Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
TU UB WIEN Universitätsbibliothek
The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Universal Layer For Schema Mapping Languages

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Florin Ioan Chertes
Matrikelnummer 9225499

an der
Fakultät für Informatik der Technischen Universität Wien
Betreuung: Univ.Prof. Dr. Reinhard Pichler

Diese Dissertation haben begutachtet:

(Dr. Reinhard Pichler) (Dr. Paolo Papotti)

Wien, 01.07.2015

(Florin Ioan Chertes)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Universal Layer For Schema Mapping Languages

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Florin Ioan Chertes

Registration Number 9225499

to the Faculty of Informatics
at the Vienna University of Technology
Advisor: Univ.Prof. Dr. Reinhard Pichler

The dissertation has been reviewed by:

(Dr. Reinhard Pichler) (Dr. Paolo Papotti)

Wien, 01.07.2015

(Florin Ioan Chertes)

Erklärung zur Verfassung der Arbeit

Florin Ioan Chertes
Süssebrunnerstr. 66/4/1 , 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_________________________ _________________________

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

I would like to express my deep gratitude to my advisor, Prof. Reinhard Pichler, for his support and encouragement through the hard years of my studies and writing this thesis. His optimism and capacity of helping to solve even the most difficult problems gave me power to continue, even in those moments when I intended to give up. He opened me horizons, I never could imagine that exist.

I would like to thank and express my gratitude to my co-author, Ingo Feinerer, for his help, support and hard work in writing the publications upon this thesis is based. He supported my difficult steps necessary in transforming a scientific text into a conference paper.

I would like to thank my family. My parents, my wife and my daughter, supported me with their help and love. I am so grateful for their efforts and patience in all these years.

This thesis and the work it is based upon have been supported by the Austrian Science Fund (FWF), project P25207-N23.

Abstract

Schema mappings are central notions, both in data exchange and data integration. They provide a precise formalism for modeling and describing the process of transforming source to target instances of a database in an information exchange scenario. The most common formalism for expressing schema mappings are logical formulae, typically in first-order logic or second-order logic. The use of logics allows for exact definitions of the syntax and semantics of schema mappings, contributing to the success of data exchange in theoretical research during the last decade. Similarly, schema mappings have been of high importance in industrial data exchange applications, e.g., in the well-known IBM Clio mapping tool presented by Popa et al. in 2002 and Fuxman et al. in 2006.

In the industrial context, visual languages for modeling schema mappings have gained increasing importance over the last years. Visual languages hide logical formalisms behind graphical notations and allow users without deep technical and mathematical background to perform data exchange. The graphical notations are suitable interfaces in bringing together stakeholders from different activity fields. This is especially relevant for big data applications, as manual compilation and inspection becomes inherently complex with increasing schema and data size. One of the most influential approaches along this line is CLIP, presented by Raffio et al. in 2008, a visual high level language for schema mappings. CLIP defines a set of custom language elements, modeling source-to-target schema mappings and introducing structural mappings in addition to value mappings.

Nonetheless, we observe a number of drawbacks. First, there is no unified formalism nor standard for the actual constructs of such a visual mapping language: supported language elements depend on the concrete schema mappings tool, thus, each visual language depicts its own graphic elements differently. Second, when automatically generating code from schema mappings, various tools (IBM Clio, Altova MapForce, Stylus Studio, etc.) differ significantly in the number of target languages and the concrete implementation of the rules. Finally, there is a lack

of easy extension mechanisms that allow the user to model additional types of schema mappings, e.g., for second-order dependencies, or mappings in the non-relational case. Consequently, these challenging tasks need to be addressed to foster the applicability of visual languages for schema mapping design in industry. To the best of our knowledge, no comprehensive middleware for visual schema mapping languages exists.

The goal of this thesis is to fill this gap. To this end, we present a new unifying layer for visual schema mapping languages that we call UMAP, which is based on standardized Unified Modeling Language (UML) class diagrams and Object Constraints Language (OCL) constraints. Both, UML and OCL are standard languages of ISO and OMG, a graphical modeling language and a constraint language, respectively. This layer is intended as a middleware, underlying high-level visual languages like CLIP or CLIO but can also be used directly to visually design, model, and maintain schema mappings. This thesis uses OCL as standard query language for schema mappings in the context of data exchange. By using only standardized and well-understood artifacts from the UML modeling language we obtain a precise syntax and semantics for our layer. Most existing UML toolkits support the generation of code from class diagrams, which we use for implementing our schema mappings in various target languages. This important step towards standardization is done in the direction of interoperability between different tools and technologies. This fosters the usability of high level schema mapping languages by opening the access for different such languages to different reference implementations interfaced by UMAP.

Kurzfassung

Ein grundlegendes Konzept in "Data Integration" (Daten-Integration) und "Data Exchange" (Daten-Austausch) sind " Schema Mappings " (Schema-Abbildungen). Dieses Konzept stellt den Formalismus für die Beschreibung und Modellierung der Prozesse zur Verfügung, der Daten von einem Quell-Schema in ein Ziel-Schema transferiert. Der übliche Formalismus, um Schema Mappings abzubilden, sind logische Formeln der Prädikaten-Logik erster und zweiter Stufe. Die Benutzung der Prädikaten-Logik ermöglicht eine präzise Definition der Syntax und Semantik der Schema Mappings, ein essentieller Beitrag zum Erfolg der Forschung im Bereich der Data Exchange des letzten Jahrzehntes. Schema Mappings sind von großer Bedeutung in den industriellen Applikationen wie zum Beispiel in dem sehr bekannten IBM CLIO Schema Mappings Produkt, das von Popa et al. in 2002 und von Fuxman et al. in 2006 präsentiert wurde.

Visuelle Sprachen für Modellierung der Schema Mappings gewinnen in den letzten Jahren stets Bedeutung im industriellen Kontext. Diese visuellen Sprachen verstecken den logischen Formalismus hinter den grafischen Notationen und ermöglichen damit TechnikerInnen, auch ohne tiefe mathematische Kenntnisse, den vollen Zugang zum Data Exchange. Gleichzeitig sind die visuellen Notationen ausgezeichnete Medien, die MitarbeiterInnen aus sehr unterschiedlichen Arbeitsbereichen zusammenbringen, und so eine nahtlose Zusammenarbeit ermöglichen. Dies ist besonders im Zusammenhang mit "Big Data" Applikationen wichtig, da sich manuelle Umwandlung und Kontrolle, durch die Datengröße und Schema-Komplexität, sehr schwierig gestalten. Die Applikation CLIP, die von Raffio at al. in 2008 präsentiert wurde, ist eines der einflussreicheren Produkte aus dem Bereich der visuellen Sprachen für Modellierung von Schema Mappings. CLIP definiert eine Reihe von Sprachelementen für die Modellierung von sogenannten source–to–target (Quelle–zu–Ziel) Schema Mappings und ist sehr bekannt für die Einführung der " Structural Mappings " (Strukturale Abbildungen) zusätzlich zu den schon bekannten "Value Mappings"(Wert Abbildungen).

Nichtsdestotrotz, bemerken wir wichtige Aspekte, die noch verbessert werden können. Ers-

tens gibt es weder einen einheitlichen Formalismus noch einen Standard für die Konstrukte einer solchen visuellen Sprache aus dem Bereich Schema Mappings. Mehr als das, sind diese aktuell etablierten Sprachkonstrukte abhängig von den dazugehörigen Applikationen, sodass konkret, jede visuelle Sprache eigene Sprachkonstrukte hat. Zweitens, wenn Source Code von diesen unterschiedlichen Applikationen (IBM CLIO, Altova MapForce, Stylus Studio, etc.) erstellt wird, dann gibt es Unterschiede in den benutzten Zielsprachen und vor allem in der angewandten Regel, die dies produziert. Schlussendlich mangelt es an Unterstützung für Erweiterungsmechanismen, besonders in Bereichen wie " SO–dependences " (zweite Stufe Abhängigkeiten) und Schema Mappings für nicht relationale Modelle. Daraus folgt, dass Lösungen gesucht gehören, um die industrielle Anwendbarkeit der visuellen Sprachen für Schema Mappings zu verbreiten. Nach unserem besten Wissen gibt es keine etablierte Middleware für visuelle Sprachen aus dem Bereich Schema Mappings.

Das Ziel dieser Dissertation ist, eine Lösung für diese gestellten Fragen zu präsentieren. Wir führen eine " unified layer " (eine vereinigte Schicht) für visuelle Sprachen aus dem Schema-Mappings-Bereich ein, die wir UMAP nennen, welche auf Klassendiagrammen der Standardsspezifikation Unified Modeling Language (UML) und "constraints" (Beschränkungen) der Standardspezifikation Object Constraints Language (OCL) basieren. Die beiden UML und OCL sind Standardspezifikationen der ISO und OMG, einer visuellen Sprache für Modellierung, bzw. einer constraints Sub-Sprache der UML. Die von uns eingeführte Schicht ist gedacht als Middleware zur Unterstützung visueller Sprachen einer höheren Ebene, wie CLIP oder CLIO. Diese Middleware kann auch direkt für Entwurf, Modellierung und Wartung von Schema Mappings benutzt werden. Diese Dissertation benutzt OCL als standardisierte Abfragesprache für Schema Mappings im Kontext von Data Exchange. Wir benutzen nur bestimmte visuelle Elemente der Sprachen UML und OCL und bekommen für unsere UMAP Sprache eine präzise Syntax und Semantik. Fast alle UML Modellierungs-Umgebungen unterstützen die Generierung von Source Code aus Klassendiagrammen, eine Eigenschaft, die es uns ermöglicht, Data Exchange in verschiedene Zielsprachen zu implementieren. Dieser wichtige Schritt in Richtung der Interoperabilität diverser Applikationen und Technologien ist nur durch konsequente Anwendung der Standards möglich. Die Anwendbarkeit vieler wichtiger Schema Mappings Sprachen höherer Ebenen wird steigen, indem der Zugang zu verschiedenen Referenz-Implementierungen durch UMAP ermöglicht wird.

Contents

1 Introduction **1**
1.1 State of the art 1
1.2 Problem Statement 3
1.3 Main Results 4
1.4 Structure and Publications 5

2 Preliminaries **7**

3 Mapping Language **11**
3.1 A simple CLIP mapping 11
3.2 A motivating example: a simple mapping using UML and OCL 12
3.3 Translating language constructs from CLIP to UML and OCL 13
3.4 The UMAP Language 15
3.5 UMAP Layer and Translation of CLIP Core Features 18
3.6 Conclusion 19

4 Complex Mappings **21**
4.1 Context propagation 21
4.2 The UML structure 23
4.3 A more complex mapping 25
4.4 A join constrained by a *Context Propagation Tree* 28
4.5 A mapping with grouping and join 31
4.6 Inverting the nesting hierarchy 35
4.7 A mapping with aggregates 38
4.8 Discussion on the correctness of the UMAP approach 42
4.9 Usage of Skolem functions 44

4.10 Target dependencies, target *egds* and *tgds* 45

5 VMAP the implementation of UMAP **47**

5.1 System architecture 47

5.2 Data exchange scenario 51

5.3 Conclusion 53

6 OCL Compiler **55**

6.1 Introduction 55

6.2 Compiler Architecture 56

6.3 Semantic Analysis And Code Generation 61

6.4 Lexical and Syntactical Analysis 64

6.5 Conclusion 67

7 Related Work **69**

7.1 Overview 69

7.2 The SPICY System 71

8 Conclusion and future work **79**

Bibliography **81**

CHAPTER 1

Introduction

Schema mappings are fundamental notions in data exchange and integration for relating source and target schemas. Visual mapping languages provide graphical means to visually describe such transformations. There is a plethora of tools and languages available. However all use different notions and visualizations making the interoperability between them hardly extensible. This thesis proposes a new universal layer for schema mapping languages, that we call UMAP, which provides a unified abstraction and middleware for high-level visual mapping languages. We use only standardized UML and OCL artifacts, which allow for easy code generation in a number of target languages (e.g. C++ code) and for a modular extension mechanism to support complex schema mappings. We show that translating key elements of established visual mapping languages to UMAP is possible and illustrate that UMAP has enough expressive power to encode fundamental features of them. Moreover, we present a strategy for automating the translation of any visual input language with a formal meta-model to UMAP.

1.1 State of the art

Schema mappings provide a precise formalism for modeling and describing the process of transforming source to target instances of a database in a data exchange scenario. The most common formalism for expressing schema mappings are logical formulae, typically in first-order or second-order logic [13]. The use of logics allows for exact definitions of the syntax and semantics of schema mappings, contributing to the success of data exchange in theoretical research during the last decade [19]. Similarly, schema mappings have been of high importance in indus-

trial data exchange applications, e.g., in the well-known IBM CLIO mapping tool [15, 17, 34]. Next to schema matching and schema mapping generation, the quality of the generated schema mapping solution is a central concern of the SPICY project [20,21,24], one of the most prominent CLIO successors.

Visual languages for schema mappings

However, in an industrial context, visual languages for modeling schema mappings have gained increasing importance over the last years. Visual languages hide logical formalisms behind graphical notations and allow users, without deep technical and mathematical background, to perform data exchange. This is especially relevant for big data applications, as manual compilation and inspection becomes inherently complex with increasing schema. One of the most influential approaches along this line is CLIP [37], a visual language for explicit schema mappings. CLIP defines a set of custom language elements modeling source-to-target and hierarchical schema mappings. CLIP design is influenced by CLIO, but the main difference relative to its predecessor is the introduction of structural mappings in addition to value mappings. This gives the users greater control over the produced transformations by inferring the structural transformation from the value mappings.

Challenges of visual languages for schema mappings

Nonetheless, we observe a number of drawbacks. First, there is no unified formalism nor standard for the actual elements of such a visual mapping language: the introduced elements depend on the concrete schema mappings supported by the tool, and every visual language uses different visualizations for its elements. Second, in the process of automatically generating code from schema mappings, various tools (IBM CLIO, Altova MapForce, Stylus Studio, etc.) differ significantly in their concrete implementations, which are adapted to their own mapping language with no concern of reusing existing ones. Finally, the transformation of a schema mapping setting between two different schema mapping languages is missing. The interoperability between different schema mapping systems is difficult, making the reuse of their components as external modules expensive. Visual languages for schema mapping profiting from higher degree of standardisation will allow the user to model easily additional types of schema mappings, e.g., for second-order dependencies, or mappings in the non-relational case. Consequently, these challenging tasks need to be addressed to foster the interoperability of visual languages for schema

mapping design in industry. To the best of our knowledge no comprehensive middleware for visual schema mapping languages exists to fill this gap.

Query language used to specify schema mappings

Most of the database constraints studied during the 1970s and the 1980s can be expressed as tuple-generating dependencies or equality-generating dependencies. Originally the database constraints were not developed to be used for data exchange. Presenting an overview of the advances in data exchange, Kolaitis [19] characterizes the tuple-generating dependencies as a first order formulae expressing the containment of one conjunctive query in another conjunctive query. The IBM data exchange application, CLIO, uses queries to perform data exchange [10]. There are other well-known formalisms in data exchange and data integration that use intensively conjunctive queries to perform the mappings. The semantics of the visual mapping language CLIP, is defined by a query-like language [19]. This enumeration shows the importance of the query languages used to define the semantics of the schema mappings and consequently used to perform the mapping.

1.2 Problem Statement

The diversity of the query languages employed to support the schema mapping languages raises the question of query language unification and standardization without losing expressive power. The OMG standard language OCL is designed to specify invariant conditions that must hold for the system being modeled or queries over objects described in the model. This thesis proposes OCL as unified standard query language for schema mappings in the context of data exchange. The thesis illustrates that OCL together with the modeling language UML, another OMG standard, have good expressive power building together a rich language capable of introducing the standardisation to the field of high level visual schema mappings. In particular the thesis illustrates how to successfully encode all CLIP mapping features including its main attributes: the nested and structural mappings.

UMAP, the middleware for high-level visual languages

The thesis proposes and defines the syntax and semantics of the visual language UMAP, a new unifying layer for visual schema mapping languages, based on standardized UML class diagrams [29] and OCL constraints [30]. This middleware could be seen also as a practical interface

that allows for different implementations to be used alternatively, improving the modular architecture of industrial frameworks. This layer is intended as a middleware underlying high-level visual languages like CLIP or schema mapping toolkits like CLIO but can also be used directly to visually design, model, and maintain schema mappings. We illustrate our layer by translating all elements of CLIP, a recent expressive visual mapping language.

UMAP and the standards

By using only standardized and well-understood artifacts from the UML modeling language (class diagrams, associations and aggregations) and OCL (selected constraints e.g. straightforward post-conditions and invariants) we obtain a precise syntax and semantics for our layer, which can be translated back to logics [4, 5]. Most existing UML toolkits support OCL and starting from class diagrams they permit the generation of source code into various target languages. Thus, UMAP is not limited to a particular toolkit in implementing schema mappings. The translation from the high-level mapping language to UMAP is achieved using another OMG standard: QVT, an evolving specification for Query, View, Transformation [27]. In this way, schema mappings defined in different languages of the same expressive power as CLIP could be translated via UMAP to a standard programming language. Source code in such a language, as ISO-IEC C++ is compiled to an executable, that performs the mapping. UMAP can also be used as standalone mapping language for data exchange.

1.3 Main Results

We conclude with the presentation of the main achieved results.

- We introduce UMAP, a new universal layer for schema mapping languages and present its syntax and semantics. Schema mappings are modeled with the help of standardized UML graphic artifacts and OCL constraints expressions, both standards having a hierarchical and strict modular structure. By allowing as part of UMAP only certain UML and OCL artifacts, i.e., using only limited subsets of each, this restriction has still its own well-defined syntax and semantics. Under these prerequisites, we can translate UMAP specifications to a broad range of target implementation languages.

- We show how to model central elements occurring in common visual mapping languages via UMAP following a generic strategy defined by the UMAP semantics. As input for our

system, we use the visual mapping language CLIP, a prominent and representative example. We map the core CLIP language elements to our UMAP-based formalism, demonstrating the translation of source-to-target mappings to UML class diagrams augmented with OCL-constraints.

- We show the handling of more complex transformations like joins with grouping in the context of nested schema mappings for tree-like data structures (e.g., necessary for XML data sources) in our proposed formalism. These transformations are characterized by more involved restructuring operations to map the source schema to the target schema. We show that UMAP has enough expressive power to capture also these CLIP features.

- UMAP can be seen as a new middleware for high-level visual schema mapping languages. We propose to use UMAP as a back-end when creating a new visual mapping language with a formal meta-model as it can be easily mapped to UMAP via QVT.

- We present a reference implementation for UMAP, which could be used with any compatible UML-OCL modeling tool, as interactive graphical interface. Our OCL compiler works at the heart of the implementation, designed as a plug-in for the modeling tool. It translates the actual mapping definitions to functions in a programming language, which perform the data exchange. The purpose of the implementation is to offer data engineers a platform for their own visual mapping languages. We also present an implementation that generates C++ code and shows the translation of typical CLIP language elements to our UML-based formalism, illustrating that our approach works in practical applications.

1.4 Structure and Publications

In this section we present the structure of the rest of this thesis and the publications on which each chapter is based.

Chapter 2 Preliminaries. We shortly introduce the notions of schema mappings and dependencies. The language elements of the graphic schema mapping language CLIP are also presented.

Chapter 3 Mapping Language. We start with a CLIP schema mapping example that we transform to our UMAP formalism. Based on this case, we show that UMAP can be used as a graphic schema mapping language too. Further, we introduce all the elements of the graphic schema mapping language UMAP. We define the syntax and semantics of this language.

Chapter 4 Complex Mappings. We show that UMAP, a graphic schema mapping language, has at least the expressive power of CLIP by translating all its reference features to UMAP. We discuss the difficulties encountered in our endeavour. This chapter and the previous two chapters are based on the following publication:

- UMAP: A universal layer for schema mapping languages. [8]
  Florin Chertes and Ingo Feinerer.
  DEXA 2013.

Chapter 5 VMAP, the implementation of UMAP. UMAP is a middleware for high level schema mapping languages. This chapter presents the architecture of VMAP, the implementation of UMAP, that consists of a chain of executables transforming the UMAP model into an executable. Query, View, and Transformation (QVT), an OMG standard, is introduced and we show how various schema mapping models, designed using different high level mapping languages can be transformed to the UMAP layer.

Chapter 6 OCL Compiler. We present the OCL compiler – the heart of the VMAP. The architecture of the compiler is shown with important details about the semantical analysis and the code generation. The UMAP model in XMI format (another OMG standard) is analysed and all the information about graphic artifacts from the UMAP model are collected and used in the phase of semantical analysis. This chapter and the previous one are based on the following publication:

- VMAP: A visual schema mapping tool. [9]
  Florin Chertes and Ingo Feinerer.
  ECAI-PAIS 2014.

Chapter 7 Related work. We consider prominent graphic schema mapping tools that are related to our contribution, a step towards standardisation. The SPICY project that uses external modules for schema mapping generation, is introduced from our perspective of UMAP and VMAP.

Chapter 8 Conclusion and future work. This chapter concludes the thesis. A summary of our contributions is given, first about the UMAP as a new middleware and then about VMAP, its implementation, last the future work is shortly presented.

CHAPTER 2

Preliminaries

Introduction

In this chapter, we give the preliminaries used in the rest of this thesis, which are based on [12, 13, 19]. We give the formal definitions for all necessary concepts, including schemas, schema mappings and classes of dependencies. CLIP, a recent and prominent visual language for schema mapping is introduced. The elements of the language are shortly presented.

Schemas and Schema mappings

A *schema* $\mathbf{R} = \{R_1, \ldots, R_n\}$ is a set of relation symbols R_i each of a fixed arity. An *instance* I over a schema $\mathbf{R}$ consists of a relation for each relation symbol in $\mathbf{R}$, s.t. both have the same arity. For a relation symbol R, we write R^I to denote the relation of R in I. We only consider finite instances here. Let $\mathbf{S} = \{S_1, \ldots, S_n\}$ and $\mathbf{T} = \{T_1, \ldots, T_m\}$ be schemas with no relation symbols in common. A *schema mapping* is given by a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ where $\mathbf{S}$ is the source schema, $\mathbf{T}$ is the target schema, and Σ is a set of logical formulae called dependencies expressing the relationship between $\mathbf{S}$ and $\mathbf{T}$.

Instances over $\mathbf{S}$ (resp. $\mathbf{T}$) are called *source* (resp. *target*) *instances*. We write $\langle \mathbf{S}, \mathbf{T} \rangle$ to denote the schema $\{S_1, \ldots, S_n, T_1, \ldots, T_m\}$. If I is a source instance and J a target instance, then $\langle I, J \rangle$ is an instance of the schema $\langle \mathbf{S}, \mathbf{T} \rangle$.

Given a (ground) source instance I, a target instance J is called a *solution for I under $\mathcal{M}$* if $\langle I, J \rangle \models \Sigma$. The set of all solutions for I under $\mathcal{M}$ is denoted by $Sol(I, \mathcal{M})$.

Dependencies

Source-to-target tuple generating dependencies (s-t tgd) are logical formulae of the form:

$$\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists \overline{y}\psi(\overline{x}, \overline{y})). \tag{2.1}$$

We write $\overline{x}$ for a tuple $(x_1, \ldots, x_n)$. However, by slight abuse of notation, we also refer to the set $\{x_1, \ldots, x_n\}$ as $\overline{x}$. Hence, we may use expressions like $x_i \in \overline{x}$ or $\overline{x} \subseteq X$, etc.

Equality-generating dependencies (egds) are of the form

$$\forall \overline{x}\,(\phi(\overline{x}\,) \rightarrow x_i = x_j) \tag{2.2}$$

with $x_i, x_j \in \overline{x}$.

A *second-order tuple generating dependency (*SO *tgd)* is a logical formula of the form

$$\exists \overline{f}((\forall \overline{x}_1(\phi_1 \rightarrow \psi_1)) \wedge \cdots \wedge (\forall \overline{x}_n(\phi_n \rightarrow \psi_n))) \tag{2.3}$$

where:

- each member of $\overline{f}$ is a function symbol,
- each ϕ_i is a conjunction of atomic formulas of the form $S(y_1, \ldots, y_k)$ (with $S \in \mathbf{S}$ and $y_j \in \overline{x}_i$), and equalities of the form $t = t'$ (with t and t' terms based on $\overline{x}_i$ and $\overline{f}$),
- each ψ_i is a conjunction of atomic formulas of the form $T(t_1, \ldots, t_\ell)$ (with $T \in \mathbf{T}$ where $t_1, \ldots, t_\ell$ are terms based on $\overline{x}_i$ and $\overline{f}$), and
- each variable in $\overline{x}_i$ appears in some atomic formula of ϕ_i.

Nested mappings

In [15] the authors discuss some difficulties encountered by schema mapping tools like CLIO: redundancy in specification and underspecified grouping semantic. In order to address these issues, they proposed an extension of the basic mappings, based on arbitrary nesting of mappings formulas within other mapping formulas. One important characteristic of the nesting mappings is that at each level, there are correlation between the current submappings and the upper-level mappings, nothing is repeated from the upper level, but instead reused.

Visual mapping language CLIP

CLIP is a mapping language for relational and XML schemas. Schema elements are visually connected from source to target by lines interpreted as mappings. Both structural mappings and simple value mappings are supported. The combination of value and structural mappings in CLIP yields expressive language elements extending those from Clio [15, 34], one of the most prominent schema mapping tools, developed by IBM Almaden Research Center and the University of Toronto, and gives users fine-grained control over generated transformations. Mappings are compiled into queries that transform the source instances into target instances. The main CLIP language elements [37, Figure 2] are as follows.

- *Value nodes* store attributes and text.
- *Single elements* consist of a *value node* and have a name.
- *Multiple elements* represent sets of elements.
- *Value mappings* are thin arrows with open ends, in order to map values from source to target.
- *Builders or object mappings* are thick arrows with closed ends connecting elements.
- *Build nodes* have at least one incoming *builder* and at most one outgoing *builder* and express a filtering condition in terms of the variables in the *builders* or enforce a hierarchy of *builders* if connected by *context arcs*.
- *Grouping nodes* are a special kind of *build nodes* used for grouping on attributes.
- *Context propagation trees* are trees with *build nodes* and *context arcs*.

CHAPTER 3

Mapping Language

3.1 A simple CLIP mapping

In this chapter we start to discuss CLIP, the visual schema mapping language. The first simple CLIP mapping scenario is introduced. We show that the same schema mapping can be achieved by using the standard modeling language UML and its extension OCL, a standard constraint language. Based on these standard specifications we introduce our schema mapping language UMAP.

A simple CLIP mapping, adapted from [37, Figure 3] is presented in Figure 3.1: an *employee* is created for each *regEmp* whose salary is greater than *11,000*. For each employee the name is also copied from source to target. The visual mapping language CLIP depicts the schemas as trees. The nodes are *single elements* or *multiple elements*. The source schema is on the left side and the target schema on the right side.

In Figure 3.1, source *regEmp* and target *employee* are *multiple elements*, represented with shaded icons. Each element from the source set is described by its *ename*, a string and its *sal*, an integer, both called *value nodes*. Each element from the target set is described only by its *@name*, a string, also a *value node*. *Value mappings* are thin arrows with open ends connecting value nodes. To show that the name is mapped from source *ename* to target *@name*, a *value mapping* is used. On the arrow a text indicates the mapped value, *$r.ename.value*.

However, the scenario maps *multiple elements* to *multiple elements*. An iterator is needed and this is achieved in CLIP with *builders*. *Builders* are thick arrows with close ends connecting elements and possibly *build nodes*. *Build nodes* have incoming and outgoing *builders* and can

filter variables in terms of the variables on the *builders*. In this scenario the first *builder* starts from source *regEmp*, connects it with a filtering *build node* and the second *builder* connects further the *build node* with the destination, the target *employee*. The text note next to the *build node* defines the filtering condition, only those *regEmp* are selected with a value of *sal* greater than 11,000. Clearly, this connection between source and target is an iterator and filter at the same time.

In [37] it is explained that this simple mapping is still expressible also in CLIO [10], but the rest of the CLIP mappings, despite some of them being only slightly more complex, build for CLIO a lot of difficulties. The authors mention further that this mapping is underspecified: there is no indication how to map the *dept* from the source to *department* on the target. Using the notion of *universal solution* [13] the authors explain that there are at least two such solutions: a universal solution with a generic *department* for each mapped *employee* or a universal solution with a single generic *department* for all mapped *employees*. By adopting the principle of *minimum-cardinality*, the authors prefer the latter solution. The schema mapping scenario from Figure 3.1 translates to the following simple tgd 3.1

$$\begin{aligned}
&\forall d \in source.dept, r \in d.regEmp \quad | \\
&r.sal.value > 11,000 \rightarrow \\
&\exists d' \in target.department, e' \in d'.employee \quad | \\
&e'.@name = r.ename.value
\end{aligned} \tag{3.1}$$

3.2 A motivating example: a simple mapping using UML and OCL

The UML class diagram in Figure 3.2 presents the structure and the OCL expressions define the operations used to map the source to the target. For clarity we present the mapping of the source set *regEmp[0..*]* to the target set *employee[0..*]* without the source *dept* and the target *department* to which they belong. Thus we represent only the essential part of Figure 3 from [37] using a class diagram in Figure 3.2. There are two classes on the source side: a class of type *regEmpSet* and a class of type *regEmp* connected with the previous by aggregation with cardinality *0..**. The class *regEmp* contains two attributes: *ename* of type *string* and *sal* of type *int*. On the target side there are two classes: *employeeSet* and *employee* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *regEmpSet* from the source is connected to the class *employeeSet* from the target by an *association class*:

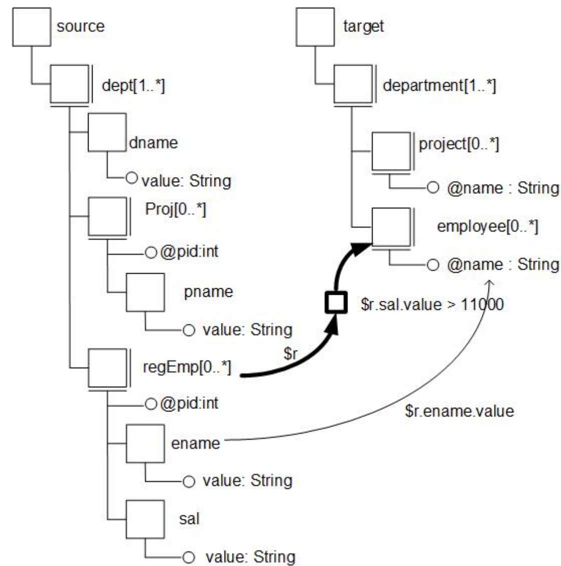

Figure 3.1: A simple CLIP mapping (adapted from [37, Figure 3])

Builder. In the same way the class *regEmp* from the source is connected to the class *employee* from the target by an *association class*: *ValueMap*. Between these two association classes there is an *association* which helps the class *Builder* to access the functionality of the class *ValueMap*. The *Builder* association class iterates through the source set using the function *build*. In each iteration by the help of the association class *ValueMap* each *regEmp* is mapped to an *employee* using the function *map*. These two functions, *Builder::build* and *ValueMap::map* are defined by OCL post-condition expressions.

3.3 Translating language constructs from CLIP to UML and OCL

Translating *value nodes*, *single elements* and *multiple elements*. The previously named classes translate the CLIP structure to UML. Both *Set* classes: *regEmpSet* and *employeeSet*, represent the *multiple nodes* in the CLIP language: *regEmp[0..*]* and *employee [0..*]*. The other two classes *regEmp* and *employee* put together all *value nodes* and *single elements* that structurally belong to the *multiple elements* such as *regEmp[0..*]* and *employee [0..*]*.

Translating *value mappings* and *builders*. The semantics of CLIP *value mappings* and *builders* is achieved in UML through artifacts of the class diagram and OCL expressions. We use in the class diagram the association class *ValueMap* that connects the source type *regEmp* to the target

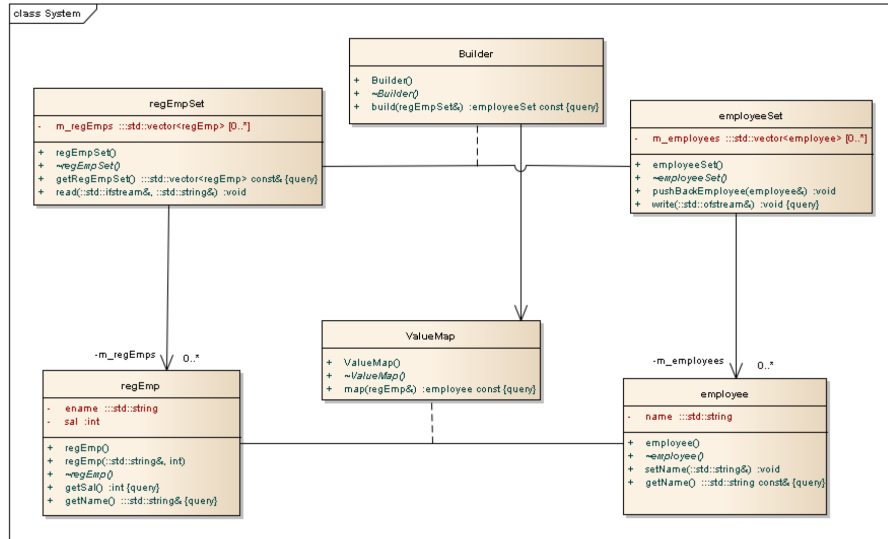

Figure 3.2: A UML class diagram with a simple mapping, corresponding to Figure 3.1

type *employee*. In the UML translation of CLIP *value mapping*, the generation of the target object from the source object is done by the help of the function *ValueMap::map*. This mapping function is defined in OCL as follows:

$$
\begin{aligned}
&context \quad ValueMap :: map(rEmp : regEmp) : employee \\
&post : result = e : employee \quad and \quad e.name = rEmp.ename
\end{aligned}
\tag{3.2}
$$

In the OCLspecification [30, Section 7.6.2] it is mentioned that an operation could be defined by a post-condition. The object that is returned by the operation can be referred to by the keyword `result`. In our case the source–to–target mapping is defined by the equality of the names. Other mapped attributes could be added here if necessary.

In the UML translation of CLIP *builders*, the generation of the target set from the source set is achieved by the use of the function *Builder::build*. This function iterates over the set *regEmpSet* generating the set *employeeSet* and by this models the CLIP *builder*. This mapping function is defined in OCL as follows:

$$\begin{aligned}
&context \quad Builder :: build(rEmpSet : regEmpSet) : employeeSet \\
&post : result = rEmpSet.regEmps \\
&- > select(r : regEmp \quad | \quad r.getSal() > 11,000) \\
&- > collect(r : regEmp \quad | \quad ValueMap.map(r) : employee)
\end{aligned} \tag{3.3}$$

The mapping definition starts from the source set *regEmpSet* and selects only those objects from the source that have a salary greater than 11,000 creating a set. For this selection the standard OCL function *select* is used.

In the next step we obtain another set of type *employeeSet* from this set. This is done by the use of the function *collect*, also a standard OCL function, that applies to each object of type *regEmp* the function *ValueMap::map*. The result is an object of type *employee*. Further the function *collect*, by its OCL definition, inserts all these newly created objects into a set, which is the return value of the function *Builder::build*. The class *Builder* is the translation of the CLIP *builder* because it iterates on the source set, it selects the nodes to be mapped to the target by the help of the OCL *select* standard function and then it creates a set of a different type, using the OCL *collect* standard function. The function *collect* uses the association to the class *ValueMap* to effectively map each object from the source to the target. The class *ValueMap* translates the CLIP *value mappings*. The presentation of this example used implicitly the modeling language for schema mapping that we call UMAP, which is introduced and described in the following chapters.

3.4 The UMAP Language

The previous section showed how to reproduce some central building blocks of a visual modeling language just with UML and OCL constructs. However, in order to provide a well-defined foundation for a middleware like UMAP, we need to restrict the allowed UML constructs and provide a clear generic strategy of translating the input language constructs to UML and OCL expressions generating the same results.

UML constructs. The UMAP language uses a small and well-defined subset of UML and OCL constructs. The main UML building blocks:

- *class*,
- *association class*,

- *association* and a special form of it called *aggregation*,

as defined by the official abstract syntax of the language [29, Section 7] are used by the UMAP language for describing the structure of the source and the target, and for the mapping from source to target. Moreover, the OCL constructs *asSet*, *collect*, *forAll*, *isUnique*, *iterate*, *result*, and *select*, are part of UMAP. According to UML [29, Section 2.1] such a visual language consists of *language units*, which are adapted to a particular paradigm or formalism. Between these language units there are no interdependencies, so they can be used one apart from the other. Some of these language units are partitioned into multiple increments leading to a horizontal stratification of UML. These layers of increasing capability are called *compliance levels*. In UML [29] there are four compliance levels. UMAP uses the language unit *Classes*, and adheres to the fourth level of compliance named *L3* because a main construct of UMAP is the *association class* from the meta model package *Classes::AssociationClasses*.

The *classes* of UMAP describe the structure of the source and the target. In UMAP the aggregation (represented as an association without the diamond notation) is the only connection inside the source or target structures. This means that the structure at the top level includes arrays of other structures that again include arrays. This description defines the source and the target structures as trees. The behavior of the *association class* is used for the actual mapping. The only active class is the *association class* at the top of such a hierarchy. We use the property described in [30, Section 7.6.3] that from an *association class* we can navigate the association-ends. This comes in contradiction with [29, Section 7.3.4] that states the contrary. No matter how this contradiction between these two standards will be solved in the future by the standard committees, the syntax and semantics of UMAP are not affected and minor adapting changes are necessary, only if [30, Section 7.6.3] is going to be modified and aligned with [29, Section 7.3.4].

There are two different usages of the *association class*: one with a function named *build* and the other named *map*, building two types. A UMAP source–to–target schema mapping employs these two types building a tree hierarchy. On each level of this hierarchy one of these types is used alternatively, i.e., a *build* calls a *map* and a *map* calls a *build*. The top level of such a hierarchy starts with the type *build*, called the active class, the one that triggers the mapping. The first type connects top level structures from source–to–target. At the same time this first type is connected through an *association* only to one object of the second type from a lower level in the tree hierarchy. The second type is connected in the same way to zero or more objects of the first type also from a lower level of the tree hierarchy.

OCL constructs. The semantics of the UMAP language describes the transformation of the

source into the target structure. As we claim standard compliance with UML, its semantics uses the standard behavior [28, Section 7.11]. Using the definition of *attribute grammars* [25], additional semantics of UMAP has been achieved through constraints expressed in the OCL language for the following constructs of the UMAP language:

- behavior constructs, i.e., the functions named *build* and *map* of the construct *association class* and
- structure constructs, i.e., attributes of the target structure.

The OCL expressions are implemented in a programming language that can be executed to create and instantiate a mapping. The usage of OCL in UMAP is limited to the following basic constructs: *asSet*, *collect*, *forAll*, *isUnique*, *iterate*, *result*, and *select*. For their semantics we assume the default interpretation as defined by the OCL standard [30].

Another characterization of the semantics of the OCL constructs in UMAP is given by logics. The translation of the OCL constructs *asSet*, *collect*, *forAll*, *isUnique* and *select* to first-order predicate logic is given in [4]. The construct *result* is used only for defining the output. The authors in [4] do not translate the operator *iterate* to first-order predicate logic but propose to express it in higher-order logic.

The active class is the *association class* connecting the top level structures from the source to the target. This *association class* must be from the first type so it must have a function named *build* and it is an iterator on included arrays. This function is described in OCL and makes a selection of the source objects and a transformation of them by calling the other type of *association class*, which is connected by an *association*. On its turn the second type of the *association class* using the function named *map* does the mapping. This is again described in OCL. A second role of the function *map* is to access through the *association* other objects of the *association class* of the first type having a function named *build*. This chain of these two types calling each other executes the mapping, thus defining the semantics of the language UMAP. A program in UMAP is a UML diagram with OCL defining the mapping functions. An input is an instance of the source structure or an instance including such an object. The output of the program is an instance of the target structure. The semantics of the language is the mechanism that transforms the source into the target.

The Language Complexity. As the mappings also use OCL expressions a major concern is their complexity, when implementing them in a programming language. UML/OCL can be translated to first-order predicate logic [4]. The tgds that we use can be considered as first order formulae

expressing the containment of one conjunctive query in another conjunctive query relative to some given database [19]. In this case, the data complexity is tractable but the query complexity and the combined complexity are Π_2^p complete [16, 33]. However, as we restrict the allowed constructs (as defined above) expensive queries are created in rather special cases that hardly reflect the practical scenarios. In fact, the implementation of the OCL functions *iterate*, the most general, or other more specialized as *select* and *collect*, can be done by a limited number of nested loops. This number is bounded by the size of the tgds, which tend to be small.

3.5 UMAP Layer and Translation of CLIP Core Features

Before defining any syntax or semantics the main high-level idea behind UMAP is summarized as follows. The UMAP layer abstracts source and target schemas as UML class diagrams. Without loss of generality, we assume both source and target as XML schemas (since relational schemas can be converted into XML schemas). The schemas represent trees consisting of nodes, attributes and sets of nodes. Individual schema elements, i.e., nodes, are modeled as classes, and attributes in the XML schema become attributes in the corresponding UML class. Sets of elements are modeled as generic container classes encapsulating the underlying class. The actual mappings between source and target schemas are done by association classes augmented by associations between them. We use OCL to specify constraints (post-conditions and invariants) on the association class functions and class attributes to achieve the desired semantics of the mapping.

We have presented above an example translating basic features of CLIP into UMAP. The exact definition of the UMAP language follows in the next chapter. Once we have motivated a set of typical constructs needed, CLIP is a good representative for recent visual mapping languages and has clearly motivated the need for the individual language constructs. We map each CLIP artifact to a UML/OCL artifact of the UMAP layer.

- The CLIP *value nodes* and *single elements* are modeled by class attributes grouped semantically in a class.
- The CLIP *multiple elements* are modeled in UML as generic container classes (sets of elements).
- The *value mappings* are modeled in UML with the help of an association class linking source to target. A class function named *map* implements the mapping.

- The definition of the mappings is achieved through OCL expressions, which include also the filtering conditions. The *builders* or *object mappings* are modeled in UML also with the help of an association class linking source to target. A class function named *build* implements the iterator defined by OCL expressions.

- The associations between association classes model the hierarchy of *builders*.

- The *context propagation tree* is achieved with the help of the hierarchies of association classes and associations between them. Each iterator modeled by a class function named *build* from one level of the hierarchy triggers only a class function named *map* from a lower level in this hierarchy, which maps source to target values.

- The class function named *map* from one level triggers zero or more class functions named *build* from a lower level of the induced hierarchy of functions.

- As a general characteristic of the translation from CLIP to UML the translations of the CLIP *value mappings* and *builders* are association classes, using functions named *map* or *build*. Successive alternations of these two functions correspond to the CLIP feature of a *context propagation tree*.

- Joins and *grouping nodes* are modeled with the help of the OCL expressions, defining class functions and attributes.

3.6 Conclusion

The first important advantage of UMAP is its definition based on ISO and OMG standards for UML and OCL. In this way, the whole range of UML and OCL tools is available. The second important advantage of the UMAP language, when compared to the Clip GUI, is that UMAP hasn't its own GUI, but it can use the GUI of any a modelling tool, provided that the tool supports at least the following standards and features:

- UML and OCL,

- export the modelled classes and their functions to various programming languages,

- export/import the UML/OCL model to XMI and

- call external executables such as VMAP.

Most of the current modelling tools support all these features, e.g., Enterprise Architect. Therefore, the most important benefits of UMAP are:

- it is based on international standards,
- it is supported by most of the current modelling tools,
- it can be used as middle-ware for other high-level schema mapping modelling languages, provided they are formal, i.e., they have a meta-model, a MOF [31] conform description in UML and
- it can use a reference implementations as VMAP.

The Clip GUI is more intuitive because is was specifically tailored for this purpose. UMAP brings more standardisation and can use the GUI of any available commercial modelling tool. UMAP, due to UML/OCL, has at least the expressive power of Clip. As a benefit, all CLIP features are represented in UMAP.

CHAPTER 4

Complex Mappings

4.1 Context propagation

Consider the CLIP [37] mapping with context propagation shown in Figure 4.1. For each *dept* from the source a *department* in the target is created and for each *regEmp* of a *dept* an *employee* of a *department*. The mapped *regEmps* are only those with a salary greater than 11,000. The mapping is performed with the help of two *builders*, each through a *build node* and a *context arc*

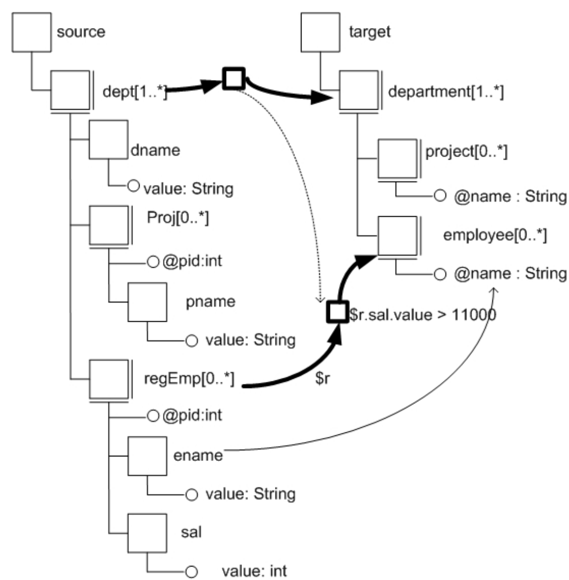

Figure 4.1: A CLIP mapping with context propagation (adapted from [37, Figure 4])

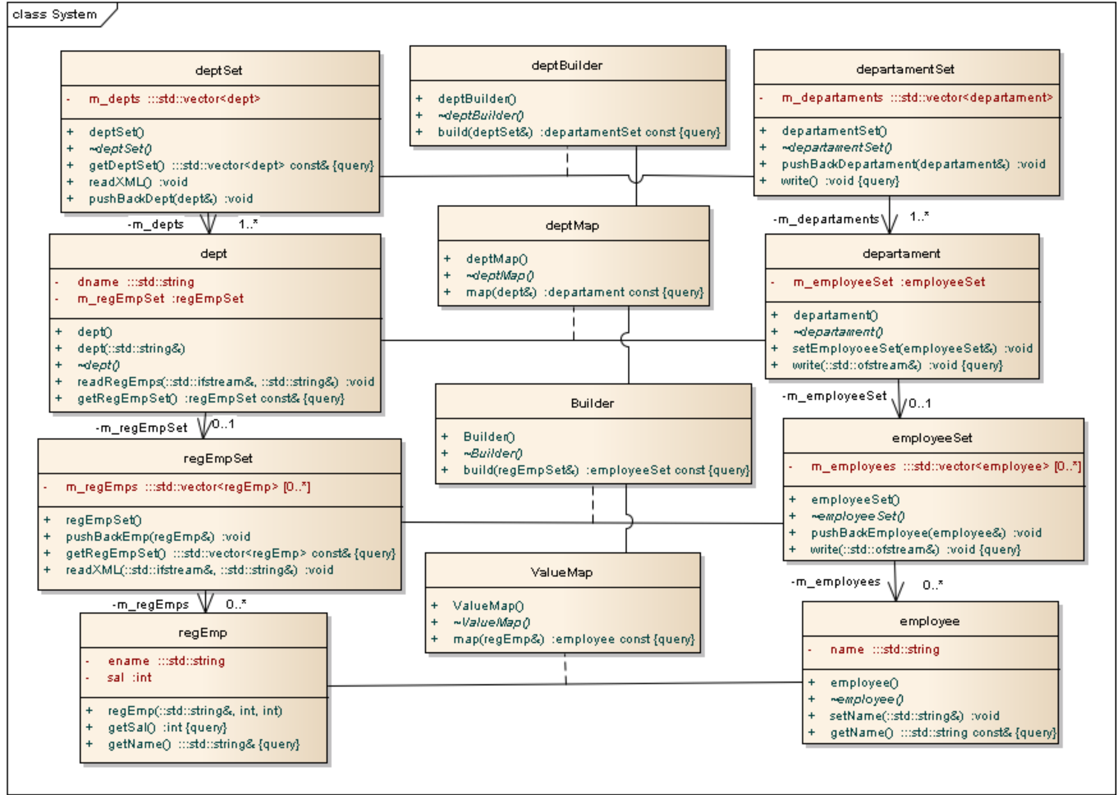

Figure 4.2: The UMAP diagram with context propagation, corresponding to Figure 4.1

connecting the *build nodes*. Thus, a hierarchy of *builders* is created. The *builder* started from *dept* acts as an outer iterator on the *builder* started from *regEmp*, an inner iterator. This has the effect that all *regEmps* of each distinguished *dept* from the source are mapped as the *employees* of the corresponding *departament* in the target. If the *context arc* is omitted in CLIP then all the *employees* are connected to each of the *departments*.

The semantics of this scenario translates to the following tgd:

$$\begin{aligned}
&\forall d \in source.dept \rightarrow \exists d' \in target.department \quad | \\
&[\forall r \in d.regEmp \quad | \quad r.sal.value > 11,000 \rightarrow \\
&\exists e' \in d'.employee \quad | \quad e'.@name = r.ename.value]
\end{aligned} \tag{4.1}$$

The CLIP authors remark that this tgd is similar to the already presented tgd 3.1. But the *context propagation*, the arc connecting the two *build nodes*, constrains the scope of the inner mapping within the context of the outer one. This is expressed by the help of the nested tgd, in

square brackets. The nested tgd is coordinated with the outer tgd by references to the variables *d* and *d'*.

4.2 The UML structure

The UML class diagram in Figure 4.2 presents the structure and the OCL expressions, define the operations used to map the source to the target. Supplementary to the previous case, in which only employees were mapped from source to target, in this case departments with employees are mapped. The mapping of the employees was already presented in Chapter 3. The classes and the associations between them are reused and we repeat only the essential facts. The employees are represented on the source side by the help of two classes: a class of type *regEmpSet* and a class of type *regEmp* connected with the previous by aggregation with cardinality 1..*. On the target side there are two classes: *employeeSet* and *employee* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *regEmpSet* from the source is connected to the class *employeeSet* from the target by an association class: *Builder*. In the same way the class *regEmp* from the source is connected to the class *employee* from the target by an association class: *ValueMap*. Between these two association classes there is an association, which helps the class *Builder* to access the functionality of the class *ValueMap*. The *Builder* association class iterates through the source set using the function *build*. In each iteration, the association class *ValueMap* maps each *regEmp* to an *employee* using the function *map*. Both functions *Builder::build* and *ValueMap::map* are defined by OCL post-condition expressions.

The association class *ValueMap* connects the source type *regEmp* to the target type *employee*. The mapping function is defined in OCL as follows:

$$\begin{aligned}&context \quad ValueMap :: map(rEmp : regEmp) : employee\\&post : result = e : employee \quad and \quad e.name = rEmp.ename\end{aligned} \tag{4.2}$$

The generation of the target set from the source set is achieved by the use of the function *Builder::build*. This function iterates over the set *regEmpSet* generating the set *employeeSet*:

$$\begin{aligned}&context \quad Builder :: build(rEmpSet : regEmpSet) : employeeSet\\&post : result = rEmpSet.m_regEmps- > select(r \quad | \quad r.getSal() > 11,000)\\&- > collect(r : regEmp \quad | \quad ValueMap.map(r) : employee)\end{aligned} \tag{4.3}$$

The mapping definition starts from the source set *regEmpSet* and selects only those objects from the source that have a salary greater than 11,000 creating a set. In the next step we obtain another set of type *employeeSet* from this set. This is done by the use of the function *collect* that applies to each object of type *regEmp* the function *ValueMap::map*. The result is an object of type *employee*. Further the function *collect* inserts all created objects in a set which is the return value of the function *Builder::build*.

The department is represented on the source side by the help of two classes: a class of type *deptSet* and a class of type *dept* connected with the previous by aggregation with cardinality 1..*. On the target side there are two classes: *departmentSet* and *department* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *deptSet* from the source is connected to the class *departmentSet* from the target by an association class: *deptBuilder*. In the same way the class *dept* from the source is connected to the class *department* from the target by an association class: *deptMap*. Between these two association classes there is an association, which helps the class *deptBuilder* to access the functionality of the class *deptMap*. The *deptBuilder* association class iterates through the source set using the function *deptBuilder::build*. In each iteration, the association class *deptMap* maps each *dept* to a *department* using the function *deptMap::map*. Both functions *deptBuilder::build* and *deptMap::map* are defined by OCL post-condition expressions.

$$\begin{aligned}
&context \quad deptMap :: map(dep : dept) : department \\
&post : result = d : department \quad and \\
&d.m_employeeSet = Builder.build(dep.m_regEmpSet)
\end{aligned} \tag{4.4}$$

The input object of this operation is of type *dept* and the object that results is of type *department*. The input object includes a set of source type *regEmpSet*, which is mapped to a set of target type *employeeSet*. This is done by the function *Builder::build*, already presented. If needed, other target attributes, based on source attributes, could be defined here. The translation of the CLIP construct *builder*, generating a *department* set from a *dept* set is done by the function *deptBuilder::build*. This function iterates over the source set, generating the target set and so translating the CLIP *builder*:

$$\begin{aligned}
&context \quad deptBuilder :: build(dSet : deptSet) : departmentSet \\
&post : result = dSet.m_depts- > asSet() \\
&- > collect(r : dept \quad | \quad deptMap.map(r) : department)
\end{aligned} \tag{4.5}$$

The post-condition starts from the source set *deptSet* and selects only those departments fulfilling some conditions. In this particular case, there are no conditions, so all the departments are selected. The next step is to obtain from this set another set of type *departmentSet*. This is done by the function *collect* that applies to each object of type *dept* the function *deptMap::map*. The result is an object of type *department*. Further the function *collect* inserts all these newly created objects in a set, which is the return value of this function. The class *deptBuilder* is the translation of the CLIP *builder* because it iterates on the source set, it selects the nodes to be mapped to the target by the help of the OCL *select* standard function and then it creates a set of a different type, using the OCL *collect* standard function.

4.3 A more complex mapping

A more complex CLIP mapping is presented in Figure 4.3. The hierarchy of builders, i.e., builders connected by context arcs, enforces the propagation of the outer iterator context to the inner iterators on *Proj* and *regEmp*. The authors explain in [37] that CLIP can achieve, through this configuration, the mapping of *depts* with *Projs* and *regEmps* from the source to the target without loss of the structure what no other state-of-the-art-tools like Clio can do.

The semantics of the scenario presented in Figure 4.3 translates to the following tgd:

$$
\begin{aligned}
&\forall d \in source.dept \rightarrow \exists d' \in target.department \quad | \\
&[\forall p \in d.Proj \rightarrow \exists p' \in d'.project \quad | \\
&p'.@name = p.pname.value] \quad \wedge \\
&[\forall r \in d.regEmp \quad | \quad r.sal.value > 11,000 \rightarrow \\
&\exists e' \in d'.employee \quad | \quad e'.@name = r.ename.value]
\end{aligned} \tag{4.6}
$$

The main idea of translating from CLIP into UML, developed in the previous case, is used and the results are presented in the class diagram Figure 4.4. The class diagram has on the source side six classes: *dept*, *Proj* and *regEmp* and the set variant of each. The association class *deptBuilder* using the other association class *deptMap* triggers the two inner iterators of the association classes *Builder* and *projBuilder*. At each outer iteration step, the function *deptMap::map* is triggered and the inner iteration transforms all the qualified *regEmp* and *Proj* objects from the source to *employee* and *project* objects of the target types. Then the outer iteration inserts them into the corresponding *department* on the target. Both inner iterators, the association classes:

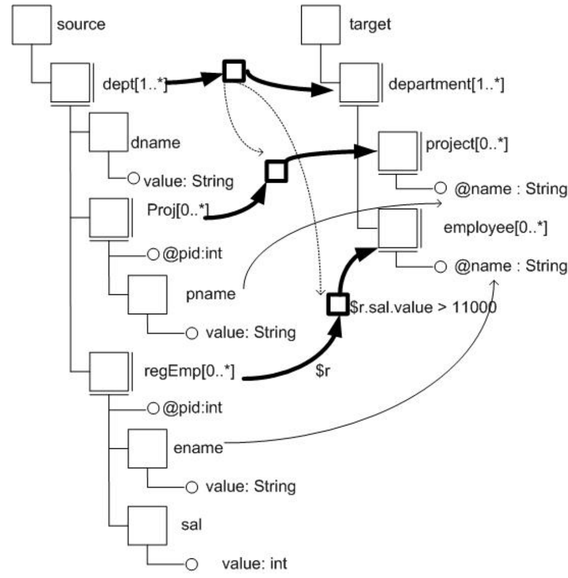

Figure 4.3: A more complex CLIP mapping (adapted from [37, Figure 5])

projBuilder and *Builder* are triggered using the associations existing between them and the association class: *deptMap*, the outer iterator. All the functions are defined in OCL and are similar to those used in the precedent two cases.

The OCL expressions for the mappings included in the association classes *ValueMap* and *Builder* were already presented in the first two cases. The corresponding OCL expressions for the association classes *projMap* and *projBuilder* are presented next.

$$\begin{aligned} &context \quad projMap :: map(p : Proj) : project \\ &post : result = pr : project \quad and \quad pr.name = p.pname \end{aligned} \tag{4.7}$$

This OCL expression defines the function *projMap::map* which maps a source object of type *Proj* to a target object of type *project*.

$$\begin{aligned} &context \quad projBuilder :: build(pSet : ProjSet) : projectSet \\ &post : result = pSet.m_Projs \\ &- > Set()- > collect(r : Proj \quad | \quad projMap.map(r) : project) \end{aligned} \tag{4.8}$$

This OCL expression defines the function *projBuilder::build* which maps a source set of *Proj* objects to a target set of *project* objects.

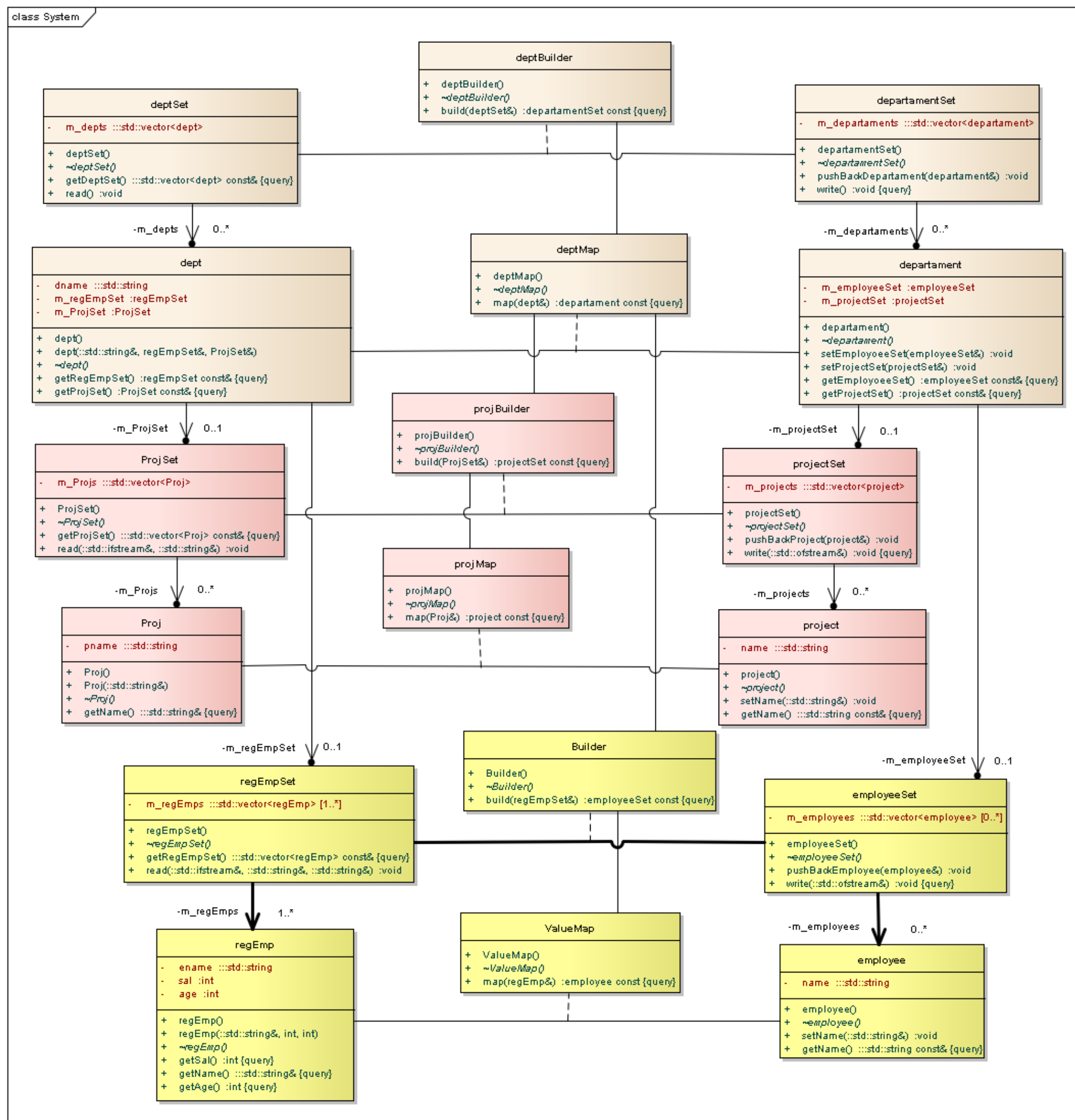

Figure 4.4: The UMAP diagram with a more complex mapping, corresponding to Figure 4.3

The OCL expression for the association class *deptMap* is similar to that of the second case. This function has to define the mapping of two sets at the same time.

$$\begin{aligned}
&context \quad deptMap :: map(dep : dept) : department \\
&post : result = d : department \quad and \\
&d.m_employeeSet = Builder.build(dep.m_regEmpSet) \quad and \\
&d.m_projectSet = projBuilder.build(dep.m_ProjSet)
\end{aligned} \tag{4.9}$$

The translations of the CLIP *builders* or *value mappings* to UML are association classes using respectively functions named *build* or *map*. These association classes are used alternatively at successive levels developing a hierarchy of functions. The function named *build* is always situated at the top level of the hierarchy and *map* at the bottom level. The function *build* from the top level or from another level of the hierarchy calls always only one function named *map* from a successive lower level and vice versa: a *map* function calls one ore more functions named *build* from a successive lower level. As already mentioned the function at the lowest level, the bottom level of the hierarchy, is always named *map*. These successive levels of alternations using functions named *build* and *map* translate the CLIP feature called Context Propagation Tree (CPT) to UML.

The OCL expression of the function *deptBuilder::build* is identical to the one with the same name in the second case above.

This illustrates again that UML diagram produces the same mapping as the CLIP diagram.

In contrast to the previous CLIP features, the next mappings join multiple source elements and create multiple target elements. This means that both functions *map* and *build* create sets of the same type, with the consequence that both *association classes* containing these functions are connected on the target side to the same class. The function *map* uses the join to map from target to source, while the function *build* iterates over the set *deptSet* and inserts target sets produced by the function *map* into the union of target sets.

4.4 A join constrained by a *Context Propagation Tree*

When a *build node* is reached from two or more source schema nodes, as in Figure 4.6, CLIP computes the *Cartesian product* of the sources selected by each *builder*. A filtering condition can be added on the label of the *build node*. If this condition involves two different variables, CLIP computes a join between the source data selected by the *build node*. The topmost *build node*

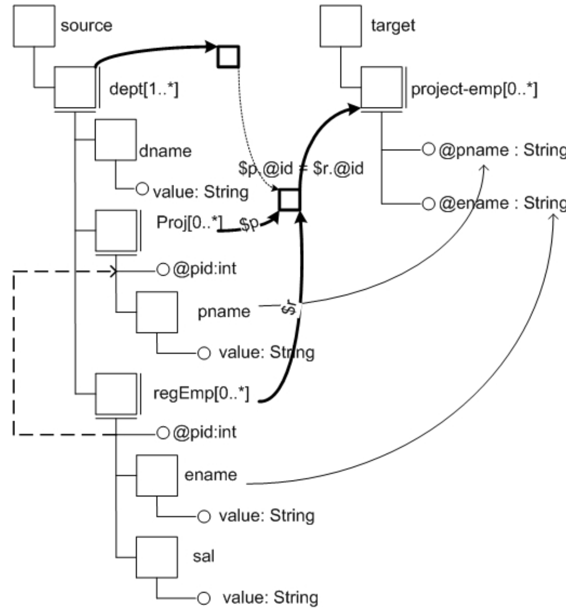

Figure 4.5: A CLIP mapping with a join constrained by a CPT (adapted from [37, Figure 6])

has no output builder. The *context arc* as shown in Figure 4.5 restricts the *Cartesian product* of *Projs* and *regEmps* to nodes within the same *dept*. The result is a flattened list of employees and projects, in which they work. The UMAP class diagram in Figure 4.6 has on the target side the class *project_empSet*. As in CLIP, *dept* is not mapped in the target.

The semantics of this CLIP scenario transforms to the following tgd:

$$\begin{aligned}
&\forall d \in source.dept \rightarrow \\
&[\forall p \in d.Proj, r \in d.regEmp \quad | \quad p.@pid = r.@id \rightarrow \\
&\exists p' \in target.project_emp \quad | \\
&p'.@pname = p.pname.value, \\
&p'@ename = r.ename.value]
\end{aligned} \tag{4.10}$$

The UML translation is based on the definition of the *Cartesian Product* in OCL by [1]. The association class *project_empSetBuilder* starts the iteration over the elements of the set *deptSet*. Each iteration maps one object of type *dept* to a set of objects *project_emp* obtained from the join of the *Proj* and *regEmp* objects of each *dept* on the attribute *pid*. The OCL expressions define the join by constructing first a *Cartesian Product* and then a selection of the elements with the same attribute *pid* associating each employee with the projects, in which she works. In this case, the

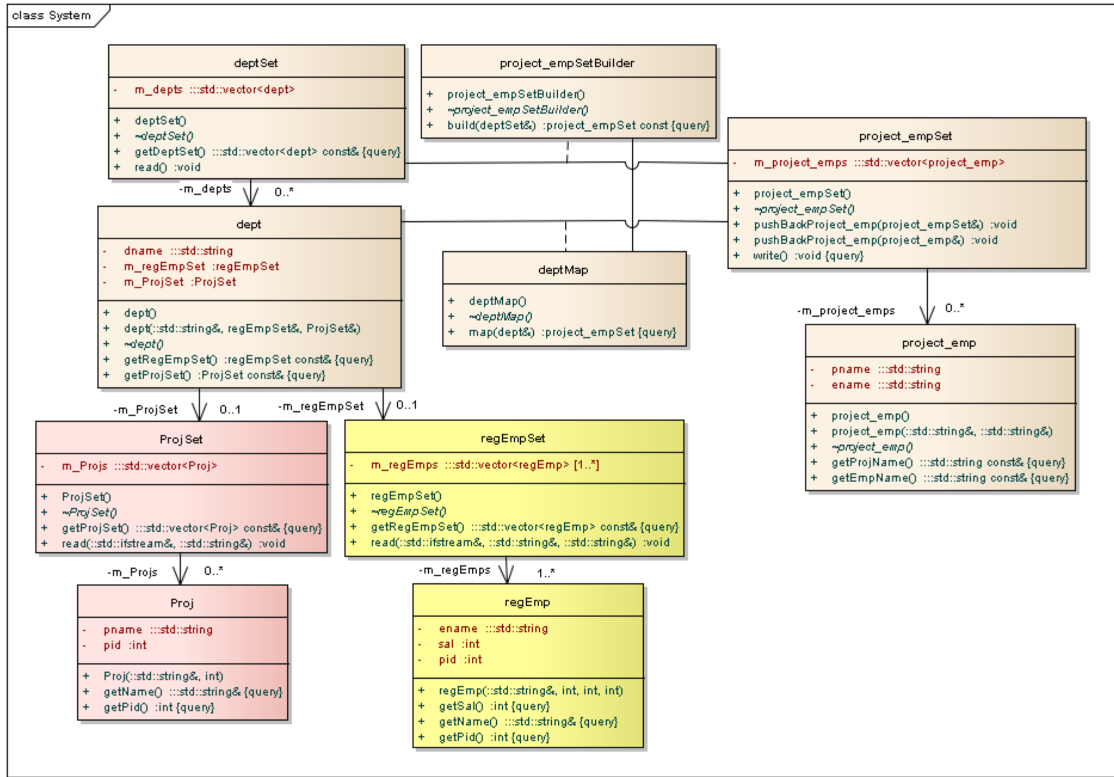

Figure 4.6: The UMAP diagram with a join, corresponding to Figure 4.5

association class *deptMap* creates from each object *dept* a set of *project_emp*. The association class *project_empSetBuild* using this functionality maps the set of *dept* objects to the union of sets of *proj_emp* objects. We define the OCL expression for the function *deptMap::map* as

$$\begin{aligned}
&context \quad deptMap :: map(dep : dept) : project_empSet \\
&def : projProdEmp = dept.m_Projset \\
&- > collect(p : Proj \quad | \quad dept.m_regEmpSet \\
&- > collect(e : regEmp \quad | \quad Proj_regEmp : TupleProj, regEmp))
\end{aligned} \tag{4.11}$$

This is the *Cartesian Product* of the two sets included in a *dept*. The result is a set of tuples composed of a *regEmp* and a *Proj* each.

$$def : projJoinPidEmp = projProdEmp$$
$$- > select(Proj_regEmp \quad | \tag{4.12}$$
$$Proj_regEmp.Proj.pid = Proj_regEmp.regEmp.pid)$$

The join is obtained by its definition from the *Cartesian Product* by selecting those tuples with the same *pid*.

$$post : result = projJoinPidEmp- > collect(Proj_regEmp \quad | \quad project_emp($$
$$Tuple\{pname = Proj_regEmp.Proj.pname, \tag{4.13}$$
$$ename = Proj_regEmp.regEmp.ename\}))$$

The result of this operation is a set of *project_emp* objects containing the name of the project and the name of the employee working in that project. The OCL definition of the function *project_empSetBuilder::build* uses the function *deptMap::map*.

$$context \quad project_empSetBuilder :: build(dSet : deptSet) : project_empSet$$
$$post : result = dSet.m_depts- > Set()- > iterate(r : dept;$$
$$peS : project_empSet = Bag \quad | \quad peS.pushBackProject_emp(deptMap.map(r))) \tag{4.14}$$

This function iterates over the set of *dept* objects and produces from each of them a set of *project_emp* objects and these sets are inserted in the *project_empSet*, a union of sets. This ensures that the CLIP join and the described UML class diagram produce the same mapping.

4.5 A mapping with grouping and join

In CLIP, *group nodes* are used to group source data on attributes. Figure 4.7 depicts such a construct. The result of a *group node* is a sequence of elements selected by the grouping attributes. The number of created sequences on the target equals the number of distinct values of the grouping attributes from the source. In Figure 4.7 the *Projs* are grouped by *pname*. The *Projs* and *regEmps* are joined by *pid* and finally the *employees* on the target are created and added to the *project* by name independently of the *dept*, in which they work.

The semantics of this CLIP scenario translates to the following tgd:

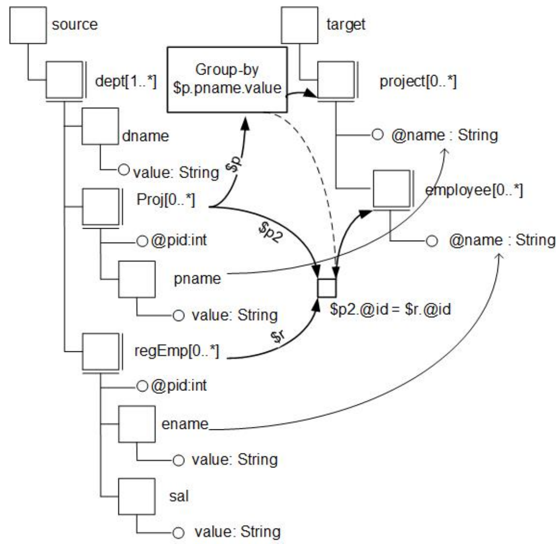

Figure 4.7: A CLIP mapping with grouping and join (adapted from [37, Figure 7])

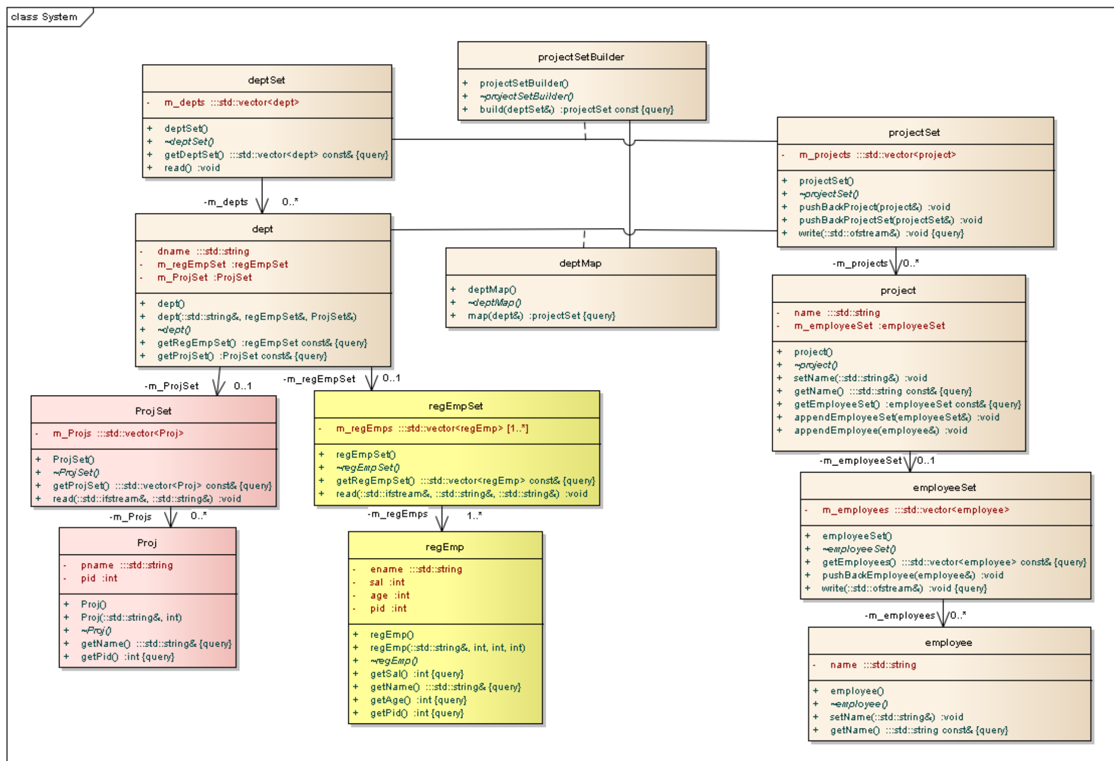

Figure 4.8: The UMAP diagram with grouping and join, corresponding to Figure 4.7

$$
\begin{aligned}
&\exists group_by \\
&(\forall d \in source.dept, p \in d.Proj \rightarrow \\
&\exists p' \in target.project \quad | \\
&p' = group_by(\perp, [p.pname.value]), \\
&p'.@name = p.name.value, \\
&[\forall d_2 \in source.dept, p_2 \in d_2.Proj, r \in d_2.regEmp \quad | \\
&p_2.@pid = r.@pid \rightarrow \\
&\exists e' \in p'.employee \quad | \quad e'.@name = r.ename.value])
\end{aligned}
\tag{4.15}
$$

The class diagram in Figure 4.8 is the corresponding translation to UMAP for the CLIP mapping scenario with join and grouping. The association class *projectSetBuilder* starts the iteration over the elements of the set *deptSet*. Each iteration using the function *deptMap::map* maps one object of type *dept* to an object of type *projectSet*. This set is obtained from the join of the *Proj* and *regEmp* objects of each *dept* on the attribute *pid*. On the target each *project* includes its *employees*. The function *deptMap::map* inserts each *project* into the *projectSet*. Each insert groups the *project* objects by name. In this case OCL expressions do not give a constructive solution but the OCL constraints define the possible implementations. The attribute *m_projects* of the type *projectSet* from the target is specified in OCL by the following expression:

$$
\begin{aligned}
&context \quad projectSet.m_projects \\
&inv : self- > isUnique(p : project \quad | \quad p.name)
\end{aligned}
\tag{4.16}
$$

This means that the elements of the set, the *project* objects, are unique by name. In this way the grouping by project name is achieved. In the UML diagram the type *project* has a set of objects of type *employee*. Because of this structure the only possible grouping is to attach all the employees to the project, in which they work. If two or more projects have the same name by the uniqueness of the project name the employees of these projects are again grouped together. This is valid by the structure of the UML diagram also for projects in different departments. The OCL expressions give the definition of the join by constructing first a *Cartesian Product* and then a selection of the elements with the same attribute *pid* associating each employee with the projects, in which she works. It follows the OCL expression for the function *deptMap::map*:

$$context \quad deptMap :: map(dep : dept) : projectSet \tag{4.17}$$

The OCL expressions defining the *Cartesian Product* and the join on *pid* have already been presented in the previous subsection.

$$\begin{aligned}&def : result_lhs = projJoinPidEmp- > collect(Proj_regEmp \quad | \\ &Tuple\{pname = Proj_regEmp.Proj.pname, \\ &ename = Proj_regEmp.regEmp.ename\})\end{aligned} \tag{4.18}$$

This OCL expression creates all the tuples from the source that are to be grouped on *project* name in the target by the mapping,

$$\begin{aligned}&def : result_rhs = projectSet.m_projects \\ &- > collect(p \quad | \quad p.m_employeeSet.m_employees \\ &- > collect(e \quad | \quad proj_emp : Tuple\{p : project, e : employee\}))\end{aligned} \tag{4.19}$$

creates the *Cartesian Product* of the tuples from the target, and

$$\begin{aligned}&post : result = projectSet(result_lhs) \quad and \\ &result_lhs- > forAll(pe \quad | \\ &result_rhs- > exists(proj_emp \quad | \\ &proj_emp.pname = pe.pname \quad and \\ &proj_emp.ename = pe.ename))\end{aligned} \tag{4.20}$$

defines the constraint that all tuples from the source must have a correspondent in the target. All elements from the target are created only from the source so it is not necessary to show that all elements from the target are only those that are created by the mapping. Every possible implementation must fulfill these constraints. The association class *deptMap* connects the class *dept* from the source with class *projectSet* from the source. The function *deptMap::map* transforms a *dept* to a *projectSet*. The association class *projectSetBuilder* connects the class *deptSet* from the source with class *projectSet* from the target.

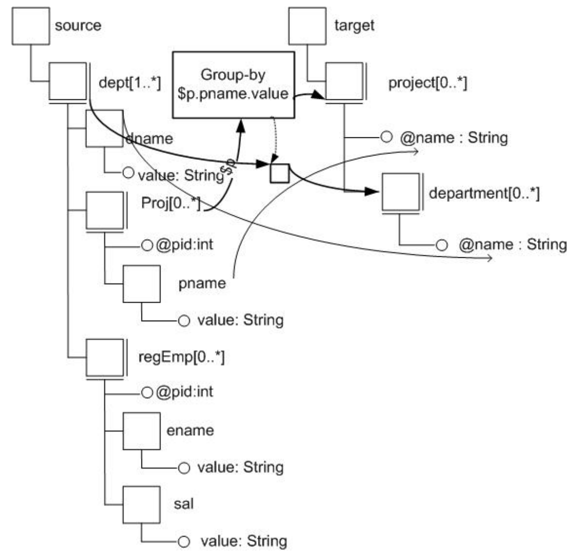

Figure 4.9: A CLIP mapping with inverting the nesting hierarchy (adapted from [37, Figure 8])

$$\begin{aligned}
&context \quad projectSetBuilder :: build(dSet : deptSet) : projectSet \\
&post : result = dSet.m_depts- > Set()- > iterate(r : dept; \\
&pS : projectSet = Bag\{\} \quad | \\
&pS.pushBackProjectSet(deptMap.map(r)))
\end{aligned} \tag{4.21}$$

The function *projectSetBuilder::build* transforms the source to the target. The OCL definition of the function *projectSetBuilder::build* uses the function *deptMap::map*.

4.6 Inverting the nesting hierarchy

Another CLIP feature, a *group node* with inverted hierarchy is presented in Figure 4.9. The source data is mapped to the target and as in the previous example, grouped on attributes, i.e., the mapping groups the *projects* by *name*. The *departments* are nested under the grouped *projects*, recall that in the source the *depts* have nested *Projs* hence, the inverted hierarchy. This CLIP scenario translates to the following tgd:

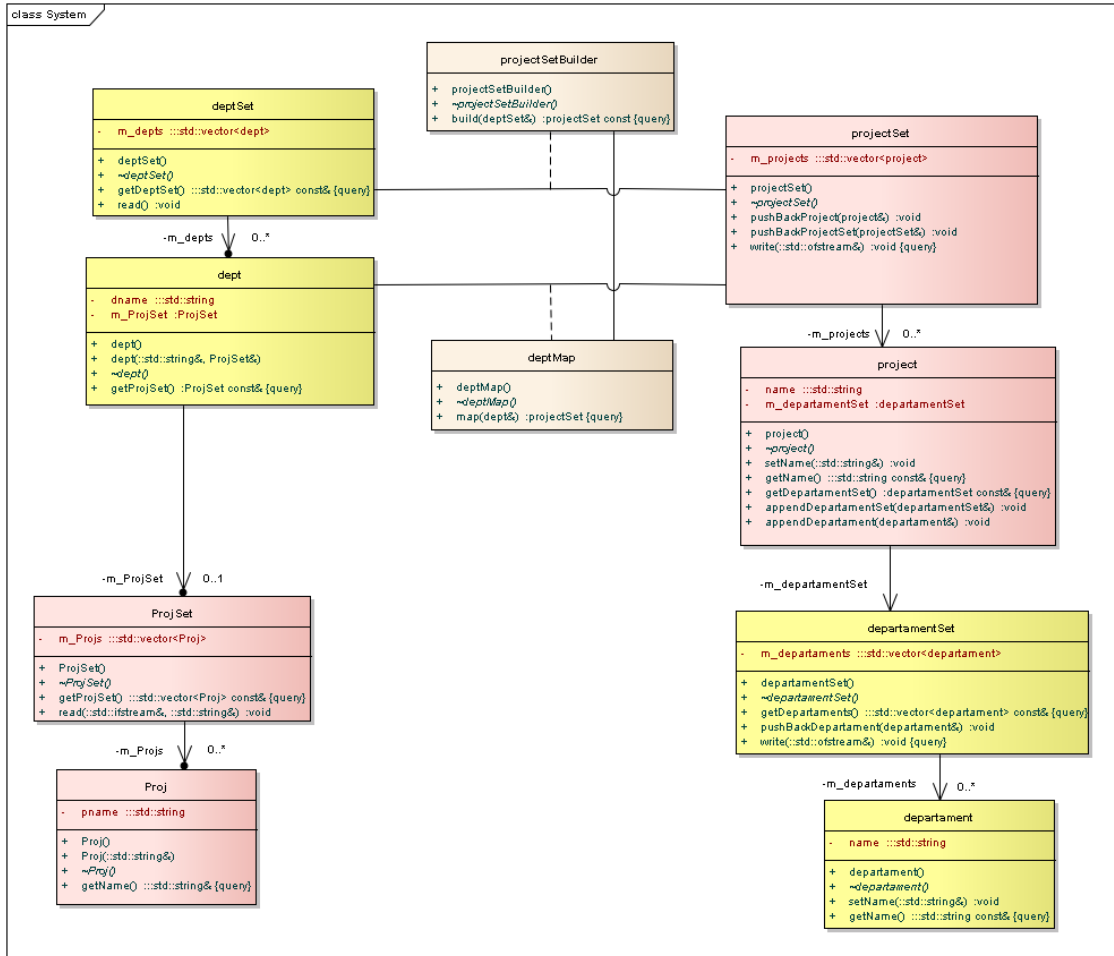

Figure 4.10: The UMAP diagram with inverting the nesting hierarchy, corresponding to Figure 4.9

$$
\begin{aligned}
&\exists group_by \\
&(\forall d \in source.dept, p \in d.Proj \rightarrow \\
&\exists p' \in target.project \quad | \\
&p' = group_by(\bot, [p.pname.value]), \\
&p'.@name = p.name.value, \\
&[\forall d_2 \in source.dept \rightarrow \\
&\exists d' \in p'.department \quad | \quad d'.@name = d_2.ename.value])
\end{aligned}
\tag{4.22}
$$

The class diagram in Figure 4.10 translates the grouping with inverted hierarchy from CLIP to UMAP. The mapping is started by the iterator from the association class *projectSetBuilder* on the elements of the set of *deptSet*.

The function *deptMap::map* in each iteration maps one object of type *dept* to an object of type *projectSet*.

This set is obtained from the *Proj* elements of the current *dept* of the iteration. The current *department* is inserted to each *project* object created by mapping from a *Proj* object. This operation actually inverts the hierarchy. Each inserted *project* is grouped by name. In this case OCL expressions do not give a constructive solution but the OCL constraints define the possible implementations. OCL allows us to express constraints, which must be satisfied by the implementation.

The attribute *m_projects* of the type *projectSet* from the target is specified by the following OCLexpresion.

$$\begin{aligned} &context \quad projectSet.m_projects \\ &inv : self->isUnique(p : project \quad | \quad p.name) \end{aligned} \tag{4.23}$$

This means that the elements of the set, the *project* objects, are unique by name. In this way the grouping by project name is achieved.

In the UML diagram the type *project* has a set of objects of type *department*. Because of this structure, the only possible grouping is to attach all the departments to the project belonging to it. If two ore more projects have the same name by the uniqueness of the project name the different departments are again grouped together. This is valid by the structure of the UML diagram also for projects in different departments.

We present next the OCL expression defining the function *deptMap::map*.

$$\begin{aligned} &context \quad deptMap :: map(dep : dept) : projectSet \\ &def : result_lhs = dept.m_ProjSet \\ &->collect(Proj \quad | \quad Tuple\{pname = Proj.pname, \\ &dname = dept.dname\}) \end{aligned} \tag{4.24}$$

This expression creates a set of tuples containing the name of the current department and each of the names of the projects of the current department that are to be inverted in this iteration.

$$
\begin{aligned}
&def : result_rhs = projectSet.m_projects \\
&- > collect(p \quad | \quad p.m_departmentSet.m_departments \\
&- > collect(d \quad | \quad Tuple\{pname = p.name, \\
&dname = d.name\}))
\end{aligned} \tag{4.25}
$$

This OCL expression computes the *Cartesian Product* of the target.

$$
\begin{aligned}
&post : result_lhs- > forAll(dp \quad | \quad result_rh \\
&- > exists(DP \quad | \quad dp.pname = DP.pname \quad and \\
&dp.dname = DP.dname))
\end{aligned} \tag{4.26}
$$

The constraint that we impose, is that the elements from the source to be inverted in this iteration are included in the target. The uniqueness of the project name and the UML structure assure that the target actually inverts the hierarchy. All the elements from the target are the result of the mapping, so no supplementary element exists in the target to those created by the mapping.

The association class *projectSetBuilder* connects the class *deptSet* from the source with class *projectSet* from the target. The function *projectSetBuilder::build* transforms the source to the target.

The OCL definition of the function *projectSetBuilder::build* uses the function *deptMap::map*.

$$
\begin{aligned}
&context \quad projectSetBuilder :: build(dSet : deptSet) : projectSet \\
&post : result = dSet.m_depts- > Set()- > collect(r : dept; \\
&pS : projectSet = Bag\{\} \quad | \\
&pS.pushBackProjectSet(deptMap.map(r)))
\end{aligned} \tag{4.27}
$$

The UML solution produces also in this case the same final result as CLIP.

4.7 A mapping with aggregates

Aggregate functions are presented in the last CLIP mapping, Figure 4.11.

The CLIP scenario from this diagram translates to the following tgd:

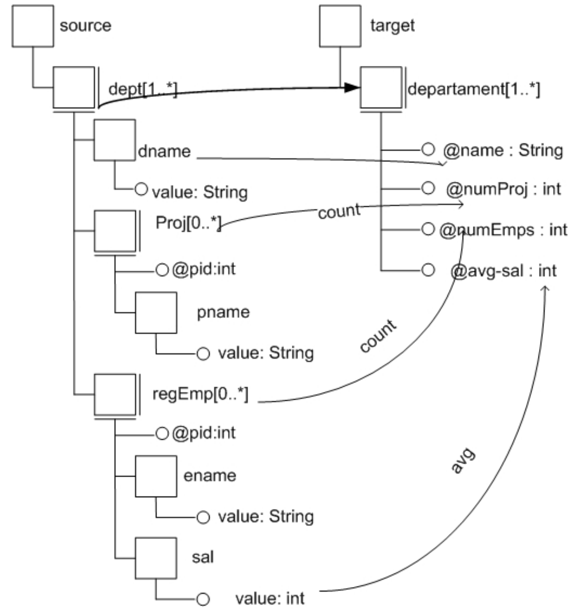

Figure 4.11: A CLIP mapping with aggregates (adapted from [37, Figure 9])

$$
\begin{aligned}
&\exists count, avg \\
&(\forall d \in source.dept \rightarrow \\
&\exists d' \in target.department \quad | \\
&d'.@name = d.dname.value, \\
&d'.numProj = count(d.Proj), \\
&d'.numEmps = count(d.regEmp), \\
&d'.avg_sal = avg(d.regEmp.sal.value))
\end{aligned}
\tag{4.28}
$$

The class diagram Figure 4.12 translates this mapping to UMAP. The *dept* objects from the source are mapped to the target as *department* objects. Target aggregate values are calculated for the source nested elements: *Projs* and *regEmps*.

The function *deptBuilder::build* starts the mapping iterating over the *deptSet*. In each iteration a *dept* object is mapped from the source to the target creating a *department* object. In this way the source *deptSet* is mapped to the target *departmentSet*. The mapping function is *depMap::map*. This function calls the functions that create the aggregates having as input sets. In a *dept* object there are two sets included: *ProjSet* and *regEmpSet*. The two functions

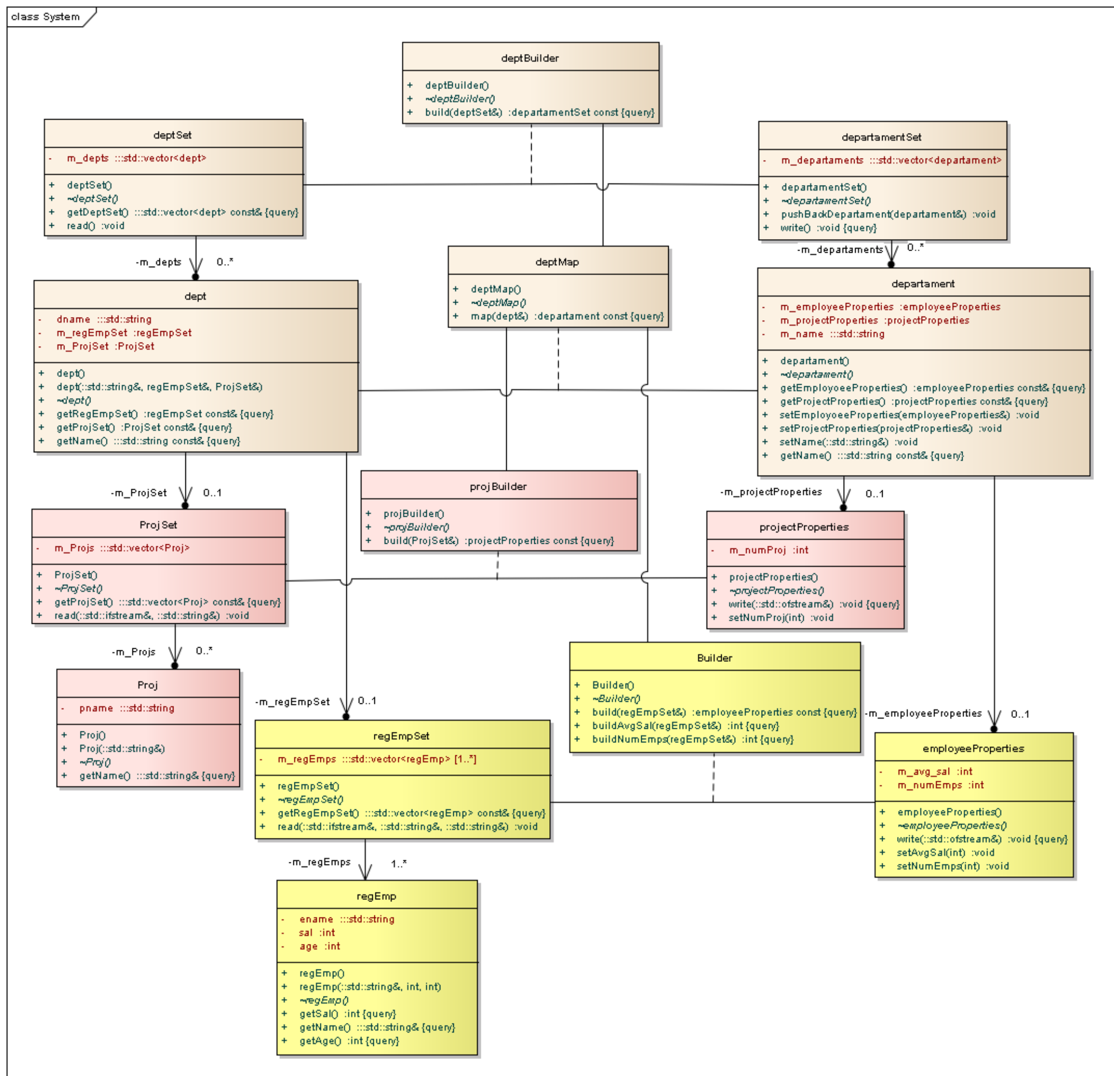

Figure 4.12: The UMAP diagram with aggregates, corresponding to Figure 4.11

projBuilder::build and *Builder::build* create the aggregate objects *projectProperties* and respectively *employeeProperties*, which are again in the object *department* included. We present next the OCL expressions, defining the mapping starting from the function creating aggregates for the *regEmpSet*.

$$\begin{aligned}&context \quad Builder :: buildNumEmps(rEmpSet : regEmpSet) : int \\ &post : result = rEmpSet.m_regEmps- > size()\end{aligned} \tag{4.29}$$

This OCL expression defines a function taking as input a *regEmpSet* and returning the size of the set.

$$\begin{aligned}&contest \quad Builder :: buildAvgSal(rEmpSet : regEmpSet) : int \\ &post : result = rEmpSet.m_regEmps \\ &- > collect(r : regEmp \quad | \quad sal)- > sum/ \\ &rEmpSet.m_regEmps- > size()\end{aligned} \tag{4.30}$$

This OCL expression defines a function using the same input and returning an average salary.

$$\begin{aligned}&context \quad Builder :: build(rEmpSet : regEmpSet) : employeeProperties \\ &post : result = employeeProperties \quad and \\ &employeeProperties.m_avg_sal = buildAvgSal(rEmpSet) \quad and \\ &employeeProperties.m_numEmps = buildNumEmpsl(rEmpSet)\end{aligned} \tag{4.31}$$

Now we use the previous two results in creating an *employeeProperties* object from a *regEmpSet*. The following OCL expression describes the creation of an object *projectProperties* containing the number of the projects belonging to a *dept* object.

$$\begin{aligned}&context \quad projBuilder :: build(pSet : ProjSet) : projectProperties \\ &post : result = projectProperties \quad and \\ &projectProperties.m_numProj = pSet- > size()\end{aligned} \tag{4.32}$$

Now we have all the elements for mapping the source to the target creating a *department* from a *dept*.

$$
\begin{aligned}
&context \quad depMap :: map(de : dept) : departament \\
&post : result = departement \quad and \\
&department.m_employeeProperties = Builder.build(de.m_regEmpSet) \quad and \\
&department.m_projectProperties = projBuilder.build(de.m_ProjSet)
\end{aligned}
\tag{4.33}
$$

The functions *Builder.build* and *projBuilder.build* create the desired aggregated values. The mapping of sets of *dept* to sets of *department* is presented in the following expression.

$$
\begin{aligned}
&context \quad deptBuilder :: build(dSet : deptSet) : departamentSet \\
&post : result = deptSet.m_depts- > Set() \\
&- > collect(r|deptMap.map(r))
\end{aligned}
\tag{4.34}
$$

As in the previous cases the UML/OCL translation describes the same mapping as the CLIP mapping.

After discussing all these complex features in detail, we show a general result that the expressive power of UMAP captures all language features of CLIP.

4.8 Discussion on the correctness of the UMAP approach

In this chapter, we have shown a translation of all CLIP schema mapping features presented in [36, 37] to UMAP. Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st})$ be a schema mapping where Σ_{st} denotes a set of CLIP s-t tgds. Let I be a source instance over $\mathbf{S}$ and let J be a universal solution over $\mathbf{T}$ satisfying $\mathcal{M}$, i.e., $\langle I, J \rangle \models \Sigma_{st}$, obtained via CLIP. As a successor of CLIO, CLIP produces universal solutions [35]. CLIO is built directly on the formal foundations of the data exchange problem [11, 13, 15, 34]. By limiting the class of possible mappings to the ones studied in these works, it is possible to affirm that the resulting transformations are sound and complete. Let Σ'_{st} be constraints materialised as OCL expressions obtained by our construction from CLIP constrains Σ_{st}. Their semantics follow the CLIP constraints by construction. Then, when UMAP is applied to the same input instance I over $\mathbf{S}$ there exists a universal solution J' over $\mathbf{T}$, which satisfies Σ'_{st}, i.e., $\langle I, J' \rangle \models \Sigma'_{st}$. We claim that both constraints produce homomorphically equivalent solutions, thus, there is a homomorphism from J' to J. In the following, we discuss all the presented translations from CLIP to UMAP and illustrate the equivalence of the two schema mappings.

- A *simple* CLIP *mapping*, presented in Section 3.1 and Figure 3.1, is by construction equivalent to the UMAP translation presented in Figure 3.2. The *value mapping* and the function *map* have the same functionality, by mapping values to values. The *builders* and the *build nodes* construction define an iterator identically to the UMAP function *build*. In this case the s-t tgds, Σ_{st}, require that all elements of the source satisfying a certain condition are to be mapped to the target. The exact mapping is also defined. Both CLIP and UMAP implement the same s-t tgds.

- CLIP mapping with one *context propagation*, presented in Section 4.1 and Figure 4.1, is by construction equivalent to the UMAP solution, presented in Figure 4.2. The basic concept of UMAP uses *associations* to connect the *association classes* situated in two different levels of the model hierarchy. This connecting element between these two hierarchy levels simulate the *context propagation*.

- CLIP mapping with multiple context propagations, presented in Section 4.3 and Figure 4.3 is by construction equivalent to the UMAP solution, presented in Figure 4.4. The previous concept can be easily extended when we change from one *context propagation* to many *context propagations* building a hierarchy.

- CLIP mapping with a join constrained by a Context Propagation Tree (CPT) is presented in Section 4.4 and Figure 4.5. Its UMAP translation is depicted in Figure 4.6. The semantics of these two mappings are defined by their s-t tgds. In this case, the CLIP join is by definition and construction identical to the UMAP defined join. The discussion in Section 4.4 gives moreover details about this case.

- CLIP mapping with a grouping and join is presented in Section 4.5 and Figure 4.7. Its UMAP translation is illustrated in Figure 4.8 and its s-t tgds define this mapping using a different technique. The CLIP mapping uses a *Skolem function* and the UMAP mapping uses, supplementary to the already presented s-t tgds, target conditions. In UMAP, the mapping function is defined by the conditions imposed on the resulting target instance. A detailed description of this case is given in Section 4.5, so that we can always find a homomorphism between the solutions obtained from CLIP and UMAP.

- CLIP mapping with inverting the nesting hierarchy, is presented in Section 4.6 and Figure 4.9, its translation to UMAP is shown in Figure 4.10. This translation uses the same

technique as the previous case, CLIP uses a *Skolem function* and the UMAP mapping functionality is deduced by the s-t tgds and target conditions.

- CLIP mapping with aggregates is presented in Section 4.7 and Figure 4.11, the UMAP translation in Figure 4.12 uses again different technics obtaining the same target instance. CLIP uses *Skolem functions* and in UMAP creates the mapping using the definitions of the aggregate functions.

The formal prove of our claim should be given by future work.

4.9 Usage of Skolem functions

In [34], the authors consider the schema *mapping problem* defined as translating an instance of the source schema to an instance of the target schema. The *primary path* is defined as the set of elements, found on the path from the root to an intermediate node or leaf in the tree structure of the source and target. The mapping is materialized with the use of *correspondences*: elements of source and target connected with arrows. The *correspondences* are modeled as *interpretations*, source–to–target referential constraints or dependencies [34]. In this context the *nested referential integrity constraint* is presented and defined using the *primary path*, a large class of referential constraints that include relational foreign key and XML schema's key references. Nested dependencies support the translation of the nested structure of the source and the target. These source–to–target dependencies are compiled into low level, language independent *rules*. These *rules* are used to obtain the transformations in concrete languages like XSLT or XQuery for XML Schemas or SQL for relational schemas. For the creation of the new values in the target that are not specified, one-to-one Skolem functions are used. In [15], the authors, based on [34], introduce *nested mappings*. The mappings are defined by lines, *correspondences* connecting the source elements with target elements. Constraints, describing the mapping can be generated from these lines by tools like Clio or directly by human experts. The *nested mapping* is introduced allowing common subexpressions to be factored out. Grouping is another feature introduced by the use of *nested mappings*. The *nested mappings* are strictly more expressive than the mappings from [34] but less expressive than languages used for composition of s-t tgds as presented in [13, 26]. The authors note that for the relational model the *nested mappings* are a sub-language of the second-order tgds (SO tgds): every *nested mapping* can be rewritten, via Skolemization into an equivalent SO tgd but not vice-versa [15]. In [37] the authors, based on [15, 34], introduce more complex mappings using second-order logical formulas expressing

grouping and aggregates by means of functions. Recall the nested tgds of Figure 7, the grouping, from CLIP [37, IV. Language Semantics] expressing the fact that for each join on *pid* from the source there must be an *employee* nested inside of a *project* of the target. The correlation between the outer mapping and the inner mapping is achieved through the variable p', a *project* on the target, defined in the outer mapping and used in the inner mapping (more, the name of the *project* must be unique). This correlating element p' is the grouping element of the mapping. This could be expressed in first-order logic but the structure of the nested tgds is not preserved. This was the reason why a special Skolem function was introduced to solve this problem.

Our UMAP approach using UML class diagrams and OCL constraints does not explicitly use *nested mappings* [15], a sub-language of SO tgds, as CLIP [37] does. Instead the diagram structure and the constraints define the possible query implementations that execute the mapping. A translation of UML/OCL to first-order predicate logic is given in [4]. The UMAP mapping, translated to first-order logic, could be compared with the nested mappings used by CLIP. CLIP introduces a special kind of Skolem function to express the grouping and the aggregates. A translation of the UML/OCL to nested tgds via a translation to first-order logic would need the same special Skolem function, showing the equivalence of the both mappings.

4.10 Target dependencies, target *egds* and *tgds*

UMAP allows the use of target *egds* and target *tgds*. The definition of an explicit mapping by nested *tgds* from [37, IV. Language Semantics] uses two expressions: C_1 and C_2. The first is a source expression. The second has three kinds of target conditions and could be used as *s-t tgds* but, by their definition, not as target *egds* or target *tgds*. In our translation, constraints could be defined for each element in the target schema. An element of the target schema defined using an OCL expression to be unique is an example of using a target *egd*. The usage of target *tgds* is possible through an OCL expression on the target. Special measures must be taken to limit the infinite cascading of tuples created in the target by restricting the target *tgds* to a *weekly acyclic set of tgds* [11].

CHAPTER 5

VMAP the implementation of UMAP

5.1 System architecture

To show the usability of our visual schema mapping language UMAP, we present VMAP, its reference implementation. VMAP consists of a chain of specializes components that transforms a schema mapping scenario depicted using the UMAP visual language into an executable capable of creating a target instance from a source instance. UMAP has enough expressive power to capture the main features of common high-level visual schema mapping languages (see, e.g., Chapter 4 for a translation of all major CLIP constructs to UMAP), which is an essential prerequisite for such a middleware. Nonetheless, an interface needs to be implemented to provide the bridge between the high-level language and the UMAP middleware.

Fortunately, this interface can be automatically generated for any high-level language, which has a so-called *formal meta-model* (in UML terminology). Then Query, View, and Transformation, QVT [27], an evolving standard, provides a methodology for automatic transformation. QVT is an OMG standard, that describes model-to-model transformations. An application that implements QVT takes as input a schema mapping model, in our case represented using the schema mapping language Clip and produces as output a model represented using the schema mapping language UMAP. The scripts using the QVT transformation languages perform actually the model-to-model transformation. QVT has for this purpose three categories of languages that could be employed to achieve the desired result. QVT works on modelling languages that are formal, i.e., they have a MOF [31] conform description in UML, the *formal meta-model*. In this thesis we do not employ QVT for the transformations. The transformations have been performed

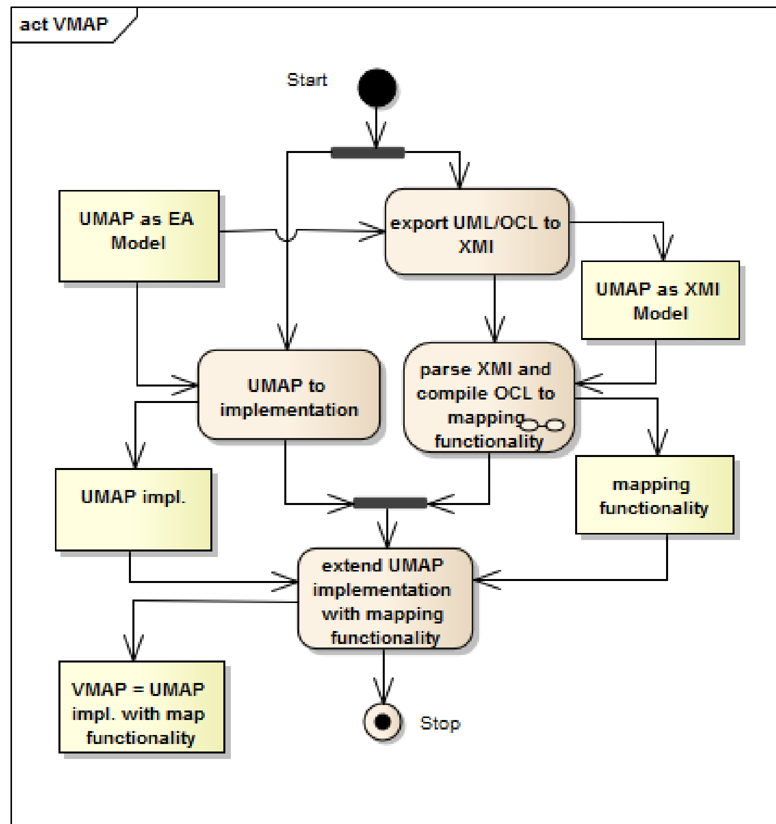

Figure 5.1: VMAP process flow.

manually with respect to the presented transformation rules. The future work mentions the need to create the Clip meta-model for a QVT transformation.

For our discussion we assume that this transformation has already been executed, either by hand or via QVT. So we have the UMAP model, as a UML class diagram containing the source and target structures with the mapping functions defined as OCL expressions. We achieve this by designing the classes with the help of a UML modeling tool and augment them with functions for reading the source from and writing the target to XML files. Further, functions for the mapping are added and defined with the help of OCL expressions. The *activity diagram* in Figure 5.1, a UML artefact, presents the VMAP work flow consisting of four tasks:

- UMAP to implementation,
- export UML/OCL to XMI,
- parse XMI and compile OCL to mapping functionality and

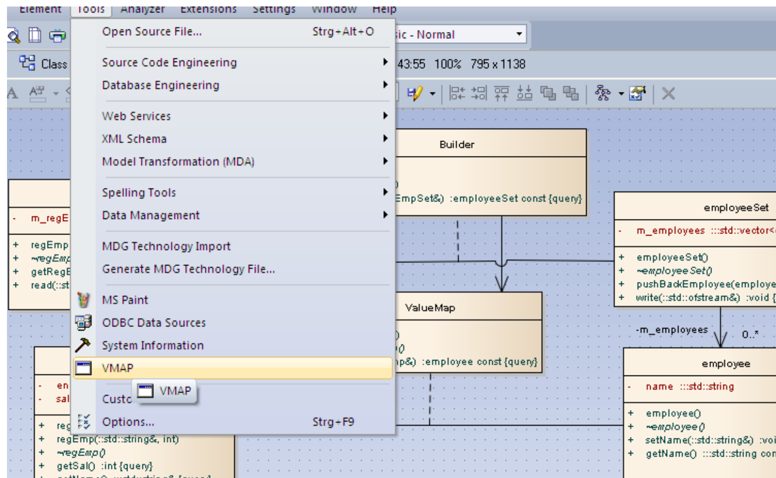

Figure 5.2: VMAP process flow, its usage in Enterprise Architect (EA).

- extend UMAP implementation with mapping functionality.

The first two tasks use the modeling tool to export the UMAP diagram and create the implementation skeleton. The last two tasks represent the VMAP, they transform an empty function skeleton to implementation by adding the mapping functionality. VMAP is called from the modeling tool as an external function, presented in Figure 5.2. The input of the first task is the UMAP schema mapping model. The output of the last task is the result of VMAP, the implementation of the mapping functions, ready to be transformed into an executable. In Figure 5.1 we also present the intermediary outputs of the tasks, subsequently used as inputs by other tasks. Further we present these tasks in detail.

Using the UML modeling tool we generate the implementation of the UMAP classes and the skeletons of the mapping functions. The implementation language used is the programming language C++, which needs for a class two files: a header file, defining the class and an implementation file for the member functions of that class. The complete UMAP model is exported using the modeling tool to XMI, which stands for XML Metadata Interchange, an OMG standard for representing object-oriented information using XML. This XMI file representation of the UMAP model is standardized and does not depend on the concrete UML modeling tool, used to create it. We parse this XMI file and extract the OCL expressions together with the referenced variables. Towards evaluating these OCL expressions, we developed an OCL compiler, specific to our context of schema mappings and data exchange. The compiler transforms the OCL expressions to functions of a programming language, in our particular case C++, inspired by

compilation techniques of OCL to a programming language as presented in [38]. For the lexical and syntactical analysis we use the tools Flex [14] and Bison [6]. From the UMAP model we extract the variables used in the OCL expressions. Together with the key words of OCL and the grammar we parse the stream of tokens from our expressions into an abstract syntax tree. The usage of OCL constructs in UMAP is limited to a few and the resulting expressions are simple. The translation is also limited to a few idioms of the target language, and as a consequence, only a subset of its constructs are used. The functionality created by our OCL compiler augments the initially empty generated skeletons of the mapping functions. The UML modeling tool generates only the skeletons of the mapping functions and their functionality is implemented through the compilation of the corresponding OCL expressions.

Using an integrated development environment the main function, a C++ specific artifact, is created. The main function is extended to use the topmost classes generated from the model: the source, the target and the mapping class. The generated files are then transformed into an executable. The execution of the created program then loads the input file, an XML file defining the source instance, and transforms it to another XML file, the target instance, thus executing the mapping.

The main tools used for VMAP are:

- as operating system (OS) Microsoft Windows (WIN32),
- as UML modeling tool, the well known and appreciated product Sparx Systems' Enterprise Architect (EA),
- as integrated development environment (IDE) for C++ Microsoft Visual C++ 2010 Express (MSVC),
- as XML parser the well known tool Xerces [39], and
- as lexical and syntactic analyzers the tools Flex [14] and Bison [6], well known for their performance in generating syntax trees in compiler construction.

No proprietary WIN32, EA or MSVC specific features are used in creating VMAP. This makes the implementation OS, modeling tool, and IDE independent. The implementation only uses standardized constructs: standard XML and XMI, standard UML and OCL, and the standardized target programming language C++11. At the same time the C++ features employed are limited to those that are also found in other common languages like JAVA, so that the implementation language of VMAP could be easily exchanged if desired.

5.2 Data exchange scenario

We present the work of a data engineer executing data exchange, using our tool chain with VMAP. The following steps are construction steps common for all scenarios presented here. The source, target and mappings are defined with the EA tool. The source and target instances are stored as XML file. The standard data exchange scenarios that we present are similar to those shown in [37]. The OCL compiler used in this context, one of our main contributions, is integrated in EA and called as an external function, presented in Figure 5.2. The compilation error messages and warnings are presented in an log file.

We start with the UMAP model, constructed with the EA tool, and saved in a file of type EAP (Enterprise Architect Project). Using MSVC, the chosen IDE, we create a project, an MSVC specific artifact. The project is saved in two files of type SLN (Microsoft Visual Studio Solution) and VCXPROJ (Microsoft Visual Studio Project), the MSVC build environment. We add the EA generated files to the project. VMAP uses the exported UMAP model to XMI to parse it and to extract the OCL expressions. The generated files are OS (Windows, Linux) and IDE (MSVC, Eclipse) independent. The MSVC IDE can immediately create an executable.

Let us consider the following OCL expression for a source to target mapping. The OCL `select` function is used to iterate over the source set of `regEmp`. Only those employees are selected with a salary greater than a certain value. The OCL `collect` function is used to transform the selected source set into the target set of type `employee`:

```
context
  Builder::build(rEmpSet: regEmpSet): employeeSet
post: result = rEmpSet.m_regEmps
  ->select(r | r.getSal() > 11000)
  ->collect(r: regEmp | ValueMap.map(r): employee)
```

The modeling tool EA creates from the UMAP model the empty function. The content of the function, the mapping functionality, is the result of our OCL to C++ compiler that creates the following function:

```
employeeSet const
Builder::build(const regEmpSet& rEmpSet) const
{
  employeeSet eS;
```

```
  std::vector<regEmp> const& rmS = rEmpSet.getRegEmpSet();
  std::vector<regEmp>::const_iterator i = rmS.begin();
  std::vector<regEmp>::const_iterator e = rmS.end();
  for (; i != e; ++i)
  {
    regEmp const& currRegEmp = *i;
    if(currRegEmp.getSal() > 11000)
    {
      ValueMap valueMap;
      const employee& e = valueMap.map(currRegEmp);
      eS.pushBackEmployee(e);
    }
  }
  return eS;
}
```

The semantics of the C++ function is similar to that of the OCL expression. The selection of the source elements is followed by the mapping and the resulting target elements are inserted into a target set, in this case, in a language specific generic vector.

Further we consider all the cases of mappings presented in [37].

- A mapping with context propagation. The parsing of the OCL expressions determines the variables, the loops, the nested loops and transformations using these variables. These patterns are identified and the corresponding code is created. The OCL expressions are compiled to C++ as in the motivating example. The case of multiple context propagation is similar and our concept can easily be extended to fulfill the supplementary requirements.

- A mapping with join. The OCL expressions are compiled using the same techniques as already presented. The join is in OCL specified as a nested loop. This pattern is recognized and used in the translation to the target language.

- A mapping with grouping and join and a mapping with inverting the nesting hierarchy. This case is much more complex and the OCL expression formulates only the constraints without suggesting an implementation roadmap as in previous cases. This is a consequence of using the OCL function 'exists'. Thus, the created function must fulfil the

constraints expressed by the OCL expression. The automatic generation of such functions requires careful design which is left for future work.

- A mapping with aggregates. Here the OCL expressions can be directly compiled to C++. The simple patterns recognized in the OCL expression are used by the translation to the target language.

5.3 Conclusion

We have presented VMAP, the reference implementation for UMAP, a recent underlying layer and middleware for high-level visual schema mapping languages. UMAP can be used as a visual mapping language itself or as middleware, having at least the expressive power of CLIP [8]. UMAP is based on the well known and standardized OMG specifications: UML and OCL. UMAP can use as interactive graphical interface any UML/OCL modeling tool, thus providing the graphical interface for VMAP. UMAP builds only on standard specifications, fact used intensively by VMAP. For this reason, VMAP works only with schema mapping models represented as UMAP models. VMAP is not designed to work directly with logical formulas, even if they are translated to OCL. OCL, a full fledged constraint language, is not very useful without UML, because it loses context information supplied by UML; that is why OCL must be embedded in UML diagrams. VMAP extracts its input from a file in XMI [32] format. This file includes the whole schema mapping model represented as a UML diagram and its OCL expressions exported by the modeling tool. VMAP identifies the functions that are relevant for the schema mapping and their OCL expressions, the OCL supplementary semantics of the UMAP model. These expressions are compiled via the OCL-to-C++ compiler to functions. Finally, VMAP puts in place these functions containing code in a standard programming language, ready to be used by the build environment, responsible for the production of the executable as final result. The purpose of VMAP is to offer data engineers an implementation platform for their own visual mapping languages.

CHAPTER 6

OCL Compiler

6.1 Introduction

We have presented the architecture of VMAP in Chapter 5. Its most important part, our OCL compiler works at its heart. The compiler translates the actual mapping definitions to functions of a programming language, which after compilation execute the data exchange. In this chapter, we present the OCL [30] compiler in much more detail. The OCL expressions are extracted from the UMAP model and transformed into source code of a specific programming language. In our case the programming language is C++. The created source code is compiled using a build environment. Further, we discuss the phases preceding the creation of the source code and succeeding it. The input is the UMAP data exchange scenario and the output is the source code, representing the data exchange functionality.

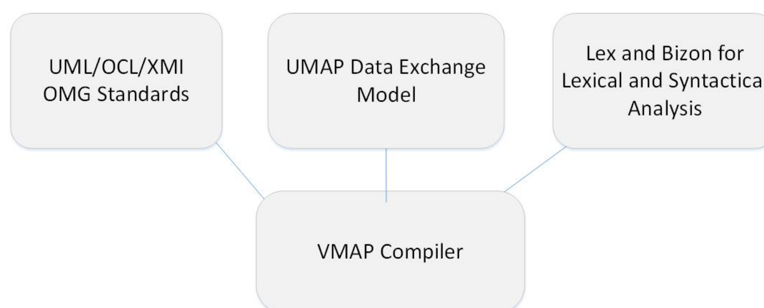

Figure 6.1: VMAP compile architecture concept

In Figure 6.1 the fundamental building blocks, on which the VMAP compilation process is based, are presented. The most difficult part in designing this architecture was to choose only standardized elements and to bind them with clear structured interfaces. The VMAP compiler consists of a pipeline of executables, each of them solving well–defined elementary problems. In this way, the test of these executables can be done in isolation with minimal knowledge of the tasks, solved by the surrounding executables. This architecture creates the condition for a simple construction capable of promptly reacting to extensions, changes and defects.

6.2 Compiler Architecture

In Figure 5.1, VMAP, the implementation of the UMAP, is presented. The complex activity, named: *parse* XMI *and compile* OCL *to mapping functionality*, depicts the compilation process and it is detailed in Figure 6.2. This process has as input the data exchange model in XMI format and as output the source code representing the functionality. In the rest of this chapter we discuss in detail the three phases of the compilation process:

- the XMI model parsing, followed by the extraction of the OCL expressions and the source and target types,
- each OCL expression is compiled to an abstract syntax tree (AST) and then transformed to flat OCL AST by traversing the tree in pre-order and
- the code generation from the flat OCL AST together with all other type information.

The first executable is the XMI model parser

Recall the data exchange scenario, depicted as UMAP model in Section 3.1. This model is exported from the Enterprise Architect (EA) tool to the standardized XMI format, which ensures the tool independency. The UMAP model in XMI format is an XML file. To parse it, we use the well–known XML parser: Xerces [39]. The XMI format, an OMG standard, is very suitable for the description of a UML data model. Every construct of the UML graphical language is described in XML. UMAP defines the data exchange scenario using the UML construct *association class*. In this context, the *source* and *target* are depicted as UML classes and their attributes. All these constructs and their details are included in the XMI format model description.

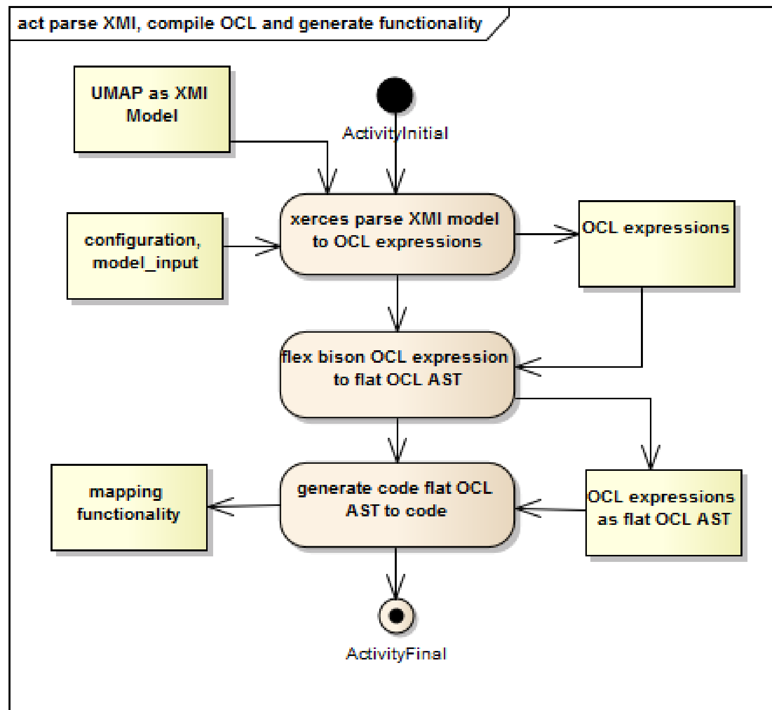

Figure 6.2: XMI to functionality.

OCL expressions

The most important role in UMAP is played by the classes called *mappers* and *builders*. They have already been presented in Section 3.1. These classes are extracted from the XMI model by parsing. They are uniquely identified by the following properties:

- only one instance function, which includes in its name the verbs: *map* or *build*,
- only this function is in UMAP described by an OCL expression,
- these classes have no state, i.e., no instance variables and
- these classes are modeled as *association classes*.

The classes named *mappers* and *builders* are of such importance because they contain the whole data exchange description. The OCL expressions define how the source is transformed into the target. The signature, i.e., the input and the output types of these functions are the data exchange source and target types, respectively. Next to the OCL expressions, these types are also extracted, together with their attributes. Algorithm 6.1 presents the work flow of this

```
Data: UMAP scenario as XMI model
Result: a summary file with all the class and function names and a file for each OCL
        expression
 1 Parse the UMAP model in XMI format;
 2 select all nodes that represent a class and its attributes in the container M;
 3 forall the elements of M do
 4 |   if current element has an OCL expression then
 5 |   |   create a file for this OCL expression;
 6 |   |   the name of the file is the class and function name;
 7 |   |   add the class and function names to the summary file;
 8 |   |   add the input and output types of this function to the summary file;
 9 |   |   select the function input and output types in the container stType;
10 |   |   forall the elements of M do
11 |   |   |   if type name is in stTaype then
12 |   |   |   |   select all the subtypes of this type;
13 |   |   |   |   add all these subtypes to the summary file;
14 |   |   |   end
15 |   |   end
16 |   end
17 end
```

Algorithm 6.1: The XMI model parsing and OCL expressions extraction.

phase. The outer loop looks for *association classes* and their OCL expression. The inner loop collects the source and target types as they are needed in the process of the OCL compilation to the programming language. The result of this phase is a summary of the functions to be implemented, together with a file for each OCL expression. At this point the road map for the rest of the executables is in place and they add their contribution to the final result.

The second executable is the OCL parser

Towards evaluating the OCL expressions, we develop an OCL compiler specific to our context of schema mappings and data exchange. The usage of OCL constructs in UMAP is limited to a few and the resulting expressions are simple. The translation is also limited to a few idioms of the target language, and as a consequence, only a subset of its constructs are used. For the lexical and syntactical analysis we use the widely accepted and well tested tools: Flex [14] and Bison [6], which transform a grammar into a parser. We have developed an OCL grammar restricted to our specific expressions. Only the used language constructs are defined, thus our grammar is a lot simplified. The OCL standard specification has a very general character under which our

interpretation is valid. The variable types, detected by compilation of the OCL expressions, are already known. They are extracted from the UMAP model in the previous phase. Together with the key words of OCL and the grammar we parse the stream of tokens from our expressions into an abstract syntax tree. The Bison generates an LALR(1) parser, well suited for our purpose. The parser creates for each OCL expression the abstract syntax tree, an example is presented in Figure 6.3. This activity checks if the OCL expression is syntactically correct and belongs to the UMAP limited expected expressions. Each OCL expression belongs to a function and this correspondence is known from the previous phase. The abstract syntax tree is traversed using the method depth first in pre-order, and the nodes are listed together with their supplementary information. We call this list the *flat* abstract syntax tree, i.e., the *flat* AST, an example is presented in Listing 6.2. The nodes, representing operations and variables, have assigned a unique identifier. The output of this second executable are the *flat* AST.

Simple OCL expression compilation

We consider the following OCL expression, already presented in (3.2).

$$post : result = (e : employee) \quad and \quad (e.name = rEmp.ename) \tag{6.1}$$

This expression defines the data exchange process at the lowest possible level. It assumes first that a target instance is created. This is defined by the following OCL construct, named descriptor.

$$e : employee \tag{6.2}$$

The type of this target instance is *employee* and its name is *e*. This type, known from the previous phase, is the return type of the initial OCL expression. It is taken from the *summary file*, and is not part of the compilation. Thus the compilation is made much simpler at this stage. The data exchange is contained in the following part of the OCL expression.

$$\textit{e.name} = \textit{rEmp.ename} \tag{6.3}$$

This expression defines the assignment of the attribute $name$, that belongs to the target instance e, mentioned above, with the attribute $ename$ of the source instance $rEmp$. This instance is the only argument of the function for which we provide the functionality, the source instance, $rEmp$ is taken from the definition of the original OCL expression (3.2). In this particular case the data exchange is limited to only a string value: the source variable *ename*. The source instance defined by the source type: *regEmp* and its name: *rEmp* are known from the previous phase and again reduce the compiling effort.

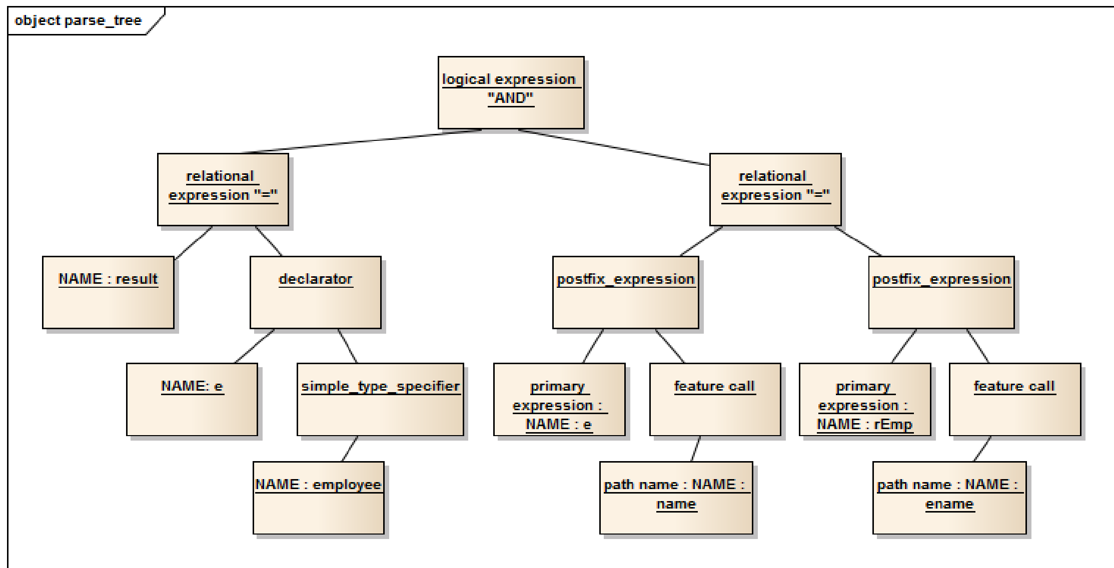

Figure 6.3: The AST of the OCL expression (6.1)

```
1  E AND ;
2  G = ;
3  D ;
4  0 e ;
5  N employee ;
6  G = ;
7  Y ;
8  N e ;
9  X ;
10 0 name ;
11 Y ;
12 N rEmp ;
13 X ;
14 0 ename ;
```

Listing 6.2: The *flat* AST of the AST presented in Figure 6.3

Simple OCL iterator expression compilation

The original CLIP expression is more complex then the already analysed OCL counterpart. The main difference between the two expressions consists in the capability of explicit iterating over sets in the OCL case. The OCL equivalent has been presented in (3.3), it is extracted by our XML

parser and here presented.

$$post : result = rEmpSet.regEmps- > select(r : regEmp \quad | \quad r.sal() > 11,000)$$
$$- > collect(r : regEmp \quad | \quad ValueMap.map(r)) \tag{6.4}$$

This expression uses two OCL language constructs as iterators: *select* and *collect*. The first selects only certain source tuples that satisfy a given condition and the second transforms the tuple from the source to the target structure. The type: *ValueMap*, an *association class*, is instantiated. Its function *map* is employed to achieve the data exchange, which was presented in (6.1).

Compiler description

The phases of our compiler are: the lexical analyzer, the syntax analyzer, the semantic analyzer and the intermediate code generator. The final product of the compilation is source code in a programming language. Actually our compiler is a compile front end. The back end of the compiler, the part of the compiler that depends on the target machine, is delegated to the programming language which is the product of our compiler.

6.3 Semantic Analysis And Code Generation

The third executable is the code generator

In Figure 6.3 we present the abstract syntax tree (AST) of the OCL expression as shown in (6.1). This OCL expression corresponds partially to the CLIP expression (3.1). This third executable takes *the flat abstract syntax tree*, *the flat* AST, with the supplementary information and by comparison with an existing collection of patterns, *flat* AST patterns, determines which functionality corresponds to the current *flat* AST. The unique codes of the AST nodes are actually compared, i.e., the *flat* AST structure is used to identify the source code suitable for the current OCL expression. The supplementary information, e.g., types, relational operators and other constructs, are taken from the AST or from the *summary file*. These both sources are used to optimize the code generation. Thus the source code patterns collection is used and augmented with types discovered by both XML and OCL parsing.

The functionality is the result of this third executable. The semantic analysis of our compiler works in two steps of granularity. Our algorithm checks if the *flat* AST resulting from the phases: lexical and syntactical analysis matches one of the predefined patterns from a collection.

If this is the case, then the pattern is used to generate code. The *flat* AST contains enough information about the input and output types or about the selection criteria used to perform the data exchange. The code pattern is augmented with the missing elements to obtain the functionality in a programming language. If the pattern for the whole *flat* AST is not recognised, then our algorithm checks portions of the *flat* AST that could be recognised. The source code is created for the identified portions of the *flat* AST and the portions of the code are concatenated.

Recall Figure 6.3 and the related *flat* AST, shown in Listing 6.2. Both represent the parsing of the OCL expression (6.1). The root of the abstract syntax tree is identified as the logical operator: *and*, which connects the two parts of the OCL expression. Left and right to the root are depicted the declarator and the data exchange part, respectively. The data exchange part uses as root the assignment operator the *relational operator* "=", which is positioned between: the target and the source, to the left and to the right, respectively. They are both identified as *postfix expressions*. This means that they are of the following form: *e.name* and *rEmp.ename*. At this moment the whole semantics behind the syntax is discovered and the code generation begins.

This semantics analysis is flexible enough to cover extended expressions, as the following example shows. If the data exchange scenario includes also the transfer of the *salary* to the target, supplementary to the *name*, then the OCL extended expression is the following.

$$\begin{aligned} post : result = (e : employee) \quad and \\ (e.name = rEmp.ename) \quad and \\ (e.salary = rEmp.sal) \end{aligned} \tag{6.5}$$

This change of the OCL expression is reflected by the AST and also by the *flat* AST changes. This modification of the *flat* AST can be easily analysed in Listing 6.3

The code generation executable is designed to extension

In the particular case of the two OCL expressions already presented the modification of the first expression with more than one attribute copied from the source to the target has been presented. A more complex expression, at this stage, could use the source attributes to calculate the target attributes, besides only copying them from source to target. For the iterator part of the OCL expression the condition is easily extendable for a sequence of conditions connected by logical operators.

```
1  E AND ;
2  E AND ;
3  G = ;
4  D ;
5  0 e ;
6  N employee ;
7  G = ;
8  Y ;
9  N e ;
10 X ;
11 0 name ;
12 Y ;
13 N rEmp ;
14 X ;
15 0 ename ;
16 G = ;
17 Y ;
18 N e ;
19 X ;
20 0 salary ;
21 Y ;
22 N rEmp ;
23 X ;
24 0 sal ;
```

Listing 6.3: The *flat* AST of the AST presented in (6.5)

The fourth executable matches the skeleton functions with the functionality

This activity is not part of the compilation process but is so close to it that is important to mention a few facts about it here. This activity puts the compiled functionality in place. The last executable uses this functionality and the list of classes and functions from the first step to match the function skeletons with the constructed functionality. Recall from the *activity diagram* in Figure 5.1 the last activity, which corresponds to this executable. The UML modeling tool generates only the skeletons of the mapping functions and their functionality is implemented through the compilation of the corresponding OCL expressions.

6.4 Lexical and Syntactical Analysis

The used OCL Grammar

We present the concrete syntax of OCL using an extended Backus-Naur format. The grammar is limited to the used production rules. The grammar described here is a context-free grammar that can be compiled with an LALR(1) parser. The start symbols are <expression>. The reserved keywords of OCL are presented in the table.

and	if	or
body	implies	package
context	in	post
def	init	pre
derive	inv	self
else	invalid	static
endif	let	then
endpackage	not	true
false	null	xor

Table 6.1: OCL reserved keywords

Syntax

```
<expression> ::= <expression> (<relational_op> <expression>)*

<expression>::= <expression> (<logical_op> <expression>)*

<expression> ::= <expression> (<add_op> <expression>)*

<expression> ::= <expression> (<multiply_op> <expression>)*

<expression> ::= "(" <expression> ")"

<expression> ::= <declarator>

<expression> ::= <declarator_list>
```

```
<expression> ::= <feature_call_parameters>

<expression> ::= <feature_iterator>

<expression> ::= <feature_call>

<expression> ::= <postfix_expression>

<expression> ::= <NAME> "=" <expression>

<expression> ::= <NAME> "(" <exp_list> ")"

<expression> ::= <NAME> "{" <exp_list> "}"

<expression> ::= <NUMBER>

<expression> ::= <NAME>

<postfix_expression> ::= <expression>
    (<navigation_op> <feature_call>)*

<feature_call> ::= <path_name> <feature_call_parameters> |
    <path_name> "(" <expression> ")" |
    <path_name> "()" |
    <path_name>  |
    <path_name> <feature_iterator>

<path_name> ::= <NAME>

<feature_iterator> ::= "(" <declarator_list> "|"
    <actual_parameter_list> ")"

<feature_call_parameters> ::= "(" <declarator> "|"
```

```
    <actual_parameter_list> ")"

<declarator_list> ::= <declarator> |
    <declarator> ";" <declarator_list>

<declarator> ::= <NAME> ":" <simple_type_specifier>


<actual_parameter_list> ::= <expression_list>

<simple_type_specifier> ::= <expresion>

<expression_list> ::= <expresion> |
    <expresion> "," <expression_list>
```

Lexicon

```
<logical_op> ::= "and" | "or" | "xor" | "implies"
<relational_op> ::= "=" | ">" | "<" | ">=" | "<=" | "<>"
<add_op> ::= "+" | "-"
<multiply_op> ::= "*" | "/"
<navigation_op> ::= "." | "->"

<NUMBER> ::= <digit> (<digit>)+
<NAME> ::= <letter> (<char>)+
<char> ::= <letter> | <digit> | "_"
<digit> ::= "0".."9"
<letter> ::= <upper> | <lower>
<upper> ::= "A".."Z"
<lower> ::= "a".."z"
```

6.5 Conclusion

We have presented the OCL compiler, the heart of VMAP implementation. The OCL constraints language extends the expressivity of visual modeling language UML. The code generation from models like UML is long known and used prerequisite of every UML modeling tool. The OCL expressions define the semantics of our schema mapping language UMAP and the implementation of OCL compiler has been necessary to show the usability of our concept.

The implementation, VMAP, consists of a chain of components that transform a UMAP schema mapping model in XMI [32] format into an executable, which is responsible for the data exchange. The implementation of the code generation (the compiler back-end) is in the phase of proof of concept. All seven Clip scenarios presented in [37] are supported. As implementation language, C++ was used, because it is best known to the author and the build environment has no dependencies on other tools. In this phase of the UMAP/VMAP, no particular performance measures have been done. The most important achievement is the implementation of an end-to-end solution starting from a graphical UML/OCL model and producing source code ready to be used by a build environment.

At the moment of our research no standard OCL compiler was available with support for our restricted field of schema mapping. The code generation from OCL is a supplementary benefit that extends the value of a UML/OCL modeling tool by using directly the requirements to generate prototypes. The future work should be directed towards extending the area of OCL expressions that could be compiled and towards the growth of the compilation performance.

CHAPTER 7

Related Work

7.1 Overview

Today, there are many commercially available products on the market of schema mapping tools. IBM InfoSphere Data Architect, MS BizTalk, Altova Mapforce (www.altova.com/mapforce) and Stylus Studio (www.stylusstudio.com) are examples of commercial mapping systems. IBM InfoSphere Data Architect is well known for its many features. Some of them are related to our work: data model transformation and mapping relations between data models. Altova Mapforce is another prominent commercial product appreciated for the source-to-target visual data mapping and code generation for data transformation.

Important research is presented in [18] with respect to the translation data-metadata. In this case, the the target schema is not a priori known. Some data from the source is used to augment the metadata of the target.The opposite situation is presented, the metadata-data transformation maps some metadata from the source in data of the target. This work introduced the novel concept of nested dynamic output schemas, which are nested schemas that may only be partially defined at compile time. Data exchange with nested dynamic output schemas involves the materialization of a target instance and, additionally, the materialization of a target schema that conforms to the structure dictated by the nested dynamic output schema.

Traditional approaches for designing schema mappings are either manual or performed through a user interface from which a schema mapping is interpreted from correspondences between attributes of the source and target schemas. These correspondences are either specified by the user or automatically derived by applying schema matching on the two schemas. In [2], an al-

ternative approach is presented that allows a user to follow the "divide-design-merge" paradigm for specifying a schema mapping. The user designs portions of the schema mappings that are finally merged by the system to obtain the desired result.

In the frameworks of the CLIO project [34], the authors introduce the schema *mapping problem*, defined as translating an instance of the source schema to an instance of the target schema. They present the original schema mapping algorithm, applied to a nested relational model to handle relational and XML data. The mapping is materialized with the use of *value correspondences*: elements of source and target connected with arrows. The *primary path* is defined as the set of elements found on the path from the root to an intermediate node or leaf in the tree structure of the source and target. A *logical relation* results by chasing a *primary path*. Based on the *correspondence* and *logical relation* an *interpretation* is computed. Thus *correspondences* are modeled as *interpretations*, source–to–target referential constraints. In this context the *nested referential integrity constraint* is presented and defined using the *primary path*, a large class of referential constraints that include relational foreign key and XML schema's key reference. Nested dependencies support the translation of the nested structure of the source and the target. These source–to–target dependencies are compiled into low level, language independent *rules*. These *rules* are used to obtain the transformations in concrete languages like XSLT or XQuery for XML Schemas or SQL for relational schemas. For the creation of the new values in the target that are not specified, one–to–one Skolem functions are used.

In [7, 22, 23] SPICY and +SPICY, schema mapping systems are introduced, a contribution to bridge the gap between the practice of mapping generation and the theory of data exchange. SPICY/+SPICY/++SPICY is an evolution of the same system from a schema matching tool to a schema mapping generation and rewriting tool. The importance of CLIP for our visual schema mapping language UMAP was presented in Chapter 3. The importance of the SPICY system is even greater for our UMAP and its implementation, VMAP. If UMAP/VMAP is used as middleware by the SPICY system we gain its capability to bring together expressive mapping generation algorithms with an efficient strategy to the computation of core solutions. The ++SPICY system brings successful results, where other tools exhibit some serious limitations that prevent their adoption in real-life scenarios. Some of these drawbacks are the limited support for target constraints and limited support for relational scenarios. By using UMAP/VMAP as middle-ware layer below the SPICY system, we can combine the powerful functionality of ++SPICY with the our concern towards standardisation of UMAP/VMAP.

7.2 The SPICY System

SPICY system and its evolution to ++SPICY is presented in a row of recent papers. Starting with [7] the new system is introduced. The notion of *mapping quality* plays an important role. The new approach consists of a three-layer architecture:

1. the schema matching module, relying on value correspondences is used to provide the input for
2. the second module, the schema mapping generation and
3. the third module, the mapping verification module, which checks the candidate mappings, choosing the ones that represent better transformations of the source into targets.

This three-layer system is used to address the problem of the *schema mapping quality*. Schema matching is a very challenging problem with no definitive solution. As a consequence, outputs of the attribute matching phase are hardly ready to be used for the schema mapping generation module. The mapping systems usually produce all the alternatives and offer the possibility to choose a preferred result. The major contribution of the SPICY approach in opening the way for automated schema mapping generation is the introduction of the mapping *verification*, based on the notion of *mapping quality*, a phase after the schema mapping generation.

The mapping quality

The mapping quality could be defined in terms of the mapping views, their efficiency, the expressive power of the mapping language, etc. The SPICY definition of quality starts with the system input:

1. besides the source and target schemas,
2. a source and a target instance

are given, these instances are considered *canonical instances*.

Definition 1. Given the *canonical instances* of a mapping, the mapping quality is determined by running the schema mapping on the *canonical source instance* and by comparing the resulting target instance with the *canonical target instance*. The more similar the two target instances are, the higher the mapping rank is.

The SPICY Architecture

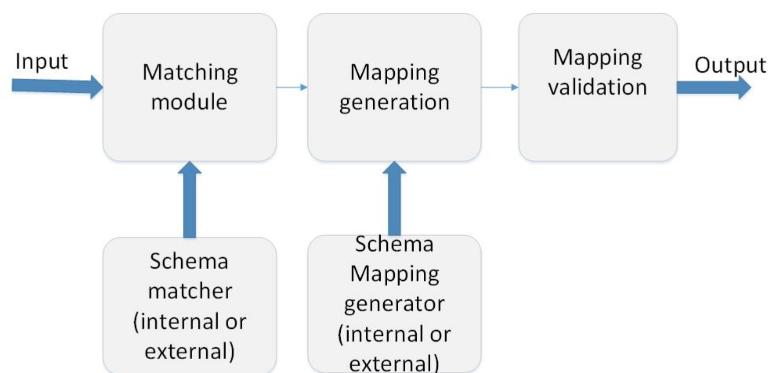

Figure 7.1: SPICY architecture concept

In the Figure 7.1, the three modules are depicted, they transform the above mentioned input into the output and thus executing the data exchange. The first module is the *mapping matcher*. Given the variety of schema matching systems available, SPICY can integrate any one. The *mapping generation* module takes the sets of correspondences discovered at the previous step and generates the mappings as *source to target tuple generating constraints* s-t tgds [34]. SPICY uses a version of CLIO's mapping algorithms [34] to derive the s-t tgds. The *mapping verification* module selects the best mapping from the available set.

The Mapping Verification Example

After the schema matching module runs and produces a number of correspondence candidates, the set of possible schema mappings is created. At this stage it is common to represent the source and target schemas using an abstract graph–base model, essentially a tree–based representation of a nested–relational model, in which set and tuple nodes alternate. The resulting s-t tgds use the syntax adopted in [34]. In Figure 7.2, the running mapping example from [7] is depicted and SPICY yields the following s-t tgd:

$$
\begin{aligned}
&for \quad d \quad in \quad companyDB.divisions, \quad m \quad in \quad d.managers,\\
&\quad p \quad in \quad companyDB.projects\\
&where \quad\quad p.project.manager = m.manager.id\\
&exists \quad p' \quad in \quad projectDB\\
&where \quad p'.project.manager = p.project.projectName \quad and\\
&\qquad p'.project.budget = p.project.budget \quad and\\
&\qquad p'.project.manager = m.manger.name\\
&UNION\\
&for \quad d \quad in \quad companyDB.divisions, \quad m \quad in \quad d.managers,\\
&exists \quad p' \quad in \quad projectDB\\
&\quad p'.project.manager = m.manger.name
\end{aligned}
\tag{7.1}
$$

The transformation is the union of two tgds. The first one constructs the target instance using a join between projects and managers. The new produced tuples contain, besides the project name and the budget, the name of the manager instead of its identification number, as in the source instance. The second s-t tgd produces tuples containing only the names of the managers. Such supplementary tuples exist, only if some managers do not manage any project and thus, they are not already selected by the previous join.

Realisation in UMAP connected to SPICY

The high–level graphic schema mapping language CLIP [37] has the expressive power to represent this mapping example. The corresponding UMAP mapping is depicted in Figure 7.3. The corresponding s-t tgds in form of OCL expressions are the following:

$$
\begin{aligned}
&context \quad value_map :: map(t : Tuple\{p : project, ds : divisions\}) : project_manager\\
&post : result = p_m : project_manager \quad and\\
&p_m.name = t.p.name \quad and\\
&p_m.budget = t.p.budget \quad and\\
&p_m.manager = t.ds- > collect(d : division \quad | \quad d.managers- > Set())\\
&- > select(m : manager \quad | \quad m.id = t.p.manager).name
\end{aligned} \tag{7.2}
$$

$$
\begin{aligned}
&context \quad builder :: build(c_db : company_db) : project_db_set\\
&def : project_managers = c_db.project_set.projects- > Set()\\
&- > iterate(p : project; \quad pms : project_managers = Bag\{\}\\
&| \quad pms- > including(value_map.map(Tupel\{p, c_db.division_set.divisions\})))\\
&post : result = project_db_set(project_managers)
\end{aligned} \tag{7.3}
$$

UMAP uses for the nested mappings two kinds of *association classes* named *builder* and *value_map*. In this example the *builder* iterates over the *project* elements and for each of them calls the function *map* that belongs to the *association class* with the name *value_map*. The argument of this function is an OCL *Tuple* containing the current *project* and a reference to the set of *divisions*. The function *map* can easily map the values for the *project*'s *name* and *budget*, but for the manager's name it has to iterate over all *divisions* and *managers* of each *division* to select the managers name, corresponding to the *id* delivered by the current *project*.

The UMAP solution separates SPICY s-t tgd (7.1) in two very different s-t tgds: the *builder* s-t tgd (7.3) and the *value_map* s-t tgd (7.2). They correspond to the CLIP graphical mapping language constructs: *value mappings*, responsible for the conversion of the source values to target values and *builders* responsible for structural transformation, acting as iterators. In UMAP they are both modeled as UML *association classes* and the OCL expression specializes them. The *value_map* tgd (7.2) takes as input an OCL standard tuple, consisting of the current *project* and the set of all *divisions* and returns as output a *project* with the manager's *id*, replaced by its *name*. The optimisations are delegated to the actual implementation. The OCL expression shows only the navigation to the manager's name using:

1. its *id*,

2. the set of *divisions* and

3. the set of *managers* belonging to each *division*.

By the help of the Figure 7.3 it is easier to follow the OCL expression. The OCL standard function *collect* iterates all the *divisions* and flattens each nested set of *managers* to a unique set of *managers*. Then, by the help of the OCL standard function *select* it identifies the desired *manager* using its *id*. Finally the manager's name is selected and transferred to the resulting tuple. The expressivity of OCL is very high and an extension of this tgd can give a solution, even if the manager's name is not found, i.e., no *manager* has the current *id*. For simplicity, the presented expression does not consider this case.

The s-t tgd *builder* (7.3), an iterator, uses the previously described s-t tgd, the *value_map* (7.2) as subroutine. The s-t tgd *builder* has as input the top level tuple node of the source schema: *company_db* and as output the top level tuple node of the target schema: *project_db_set*. The two s-t tgds are connected by an *association* witch permits the access from *builder* to *value_map*.

Recall the expression (7.3). In this expression the s-t tgd *builder* defines the set of *project_managers*, used further to obtain the final result the target tuple *project_db_set*. It uses a function named *iterate*, not surprisingly, the most general form of the OCL iterators. This function iterates the *projects* set and defines the output set *project_managers*. The output set is initialized as OCL empty standard *Bag*. The next OCL standard function, called *includes*, shows how is constructed each element of this set. This function calls the already described function *map* applied to an OCL standard *Tuple* containing the current *project* element and the whole *division* set. This description restricts itself to the definition of the transformation and delegates to the implementation the optimization of the process. The differences compared to the SPICY syntax consist in the separation of the *iterators* from the *mappers* in two different expressions. An *iterator* expression can call one *mapper* expression and a *mapper* expression can call many *iterator* expressions. This concept is similar to the *context propagation tree* from CLIP [37]. SPICY studies the tree-based, representations of nested-relational model, in which set and tuple nodes alternate, the UMAP representation separates strictly the nested levels, thus the complexity of the OCL expressions do not depend on their number. This is very important in industrial applications where the number of successive levels could be very high.

The usage of UMAP is as difficult as the usage of every high level programming language. A supplementary difficulty arises from the necessity to identify the elements from the source and target that must be connected by the *association classes*. The succession of the two kinds

of this *association classes* named *builder* and *mappers* needs some experience to understand its flexibility.

The SPICY system has a *mapping generation module*. The flexible SPICY architecture allows such a module to be external. UMAP middleware platform offers a standard solution for such an external module with a standard implementation provided by VMAP.

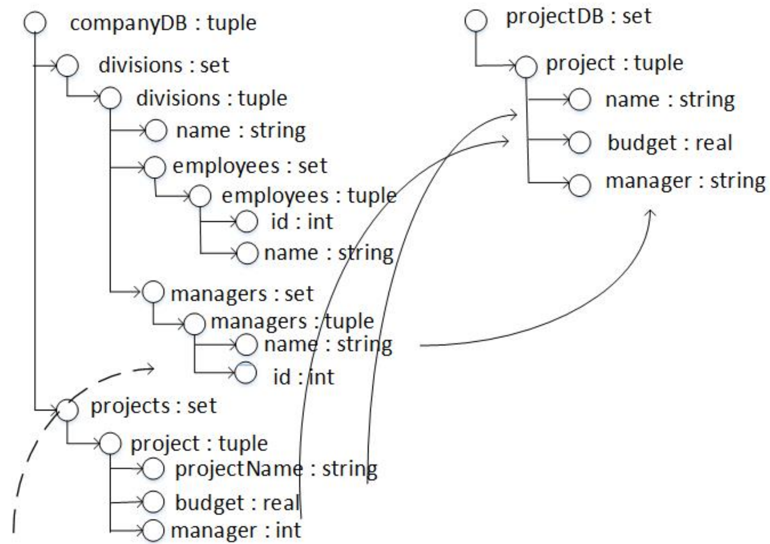

Figure 7.2: SPICY schema mapping

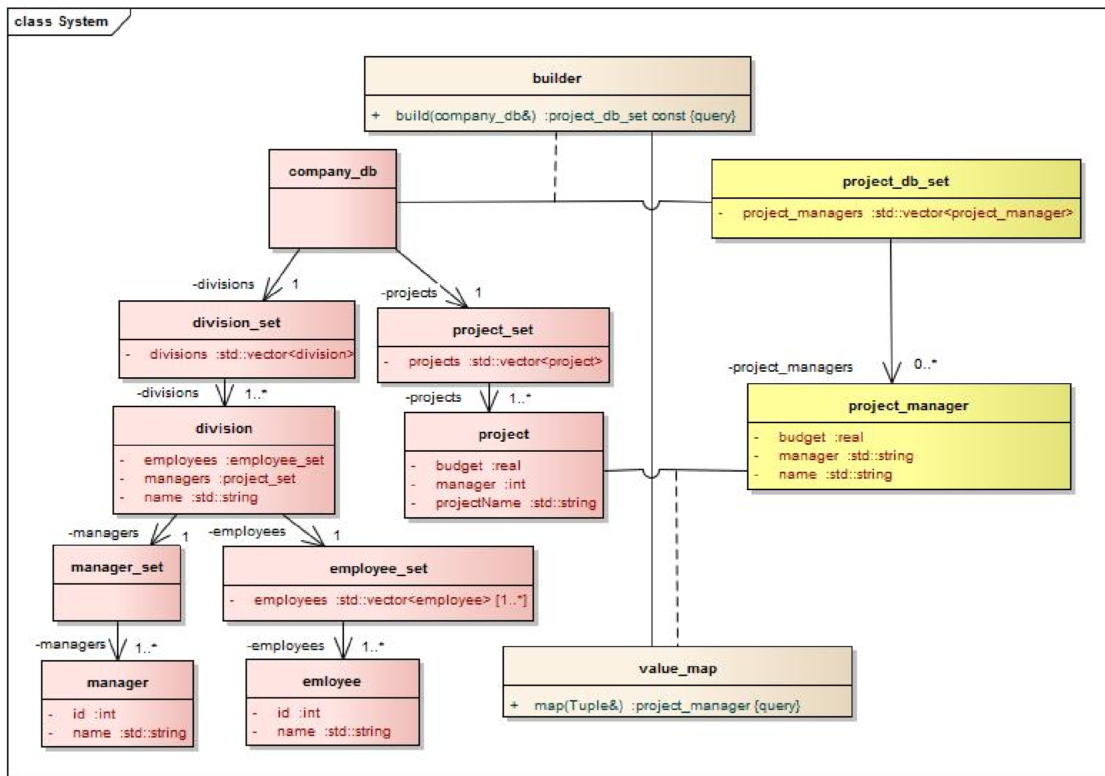

Figure 7.3: SPICY schema mapping using UMAP

CHAPTER 8

Conclusion and future work

In this thesis, we have introduced UMAP, a new universal layer for schema mapping languages and VMAP, its implementation. Schema mappings are modeled by standardized UML class diagrams and OCL expressions. By restricting the UML artifacts to well-understood elements (e.g., classes, associations, aggregations, class functions, and straightforward post-conditions and invariants), there is a well-defined semantics. This allows us to translate UMAP specifications to a broad range of target languages (like C++). We have modeled a set of common schema mapping operations in UMAP, like basic source-to-target dependencies, join, and grouping operations. We have translated all core features of CLIP to UMAP. There is also an implementation available (see `http://www.dbai.tuwien.ac.at/research/project/umap`) generating C++ code showing the translation of typical CLIP language elements to our UML-based formalism, illustrating that our approach works in practical usage. UMAP can be seen as a new middleware for high-level visual schema mapping languages. We propose to use UMAP as a back-end when creating a new visual mapping language. This language, equipped with a formal meta-model can be automatically translated to UMAP via QVT, an evolving standard for Query/View/Transformation. The high-level process for using QVT with UMAP is described in Chapter 5.

We give a summary of the main achieved results.

- First, this thesis presents the syntax and semantics of the UMAP layer. We have shown how to model central elements occurring in common visual mapping languages via UMAP, following a generic strategy defined by the UMAP semantics. As a recent and prominent

example we use CLIP as input language: we map the core CLIP language constructs to our UMAP formalism, demonstrating the translation of source-to-target mappings to UML class diagrams, augmented with OCL-constraints.

- Second, we show the handling of more complex transformations like joins, with grouping in the context of nested schema mappings for tree-like data structures. These transformations are characterized by more involved restructuring operations to map the source schema to the target schema. We show that UMAP has enough expressive power to capture all features of CLIP.

- Last, this thesis presents VMAP, the implementation of UMAP. VMAP consists of a pipeline of executables that transforms a UMAP data exchange model into an executable that creates the target database from the source. The OCL compiler, at the heart of VMAP, performs the semantic analysis and generates source code.

We conclude with directions of future work. Future work includes steps in the direction of extending the use of QVT, the bridge that connects our contribution to potential users: the high level schema mapping tools. The schema mapping languages used together with UMAP must be formal, by the OMG definition, they must have a *meta model*, a UML description of the language. This is a prerequisite in employment of QVT as a vehicle for model transformation of a particular schema mapping language to UMAP.

In Chapter 4 a translation of the *Skolem functions* to UMAP is given by the introduction of supplementary constraints that guarantee that some attribute is unique. The proper definition of the resulting *Skolem function* is delegated to the implementation. Future work should investigate the possibility of using similar constraints for the target *tgds* and *egds*. This could show the flexibility of the UMAP language.

In Chapter 6 is presented our VMAP implementation. The present implementation is restricted to the needs of VMAP. The implementation performance must be analysed using the STBanchmark [3] which offers a great variety of mapping scenarios with nested sources. Further work is needed to extend the compiler to the whole OCL language. Research could extend the OCL compiler to a standalone module, to be used outside our project. OCL is a widely accepted standard in software engineering for requirements gathering. Generating source code for prototyping from requirements is a supplementary benefit.

Bibliography

[1] David H. Akehurst and Behzad Bordbar. On querying UML data models with OCL. In *UML 2001*, volume 2185 of *LNCS*, pages 91–103. Springer, 2001.

[2] Bogdan Alexe and Wang-Chiew Tan. A new framework for designing schema mappings. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, pages 56–88, 2013.

[3] Bogdan Alexe, Wang Chiew Tan, and Yannis Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

[4] Bernhard Beckert, Uwe Keller, and Peter Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *VERIFY, FLoC*, 2002.

[5] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005.

[6] Bison, Parser Generator. *Official Homepage*. `http://www.gnu.org/software/bison/bison.html`.

[7] Angela Bonifati, Giansalvatore Mecca, Alessandro Pappalardo, Salvatore Raunich, and Gianvito Summa. The spicy system: towards a notion of mapping quality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1289–1294, 2008.

[8] Florin Chertes and Ingo Feinerer. UMAP: A universal layer for schema mapping languages. In Hendrik Decker, Lenka Lhotská, Sebastian Link, Josef Basl, and A Min Tjoa, editors, *Proceedings of DEXA2013, Prague, Czech Republic, August 26–29, 2013*, volume 8056 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2013.

[9] Florin Chertes and Ingo Feinerer. VMAP: A visual schema mapping tool. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 1223–1224, 2014.

[10] Ronald Fagin, Laura M. Haas, Mauricio A. Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.

[11] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224. Springer, 2003.

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

[13] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM TODS*, 30(4):994–1055, 2005.

[14] Flex, Lexical Analyzer. *Official Homepage*. `http://flex.sourceforge.net`.

[15] Ariel Fuxman, Mauricio A. Hernandez, Howard Ho, Renee J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: schema mapping reloaded. In *VLDB*, pages 67–78, 2006.

[16] Georg Gottlob and Pierre Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2), 2010.

[17] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 805–810, 2005.

[18] Mauricio A. Hernández, Paolo Papotti, and Wang Chiew Tan. Data exchange with data-metadata translations. *PVLDB*, 1(1):260–273, 2008.

[19] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[20] Bruno Marnette, Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Donatello Santoro. ++spicy: an opensource tool for second-generation schema mapping and data exchange. *PVLDB*, 4(12):1438–1441, 2011.

[21] Giansalvatore Mecca and Paolo Papotti. Schema mapping and data exchange tools: Time for the golden age. *it - Information Technology*, 54(3):105–113, 2012.

[22] Giansalvatore Mecca, Paolo Papotti, and Salvatore Raunich. Core schema mappings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 655–668, 2009.

[23] Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Marcello Buoncristiano. Concise and expressive mappings with +spicy. *PVLDB*, 2(2):1582–1585, 2009.

[24] Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Donatello Santoro. What is the IQ of your data transformation system? In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 872–881, 2012.

[25] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. P.-H., 1990.

[26] Alan Nash, Philip A. Bernstein, and Sergey Melnik. Composition of mappings given by embedded dependencies. In *PODS 2005*, pages 172–183. ACM, 2005.

[27] OMG. *MOF 2.0 Query, View, and Transformation*, 2011. www.omg.org.

[28] OMG. *Unified Modeling Language Infrastructure 2.4.1*, 2011. www.omg.org.

[29] OMG. *Unified Modeling Language Superstructure 2.4.1*, 2011. www.omg.org.

[30] OMG. *Object Constraint Language 2.3.1*, 2012. www.omg.org.

[31] OMG. *Meta Object Facility 2.4.2*, 2014. www.omg.org.

[32] OMG. *XML Metadata Interchange 2.4.2*, 2014. www.omg.org.

[33] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. In *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011*, 2011.

[34] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB 2002*, pages 598–609. Morgan Kaufmann, 2002.

[35] Alessandro Raffio. *Schema Mapping for Semi-structured Data*. PhD thesis, Politecnico di Milano Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32 I 20133 — Milano, 2008.

[36] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a tool for mapping hierarchical schemas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1271–1274, 2008.

[37] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE 2008*, pages 30–39, 2008.

[38] Tamás Vajk, Gergely Mezei, and Tihamer Levendovszky. An incremental OCL compiler for modeling environments. *Electronic Communication of the European Association of Software Science and Technology*, 15, 2008.

[39] Xerces, Parser Generator. *Official Homepage*. `http://xerces.apache.org/xerces-c/`.