thus, the current node is checked for its satisfiability inside the ontology; if the class is unsatisfiable there is no need to ask the reasoner additional queries over that class. If is satisfiable, then from that class possible equivalences, disjointnesses and mandatories are also calculated. Uniqueness is computed separately since it needs to scan the ontology to find all the OWL class with the tag PROJ, according to the related query in Table 7.4.

The reasoning procedure encodes the inferences in a set of data structure that can be easily queried via API 7.3.6. The main class where all inferences resides is UDerivedModel. ORM extends this class with its one Java class named UORMDerivedModel. This class is composed of a set of other classes where each one of them represent an inferred ORM constraint. Figure 7.11 shows the corresponding UML diagram:

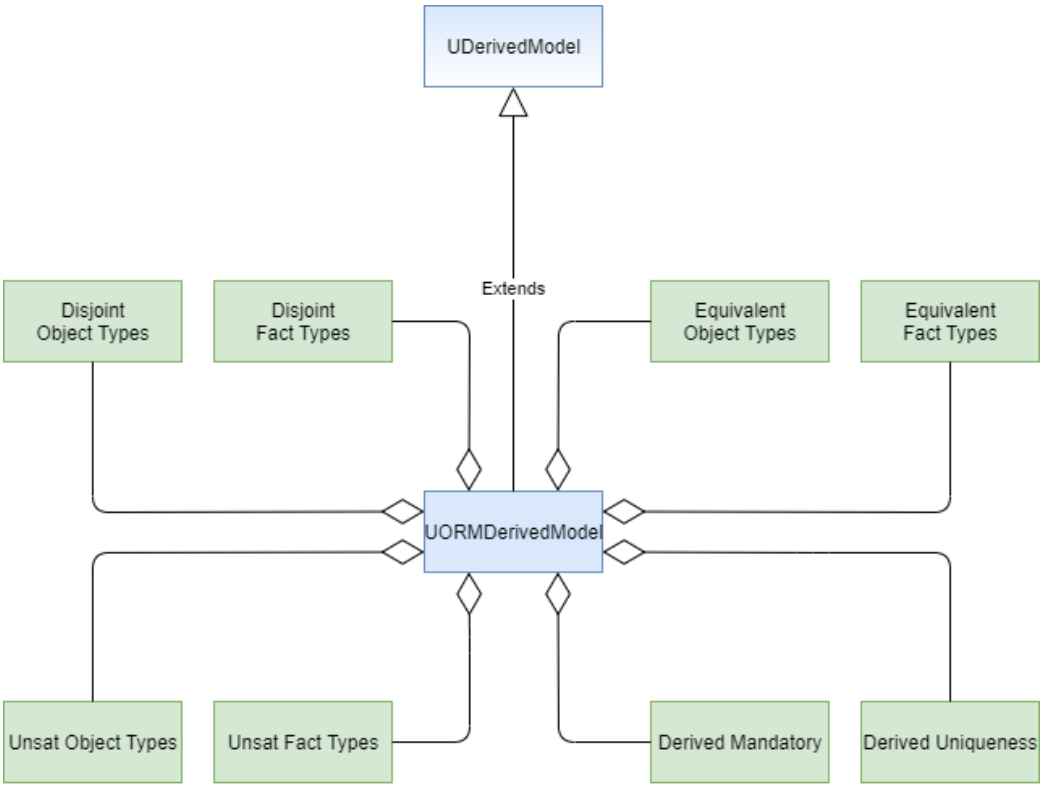


Figure 7.11: Derivation data structure in UModel

7.3.5 GUI

Once the inferences are collected in the aforementioned data structures, it is optionally possible to display them in a graphical interface as shown in Figure 7.12 and Figure 7.13.

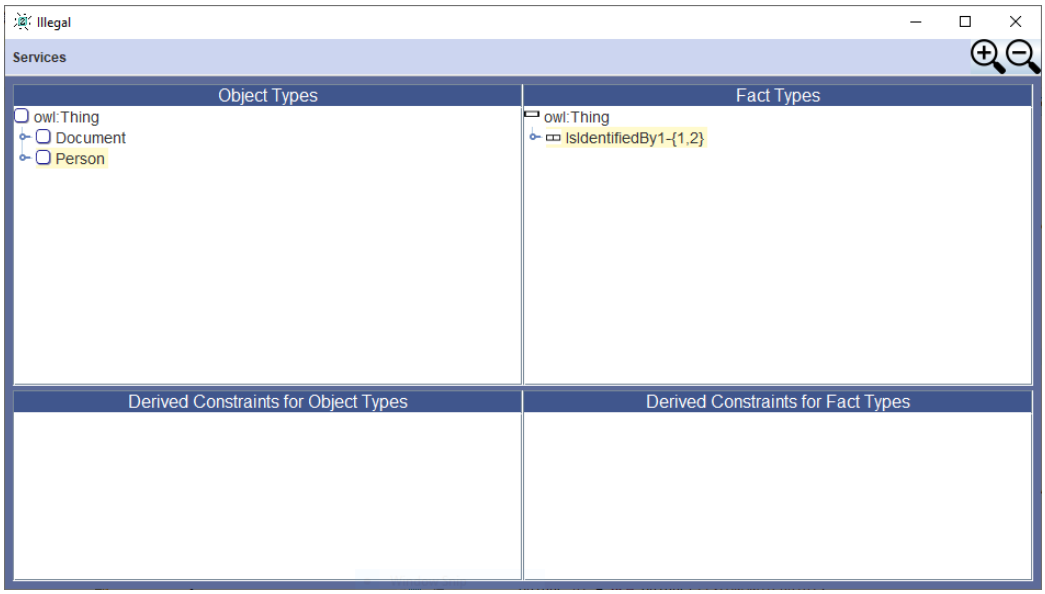


Figure 7.12: In yellow are highlighting the tree paths containing the inferences

The GUI has been designed to be partitioned into two sides: the left side for the object types information and the right side for the fact types. The top horizontal panel contains the hierarchies for both object and fact types; the bottom horizontal panel contains the relevant derivations. In the hierarchy, the derivations are marked with green, or in red to indicate that an object type or fact type is unsatisfiable. In order to ease the reading for large diagrams, the tree paths containing the relevant inferences are highlighted in yellow. In this way the user can quickly locate the relevant inferences expanding the yellow nodes. This feature is particularly useful for large diagrams where the user has to scroll down and perform numerous paths. In this way, we are guiding the user directly to the relevant inferences without wasting time by clicking every single node. When a node containing an inference is clicked more information appear inside the bottom panel: the inferred constraints are marked in green and the asserted ones in purple. The design choices made

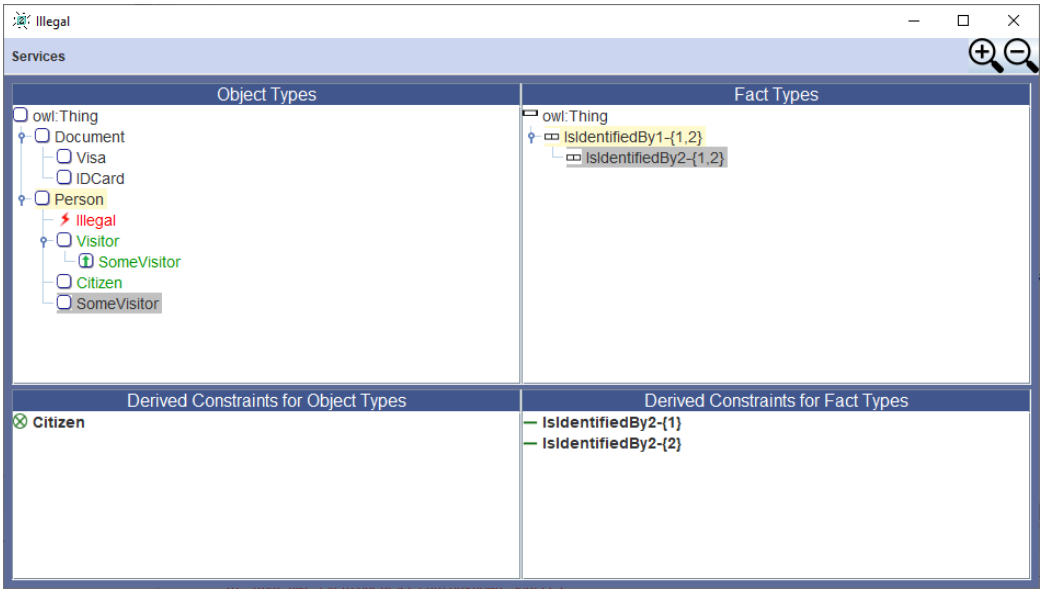


Figure 7.13: Expanding the tree paths to see the inferences

for the object type are the same for the fact types. The inconsistencies are marked in red (e.g., Illegal); the ISA derived are marked in green with an arrow in the icon (e.g., SomeVisitor).

The GUI is also equipped with a menu bar on the top side. The label named *Services* gives access to a menu where it is possible to export the OWL to a file or sync the OWL output with Protégé (this feature is quite useful for ontology engineers and/or for any debug reason).

7.3.6 The API system

UModel has been designed to expose a set of API commands to create conceptual diagrams and to manage them; in this way, the developer can benefit from the API system to easily integrate UModel in any external software, or to build on the fly some conceptual models and perform reasoning operation over them. The API system consists of a set of Java objects that must be called in a precise order, consequently the procedure is divided in 5 steps:

1. Init model - Instantiating an empty model that could be one in ORM, ER or UML one.

2. Create constraints - A conceptual diagram is made up of a set of constraints, therefore each constraint in the diagram must be declared in this step. Some constraints are dependent on others, for example the IsA must have two existing Object Types.

3. Assert constraints in the model - In this step the constraints are asserted into the model. Precisely, the Java objects representing the constraints are aggregated into the main Java data structure representing the diagram, namely UORMModel.

4. Call the reasoner - The reasoner now takes the created model as an input in order to generate the inferences.

5. Read inferences - In the last step all the inferences are encoded in a data structure ready to be queried by the user.

The first step is to encode the conceptual model. The following example aims to represent via API the ORM conceptual diagram in Figure.7.14.

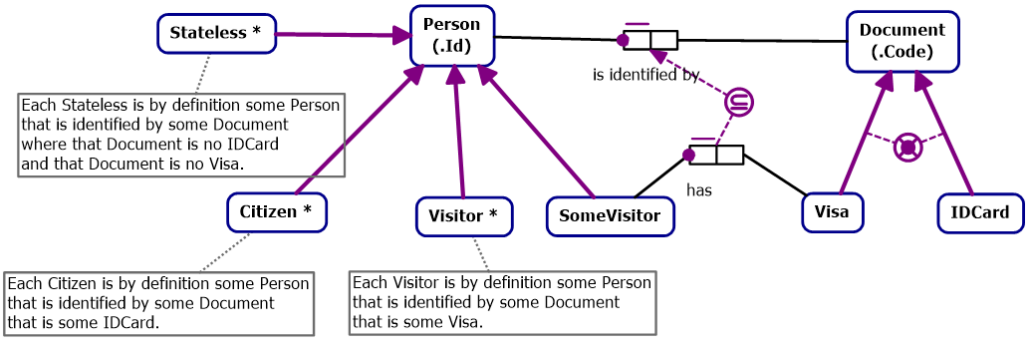


Figure 7.14: Visitor ORM diagram

We first need to create the structure for the model with the command:

```
UORMModel model = new UORMModel();
```

The UORMModel Java class is the master data structure to deal with ORM. The variable *model* will be used to encode a set of ORM constraints.

```
//Object Types
ORMEntityType stateless = new ORMEntityType("Stateless");
ORMEntityType person = new ORMEntityType("Person");
ORMEntityType document = new ORMEntityType("Document");
ORMEntityType someVisitor = new ORMEntityType("SomeVisitor");
ORMEntityType visa = new ORMEntityType("Visa");
ORMEntityType idCard = new ORMEntityType("IDCard");
ORMEntityType visitor = new ORMEntityType("Visitor");
ORMEntityType citizen = new ORMEntityType("Citizen");

//Fact types
ORMFactType isIdentifiedBy = new ORMFactType("IsIdentifiedBy1", new ArrayList<
    ORMEntityType>(Arrays.asList(person,document)));
ORMFactType has = new ORMFactType("has", new ArrayList<ORMEntityType>(Arrays.asList(
    someVisitor,visa)));
```

ORMEntityType is the Java structure to create an object type taking as input the name of the entity (e.g., Stateless, Person, etc.). ORMFactType creates fact types taking as input the name of the predicate (e.g., is identified by, has) and a list of ORMEntityType, as it is in the ORM syntax.

We can now define the ISA relationships between object types and fact types. The SubtypeOf Java class encodes the ISA relationships between two object type, where the first one is the subclass and the second one is the superclass. The SubsetOf Java class works in the same way, with the difference being that the arguments are collections of roles belonging to the fact types involved in the subset constraint. The roles are obtained by calling the function getRole(arity) on the previously defined ORMFactType class. In this example we are interested in the full arity of both relationships, so we take into account all the roles (the first and the second one for both fact types). The first role has index set to zero.

```
//ISA
SubtypeOf somevSubPerson = new SubtypeOf(someVisitor, person);
SubtypeOf visaSubDoc = new SubtypeOf(visa, document);
SubtypeOf idcardSubDoc = new SubtypeOf(idCard, document);
SubtypeOf statelessSubPerson = new SubtypeOf(stateless, person);
SubtypeOf visitorSubPerson = new SubtypeOf(visitor, person);
SubtypeOf citizenSubPerson = new SubtypeOf(citizen, person);

//Subset
ArrayList<ORMRole> roles1 = new ArrayList<ORMRole>();
ArrayList<ORMRole> roles2 = new ArrayList<ORMRole>();
roles1.add(isIdentifiedBy.getRole(0));
roles1.add(isIdentifiedBy.getRole(1));
roles2.add(has.getRole(0));
roles2.add(has.getRole(1));
SubsetOf subset = new SubsetOf(roles2, roles1);
```

In the model we also have the mandatory and uniqueness constraints. Since they deal with roles, they follow the same logic as subset constraints.

```
//Uniqueness
ArrayList<ORMRole> listORMRoleuniq1 = new ArrayList<ORMRole>();
listORMRoleuniq1.add(isIdentifiedBy.getRole(0));
Unique u1 = new Unique(listORMRoleuniq1);
```

```
ArrayList<ORMRole> listORMRoleuniq2 = new ArrayList<ORMRole>();
listORMRoleuniq2.add(isIdentifiedBy.getRole(1));
Unique u2 = new Unique(listORMRoleuniq2);

//Mandatory
ArrayList<ORMRole> listORMRoleMand = new ArrayList<ORMRole>();
listORMRoleMand.add(isIdentifiedBy.getRole(0));
Mandatory mand = new Mandatory(person, listORMRoleMand);

ArrayList<ORMRole> listORMRoleMand2 = new ArrayList<ORMRole>();
listORMRoleMand2.add(has.getRole(0));
Mandatory mand2 = new Mandatory(someVisitor, listORMRoleMand2);
```

Finally, set operators to represent the disjointness and the covering.

```
//Exclusive set
ExclusiveTypes visaEXidcard = new ExclusiveTypes(new ArrayList<ORMEntityType>(Arrays.
    asList(visa, idCard)));

//Covering
ExhaustiveTypes visaIdcardCOVDocument = new ExhaustiveTypes(new HashSet<ORMEntityType>(
    Arrays.asList(visa, idCard)), document);
```

Since this ORM diagram is equipped with ORM Derivation Rules, we need to add them in the ORM model. The Java data structures used to built the ORM Derivation Rules are based on the syntax defined in [128].

```
//DERIVATION RULES
//Visitor rule
SubtypeDerivationRule visitorRule = new SubtypeDerivationRule();
visitorRule.addLeftSubclass(visitor);
visitorRule.addRightSuperclass(person);
DRJoin visitor_join = new DRJoin(isIdentifiedBy, person, document);
DRPath isIdentifiedByDocument_path = new DRPath(visitor_join, new DRPath(document));
DRPath visa_path = new DRPath(visa);
isIdentifiedByDocument_path.setPath(visa_path);
visitorRule.addRuleBody(isIdentifiedByDocument_path);

//Citizen rule
SubtypeDerivationRule citizenRule = new SubtypeDerivationRule();
citizenRule.addLeftSubclass(citizen);
citizenRule.addRightSuperclass(person);
DRJoin citizen_join = new DRJoin(isIdentifiedBy, person, document);
DRPath citizen_path = new DRPath(citizen_join, new DRPath(idCard));
citizenRule.addRuleBody(citizen_path);

//Stateless rule
SubtypeDerivationRule statelessRule = new SubtypeDerivationRule();
ArrayList<DRPath> listOfPaths = new ArrayList<DRPath>();
statelessRule.addLeftSubclass(stateless);
statelessRule.addRightSuperclass(person);
DRJoin stateless_join = new DRJoin(isIdentifiedBy, person, document);
DRPath statelessIsIdentifiedByDocument_path = new DRPath(stateless_join, new DRPath(
    document));
DRPath notIDCard_path = new DRPath(Operator.NOT, new DRPath(idCard));
DRPath notVisa_path = new DRPath(Operator.NOT, new DRPath(visa));
listOfPaths.add(notIDCard_path);
listOfPaths.add(notVisa_path);
statelessIsIdentifiedByDocument_path.setPath(new DRPath(Operator.AND, listOfPaths));
statelessRule.addRuleBody(statelessIsIdentifiedByDocument_path);
```

Three rules are defined: Visitor, Citizen and Stateless. Each rule reuses the same variables defined in the code; additionally, it has dedicated data structures the encode ORM Derivation Rules (SubtypeDerivationRule for subtype rule and FactTypeDerivationRule for fact types). The recursive

structure of the rule is represented by the *DRPath* class which is defined recursive in the syntax. A rule consists of a left and the right-hand side plus the body where a set of constraints are defined.

The next step is to assign each of this constraints to the model:

```
//Assert the constraint into the model
model.tell(idCard);
model.tell(document);
model.tell(visa);
model.tell(someVisitor);
model.tell(person);
model.tell(stateless);
model.tell(visitor);
model.tell(citizen);
model.tell(somevSubPerson);
model.tell(visaSubDoc);
model.tell(idcardSubDoc);
model.tell(isIdentifiedBy);
model.tell(has);
model.tell(visaEXidcard);
model.tell(visaIdcardCOVDocument);
model.tell(u1);
model.tell(u2);
model.tell(mand);
model.tell(mand2);
model.tell(subset);
model.tell(visitorRule);
model.tell(citizenRule);
model.tell(statelessRule);
model.tell(statelessSubPerson);
model.tell(visitorSubPerson);
model.tell(citizenSubPerson);
```

The *tell* function asserts the constraints defined in the argument. In this step, the model is complete since it has all the constraints inside; this means we can go on calling the reasoner to compute the inferences.

```
//CALL THE REASONER
UReasoner reasoner = new UReasoner(model);
reasoner.start();
```

We have created an instance of the Java class, UReasoner, with the only argument the model previously defined. The constructor of this class translates the ORM model into OWL and runs a set of reasoning algorithms (that are transparent to us, since we are playing the role of the developer). The reasoner is executed when the function *start* is called.

Now we are interested in reading the result of the computation.

```
//GET INFERENCES
ORMDerivedModel udmodel = reasoner.getInferredModel();
udmodel.getUnsatObjectTypes();
System.out.println(udmodel.getUnsatObjectTypes().getEmptySet());
```

The reasoner has a function named getInferredModel which return the Java structure ORMDerivedModel, that is the one where all the inferences are stored. For simplicity we have created a variable named *udmodel* to get the

references to the inferences. Now we can access all the functions from the object ORMDerivedModel, in order to retrieve the information we need. In the example we are interested in the unsatisfiable object types.

Optionally, we can launch the GUI to visualize the relevant inferences. For example, Figure 7.15 shows the GUI suggesting to expand the tree node Person, because under this node some inferences have been computed. Figure 7.16 highlights the inferences under Person node and here we can find that Stateless is marked in red because it is inconsistent.

```
//Optionally, display the gui
GUI gui = new GUI();
GuiLauncher guiLauncher = new GuiLauncher();
guiLauncher.launchGui(gui, udmodel, "Stateless");
```

and we obtain:

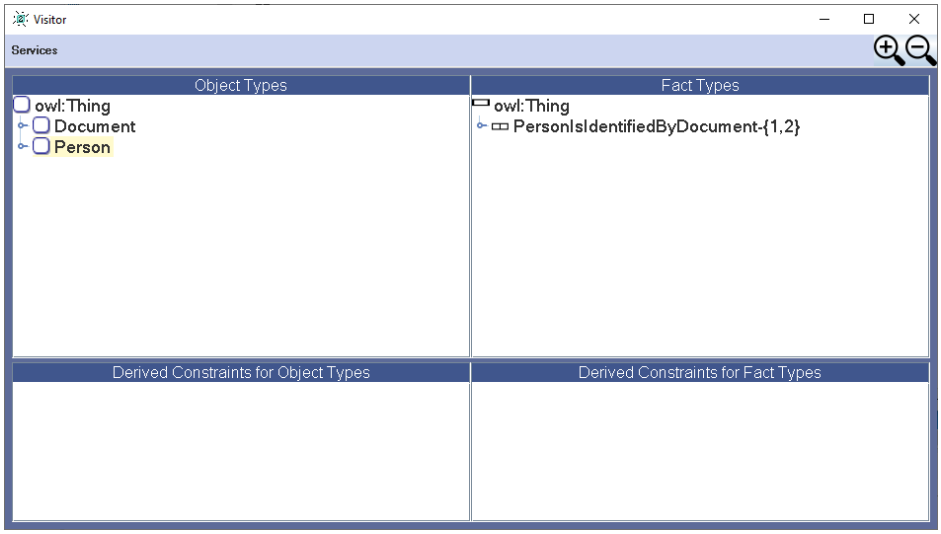


Figure 7.15: Highlighted in yellow the tree node containing the inferences

7.4 Future works

Further improvements and research tracks can be made in a future version of the UModel framework development:

- Multithreading support - Although performances are considered good in a real industrial-case scenario (see Chapter 7), a further improvement

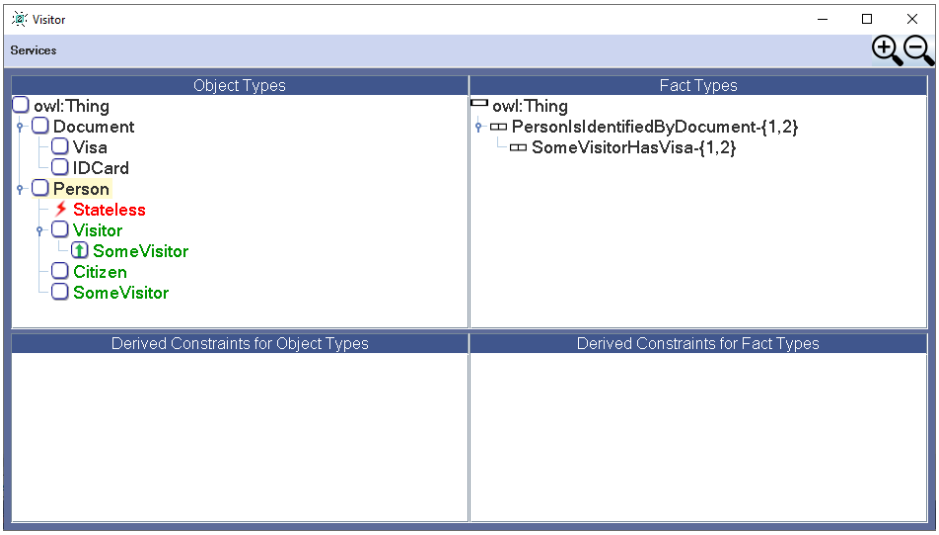


Figure 7.16: The inferences in the expanded node

consists in supporting the multithreading in the automated reasoning procedures. At the current stage of development UModel performs the reasoning tasks sequentially, so each time a query is sent to the reasoner the system has to await the result. The benefit coming from the multithreading may result in a dramatic increment of performance because of a set of threads, where each one asks a query and all are executed concurrently. In order to implement this feature, some parts of the code must be carefully redesigned.

- Explanatory - Detecting formal properties may be useful during the modelling step. For example, if the modeller reads an unexpected inference, he should try to understand the reason behind that in order to fix possible issues, but this could be a time consuming activity especially for large diagrams. The adoption of an explanatory service could significantly help the modeller to trace the selected inference, since there is no indication about the tracking of that particular inference in order to understand the source of possible issues. Equipping the reasoner with an explanatory service may surely increase the control over the diagrams and it can also speed up the procedure of fixing the models in case of mistakes. An approach used to explain how inferences have been obtain is the usage of axiom pinpointing [119]; this research

track requires the integration of this technique inside the framework. An alternative approach is to take advantage of OWLAPI, since it comes with an integrated explanatory service, as in [129].

- UML and ER support - Another research track is to enable the automated reasoning for this UML and ER; this means to encode them into $\mathcal{DLR}^{\pm}$.

- Further integrations in other CASE tools - UModel has been used in the NORMA tool (see Chapter 7), but it can also be easily integrated into any CASE tool.

8

ORMIE

In this chapter we integrate UModel into the CASE tool NORMA. NORMA is a software for ORM conceptual modelling that can be easily extended by plugins. For this reason, a plugin named ORMiE (ORM Inference Engine) has been developed in order to extend NORMA. The purpose of ORMiE is to enable the automated reasoning over those ORM diagrams loaded inside NORMA and show the inferences to the user. In order to achieve this, the UModel framework has been integrated into ORMiE [125], [56]. We recall that NORMA is an extension of Microsoft Visual Studio, which is one of the most popular IDE with many beneficial features. The purpose of NORMA is to extend Microsoft Visual Studio, allowing the user to manage ORM diagrams inside Visual Studio.

The chapter starts with the overview of the ORMIE tool, then we go in deeper with a system description, focusing on the architecture in order to identify the components and what roles they play in the infrastructure. We present some user scenarios where ORMiE is used to perform reasoning over some ORM models, after that we show the corresponding deductions in the ORMiE interface. We also present additional ORMiE features.

8.1 Overview

ORMiE is a conceptual modelling tool specifically designed to perform automated reasoning over ORM diagrams. The purpose of ORMIE is to help the modeller in checking the semantics of ORM conceptual diagrams, in order to fully control the workflow of the modelling phase. ORMIE works with a

background reasoner useful to verify the software specification, infer implicit facts, devise stricter constraints, and manifest any inconsistency. Usually ORM diagrams, especially the extended ones, must meet clear and measurable quality criteria. Large numbers of conceptual diagrams have, however, usually been developed in an ad hoc manner by domain experts, often with only a limited understanding of the semantic level. The full control of the semantic level, which usually is lacking in CASE tools, may result in a very critical problem and serious consequences for the whole infrastructure based on that software. The outcome is that a conceptual diagram could be of low quality and not accurate in the description of the domain. Moreover, many errors can occur where there is no control over the semantics, leading to a degradation of the software and increasing the development costs. This problem becomes even more acute as conceptual diagrams are maintained and extended over time, often by multiple authors. To overcome these problems, tools are needed that support the design and the development of the basic infrastructure for building, merging, and maintaining conceptual diagrams. The leverage of automated reasoning to support domain modelling is enabled by a precise semantic definition of all the elements of ORM diagrams, and constraints are internally translated into a logic formalism. In the context of ORM diagram design, ORMiE is useful to support the modeller during the early stages of the development in order to check the consistency of the diagram or, inter-domain diagrams. Moreover, the usage of complex automated reasoning tasks to deduce implied facts increases the chance to uncover some mistakes during the software development that could be hidden to the final user control. The reasoning applied to ORM involves and deduces even ORM constraints, as opposed to mere subsumption (classification) and consistency, making it more reliable for the modeller. Another key point is the reasoning which covers the ORM Derivation Rules, special ORM constructs which are able to express knowledge that is beyond standard ORM capabilities. ORMiE supports the following: checking entity types and fact types consistency, subsumption, cardinality, exclusion, mandatory and uniqueness constraints, and in general discovering any implied but originally implicit ORM graphical construct. Customarily, ontology design tools just provide a support limited to class subsumption and consistency. ORMIE is also able to provide the

OWL counterpart of the diagram for ontology design and an integration with Protege.

8.2 System description

ORMIE is not intended to be a stand-alone software, but a way to integrate the UModel framework inside the NORMA tool. Since NORMA is a modular system, the user has the possibility to create plugins to extend the behaviour of NORMA. ORMIE is a plugin for NORMA that embodies UModel framework in order to activate automated reasoning over those ORM diagrams activated inside the NORMA tool.

To better understand how those components works together, we can see Figure 8.1.

The Visual Studio 2019 Community Edition is free of charge and can be installed in any Microsoft OS. It's enough for us to see Visual Studio as a rich container of features that can be inherited by any extension, like NORMA.

NORMA indeed is an extension of Visual Studio which means it automatically inherits all the nice features coming from Visual Studio and this is a great advantage since it improves the quality and the reliability of the system. NORMA is a software specifically designed to deal with ORM diagrams, which means it is possible to create, modify, delete ORM diagrams and benefits from many features that helps the modeller in managing any ORM diagram.

Another interesting feature of NORMA is the modular system, since it is possible to extend its functionalities by means of plugins. For this reason, ORMiE exists to add reasoning capabilities to NORMA. The benefit of enabling ORMiE in NORMA is the automated reasoning applied to any ORM diagram.

In Figure 8.2 we can take a deep look into the inner structure of ORMIE and observe the ORMIE components:

- code to integrate it inside NORMA infrastructure;

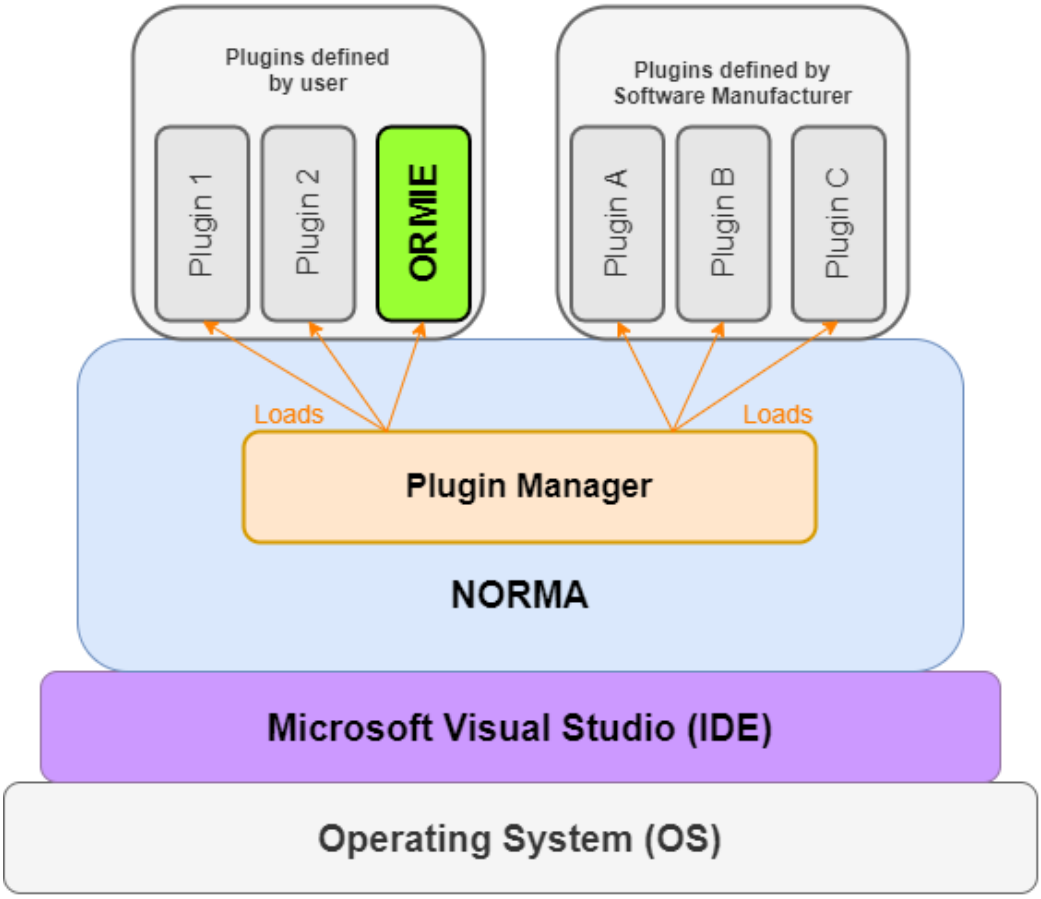


Figure 8.1: NORMA and ORMIE as Microsoft Visual Studio components

- a parser to capture every constraints of the given ORM diagram;

  each constraint is asserted into UModel;

- a module to call the reasoner from UModel;

- a module to call the ORMIE interface from UModel.

Essentially, ORMiE features a parser to detect the ORM constraints which become the input of UModel framework which does the rest of the job (reasoning and GUI generation).

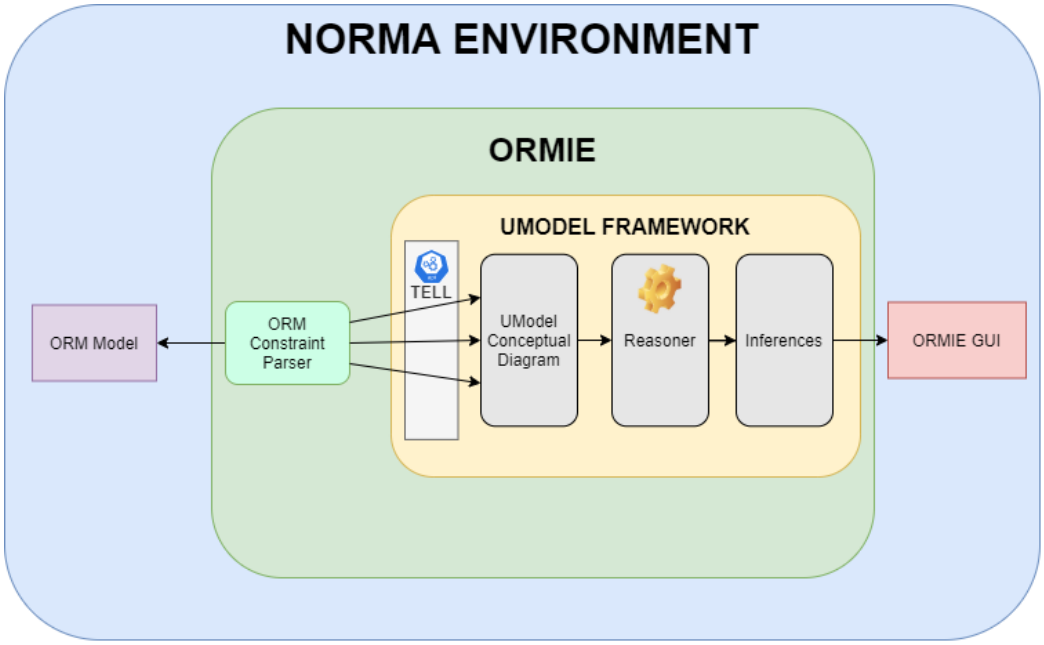


Figure 8.2: ORMIE Architecture

8.3 Example User Scenario

In this section we show how ORMiE works on a basic scenario, emphasising the added value of the main ORMiE functionalities. The inference engine helps the modeller with validating the ORM diagrams. In fact, if the derived constraints make little sense to the modeller, they may help in suggesting changes, or they may show conceptual mistakes. The advantage of this approach is that examples may be clear even to people who are not IT-experts due to their graphical nature and to the fact that ORM diagrams are quite intuitive. Complete reasoning over ORM diagrams supports the modeller in creating and maintaining ORM diagrams. The following example shows the kind of insights that a reasoning enabled system could provide during the ORM diagram modelling phase.

We consider the diagram in Figure 8.3 without reasoning.

After enabling ORMiE, thus the reasoning, the outcome is depicted in Figure 8.4:

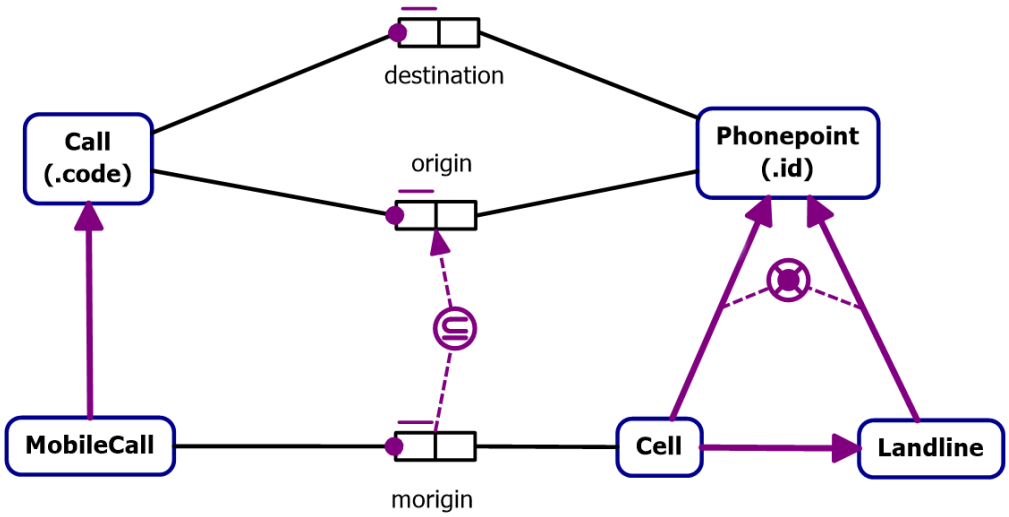


Figure 8.3: Without reasoning

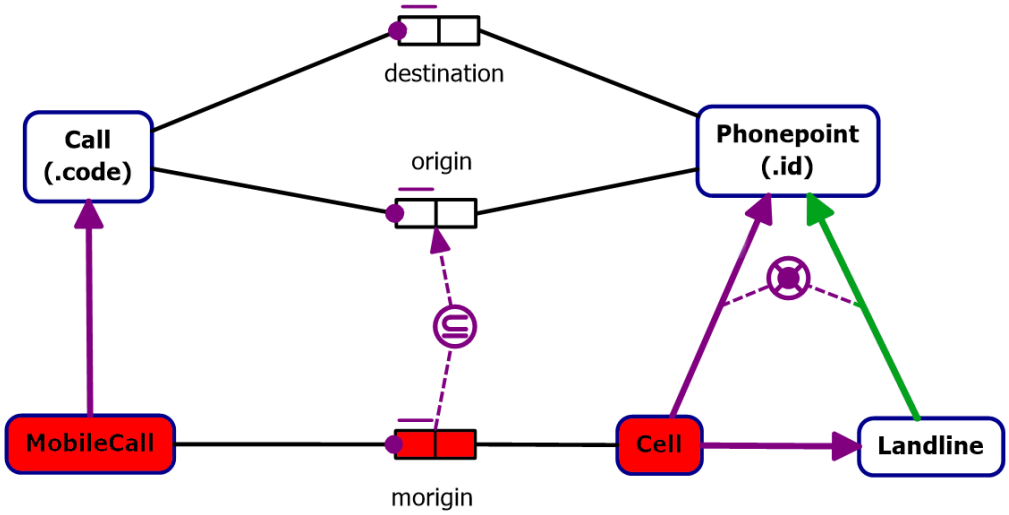


Figure 8.4: With reasoning

The results from the reasoning are displayed in the ORMiE graphical interface, as in Figure 8.5 and 8.3. The structure of the ORMIE interface is composed by three components: a top menu named *Services*; a centred block divided into two parts where the left side deals with the hierarchy for object types and the right one for the hierarchy of the Fact Types; the bottom block divided again into two parts, the left side to show the deductions referring to the Object Types and the right one for the deductions related to the Fact Types.

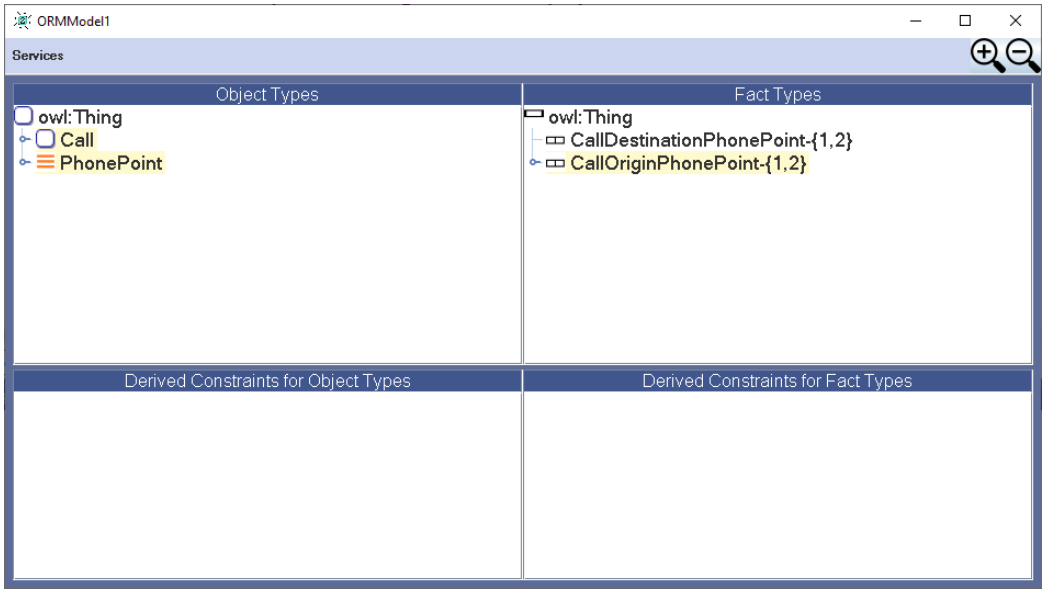


Figure 8.5: Reasoning outcome

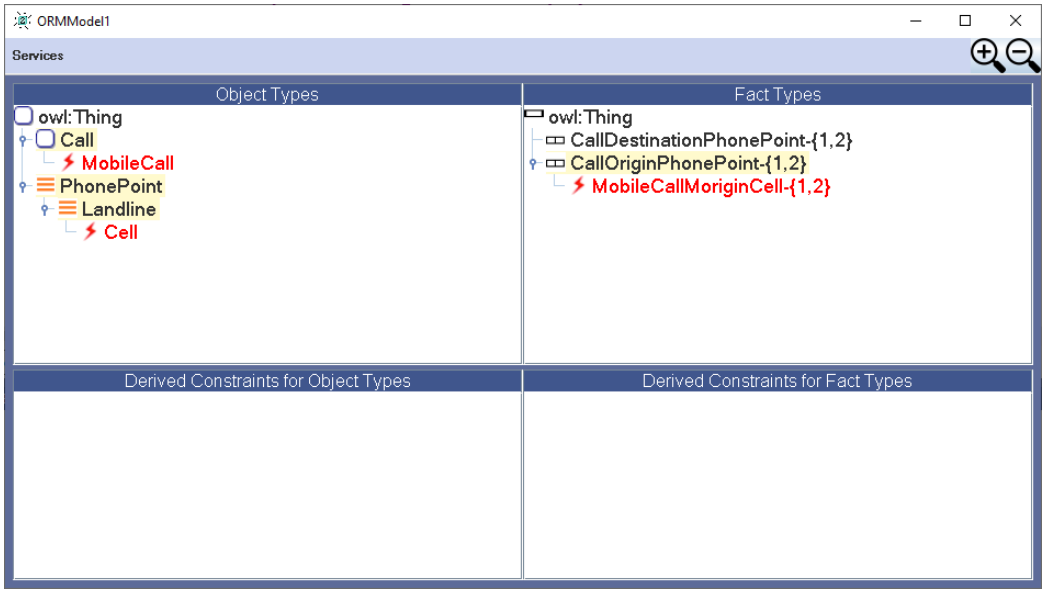


Figure 8.6: Reasoning outcome

The menu comes with different options where the user can export an ORM diagram into OWL. It's also possible to constantly sync the OWL counterpart of the ORM diagram if requested, so every time the ORM diagram is edited, a new OWL file is generated and overwrites the previous one. Moreover,

ORMIE is also able to detect if Protégé is installed on the system, in this case another feature comes to help the modeller that consists in opening the generated OWL file inside Protégé tool. This feature could be useful for ontology modelling or even for debug purposes (i.e., explanatory services in Protégé may help to detect the reason of a specific inference). Please note that the OWL output follows the mapping explained in Chapter 5, which may be not easy to read for the final user.

Below the menu, we have the blocks embedding the hierarchies. Each hierarchy (object types and fact types) is represented by a browsable tree structure, where object types and fact types are organized in tree nodes according to the hierarchy relationships; the nodes can be coloured in green or in red. If the reasoner says that a node is inconsistent, then it is highlighted in red; if the node representing an object type or fact type is involved in relevant inferences, then it is highlighted green. In the Hierarchy Types, the Object Types are represented by a box followed by the name of the object type; the fact types are represented by a sequence of tiny boxes according to the arity, plus the name of the fact type. The tool implements a methodology for handling very large conceptual diagrams and provides the user to easily find where the interesting inferences are, so the the interface has a design feature that suggest in which part of the tree an inferred object type (or fact type) is, by colouring all its ancestors in dark green and the inferred node in light green. The user can also interact with the hierarchy panels by clicking on a specific element, this is useful if the user may want to see its relevant properties in the corresponding bottom panel.

The bottom panels are dedicated to show the relevant inferences either for Object Types or Fact Types. For example, clicking on the Landline it says in the Object Type Derivation panel that, Landline is equivalent to Phonepoint. This is depicted as a yellow equivalence symbol in order to suggest the user that this is a warning since Landline and Phonepoint share the same instances. Therefore in a scenario where that diagram has a database counterpart, this is not an optimal way of modelling. In this case the modeller understands that, since Phonepoint is exactly the same as Landline, there is an error in the ORM diagram and this automatically suggests revision. The bottom

panels also shows the relevant and inferred exclusion constraints, mandatory and uniqueness constraints as well.

The ORMIE gui is automatically handled by the bottom panel of NORMA as in Figure 8.7.

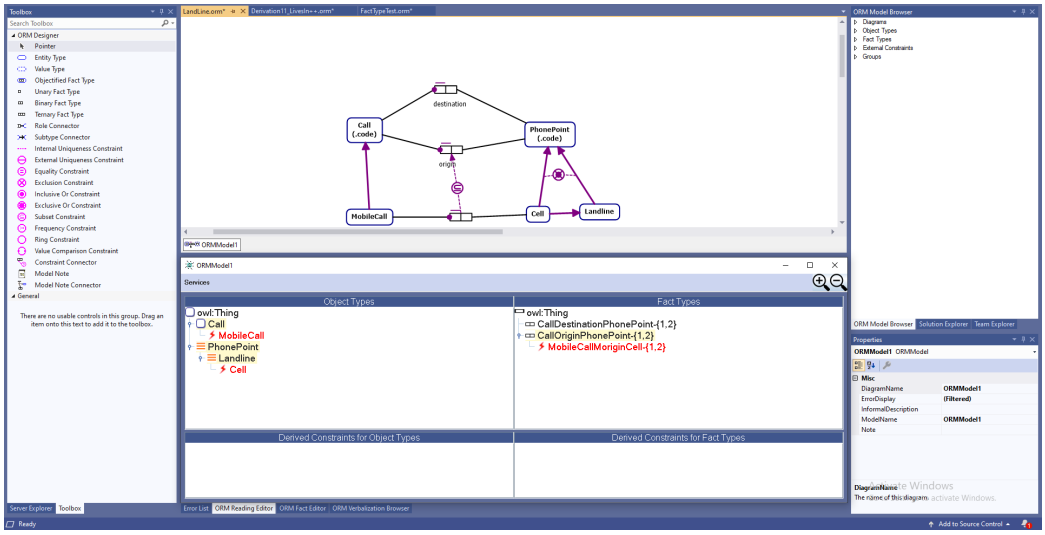


Figure 8.7: ORMIE gui inside NORMA

8.3.1 ORM diagrams Integration and Views

As seen in Section 2.4 and in Figure 2.20, NORMA can display ORM diagrams on different "pages". ORMiE is able to work even with those diagrams having multiple pages. This feature is useful to conceptually separate the sub-domains of the whole domain represented by the ORM diagram, in a way similar to ontology integration. A single page represents a subdomain and it is possible that some pages share some ORM constraints. An example is shown in Figures 8.8 and 8.9.

Page 1 depicts a situation where the main entity is Company. A company employs the employees and contacts some contact person; a company is also made of sectors. In this ORM diagram there is also a sub entity of Company, named Italian Company which represents the set of italian companies. This entity has a unary fact type placed on it, stating that an italian company may be classified as an ISO company.

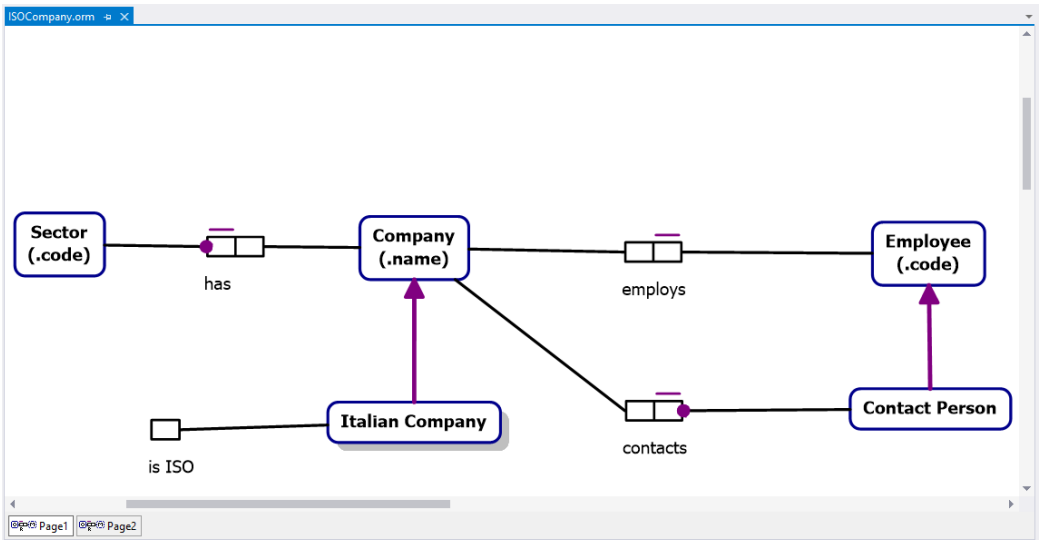


Figure 8.8: Page 1

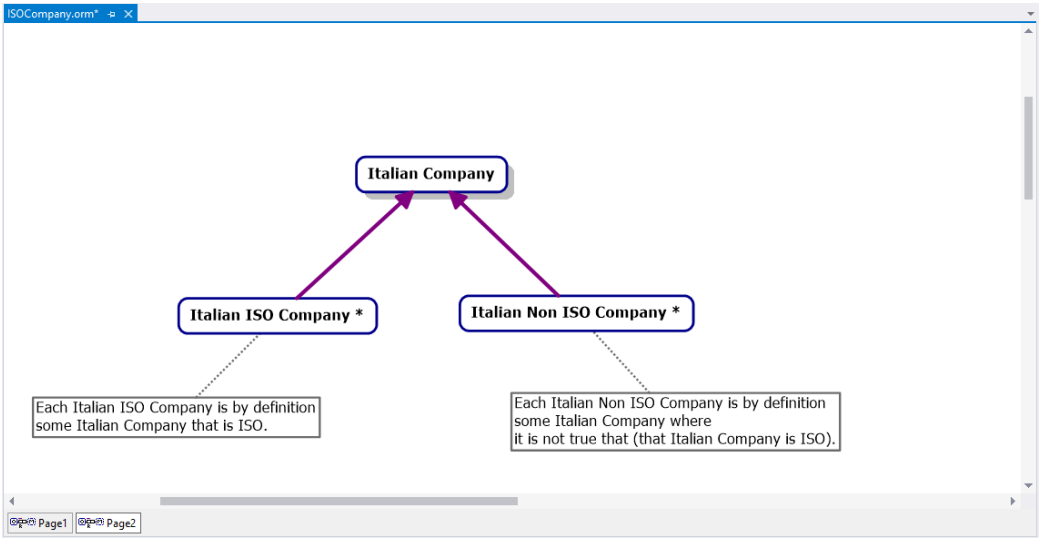


Figure 8.9: Page 2

Page 2 is dedicated to the subdomain of the italian companies, where these are simply divided into Italian ISO Company and Italian Non ISO Company. It is trivial to say that these two entities are disjoint and a simple disjoint ORM constraint would be enough to represent the disjointness. But, the ISA constraint is not enough to semantically capture all the italian ISO companies and the italian non ISO companies. For this reason this ORM diagram is

equipped with two derivation rules defining the constraints for both entities. The outcome of the reasoning is depicted in Figure 8.10, where there is a disjointness between the two entities.

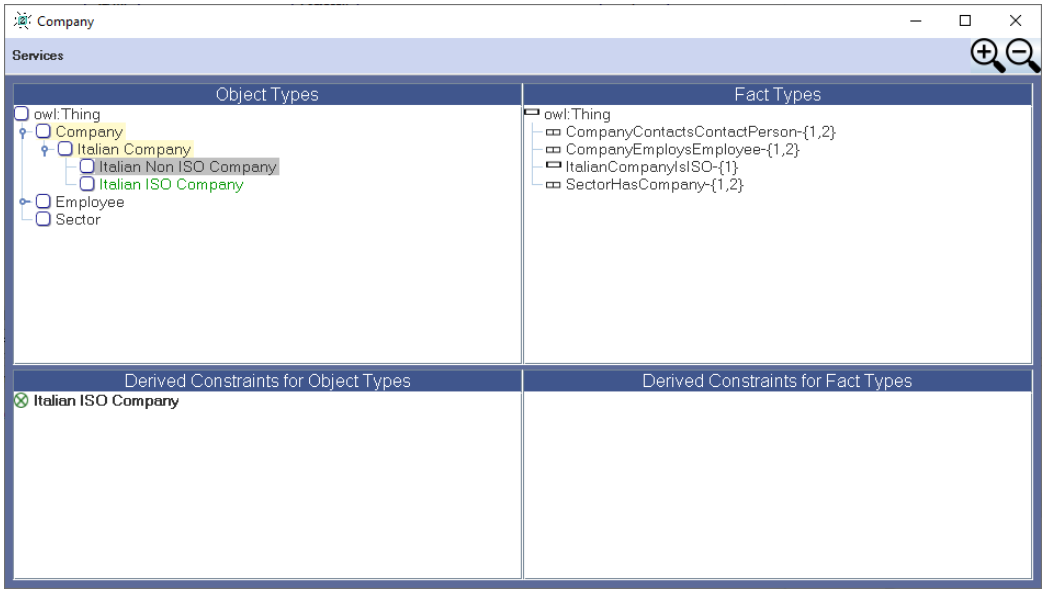


Figure 8.10: Reasoning over two pages

In the Section 8.5 this feature plays an important role in order to organize huge ORM diagrams.

8.4 Automated Reasoning

Although NORMA can be used as a powerful modelling tool, exploiting its full capabilities requires the coupling of the system with a Description Logic reasoner. Without ORMIE, NORMA would be unable to perform automated reasoning over the ORM diagrams. As we noted, this includes checking object types and fact types consistency, discovering implied constraints, subsumptions, or cardinality constraints, and in general discovering any implied but originally implicit class diagram graphical construct. Instead of implementing its own dedicated reasoner, ORMiE uses Fact++ [47], a powerful reasoner used also as a plugin for Protégé [129]. It does not need any dependency installation because the UModel framework is embedded in

the ORMIE plugin as a DLL file, since it was ported from Java to .NET; since this DLL file is the UModel framework, it comes also with OWLAPI as background engine to process the generated ontologies and Fact++ to perform the reasoning tasks. The scalability of the system is the same as any $\mathcal{ALCQI}$ [10] ontology processed by Fact++, moreover, some optimizations are been made to increase performances and to quickly deal with big ORM diagrams.

The so called verification process can be computationally expensive, so it is activated only on user's request. This process includes the following operations. The ORM diagram is encoded into a Description Logics knowledge base and shipped to the Fact++ reasoner. Each object type and fact type is checked for satisfiability (i.e. non-emptiness). For each object type and fact type, IsA and subset are determined. Uniqueness and mandatory constraints are calculated as well. To perform these operations, the system formulates a sequence of queries to be sent to the Fact++ reasoner. Accordingly to the received answers which are encoded in Java data structures, ORMIE shows inferred properties of the diagrams in the interface.

Due to performance optimizations the tool can manage diagrams with several hundreds of object types and fact types. After the verification process, the system provides the user with a visual account of the deductions by highlighting the relevant inferences in the interface. All unsatisfiable object types and fact types will appear in red, while the relevant deductions (the one not redundant) are shown in green. The redundancies, like an equivalence between two object types, are marked as yellow since they are considered as warnings. The deduction appearing in the lower part of the interface can be uniqueness, mandatory, equivalence and also disjointness constraints.

Although the deductions are displayed in the interface, it is up to the modeller to decide whether they should be permanently added to the models or discarded. The reason behind this behaviour is that the automated reasoning process may detect unwanted deductions caused by a wrong modelling of the domain. In this case the user should correct the project before any subsequent editing. Another reason is that, in spite of the fact that only the non-trivial deductions are presented, the user is satisfied by the fact that they are implicit

without the need of having them explicitly asserted. The ORMIE reasoner starts every time the interface is manually loaded, but also when the modeller makes some changes to the diagram. Trivial changes like renaming an object type do not trigger the reasoner. ORMIE is configured to ignore the situations where it's useless to recompute the deductions.

8.5 Evaluation of ORMIE in a real-world industrial scenario

This section presents results about ORMiE's performance in a real-world industrial scenario. ORMiE has been tested with ORM models provided by the European Space Agency (ESA) in a context where an ESA team uses the fact-based methodology with NORMA to model spacecraft related domains. The ORM models used are involved in an ESA project named *Intelligent Reasoner for Fact Based Models* whose goal is to extend NORMA capabilities in checking the semantics of the ORM diagrams. In order to accomplish this goal, NORMA has been equipped with ORMiE since the latter enriches NORMA by automated reasoning. The purpose of this evaluation is to prove the real efficiency of ORMIE and more in general of the entire methodology in a real-world industrial scenario.

8.5.1 Overview of the ESA Project

Developing and operating space systems implies complex activities involving many parties, distributed in location and time. This development requires efficient and effective interoperability during the overall space system development and operations lifecycle.

Interoperability is often described from a syntactic viewpoint, focusing on data exchange formats. While syntactic interoperability is required, the pre-requisite for any successful information exchange is to ensure that all actors involved share a common understanding of the information that will be exchanged. This aspect of interoperability is known as semantic interoperability. This is a mechanism whereby data can be reliably and effectively

exchanged between all suppliers and customers involved, between all engineering/quality/management disciplines involved, at all levels of the space system decomposition, within and across all space system life cycle phases.

ESA considers offering a solution to semantic modelling and semantic interoperability capturing the semantics of a "universe of discourse" and a logic based generic language to express that semantics.

Models can be visualized graphically using the ORM formal graphical language but also verbalized using a controlled natural language that enables each stakeholder to fully validate the semantics contained within the models.

In support of the development of information systems, the algorithms required to automatically transform the conceptual models into logical and physical models (relational, hierarchical, object oriented) to ease the development of software. Such automation ensures the quality of the generated logical models (e.g. fulfilling the standardized normal form rules of relational modelling).

Developing and operating large systems such as space systems implies constant exchanges of information and knowledge, through multi-levels sets of customer/supplier relationships. In such a global environment, interoperability cannot just be limited to assessing how to exchange data contained within data files, how to understand, to interpret an interface control document. Semantics is expressed using conceptual modelling. Any partner involved, independently of the customer or supplier role(s) played, has the full freedom to define and to limit their universe of discourse to the responsibility they play in the overall Space System. This implies the freedom, for any partner, to define locally the information systems needed, as long as the overall Space System needs are satisfied.

Semantic interoperability ensures that the information exchanged between all partners satisfies the overall Space System needs. Illustrated by Figure 8.11, the concepts of global model and local model are introduced:

- a global model is a conceptual data model whose purpose is to capture and put in relations among the universes of discourse of partners involved, i.e. producing the semantic links between the vocabulary used by each community, deriving how the same semantics modelled by some partner

in a given conceptual way maps the conceptual representation of the others;

- a local model is a conceptual data model that represents the view (a local view) that a given partner has of the global model, meaning the subset of the global model of relevance to that partner.

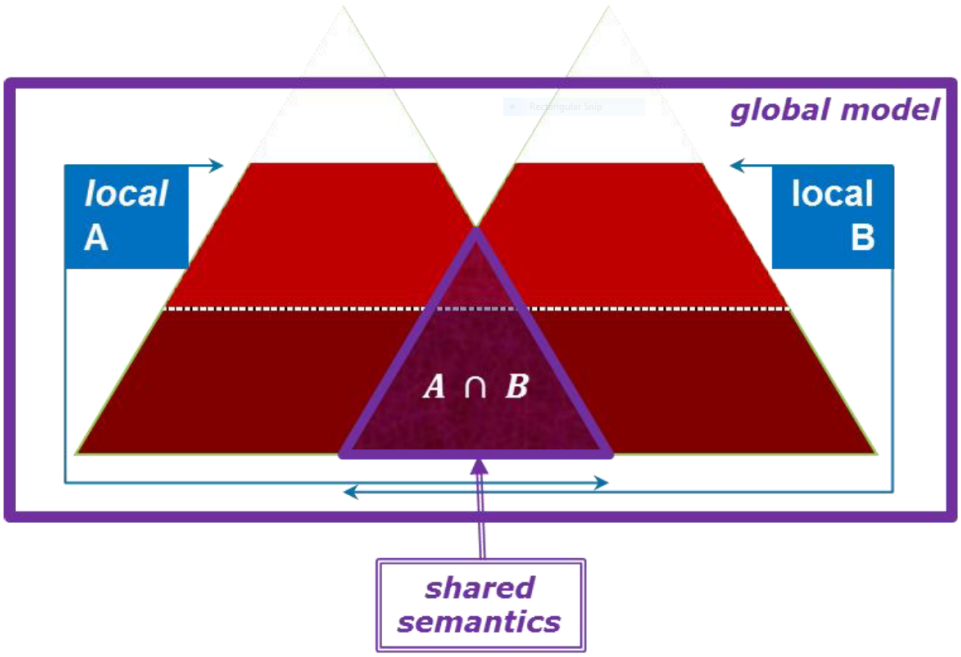


Figure 8.11: Shared semantics between two local conceptual models

Applying semantic interoperability implies the need for a common generic conceptual language used to express the semantics of the domain-specific conceptual models and to map the conceptual definitions that carry the same semantics. Each domain-specific model is expressed in a generic language that is either conceptual, logical or physical. Transforming a conceptual domain-specific model into a logical domain-specific model, translating a logical domain-specific model into a physical model and reverse-engineering physical or logical domain-specific models into conceptual domain-specific models requires mapping the conceptual definitions of these generic languages and identifying what semantics can be shared.

Having the conceptual maps between the generic languages of interest enables the capability to automate the transformation of the domain-specific schemas expressed in a generic language into domain-specific schemas expressed into another generic language. It is worthwhile to mention that the semantic equivalence of the source schemas and the schemas resulting from the transformations is fully dependent of the semantic expressiveness of the used generic languages. Formalising the generic languages' conceptual mappings and the domain-specific conceptual mappings provides the foundations for the success of the "Semantic Interoperability" objective.

Semantic modelling large information systems such as a Spacecraft Reference Database used by industry to support the spacecraft development and manage all monitoring and control definitions for testing the spacecraft and its components and later operating the spacecraft in-flight, produces large models.

Semantic interoperability implies integrating many (potentially very large) semantic models and managing for each stakeholder the views (subsets of the overall model) of interest. Ensuring the overall quality and integrity of the each semantic model is challenging. Ensuring the overall quality and integrity of the Space System Ontology (i.e. the result of integrating several semantic models for ensuing the interoperability at semantic level) is a very difficult task. The main objective of this activity is to assess the feasibility of developing a "semantic reasoner", to specify it and to prototype its key feature that is the automatic verification of the quality of the semantic models produced in NORMA. The NORMA software tool (in its professional version) is currently operationally used by ESA to model at semantic level. The page feature of NORMA is therefore essential in the development of large ORM diagrams since each page represents a local model concurring to produce the global model (i.e. the complete ORM diagram). For this reason the ORM models that are going to be tested are all composed of a multitude of NORMA pages.

8.5.2 Requirements for Benchmarking ORMiE

To prove that ORMIE is useful in an industrial setting, it should check the semantics in a reasonable amount of time. This task is particularly challenging in the context where potentially large ORM diagrams are modelled, such as in the European Space Agency environment. To properly evaluate such performance one would need benchmarks tailored towards the usage requirements proper for the ORMIE setting. Hence, there is a need for benchmarks resembling a typical real-world industrial scenario, in terms of the size of the ORM models, the complexity of the generated ontology, the complexity of the mappings, and the complexity of the queries. The purpose of this benchmark is to evaluate if ORMiE is a suitable candidate to be used in a real-world industrial scenario, but before doing so we need to identify the features of an industrial environment in order to properly design a benchmark test.

We start by identifying the industrial settings features: as stated in the previous section where the ESA context has been described, we notice that an industrial setting is characterized by large data, in this case ORM models with hundreds of ORM constraints. Moreover, since ESA models are composed by many views where each stakeholder is taking part in the project, it necessarily means that in the real-world an ORM model is developed by several iterations. We define an iteration as a change applied to the model performed by a stakeholder, like adding, removing, or editing one or more ORM constraints in the diagram. This action puts the model in a new configuration requiring a new reasoning computation.

We can summarize the real-world industrial features as follows:

- large data;
- many partners;
- several iterations.

Computing large data may be a time consuming task and if for each iteration the reasoning is slow the entire workflow is significantly slowed down and the

usage of ORMiE may even result in it being an obstacle to the development of the system.

Considering these requirements, the suitable data to be used in the experiment must reflect the features of a real-world scenario that are given by large ORM diagrams used by multiple stakeholders.

A positive outcome of the benchmark would be computing the inferences in a *reasonable* amount of time; otherwise, it could be a limitation since many "try and catch" iterations are used to manage potentially large diagrams.

But, what is defined as a reasonable time in this context?

This question has two possible answers, according to the two ways ORMiE may be activated. As we have seen in Section 8.4, ORMiE can be activated manually or automatically. A manual activation runs ORMiE once when the user decides; the automatic activation runs ORMiE every time the model is changed. The first scenario is usually taken into account to check the semantics of a finished model; the second scenario is used when more iterations are needed, but the automatic activation can also be a default choice to continuously run the reasoning every time the model changes. The adoption of the second scenario must not constitute a huge loss of time (e.g. waiting too much time every time the model changes could be frustrating for the modeller), so a reasonable time for this should be a computation that takes the least possible amount of time, best non perceivable by the modeller. In other words, few milliseconds. Unlike this scenario, when ORMiE is called manually there is an higher tolerance, so a reasonable computation time may also be longer.

8.5.3 Experiment Design

For each ORM model we consider the following characteristics:

- Number of ORM constraints - An ORM model is composed by a set of ORM constraints determining the size of the model.

- Number of TBox axioms - Each ORM model has a corresponding generated ontology that will be processed by the reasoner. The number of TBox axioms indicates the size of the ontology.

- Overall execution time - Time elapsed from the start up of the system to the inferences are ready. This metric is expressed in milliseconds (ms).

- Memory - The memory consumption taken by the ORMiE process. This metric is expressed in megabytes (MB).

Since the focus of the experiment is to measure the execution time in order to evaluate if ORMiE is able to compute the ORM models in a reasonable time, this metric needs more details. The overall execution time is obtained by a sequence of actions performed by ORMiE, so detecting these steps in the benchmark may help to better locate possible bottlenecks or having a more accurate evaluation of the data.

1. Parser - Time spent parsing the ORM model. This task is the first one to be executed.

2. Ontology generator - Time spent to generate an ontology from the given ORM model.

3. Reasoner - Time spent by the reasoner processing and querying the generated ontology. The reasoning is a set of operations handled by the reasoner used in ORMiE (Fact++) to retrieve the inferences.

4. Gui - Time spent by the GUI loading the results. The step involving the ORMiE graphical interface is related to the time that occurs to collect and display all the information for the final user.

After taking into account these data, a set of 5 runs are performed for each model in order to identify the average execution time and memory values.

Data - ORM models

To have a better insight into the data, we describe each single ORM model and its role in the test. The test has been performed using three ORM models with different sizes:

1. PUS.orm - The largest file used in this ESA project. An ORM diagram divided into 42 pages representing a space-craft information system. This is considered a suitable candidate for the test since it is has a large size and it is comprehensive of all ORM constraint, along with subtype and fact type derivation rules. PUS digram is composed of 46 views. This means that different stakeholders are taking part to the development of this ORM model by doing multiple iterations from different sources: this features makes this model a suitable candidate for a real-world industrial scenario.

2. PUS-lite.orm - As the name suggests, it is a lightweight version of the previous one, approximately half-size and divided into 26 pages. Although it is considered a large ORM model, its overall execution time should be compared to the previous one in order to see if there are some significant differences.

3. Visitor.orm - The running example in Figure 4.11 is the smallest in the test with a size that is typically used to design small ORM diagrams that fit into one page. Despite this not being suitable for testing a real-world scenario, it takes part in the test because it may help to detect possible differences in the overall execution time among the models.

The number of pages has no effect on the overall execution time since they are virtual views. Technically speaking, the ORMiE parser sees the entire ORM diagram as a whole without taking into account the views, so the system generates a single ontology then passes it on the reasoner to be processed. The number of pages is an additional information to provide an approximation of the ORM model size. In Figure 8.12 is provided an example of reasoning over the ORM diagram PUS-lite.orm:

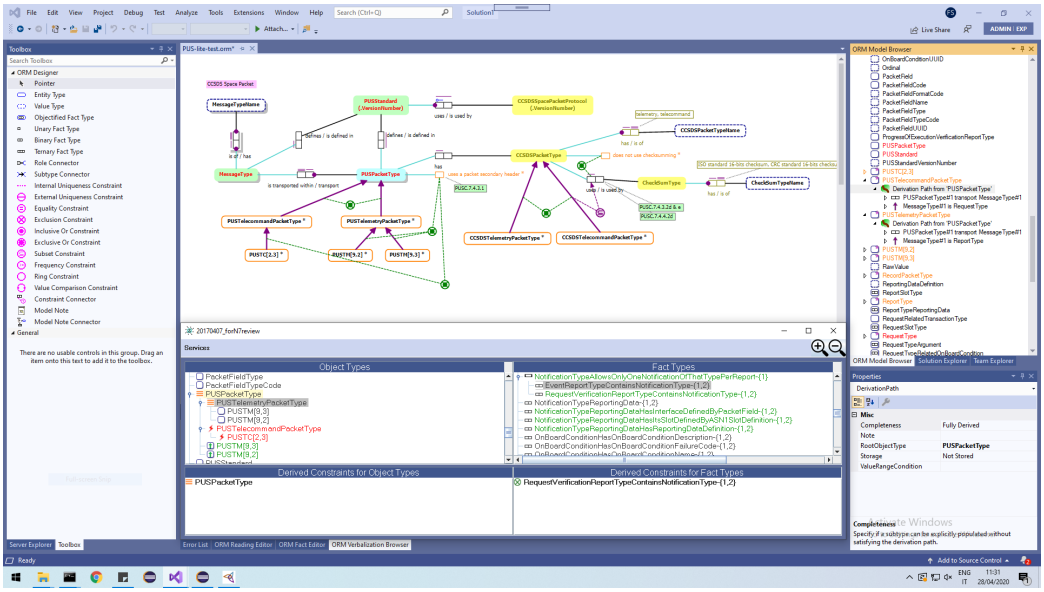


Figure 8.12: ORMiE reasoning with an ESA conceptual model

The ORM models are the initial input of the test. After the ORMiE parser finishes his job, an ontology is generated to represent the parsed ORM model. Generated ontologies must be considered since they serve as the input for the reasoner. The generated ontology contains OWL axioms specifying comprehensive information about the underlying classes in the ORM models; in particular, the generated ontology presents rich hierarchies of classes, axioms that infer new objects, disjointness and equivalence assertions. Since the backbone of the ORM formalisation is entirely based on $\mathcal{DLR}^{\pm}$, then the Description Logics used is $\mathcal{ALCQI}$; this means that the generated ontology is encoded in the OWL fragment named OWL2 DL. The generated ontology coming from the ESA ORM model is suitable for benchmarking reasoning tasks, given that it is a representative of the real-world ontology in terms of number of classes and maximum depth of the class hierarchy (hence, it allows for reasoning with respect to class hierarchies).

Hardware and software settings

The benchmark has been performed on a machine with the following specifications:

- Operating System: Microsoft Windows 10 Professional
- Architecture: x64
- CPU: AMD Ryzen 2700X
  - Number of cores: 8
  - Number of CPU threads: 16
  - Default clock frequency: 3.7Ghz
- System Memory
  - Capacity: 32 GB
  - Frequency: 2133 Mhz

As for the software specifications, we must consider ORMiE (this implies the usage of UModel) and its environment (NORMA).

- Programming language: C#
- Framework: Microsoft .NET 4
- Core system: UModel (ported in C# by IKVM)
- Reasoner: Fact++ 1.6.4
- OWLAPI: version 4.1.4

8.5.4 Benchmark Results

The results are presented in Table 8.1. Marked in bold overall execution time and the memory consumption.

The largest ORM model named PUS.orm is computed in an average time of 371ms; a smaller version of the same model with less constraints, namely PUS-lite.orm, has been executed in an average time of 279ms; finally, the running example named Visitor has been computed in an average time of 17ms. At a first glance, the execution time for large ORM models is considered

Table 8.1: Overall time (in milliseconds) - Memory (in megabytes)

Model	# constr	# axioms	avg_pars	avg_onto	avg_reas	avg_gui	**avg_all**	**mem**
PUS	737	5143	3	169	196	3	**371**	**182**
PUS-lite	479	4098	2	151	124	2	**279**	**158**
Visitor	43	130	1	6	8	1	**16**	**45**

reasonable and also efficient. It is easy to observe that the parser runs in very few milliseconds for every input. The reason behind this efficiency is because the ORMiE parser just needs to read the ORM constraints because the model has been already loaded inside NORMA. It is important to specify that the parser lies on the same stack where ORMiE is executed, the .NET framework.

The average execution time for the ORMiE graphical interface is efficient as well because the inferences are already stored in memory.

The ontology generation execution time depends on the input size and it grows linearly. Nothing surprising since ORMiE has to write the axioms in the generated ontology each time an ORM constraint is encoded, plus all the axioms related to the $\mathcal{DLR}^{\pm}$ encoding.

The memory consumption grows linearly as well, since the generated ontology and its results are stored in memory as well.

The reasoning execution time is quite efficient as well, but it seems to grow depending on the input size. As it is known from Description Logics theory [36] (Chapter 9), the size of an ontology is only loosely related to the complexity of reasoning on it. For this reason, the reasoning section deserves further insights in order to detect possible sources of complexity. The details about the reasoning execution time are shown in Table 8.2:

We can observe that the computation of disjointnesses is one of the major source of complexity. The reason behind this is the $\mathcal{DLR}^{\pm}$ tree which is built upon a set of disjointnesses, so the number of disjointess axioms depends on the size of the tree. Recalling Section 5.1.5, the $\mathcal{DLR}^{\pm}$ mapping function *dsj* ensures that relations with different signatures are disjoint. In the worst case all relationships have different signatures, this means that the complexity

Table 8.2: Reasoning execution time details (in milliseconds)

	Run/Task	Hierarchy	Unsat	Equivalence	Disjointness	Mandatory	Uniqueness
PUS.orm	RUN 1	16	2	2	95	5	91
	RUN 2	22	1	7	67	4	89
	RUN 3	23	1	2	62	13	80
	RUN 4	21	2	5	65	9	81
	RUN 5	24	2	3	63	9	92
PUS-lite.orm	RUN 1	15	1	1	44	1	54
	RUN 2	18	1	1	41	4	64
	RUN 3	19	1	2	53	0	53
	RUN 4	15	1	5	45	3	56
	RUN 5	14	1	0	48	3	57
Visitor.orm	RUN 1	2	0	0	2	0	4
	RUN 2	1	0	0	0	0	2
	RUN 3	1	0	0	2	0	3
	RUN 4	1	0	0	3	0	8
	RUN 5	3	0	0	0	0	4

grows at most linearly since the reasoner must compute all the disjointness combinations. This implies that the size of the ontology does not necessarily affect the computation time of disjointness, but the $\mathcal{DLR}^{\pm}$ tree size does.

Computing uniqueness is also a source of complexity. As we have seen in Section 7.4, the query for uniqueness compares every class marked as UNIQ with all the classes marked as PROJ which belonging to the $\mathcal{DLR}^{\pm}$ tree. The number of comparisons depends on the number of nodes in the $\mathcal{DLR}^{\pm}$ tree. Similar to the disjointness case this function grows at most linearly in time.

8.5.5 Findings and Conclusions

Observing the data we can state that ORMiE is able to perform the automated reasoning on real-world models in few hundreds milliseconds. This timing can be considered enough reasonable.

Findings related to the benchmark:

- The parser and the gui are very efficient since they read data that is already loaded.

- The ontology generation time depends on the size of the ORM model, precisely, on the number of ORM constraints and the number of TBox axioms that need to be written in the generated ontology. So, the complexity grows linearly. The biggest ontology has been generated in ∼150ms, still a reasonable amount of time.

- ORM diagram size does not necessarily affect the reasoning execution time. The reason behind this is the nature of the encoding in $\mathcal{ALCQI}$. So the scalability of the ORMIE automated reasoning is strictly related to the scalability of the reasoning algorithms that process ontologies encoded in the fragment $\mathcal{ALCQI}$, so it is possible to conclude that even for huge diagrams the system is efficient.

- The major source of complexity are disjointness and uniqueness because they depend on the number of nodes in the $\mathcal{DLR}^{\pm}$ tree, so the complexity grows depending on the size of the $\mathcal{DLR}^{\pm}$ tree, not necessarily the size of ontology (e.g. best case a large ORM diagram made only by Entity Types generates a large ontology with no tree nodes, so disjointnesses and uniquenesses are not even computed). Despite their complexity, the computation for real-world data can be still considered efficient since the amount of time in an average case is under 100ms.

Additional observations:

- The NORMA pattern control (Section 2.4) forces the modeller to avoid trivial bad-modelling patterns. If this function is disabled, or the bad modelling involves an undetected pattern, the reasoning covers the job detecting the related inferences. Where the NORMA pattern control relies on mere syntax, the reasoner relies on the semantic ensuring further control over the model.

- NORMA pattern control does not work on ORM Derivation Rules. Since NORMA alone has no control over the semantics, the modeller cannot

immediately see the consequence of defining a derivation rule in the ORM diagram. A derivation rule can raise interesting inferences, or in the worst case can raise inconsistencies; for this reason the adoption of the automated reasoning adds more control over this task.

- The inferences are presented in a way to make them easy to spot in the graphical interface. Since large diagrams have several object types and fact types, the user has to scroll-down every object type and fact type resulting in a time-consuming activity. The graphical interface has the feature to highlight even those relevant inferences that are nested in the hierarchy, in this way the modeller is able to quickly detect the wanted inference.

- The core engine of ORMiE is the UModel framework, so the further research tracks and possible improvements discussed in Section 7.4 are inherited by ORMiE.

In conclusion, ORMiE is able to handle real-world ORM diagrams in a reasonable amount of time. We must also consider that ORMiE is not a proof-of-concept tool, instead it is a ready-to-use software that is used by the European Space Agency to support the conceptual modelling about the design of spacecraft systems [4], [5]. Additional research directions are suggested by the aforementioned points, like improving performance related optimizations and implementing new functionalities like the explanatory service.

9 Related works

In this chapter we compare ORMiE against other conceptual modelling-related tools. The comparison is based on some key features considered relevant for a conceptual modelling application:

- Support for automated reasoning - it is the key feature discussed in this work since it carries interesting benefits for the modelling workflow;
- Support for automated reasoning over rules, such as ORM Derivation Rules for ORM and OCL for UML;
- Industry-ready environment - a feature that differentiates when to use a software in an industrial environment, where multiple stakeholders take part to the project through several iterations;
- Additional features for the workflow - like import/export of models in different format, support for verbalization, etc.

We are going to point out the possible weaknesses, similarities and advantages of each tool.

9.1 ICOM

ICOM is a conceptual modelling tool which allows the modeller to design multiple extended ontologies [48] [54] [53] [52]. Each project can be organised into several ontologies, with the possibility to include inter and intra ontology constraints. The logical reasoning is taken out by Racer as a reasoning

engine in order to verify the specification, infer implicit facts, devise stricter constraints, and manifest any inconsistency [71], [70], [112]. The diagrams are modelled with the DIG language, the Description Logics Interface [25]. The intention behind ICOM is to provide a simple conceptual modelling tool that demonstrates the use of the novel and powerful knowledge representation based technologies for database and ontology design.

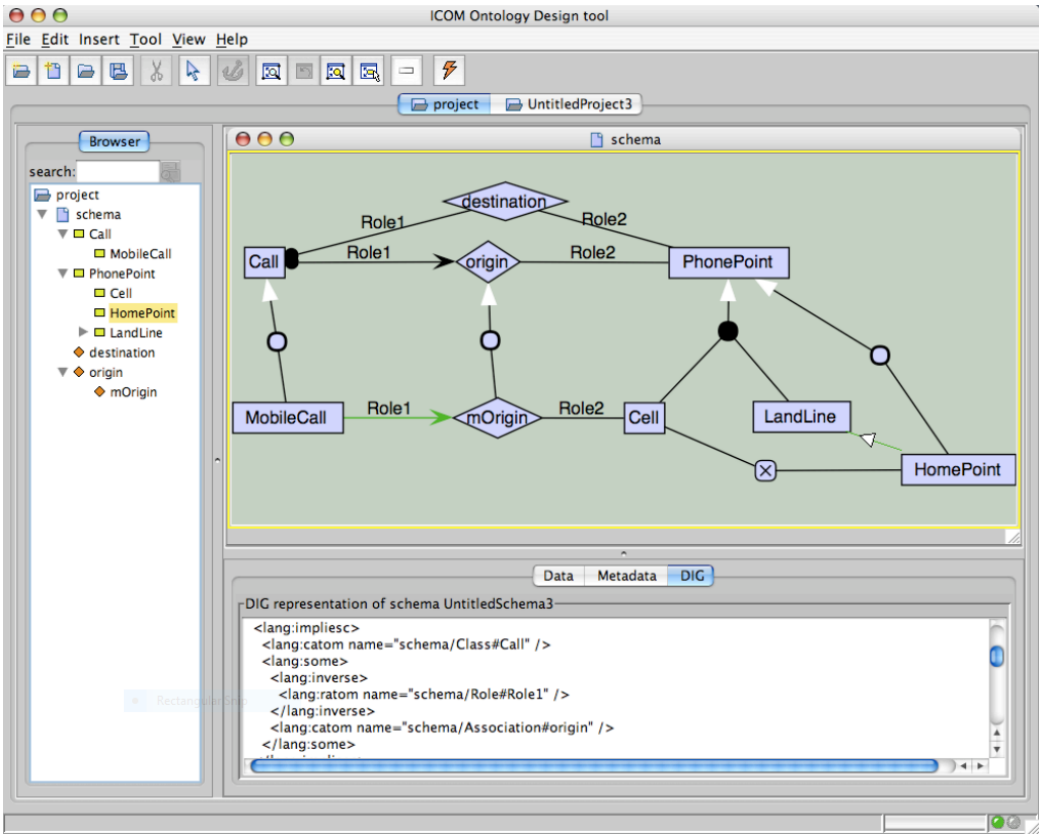


Figure 9.1: ICOM

In Figure 9.1 we see how the ICOM gui looks like. There is a browser for object navigation and the project panel where most of the model editing is done, where each model in the project is displayed in a separate model panel.

The inferences are directly depicted in the project panel, as for the Role1 constraint in Figure 9.2 where the green arrow is shown.

ICOM is a proof-of-concept software which is not maintained any more. It was written in standard Java 5 and distributed for Linux, Mac and Windows.

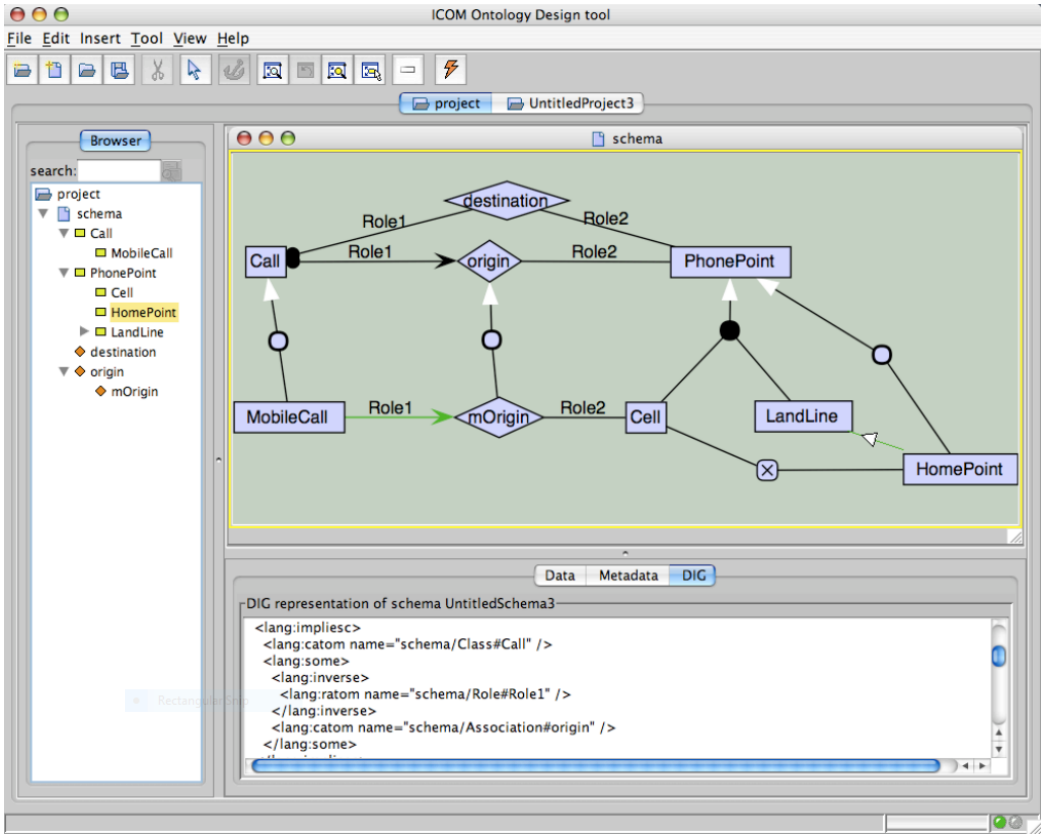


Figure 9.2: ICOM

The system has the limitation to be standalone and it does not come with a rich set of features like an environment as ORMiE.

Despite this, ICOM has some similarities with ORMiE like the usage of the automated reasoning and the capability to work with inter and intra-ontology constraints and diagrams. ORMiE and ICOM differ in displaying the inferences: ORMiE shows the inferences inside an browsable panel; ICOM shows the inferences directly into the diagram. This approach could be efficient for small diagrams where the readability is reduced, but for large diagrams a specific window may improve the user experience especially where a search for a particular inference among several diagram elements is needed.

9.2 DOGMA Modeler

DogmaModeler [114] is an ontology modelling tool based on ORM. The first version of DogmaModeler was developed at the Vrije Universiteit Brussel [101], [95].The goal of DogmaModeler is to enable non-IT experts to model ontologies with little or no involvement of an ontology engineer. This is carried out by the usage of ORM as a graphical notation, the ORM verbalization feature that represents the ORM diagrams into pseudo natural language that allows non-experts to check, validate, or build diagrams. Similar to ICOM, DogmaModeler makes use of the automatic mapping of ORM diagrams into the DIG description logic interface and reasoning using Racer.

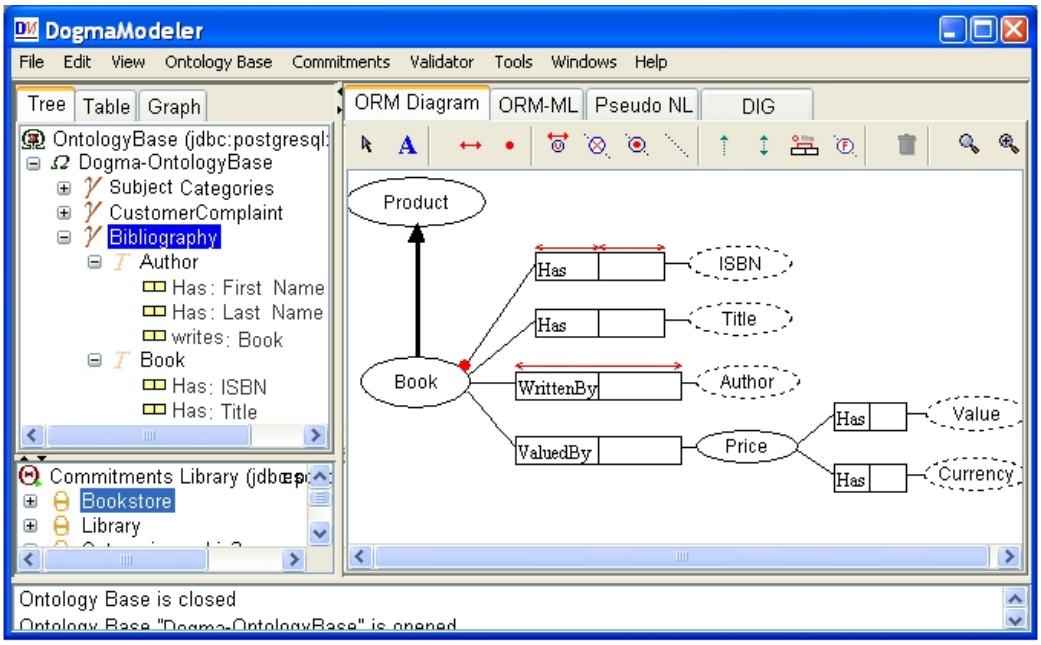


Figure 9.3: Dogma Modeler

DogmaModeler relies on the formalisation that can be found in [102], where the $\mathcal{DLR}$ description logic is used to encode ORM constraints in $\mathcal{SHOIN}$ OWL. As stated in [106] and also in [61], this formalisation has some limitations and formal inconsistencies. The provided encoding is sloppy with respect to the underlying DL formalism: distinct extensions of the adopted logic (e.g. $\mathcal{DLR}$ plus $\mathcal{DLR}$-Lite) and distinct DL languages (e.g. $\mathcal{DLR}$, plus $\mathcal{DLR}$-Lite, plus

$\mathcal{SROIQ}$, plus role composition operator) are mixed together. No semantics or complexity results are provided for these combinations.

DogmaModeler like ICOM has some software limitations: the software is outdated and dismissed and the system is standalone, so it does not have a rich set of features designed for industry standard workflow, as ORMiE. DogmaModeler does not support the automated reasoning for ORM Derivation Rules.

9.3 Protégé

Protégé [129] is a free, open source ontology editor and a knowledge management system developed at Stanford University. Earlier versions of the tool were developed in collaboration with the University of Manchester. Protégé provides a graphic user interface to define ontologies. It also includes deductive classifiers to validate the consistency of models and to infer new information based on the analysis of an ontology. Protégé is written in Java and heavily uses Swing to create the user interface. Protégé can claim to be utilized by a large community and according to [63] it is "the leading ontological engineering tool". An important feature is the capability to easily extend the system by the development of user plugins and a public repository where to download them [120]. The backbone of Protégé is based on OWLAPI, a Java interface and implementation for OWL. OWLAPI is focused towards OWL 2 which encompasses OWL-Lite, OWL-DL and OWL-Full [100].

The interface shown in Figure. 9.4 depicts a common scenario where Protégé is used. Classes are organized by hierarchy and additional information are displayed in the right part of the interface where ontology annotations and class descriptions are also provided. In the bottom-left panel the object properties are displayed. Protégé can be equipped with many reasoners (like FaCT++, Pellet, Hermit, etc.) in order to perform the automated reasoning over the ontology and highlight in yellow the inferences. The provided inferences are the equivalences, disjointness and subsumption among classes and object properties, plus the inconsistencies.

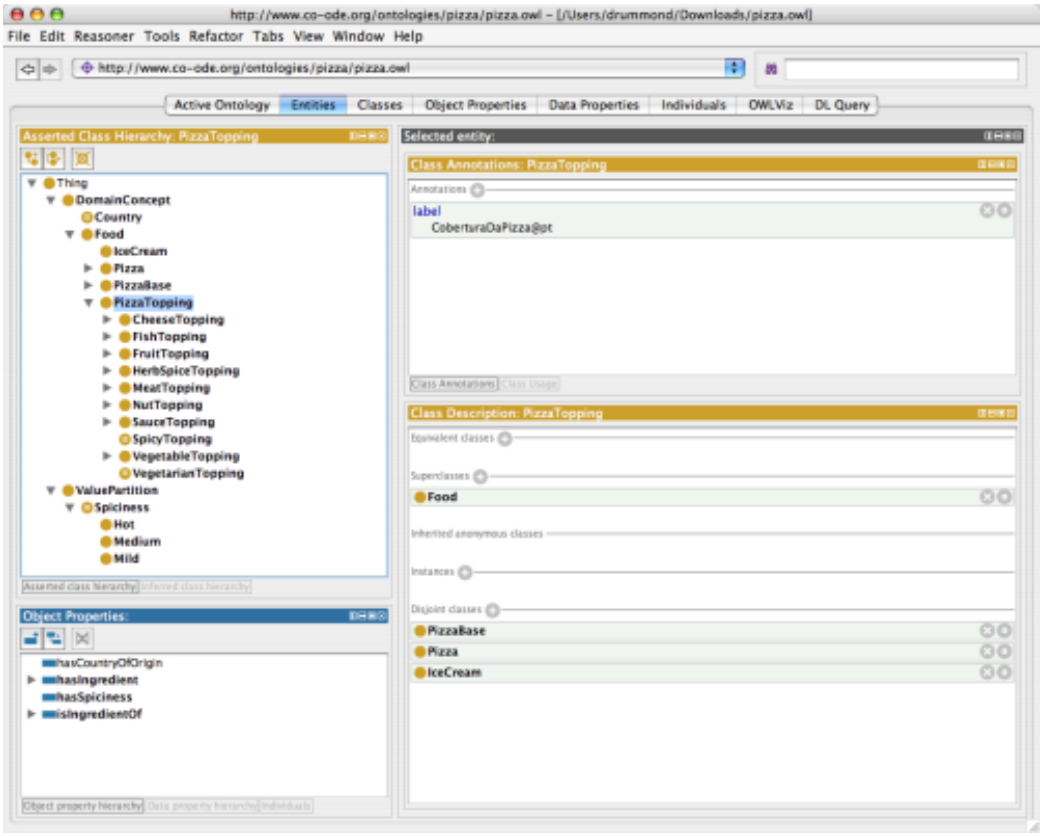


Figure 9.4: Protégé

ORMiE and Protégé are two tools that serve different purposes: one hand we have Protégé, it is designed to perform a set of operations on ontologies, like viewing, editing, exporting and running reasoners; on the other hand we have ORMiE which is specifically designed for ORM conceptual modelling. ORMiE follows the fact-based approach implemented by the ORM language; Protégé deals directly with OWL. As for the automated reasoning, we already know from Chapter 7 that ORMiE uses the same reasoner available in Protégé to deal with $\mathcal{ALCQI}$ ontologies, namely Fact++. Despite this, since ORMiE focuses on the conceptual modelling the way of displaying the inferences is different: in Protégé the way of showing the inferences is pretty straightforward, since all the inferences are displayed in the interface according to the hierarchy and the inferences from the ontology; here ORMiE differs since the hierarchy strictly reflects the ORM diagram structure, ignoring all those axioms coming

from the $\mathcal{DLR}^{\pm}$ mapping that are not-relevant from the user perspective (e.g., the one belonging to the multitree data structure). Moreover, in ORMiE the inferences are depicted with the ORM graphical notation, for example if a Uniqueness constraints is derived, an the uniqueness icon is displayed near that constraint. Protégé instead simply highlights the inference. We have to clarify that this is not a Protégé limitation, but an ORMiE feature needed in order to deal with a context based on the conceptual modelling.

The Protégé hierarchy highlights (in yellow) a specific class only if it has already been expanded in the hierarchy; the inefficiency of this approach is not to allow the user to immediately see where the inferences are. ORMiE, on the other hand, has a hierarchy system specifically designed to quickly suggest the user where to look for relevant inferences, marking all the ancestors of a specific node which has some inferences. This feature is particular useful for big diagrams. Another difference in the automated reasoning is the support for individuals: Protégé, unlike ORMiE, supports the assertion and the reasoning over individuals. The reason behind this is that ORMiE relies only on the conceptual level. An interesting feature of Protégé is the extensibility by plugins, as ORMiE is for NORMA. This feature has lead the Protégé community to expand its functionalities adding several plugins and different reasoners as well.

9.4 Boston

Boston, published in 2015 by Viev Pty Ltd [135], is a tool for conceptual modelling that implements the fact-based methodology through ORM. The main Boston feature is the capability to convert ORM diagrams to Entity Relationship Diagrams and Property Graph Schemas (PGS), since it is oriented towards the modelling software for databases, supporting both relational and graph models within the database. Boston allows the modellers to do their conceptual modelling in ORM, and then transform models to the required ER or PGS model.

Boston's graphical interface is shown in Figure 9.5. The Boston user builds ORM diagrams writing in the controlled natural language that in the case of

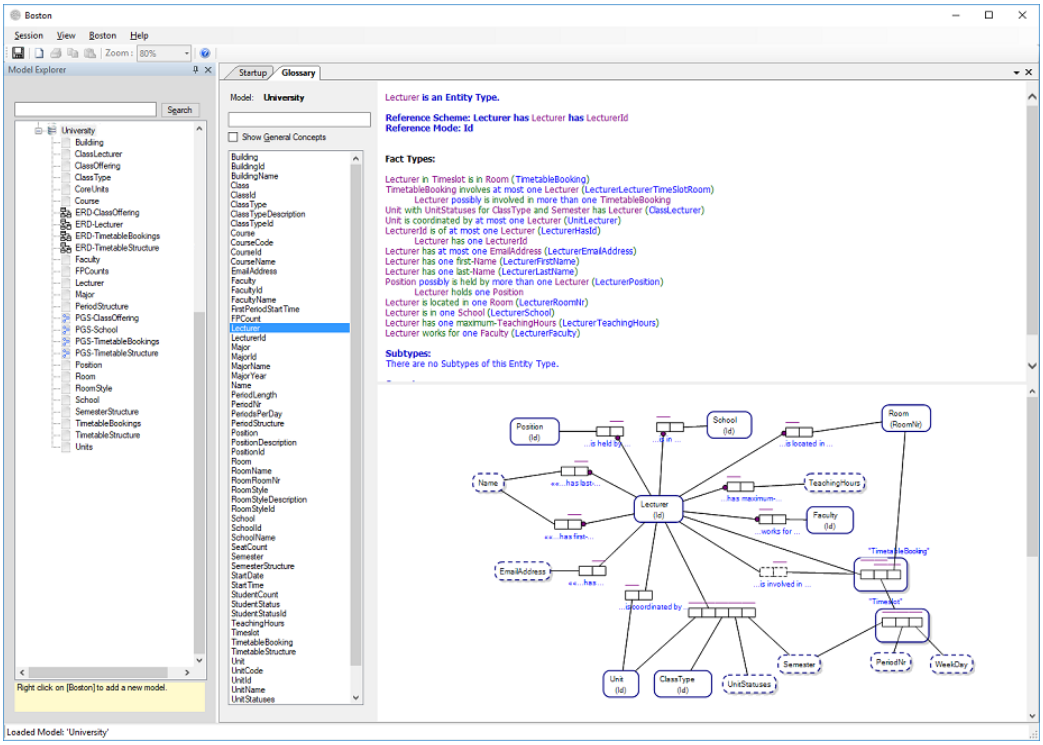


Figure 9.5: Boston

ORM is FORML [82], [94]; then, the ORM diagram is automatically generated in the main panel. Boston focuses the user experience on the usage of FORML language, rather then using the drag and drop feature like NORMA does.

Boston does not support any automated reasoning services so it is not able to check the semantics of the diagrams. Moreover, Boston does not support the creation and the editing of ORM Derivation Rules. We can say that Boston is a good standalone tool limited to ORM modelling coming with a good set of features, but it has no support for automated reasoning.

9.5 CASETalk

CASETalk [24] is a suite of products that work combined, supporting the ability to model, store and administrate business knowledge and the corresponding technological artifacts. The methodology used is the fact-based modelling based on the method FCO-IM [19], a successor of the well known

method NIAM [138]. CASETalk is a powerful framework oriented to manage corporate fact-models and with its rich set of features is suitable for large teams working on the same project.

Using CASETalk, the business analysts are able to create fact-based models using natural language for the IT personnel that can view these conceptual models. In this way, CASETalk aims to bridge the gap between these two stakeholders taking part in the same project. Moreover, it is also possible to store multiple model versions and manage access to them for multiple users, run impact analysis and change reports. This means that the CASETalk is also oriented towards large organisations. CASETalk comes with a rich set of features, which includes exporting the fact-based models in UML and ER.

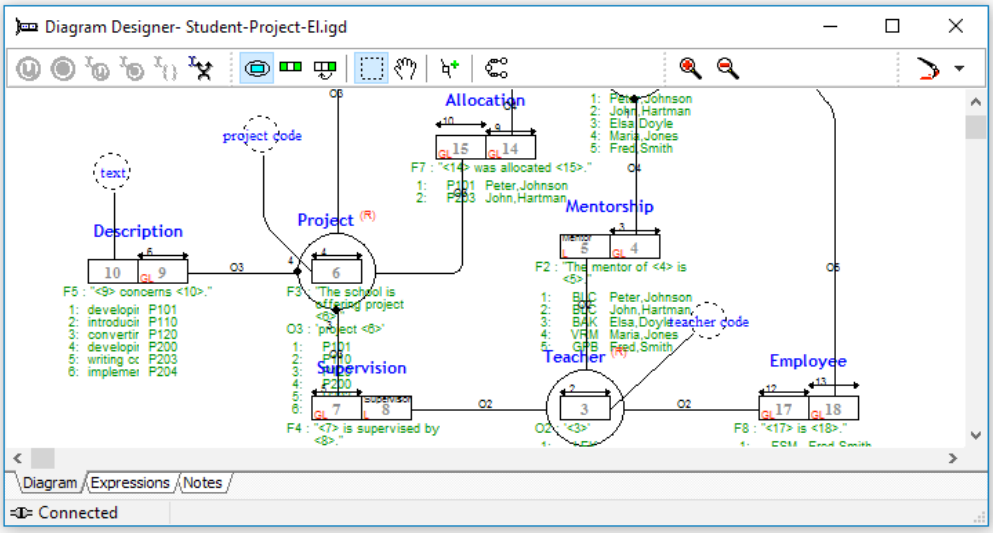


Figure 9.6: CASETalk

In Figure 9.6 is depicted a CASETalk scenario where a digram represents a specific domain.

CASETalk, unlike other tools mentioned in this chapter, is “industry-ready”; this means that it is already used in the industry so it is able to manage real-world scenarios with potentially large diagrams. It is also able to handle the definition of rules inside any diagrams, but it does not support automated reasoning services.

9.6 USE

We now compare ORMiE with a tool that does not make use of fact-based modelling, but it applies automated reasoning to UML class diagrams involving OCL constraints as well. In our context, OCL constraints can be seen as the counterpart of ORM Derivation Rules. USE allows UML models with OCL constraints to be validated against developer's assumptions. It is able to check the formal properties of a UML class diagram to control the consistency of UML models aiming to support the developers in analysing the model structure in order to suggest revision where something can harm the development of the software (e.g., inconsistencies) [51], [32], [31], [33], [34]. USE started as a dissertation project in 1998 [66] and its first version was already available. Then, other USE versions were developed further by diploma theses and other student projects. Nowadays, it is a robust tool used to validate OCL queries.

Figure 9.7 shows the USE graphical interface. In the centre there is the Object Diagram panel that shows objects with attribute values and its relationships. On the right side of the Object Diagram we have the Class Invariant window. This window shows the classes evaluation in the current system state. An invariant can be evaluated to true, false, or can be not applicable (n/a). A false invariant indicates an invalid system state which can be inspected by double-clicking the invariant name and this opens the Evaluation browser. The evaluation browser takes a detailed view on a chosen invariant in order to display variable assignments. This allows the user to understand invariant failure and to detect the violating parts of the object diagram. This feature is in some way similar to the reasoning procedure used in ORMiE to check if an object type or a fact type is consistent or not. USE represents an efficient solution to validate those UML class diagrams equipped with OCL constraints.

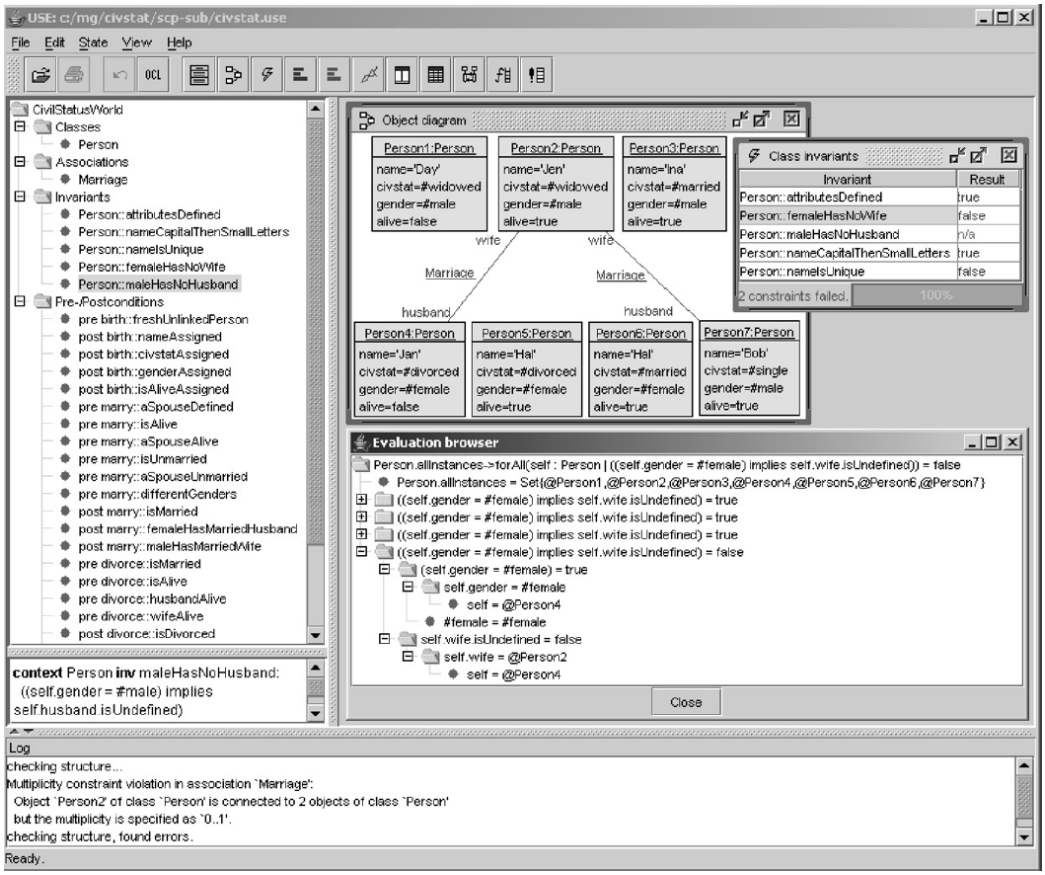


Figure 9.7: USE

9.7 HOL-OCL

HOL-OCL [2] is an interactive theorem proving environment for the OCL constraints that is integrated in a Model-driven Engineering (MDE) framework. It is implemented as a shallow embedding of OCL into the Higher-order Logic (HOL) instance of the interactive theorem prover Isabelle [133]. HOL-OCL is developed by Achim D. Brucker and Burkhart Wolff. HOL-OCL allows the user to reason over OCL specifications, refine OCL specifications, and builds the basis for further tool support, e.g. for the automatic test-case generation. HOL-OCL provides several derived proof calculi that allow for formal derivations establishing the validity of UML/OCL formulae. These formulae arise naturally when checking the consistency of class models, when

formally refining abstract models to more concrete ones or when discharging side-conditions from model-transformations [32], [31], [33], [34].

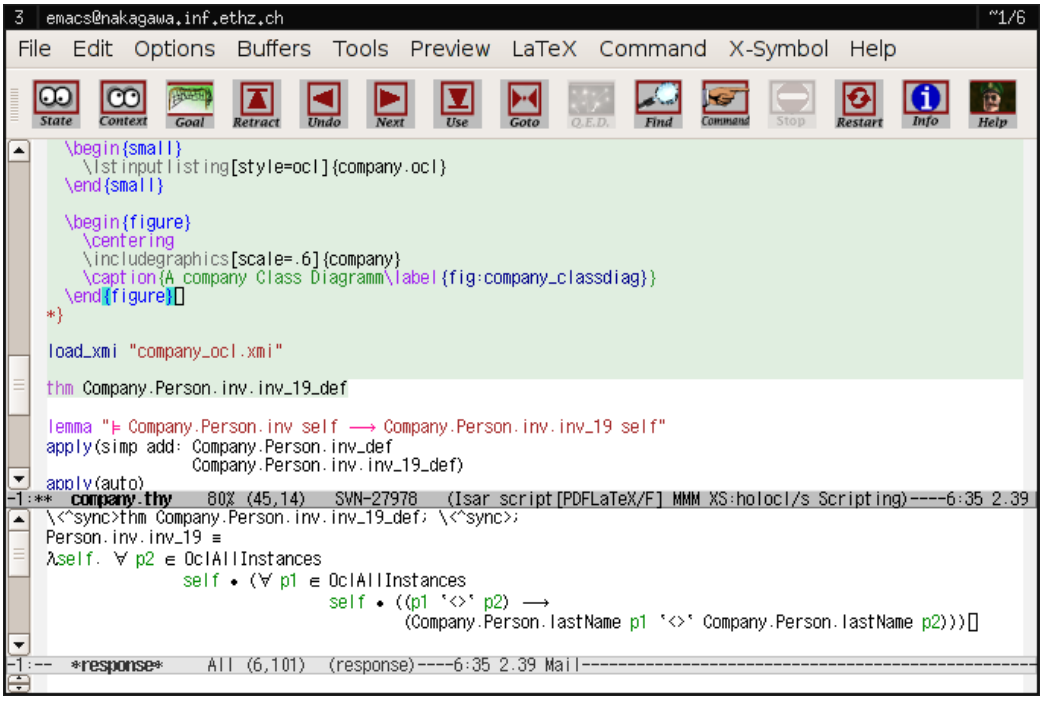


Figure 9.8: HOLOCL

As ORM Derivation Rules and ORMiE, HOL-OCL allows to reason over UML class models annotated with OCL specifications, in this way HOL-OCL extends the reasoning over those UML class diagrams equipped with rules. HOL-OCL differs from ORMiE because it is oriented to be a proof-environment system; on the other hand, ORMiE is a tool designed to visually model potentially large diagrams living inside a complete environment like Visual Studio.

9.8 Menthor

Menthor Editor [131] is an ontology-driven conceptual modelling platform which incorporates the theories of the Unified Foundational Ontology (UFO) [69]. Menthor is an implementation of the language OntoUML which is a well founded language based on UFO. The goal of Menthor is to improve

the design of domain ontologies, expressed in the language OntoUML, by using the theories behind UFO in order to build, validate and implement ontologies. Among its features, it also includes OntoUML syntax validation, Alloy simulation, Anti-Pattern verification,[123] and MDA transformations from OntoUML to OWL.

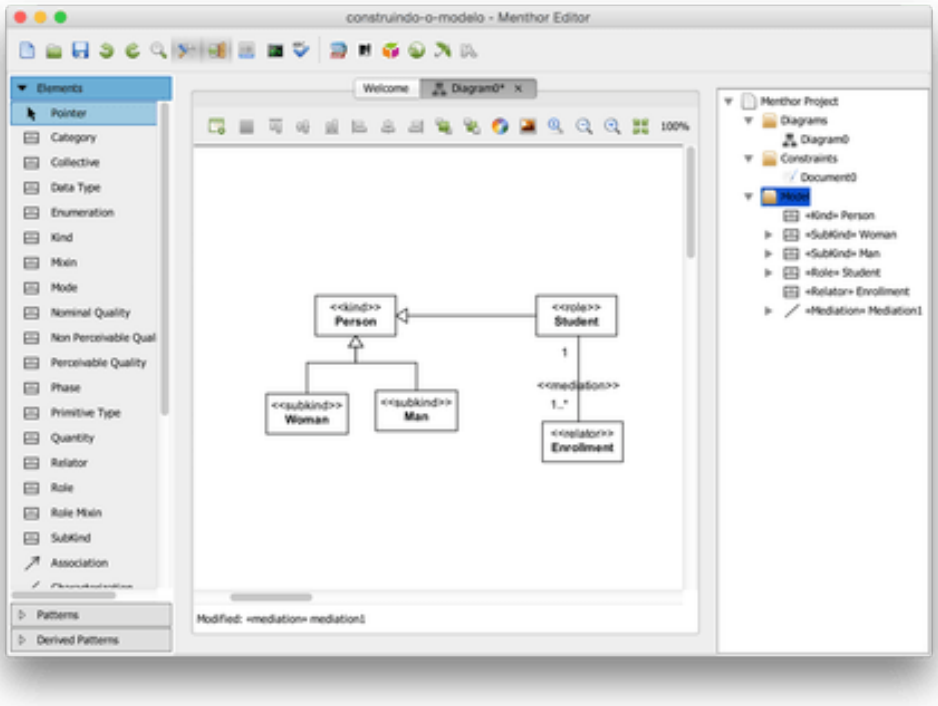


Figure 9.9: Menthor

In the context of the conceptual modelling Menthor has some similarities with ORMiE, since it focuses on modelling a domain by UML. Moreover, it supports alloy validation and so OCL constraints. Despite Menthor not being provided with a reasoning engine, it validates the models based on the UFO framework and the anti-pattern verification.

Menthor may be considered a good candidate to be equipped with the UModel framework in order to activate the automated reasoning over UML diagrams.

10

Conclusions and Future Work

This work introduced a methodology to support conceptual modelling by automated reasoning. The benefits coming from this methodology have been presented in a real-world industrial scenario where the control over the semantics of the conceptual diagrams has been improved, speeding up the development of a system from the early stages of software life cycle preventing software degradation. Checking the consistency of a conceptual diagram, or detecting redundancies and errors, may save time during the modelling step and next development steps as well. The conceptual modelling has been carried out as an implementation of the fact-based methodology with the language ORM, which in this work has been used to model the conceptual diagrams. An ORM formalisation into a logical language has been provided, alongside a formalisation concerning the ORM Derivation Rules constraints. A decidable fragment named $\mathcal{ORM}^{\pm}$ has been detected revealing that an ORM diagram containing a subset of ORM constraints and ORM Derivation Rules, can be processed by a reasoner to detect relevant inferences. $\mathcal{ORM}^{\pm}$ has been encoded in $\mathcal{DLR}^{\pm}$, a language from the Description Logics family which is specifically designed to deal with n-ary relationships. Moreover, a mapping into the OWL language has also been provided to make possible to build an implementation of this theoretical work. In particular, this work has been implemented in a framework called UModel that has been used in a real case scenario to test its efficiency. This framework implements $\mathcal{DLR}^{\pm}$ to be used as a backbone language; in this way UModel is able to cover the $\mathcal{ORM}^{\pm}$ fragment enabling the automated reasoning over ORM diagrams. The main idea behind this framework is to provide a general purpose methodology that

is useful for any conceptual modelling language, with the consequences to easily extend the approach to any CASE tool that makes use of conceptual modelling languages such as ORM, UML and ER.

The thesis focuses in particular on the ORM Derivation Rules formalisation which represents a novelty in the ORM formalisation. As consequence of this, another contribution is the implementation of a solution that deals with these constructs.

The future research may follow different tracks. The reasoning could be extended beyond the conceptual level supporting the population of a conceptual diagram. At this stage ORMIE is able to present the inferences to the final user. A research track that could improve this feature is to add explanatory services in order to show why that inferences have been detected, to make easier to understand possible mistakes by tracking back the inferences. In this way the modeller's control over the conceptual diagram is enhanced. A more practical flavour has the research going in the direction of integrating other conceptual modelling languages in UModel framework to enable automated reasoning over other languages and systems, such as UML and ER.

This work has shown that the presented methodology can be used in a real-world industrial scenario, usually characterized by several users working on the same project with multiple iterations. In this context, the execution time to perform the reasoning is a crucial point due to the expected several iterations. The results have shown that the presented framework runs in a quite reasonable time even for large inputs, suggesting that the system is ready to be used in an industrial scenario.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Achim D. Brucker. HOL-OCL. `https://www.brucker.ch/projects/hol-ocl/`, 2018.

[3] European Space Agency. `http://www.esa.int/ESA`, 2019.

[4] European Space Agency. Esa - semantic modelling and semantic interoperability. `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/Semantic_Modelling_and_Semantic_Interoperability_-_FAMOUS-2`, 2019.

[5] European Space Agency. famous2.eu. `https://www.famous2.eu`, 2019.

[6] Alessandro Artale, Diego Calvanese, Roman Kontchakov, Vladislav Ryzhikov, and Michael Zakharyaschev. Reasoning over extended ER models. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *Conceptual modelling - ER 2007, 26th International Conference on Conceptual modelling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, volume 4801 of *Lecture Notes in Computer Science*, pages 277–292. Springer, 2007.

[7] Alessandro Artale, Diego Calvanese, Roman Kontchakov, Vladislav Ryzhikov, and Michael Zakharyaschev. Reasoning over extended ER models. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *Conceptual modelling - ER 2007, 26th International Conference on Conceptual modelling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, volume 4801 of *Lecture Notes in Computer Science*, pages 277–292. Springer, 2007.

[8] Alessandro Artale and Enrico Franconi. Extending DLR with labelled tuples, projections, functional dependencies and objectification. In *Proceedings of the 29th International Workshop on Description Logics (DL-2016)*, 2016.

[9] Alessandro Artale and Enrico Franconi. Extending DLR with labelled tuples, projections, functional dependencies and objectification (full version). *CoRR*, abs/1604.00799, April 2016.

[10] Alessandro Artale, Enrico Franconi, Rafael Peñaloza, and Francesco Sportelli. A decidable very expressive description logic for databases. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2017.

[11] Alessandro Artale, Enrico Franconi, Rafael Peñaloza, and Francesco Sportelli. A decidable very expressive description logic for databases (extended version). *CoRR*, abs/1707.08468, 2017.

[12] Alessandro Artale, Enrico Franconi, Rafael Peñaloza, and Francesco Sportelli. *A Decidable Very Expressive Description Logic for Databases(Extended Version)*, 2017.

[13] Alessandro Artale, Enrico Franconi, Rafael Peñaloza, and Francesco Sportelli. A decidable very expressive n-ary description logic for database applications (extended abstract). In Sergio Flesca, Sergio Greco, Elio Masciari, and Domenico Saccà, editors, *Proceedings of the 25th Italian Symposium on Advanced Database Systems, Squillace Lido (Catanzaro), Italy, June 25-29, 2017*, volume 2037 of *CEUR Workshop Proceedings*, page 33. CEUR-WS.org, 2017.

[14] Alessandro Artale, Roman Kontchakov, Vladislav Ryzhikov, and Michael Zakharyaschev. Complexity of reasoning over temporal data models. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C.

Woo, and Yair Wand, editors, *Conceptual modelling - ER 2010, 29th International Conference on Conceptual modelling, Vancouver, BC, Canada, November 1-4, 2010. Proceedings*, volume 6412 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2010.

[15] Fazat Nur Azizah and Guido Bakema. Data modelling patterns using fully communication oriented information modelling (FCO-IM). In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET, OnToContent, ORM, PerSys, OTM Academy Doctoral Consortium, RDDS, SWWS, and SeBGIS 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part II*, volume 4278 of *Lecture Notes in Computer Science*, pages 1221–1230. Springer, 2006.

[16] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[17] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 21–43. Springer, 2009.

[18] Dang Bui Bach, Robert Meersman, Peter Spyns, and Damien Trog. Mapping OWL-DL into ORM/RIDL. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, volume 4805 of *Lecture Notes in Computer Science*, pages 742–751. Springer, 2007.

[19] G Bakema, JP Zwart, and H Van der Lek. Fully communication oriented information modelling (fco-im). *Academic Service*, 2002.

[20] Herman Balsters. Modelling database views with derived classes in the uml/ocl-framework. In *«UML» 2003 - The Unified modelling Language, modelling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, pages 295–309, 2003.

[21] Herman Balsters, Andy Carver, Terry A. Halpin, and Tony Morgan. modelling dynamic rules in ORM. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET, OnToContent, ORM, PerSys, OTM Academy Doctoral Consortium, RDDS, SWWS, and SeBGIS 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part II*, volume 4278 of *Lecture Notes in Computer Science*, pages 1201–1210. Springer, 2006.

[22] Herman Balsters and Terry A. Halpin. Formal semantics of dynamic rules in ORM. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2008 Workshops, OTM Confederated International Workshops and Posters, ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent + QSI, ORM, PerSys, RDDS, SEMELS, and SWWS 2008, Monterrey, Mexico, November 9-14, 2008. Proceedings*, volume 5333 of *Lecture Notes in Computer Science*, pages 699–708. Springer, 2008.

[23] Herman Balsters and Terry A. Halpin. Formal semantics of dynamic constraints and derivation rules in ORM. *IJISMD*, 7(2):31–47, 2016.

[24] BCP Software. CaseTalk. `https://https://www.casetalk.com/`, 2018.

[25] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG description logic interface. In Diego Calvanese, Giuseppe De Giacomo, and Enrico

Franconi, editors, *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy September 5-7, 2003*, volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[26] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artif. Intell.*, 168(1-2):70–118, 2005.

[27] Michael R. Blaha, William J. Premerlani, and James E. Rumbaugh. Relational database design using an object-oriented methodology. *Commun. ACM*, 31(4):414–427, 1988.

[28] Anthony C. Bloesch and Terry A. Halpin. Conceptual queries using conquer-ii. In David W. Embley and Robert C. Goldstein, editors, *Conceptual modelling - ER '97, 16th International Conference on Conceptual modelling, Los Angeles, California, USA, November 3-5, 1997, Proceedings*, volume 1331 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 1997.

[29] Alexander Borgida and Ronald J. Brachman. Conceptual modelling with description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 349–372. Cambridge University Press, 2003.

[30] Alexander Borgida, Maurizio Lenzerini, and Riccardo Rosati. Description logics for databases. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 462–484. Cambridge University Press, 2003.

[31] Achim D. Brucker and Burkhart Wolff. HOL-OCL: experiences, consequences and design choices. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML 2002 - The Unified modelling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2002.

[32] Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in isabelle/hol. In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2002.

[33] Achim D. Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer, 2008.

[34] Achim D. Brucker and Burkhart Wolff. Featherweight OCL: a study for the consistent semantics of OCL 2.3 in HOL. In Mira Balaban, Jordi Cabot, Martin Gogolla, and Claas Wilke, editors, *Proceedings of the 12th Workshop on OCL and Textual Modelling, Innsbruck, Austria, September 30, 2012*, pages 19–24. ACM, 2012.

[35] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modelling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*. Kluwer, 1998.

[36] Diego Calvanese and Giuseppe De Giacomo. Expressive description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 178–218. Cambridge University Press, 2003.

[37] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Identification constraints and functional dependencies in description logics. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 155–160. Morgan Kaufmann, 2001.

[38] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Daniele Nardi. Reasoning in expressive description logics. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1581–1634. Elsevier and MIT Press, 2001.

[39] Diego Calvanese, Wolfgang Fischl, Reinhard Pichler, Emanuel Sallinger, and Mantas Simkus. Capturing relational schemas and functional dependencies in RDFS. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1003–1011. AAAI Press, 2014.

[40] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Conjunctive query containment and answering under description logic constraints. *ACM Trans. Comput. Logic*, 9(3):22:1–22:31, June 2008.

[41] Andy Carver and Terry A. Halpin. Reference scheme reduction on subtypes in ORM. In Yan Tang Demey and Hervé Panetto, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Workshops - Confederated International Workshops: OTM Academy, OTM Industry Case Studies Program, ACM, EI2N, ISDE, META4eS, ORM, SeDeS, SINCOM, SMS, and SOMOCO 2013, Graz, Austria, September 9 - 13, 2013, Proceedings*, volume 8186 of *Lecture Notes in Computer Science*, pages 457–466. Springer, 2013.

[42] Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Compututing*, 14(3):671–677, 1985.

[43] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[44] Matthew Curland and Terry A. Halpin. The NORMA software tool for ORM 2. In Pnina Soffer and Erik Proper, editors, *Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers*, volume 72 of *Lecture Notes in Business Information Processing*, pages 190–204. Springer, 2010.

[45] Matthew Curland and Terry A. Halpin. Enhanced verbalization of ORM models. In Pilar Herrero, Hervé Panetto, Robert Meersman, and

Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012, Rome, Italy, September 10-14, 2012. Proceedings*, volume 7567 of *Lecture Notes in Computer Science*, pages 399–408. Springer, 2012.

[46] Matthew Curland, Terry A. Halpin, and Kurt Stirewalt. A role calculus for ORM. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009, Vilamoura, Portugal, November 1-6, 2009. Proceedings*, volume 5872 of *Lecture Notes in Computer Science*, pages 692–703. Springer, 2009.

[47] Dmitry Tsarkov. FaCT++. `https://bitbucket.org/dtsarkov/factplusplus/src/master/`, 2019.

[48] Enrico Franconi. ICOM. `https://www.inf.unibz.it/~franconi/icom/`, 2014.

[49] F. Sportelli. UModel - A reasoning framework for conceptual modelling. `https://gitlab.inf.unibz.it/fsportelli/umodel`, 2019.

[50] Eckhard D. Falkenberg. Concepts for modelling information. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 95–109. North-Holland, 1976.

[51] FBM Community. FBM. `http://www.factbasedmodeling.org/contact.aspx/`, 2015.

[52] Pablo R. Fillottrani, Enrico Franconi, and Sergio Tessaris. The new ICOM ontology editor. In Bijan Parsia, Ulrike Sattler, and David

Toman, editors, *Proceedings of the 2006 International Workshop on Description Logics (DL2006), Windermere, Lake District, UK, May 30 - June 1, 2006*, volume 189 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[53] Pablo R. Fillottrani, Enrico Franconi, and Sergio Tessaris. Ontology design and integration with ICOM 3.0 - tool description and methodology. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Proceedings of the 24th International Workshop on Description Logics (DL 2011), Barcelona, Spain, July 13-16, 2011*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[54] Pablo R. Fillottrani, Enrico Franconi, and Sergio Tessaris. The ICOM 3.0 intelligent conceptual modelling tool and methodology. *Semantic Web*, 3(3):293–306, 2012.

[55] Pablo R. Fillottrani, C. Maria Keet, and David Toman. Polynomial encoding of ORM conceptual models in *CFDI*. In *Proceedings of the 28th International Workshop on Description Logics*, 2015.

[56] Francesco Sportelli. ORMiE. `https://gitlab.inf.unibz.it/fsportelli/ormie`, 2019.

[57] Enrico Franconi and Alessandro Mosca. Towards a core ORM2 language (research note). In Yan Tang Demey and Hervé Panetto, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Workshops - Confederated International Workshops: OTM Academy, OTM Industry Case Studies Program, ACM, EI2N, ISDE, META4eS, ORM, SeDeS, SINCOM, SMS, and SOMOCO 2013, Graz, Austria, September 9 - 13, 2013, Proceedings*, volume 8186 of *Lecture Notes in Computer Science*, pages 448–456. Springer, 2013.

[58] Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. Logic foundations of the OCL modelling language. In Eduardo Fermé and Joao Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*. Springer, 2014.

[59] Enrico Franconi, Alessandro Mosca, Xavier Oriol, Guillem Rull, and Ernest Teniente. OCL_{FO}: first-order expressive OCL constraints for efficient integrity checking. *Software & Systems modelling*, pages 1–24, 2018.

[60] Enrico Franconi, Alessandro Mosca, and Dmitry Solomakhin. The formalization of ORM2 and its encoding in OWL2. In *International Workshop on Fact-Oriented modelling (ORM 2012)*, 2012.

[61] Enrico Franconi, Alessandro Mosca, and Dmitry Solomakhin. ORM2 encoding into description logic (extended abstract). In Yevgeny Kazakov, Domenico Lembo, and Frank Wolter, editors, *Proceedings of the 2012 International Workshop on Description Logics, DL-2012, Rome, Italy, June 7-10, 2012*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[62] Enrico Franconi, Alessandro Mosca, and Dmitry Solomakhin. ORM2: formalisation and encoding in OWL2. In Pilar Herrero, Hervé Panetto, Robert Meersman, and Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012, Rome, Italy, September 10-14, 2012. Proceedings*, volume 7567 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 2012.

[63] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. Springer, 2009.

[64] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems modelling*, 4(4):386–398, 2005.

[65] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A uml-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[66] Martin Gogolla and Mark Richters. On constraints and queries in UML. In Martin Schader and Axel Korthaus, editors, *The Unified modelling Language - Technical Aspects and Applications, Workshop "Einsatz, Bewertung und Stand der Unified modelling Language", Mannheim, Germany, November 10-11, 1997*, pages 109–121. Physica-Verlag, Heidelberg, 1997.

[67] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[68] John Guerson, Tiago Prince Sales, Giancarlo Guizzardi, and João Paulo A. Almeida. Ontouml lightweight editor: A model-based environment to build, evaluate and implement reference ontologies. In Jens Kolb, Barbara Weber, Sylvain Hallé, Wolfgang Mayer, Aditya K. Ghose, and Georg Grossmann, editors, *19th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2015, Adelaide, Australia, September 21-25, 2015*, pages 144–147. IEEE Computer Society, 2015.

[69] Giancarlo Guizzardi, Gerd Wagner, João Paulo Andrade Almeida, and Renata S. S. Guizzardi. Towards ontological foundations for conceptual modelling: The unified foundational ontology (UFO) story. *Applied Ontology*, 10(3-4):259–271, 2015.

[70] Volker Haarslev and Ralf Möller. Description of the RACER system and its applications. In Carole A. Goble, Deborah L. McGuinness, Ralf Möller, and Peter F. Patel-Schneider, editors, *Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA, USA, August 1-3, 2001*, volume 49 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[71] Volker Haarslev and Ralf Möller. RACER system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, 2001.

[72] T. Halpin. The orm foundation. `https://www.ormfoundation.org`, 2008.

[73] T. Halpin. Object-role modelling. Technical report, Neumont University, USA, 2012.

[74] Terry Halpin. *A Logical Analysis of Information Systems: Static Aspects of the Data-oriented Perspective*. PhD thesis, Department of Computer Science, University of Queensland, Australia, 1989.

[75] Terry Halpin. Orm 2 graphical notation. *Technical Report ORM2–02*, 2005.

[76] Terry Halpin. *Fact-Oriented modelling: Past, Present and Future*, pages 19–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[77] Terry Halpin. *Object-Role modelling Fundamentals: A practical guide to data modelling with ORM*. Technics Publications, 2015.

[78] Terry Halpin. *Object-Role modelling Workbook: Data modelling Exercises Using ORM and NORMA*. Technics Publications, 2016.

[79] Terry A. Halpin. Information analysis in UML and ORM: A comparison. In Keng Siau, editor, *Advanced Topics in Database Research, Vol. 1*, pages 307–323. Idea Group, 2002.

[80] Terry A. Halpin. Metaschemas for ER, ORM and UML data models: A comparison. *J. Database Manag.*, 13(2):20–30, 2002.

[81] Terry A. Halpin. Comparing metamodels for ER, ORM and UML data models. In Keng Siau, editor, *Advanced Topics in Database Research, Vol. 3*, pages 23–44. Idea Group, 2004.

[82] Terry A. Halpin. FORML position paper for W3C workshop on rule languages for interoperability. In *W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*. W3C, 2005.

[83] Terry A. Halpin. Information modelling in UML and ORM. In Mehdi Khosrow-Pour, editor, *Encyclopedia of Information Science and Technology (5 Volumes)*, pages 1471–1475. Idea Group, 2005.

[84] Terry A. Halpin. ORM 2. In Robert Meersman, Zahir Tari, Pilar Herrero, Gonzalo Méndez, Lawrence Cavedon, David B. Martin, Annika Hinze, George Buchanan, María S. Pérez, Víctor Robles, Jan Humble, Antonia Albani, Jan L. G. Dietz, Hervé Panetto, Monica Scannapieco, Terry A. Halpin, Peter Spyns, Johannes Maria Zaha, Esteban Zimányi, Emmanuel Stefanakis, Tharam S. Dillon, Ling Feng, Mustafa Jarrar, Jos Lehmann, Aldo de Moor, Erik Duval, and Lora Aroyo, editors, *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, GADA, MIOS+INTEROP, ORM, PhDS, SeBGIS, SWWS, and WOSE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings*, volume 3762 of *Lecture Notes in Computer Science*, pages 676–687. Springer, 2005.

[85] Terry A. Halpin. Object-role modelling (ORM/NIAM). In Peter Bernus, Kai Mertins, and Günter Schmidt, editors, *Handbook on Architectures of Information Systems*, International handbooks on information systems, pages 81–103. Springer, 2006.

[86] Terry A. Halpin. Object-role modelling: Principles and benefits. *IJISMD*, 1(1):33–57, 2010.

[87] Terry A. Halpin. Formalization of ORM revisited. In Pilar Herrero, Hervé Panetto, Robert Meersman, and Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2012 Workshops, Confederated International Workshops: OTM Academy, Industry Case Studies Program, EI2N, INBAST, META4eS, OnToContent, ORM, SeDeS, SINCOM, and SOMOCO 2012, Rome, Italy, September 10-14, 2012. Proceedings*, volume 7567 of *Lecture Notes in Computer Science*, pages 348–357. Springer, 2012.

[88] Terry A. Halpin and Anthony C. Bloesch. Data modelling in UML and ORM: A comparison. *J. Database Manag.*, 10(4):4–13, 1999.

[89] Terry A. Halpin, Andy Carver, and Kevin M. Owen. Reduction transformations in ORM. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, volume 4805 of *Lecture Notes in Computer Science*, pages 699–708. Springer, 2007.

[90] Terry A. Halpin and Matthew Curland. Automated verbalization for ORM 2. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET, OnToContent, ORM, PerSys, OTM Academy Doctoral Consortium, RDDS, SWWS, and SeBGIS 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part II*, volume 4278 of *Lecture Notes in Computer Science*, pages 1181–1190. Springer, 2006.

[91] Terry A. Halpin and Matthew Curland. The NORMA tool for ORM 2. In Pnina Soffer and Erik Proper, editors, *Proceedings of the CAiSE Forum 2010, Hammamet, Tunisia, June 9-11, 2010*, volume 592 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[92] Terry A. Halpin and Matthew Curland. Recent enhancements to ORM. In Yan Tang Demey and Hervé Panetto, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Workshops - Confederated International Workshops: OTM Academy, OTM Industry Case Studies Program, ACM, EI2N, ISDE, META4eS, ORM, SeDeS, SINCOM, SMS, and SOMOCO 2013, Graz, Austria, September 9 - 13, 2013, Proceedings*, volume 8186 of *Lecture Notes in Computer Science*, pages 467–476. Springer, 2013.

[93] Terry A. Halpin and Tony Morgan. *Information modelling and relational databases (2. ed.)*. Morgan Kaufmann, 2008.

[94] Terry A. Halpin and Jan Pieter Wijbenga. FORML 2. In Ilia Bider, Terry A. Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, and Roland Ukor, editors, *Enterprise, Business-Process and Information Systems modelling - 11th International Workshop, BP-MDS 2010, and 15th International Conference, EMMSAD 2010, held at CAiSE 2010, Hammamet, Tunisia, June 7-8, 2010. Proceedings*, volume 50 of *Lecture Notes in Business Information Processing*, pages 247–260. Springer, 2010.

[95] Rami Hodrob and Mustafa Jarrar. Mapping ORM into OWL 2. In Ayman Alnsour and Shadi Aljawarneh, editors, *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA 2010, Amman, Jordan, June 14-16, 2010*, page 9. ACM, 2010.

[96] Matthew Horridge, Nick Drummond, John Goodwin, Alan L. Rector, Robert Stevens, and Hai Wang. The manchester OWL syntax. In *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, 2006.

[97] Ian Horrocks, Ulrike Sattler, Sergio Tessaris, and Stephan Tobies. How to decide query containment under constraints using a description logic. In *7th International Conference on Logic for Programming and Automated Reasoning (LPAR00), 2000*, pages 326–343, 2000.

[98] Richard Hull and Roger King. Semantic database modelling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.

[99] Ignazio Palmisano. JFact. `https://github.com/owlcs/jfact`, 2014.

[100] Ignazio Palmisano. OWLAPI. `https://github.com/owlcs/owlapi`, 2018.

[101] Mustafa Jarrar. Mapping ORM into the SHOIN/OWL description logic. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the*

Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I, volume 4805 of *Lecture Notes in Computer Science*, pages 729–741. Springer, 2007.

[102] Mustafa Jarrar. Towards automated reasoning on ORM schemes. In Christine Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *Conceptual modelling - ER 2007, 26th International Conference on Conceptual modelling, Auckland, New Zealand, November 5-9, 2007, Proceedings*, volume 4801 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2007.

[103] Mustafa Jarrar and Stijn Heymans. Unsatisfiability reasoning in ORM conceptual schemes. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers*, volume 4254 of *Lecture Notes in Computer Science*, pages 517–534. Springer, 2006.

[104] Mustafa Jarrar and Robert Meersman. *Ontology Engineering -The DOGMA Approach*, volume 4891 of *LNCS*, chapter 3, pages 7–34. Springer, 2008.

[105] Paris C. Kanellakis. Elements of relational database theory. In A.R. Meyer, M. Nivat, M.S. Paterson, D. Perrin, and J. van Leeuwen, editors, *The Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1075–1144. North Holland, 1990.

[106] C. Maria Keet. Mapping the object-role modelling language ORM2 into description logic language dlrifd. *CoRR*, abs/cs/0702089, 2007.

[107] Inge M. C. Lemmens, Francesco Sgaramella, and Serge Valera. Development of tooling to support fact-oriented modelling at ESA. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Confederated International Workshops and Posters, ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009, Vilamoura, Portugal, November 1-6, 2009. Proceedings*, volume 5872 of *Lecture Notes in Computer Science*, pages 714–722. Springer, 2009.

[108] Shin Huei Lim and Terry A. Halpin. Automated verbalization of ORM models in malay and mandarin. *IJISMD*, 7(4):1–16, 2016.

[109] Thomas Lukasiewicz, Andrea Cali, and Georg Gottlob. A general datalog-based framework for tractable query answering over ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14(0), 2012.

[110] Matthew Curland. NORMA on GitHub. `https://github.com/ormsolutions/NORMA`, 2019.

[111] R Meersman. The ridl conceptual language. Technical report, Research report, International Centre for Information Analysis Services . . . , 1982.

[112] Ralf Möller and Volker Haarslev. Description logics for the semantic web: Racer as a basis for building agent systems. *KI*, 17(3):10, 2003.

[113] João L. R. Moreira, Tiago Prince Sales, John Guerson, Bernardo Ferreira Bastos Braga, Freddy Brasileiro, and Vinicius Sobral. Menthor editor: An ontology-driven conceptual modelling platform. In Oliver Kutz, Sergio de Cesare, Maria M. Hedblom, Tarek Richard Besold, Tony Veale, Frederik Gailly, Giancarlo Guizzardi, Mark Lycett, Chris Partridge, Oscar Pastor, Michael Grüninger, Fabian Neuhaus, Till Mossakowski, Stefano Borgo, Loris Bozzato, Chiara Del Vescovo, Martin Homola, Frank Loebe, Adrien Barton, and Jean-Rémi Bourguet, editors, *Proceedings of the Joint Ontology Workshops 2016 Episode*

2: The French Summer of Ontology co-located with the 9th International Conference on Formal Ontology in Information Systems (FOIS 2016), Annecy, France, July 6-9, 2016, volume 1660 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[114] Mustafa Jarrar. DogmaModeler. `http://www.jarrar.info/Dogmamodeler/`, 2007.

[115] G Nalepa, A Giurca, D Gasevic, and K Taveter. Handbook of research on emerging rule-based languages and technologies: Open solutions and approaches, 2009.

[116] OWLAPI. OWL Functional Syntax Document Format. `https://www.w3.org/TR/owl2-syntax/#Functional-Style_Syntax`, 2009.

[117] Wen-lin Pan and Da-xin Liu. Abstract syntax of object role modelling. In *Fourth International Conference on Machine Vision*, volume 8350, 2011.

[118] Peter F. Patel-Schneider and Enrico Franconi. Ontology constraints in incomplete and complete data. In *ISWC 2012 - 11th International Semantic Web Conference*, volume 7649 of *Lecture Notes in Computer Science*, pages 444–459. Springer-Verlag, 2012.

[119] Rafael Peñaloza. Axiom pinpointing. *CoRR*, abs/2003.08298, 2020.

[120] protegewiki.stanford.edu. Protege Repo. `https://protegewiki.stanford.edu/wiki/Protege_Plugin_Library`, 2020.

[121] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. Ocl-lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, 73:1–22, 2012.

[122] Amir Jahangard Rafsanjani and Seyed-Hassan Mirian-Hosseinabadi. Lightweight formalization and validation of ORM models. *J. Log. Algebraic Methods Program.*, 84(4):534–549, 2015.

[123] Tiago Prince Sales and Giancarlo Guizzardi. Ontological anti-patterns: empirically uncovered error-prone structures in ontology-driven conceptual models. *Data Knowl. Eng.*, 99:72–104, 2015.

[124] Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. Scalable satisfiability checking and test data generation from modelling diagrams. *Autom. Softw. Eng.*, 16(1):73–99, 2009.

[125] Francesco Sportelli. NORMA: A software for intelligent conceptual modelling. In *Proceedings of the Joint Ontology Workshops 2016 (JOWO-2016)*, 2016.

[126] Francesco Sportelli. Supporting conceptual modelling in ORM by reasoning. In Marite Kirikova, Kjetil Nørvåg, George A. Papadopoulos, Johann Gamper, Robert Wrembel, Jérôme Darmont, and Stefano Rizzi, editors, *New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, volume 767 of *Communications in Computer and Information Science*, pages 422–431. Springer, 2017.

[127] Francesco Sportelli and Enrico Franconi. Formalisation of ORM derivation rules and their mapping into OWL. In *ODBASE Conference 2016*, pages 827–843, 2016.

[128] Francesco Sportelli and Enrico Franconi. A formalisation and a computational characterisation of ORM derivation rules. In Hervé Panetto, Christophe Debruyne, Martin Hepp, Dave Lewis, Claudio Agostino Ardagna, and Robert Meersman, editors, *On the Move to Meaningful Internet Systems: OTM 2019 Conferences - Confederated International Conferences: CoopIS, ODBASE, C&TC 2019, Rhodes, Greece, October 21-25, 2019, Proceedings*, volume 11877 of *Lecture Notes in Computer Science*, pages 678–694. Springer, 2019.

[129] Stanford University. Protege. `https://protege.stanford.edu/`, 2019.

[130] Terry Halpin. ORM2 Tech Report. `http://www.orm.net/pdf/ORM2_TechReport1.pdf`, 2005.

[131] Tiago Prince Sales. Menthor. `https://github.com/MenthorTools`, 2018.

[132] David Toman and Grant E. Weddell. Applications and extensions of PTIME description logics with functional constraints. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 948–954, 2009.

[133] University of Cambridge. Isabelle. `https://isabelle.in.tum.de/`, 2020.

[134] University of Oxford. Hermit. `http://www.hermit-reasoner.com/`, 2014.

[135] Victor Morgante. Boston. `https://viev.com/index.php/products/boston-professional`, 2018.

[136] W3C. OWL2. `http://w3.org/TR/owl2-overview/`, 2009.

[137] Heba M Wagih, DSE Zanfaly, and Mohamed M Kouta. Mapping object role modelling 2 schemes into sroiq (d) description logic. *International Journal of Computer Theory and Engineering*, 5(2):232–237, 2013.

[138] J. V. R. Wintraecken. *NIAM Information Analysis Method: Theory and Practice*. Kluwer Academic Publishers, USA, 1990.

[139] Jean-Jacques VR Wintraecken. *The NIAM information analysis method: theory and practice*. Springer Science & Business Media, 2012.

[140] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

[141] C. T. Yu and M. Z. Ozsoyoglu. *Algorithm for tree-query membership of a distributed query.*, pages 306–312. IEEE, 1979.

A

ORM Derivation Rules Taxonomy

This appendix shows a set of ORM diagrams involving ORM Derivation Rules. The appendix is divided into two sections: the first one is about the Subtype Derivation Rules and the second one about the Fact Type Derivation Rules.

A.1 Subtype Derivation Rules

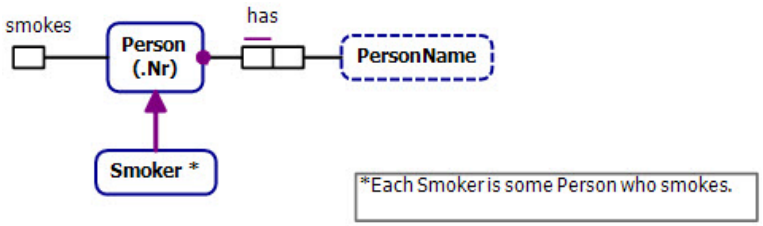


Figure A.1: Simple Subtype Derivation Rule

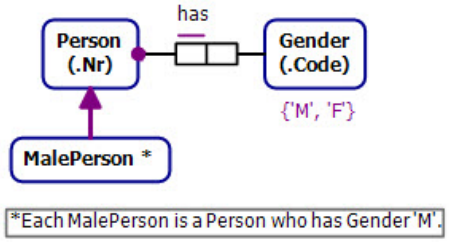


Figure A.2: Value Equality

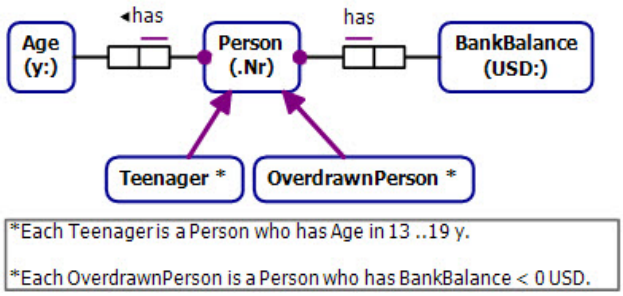


Figure A.3: Other value comparison

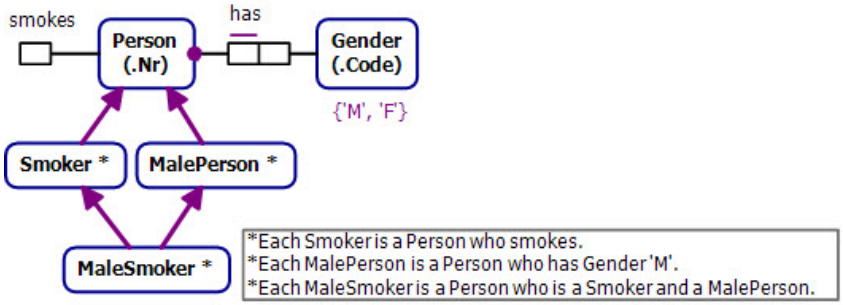


Figure A.4: Multiple Inheritance

A.2 Fact type Derivation Rules

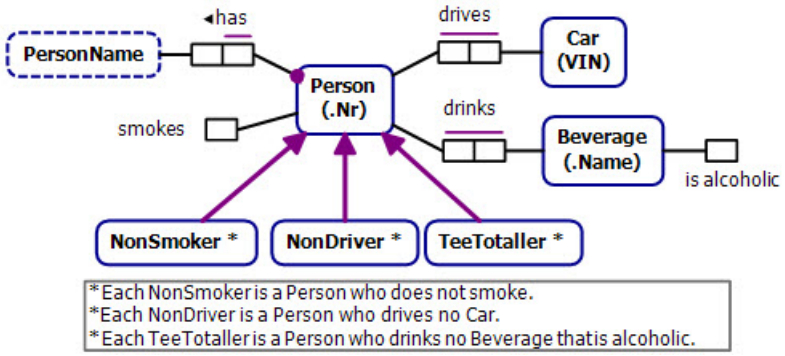


Figure A.5: Negation

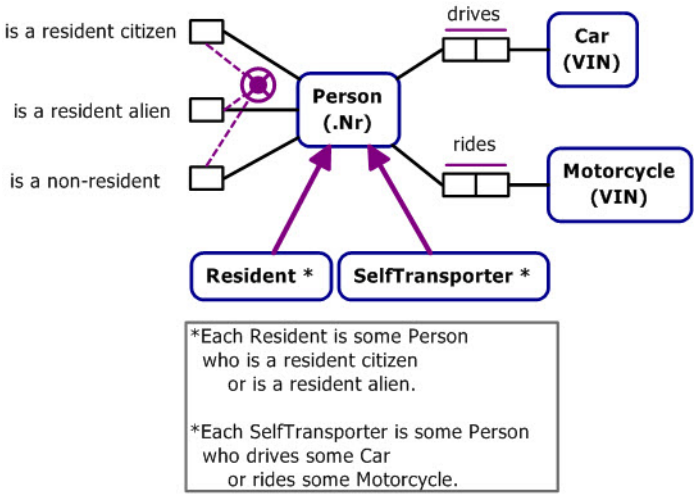


Figure A.6: Disjunction

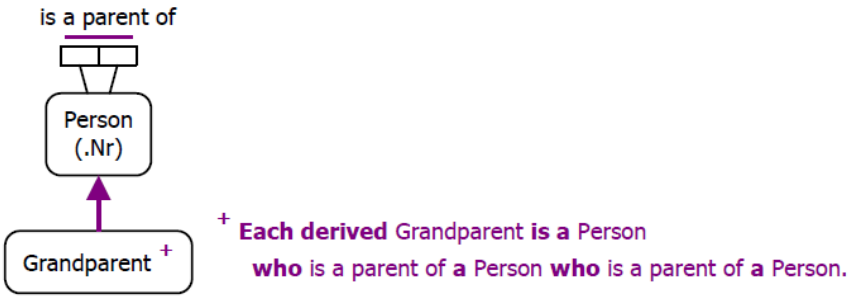


Figure A.7: Semi Derived Rule

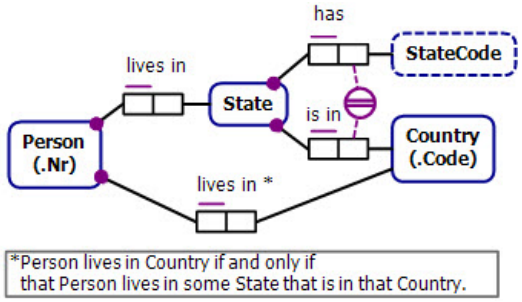


Figure A.8: Linear Path

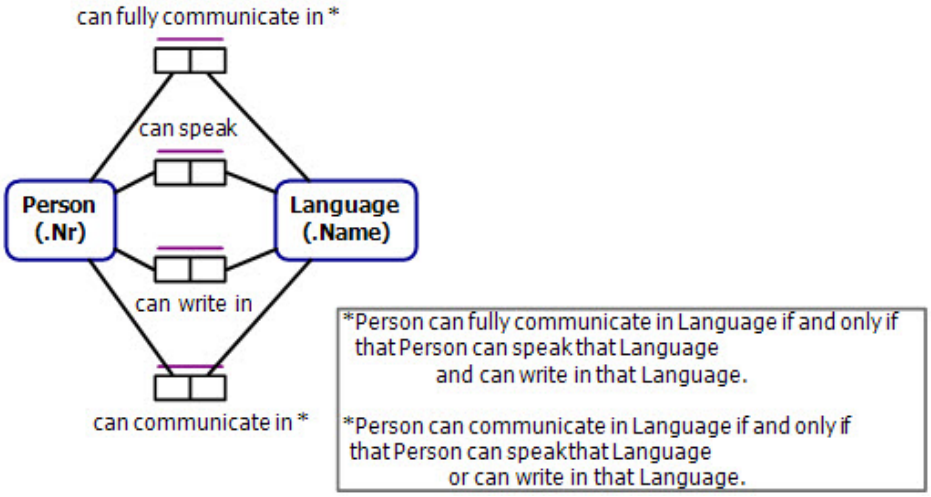


Figure A.9: Correlating Path Variables

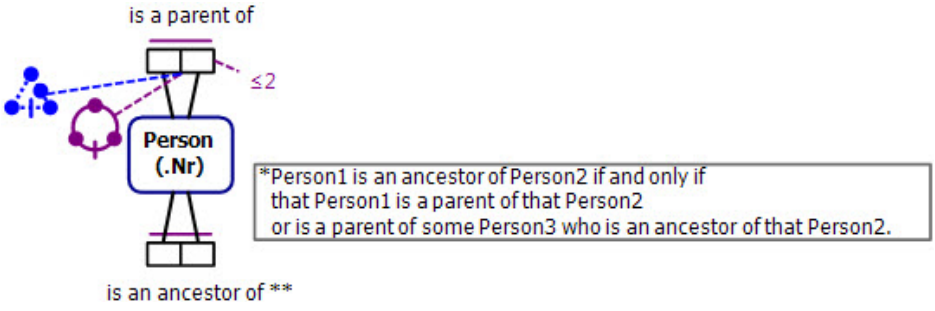


Figure A.10: Recursive Rules

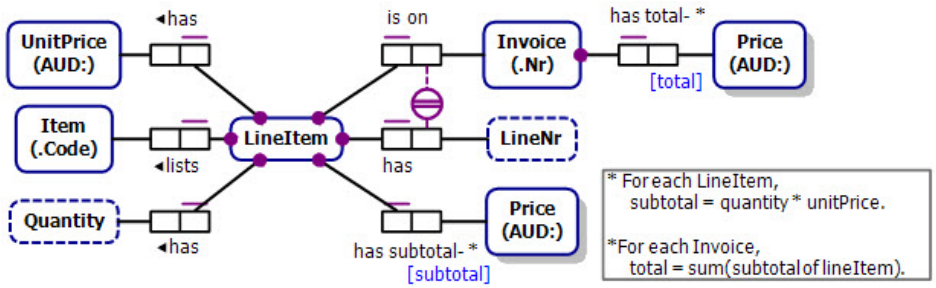


Figure A.11: Simple Calculation

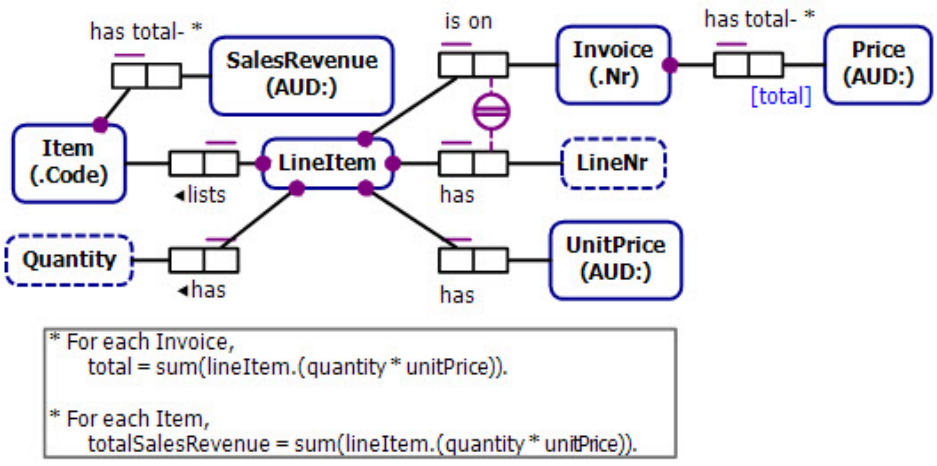


Figure A.12: Aggregate Functions

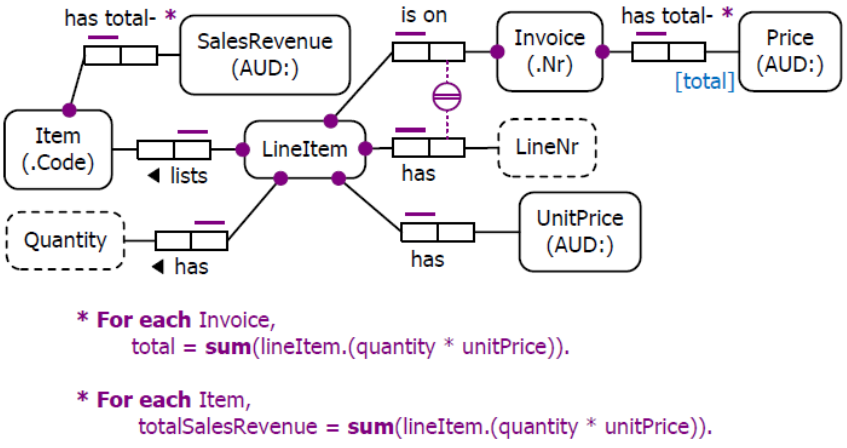


Figure A.13: Nested Calculation

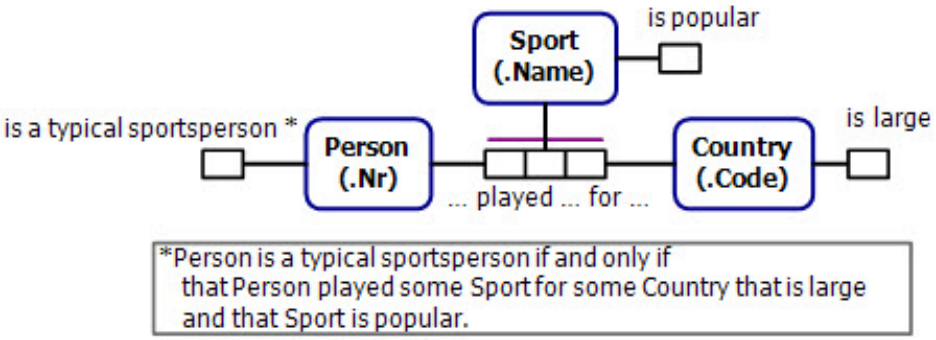


Figure A.14: N-ary Fact Types

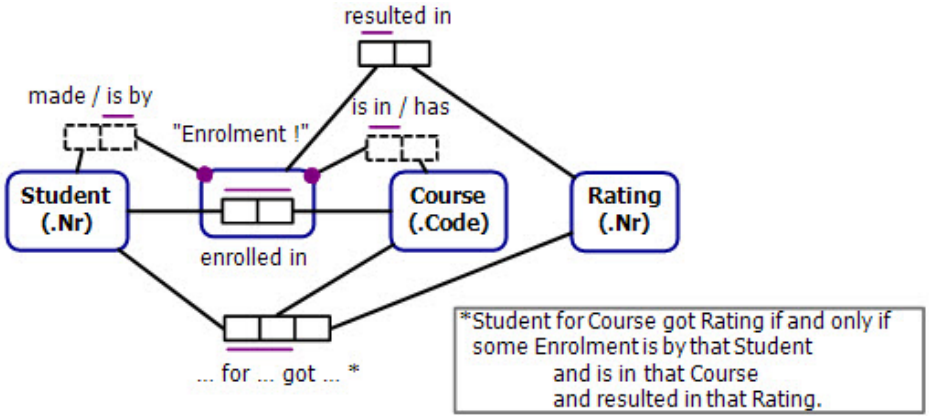


Figure A.15: Objectification

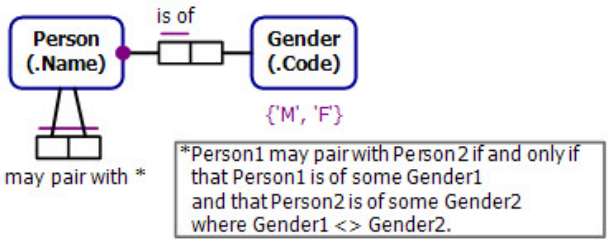


Figure A.16: Cartesian Product

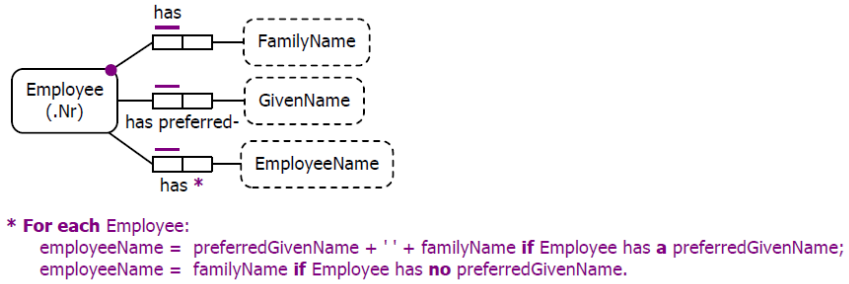


Figure A.17: Multi Path

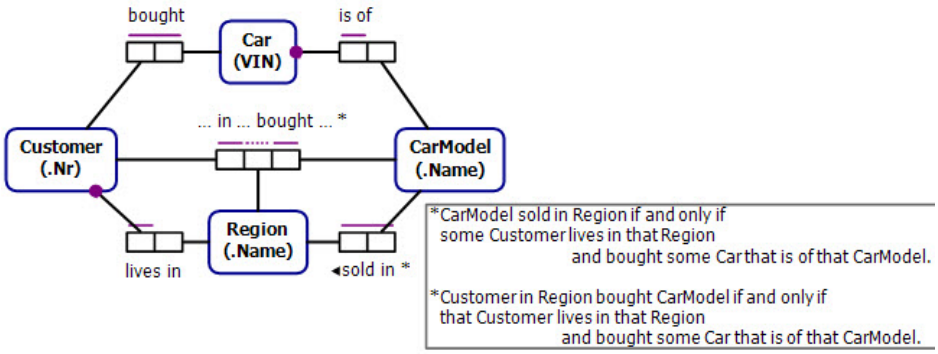


Figure A.18: Shared Path

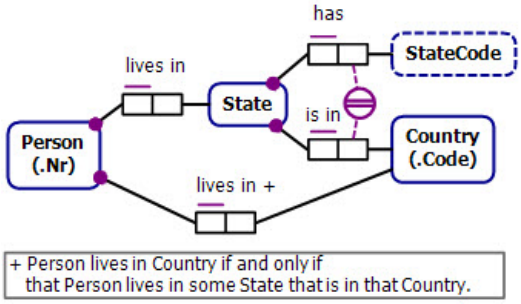


Figure A.19: Semiderived Fact Type

B

List of Figures

List of Tables

D

List of Algorithms