Roma Tre University
Ph.D. in Computer Science and Engineering

Metadata Management

Giorgio Gianforme

Metadata Management

A thesis presented by
Giorgio Gianforme
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Engineering

Roma Tre University
Dept. of Informatics and Automation

April 2009

Committee:
Prof. Paolo Atzeni

Reviewers:
Prof. Denilson Barbosa
Prof. Georg Gottlob

to my family

Abstract

The management of heterogeneous databases, in integrated or collaborative contexts, always involves the need for solutions to *data programmability* issues. In general, data programmability addresses problems dealing with evolving scenarios: changes in a database which collaborates in a heterogeneous environment often imply a sequence of propagating changes in related databases at any level, model, schema, and data.

In this scenario there is the need to translate data and their descriptions from one model (i.e. data model) to another. Even small variations of models are often enough to create difficulties. For example, while most database systems are now object-relational, the actual features offered by different systems rarely coincide, so data migration requires a conversion. Every new database technology introduces more heterogeneity and thus more need for translations.

According to the model management proposal, these problems can be solved conveniently applying the ModelGen operator, that can be defined as follows using our terminology: given two models M_1 and M_2 and a schema S_1 of M_1, ModelGen translates S_1 into a schema S_2 of M_2 that properly represents S_1.

In this dissertation we will be presenting our theoretical and practical contribution to the development of an effective implementation of a generic (i.e. model independent) platform for schema and data translation.

We improve the expressive power of its supermodel, that is the set of models handled and accuracy and precision of such models representation. We show how it is possible to automatically reason on models and schemas and how to find a suitable translation given a source and a target model exploiting a formal system, proved to be sound and complete. Then we propose an extension of Datalog based on the use of hierarchies and a sort of polymorphism, that provides a significant simplification in the definition of translations and a higher level of reuse in the specification of elementary translations. Finally we present a new, lightweight, runtime approach to the translation problem, where translations of data are performed directly on the operational system.

Acknowledgments

I would like to express my deep gratitude to Prof. Paolo Atzeni, who has been conducting me during the long walk that led me to this thesis.

I would like to thank Microsoft Research Europe that trusted in and bet on me funding my PhD through their European PhD Scholarship Programme.

I am grateful to Prof. Georg Gottlob, who welcomed and supported me during my visiting at Computing Laboratory of Oxford University.

Finally, I would like to thank all my friends and colleagues of the Database Group and the whole Department of Informatics and Automation of Roma Tre University, for fruitful collaboration and time enjoyed together during the endless days spent at the department.

Contents

List of Figures **xii**

1 Introduction **1**
1.1 Motivations 1
1.2 Overview 3
1.3 Outline and Contributions 8

2 A Metamodel Approach **11**
2.1 Overview 11
2.2 Related Work 18
2.3 Models, Schemas, and the Dictionary 19
2.4 Supermodel Explained 31
2.5 The Translations 39

3 A Formal System **47**
3.1 Overview 47
3.2 Related Work 48
3.3 Descriptions of Constructs and Models 49
3.4 Signatures of Rules and Their Application 53
3.5 Soundness and Completeness 58
3.6 Applications of the Results 64

4 Refactoring the Supermodel **71**
4.1 Coherent and Cohesive 71
4.2 Refactoring the Formal System 72

5 PolyDatalog **75**
5.1 Overview 75

5.2 Related Work 76
5.3 Extending Datalog 77
5.4 Implementation 83
5.5 Experimental results 85

6 Toward an On-line Operator **89**
6.1 Overview 89
6.2 Overall Approach 91
6.3 Generating Views 94
6.4 The View-Generation Algorithm 100
6.5 Related work 110

Conclusions **113**

Appendices **116**

A. Basic Translations **119**

B. Rules **123**

Bibliography **129**

List of Figures

1.1 A simple entity-relationship schema. 5
1.2 A simple object-oriented version of the schema in Figure 1.1. . . . 5
1.3 A simple object-oriented schema. 6
1.4 A relational version of the schema in Figure 1.3 with new key attributes. 6
1.5 A relational version of the schema in Figure 1.3 without new key attributes. 7
1.6 Models and transformations. 7

2.1 The translation process. 13
2.2 A translation composed of three steps. 14
2.3 Some models and translations between them. 16
2.4 A simplified conceptual view of models and constructs. 20
2.5 The relational implementation of a portion of the supermodel part of the dictionary. 21
2.6 The relational implementation of a portion of the models part of the dictionary. 22
2.7 The dictionary for a simple entity-relationship model. 25
2.8 The entity-relationship schemas described in Figure 2.7. 26
2.9 The dictionary for a simple object-oriented model. 26
2.10 The object-oriented schema described in Figure 2.9. 27
2.11 A model-generic dictionary, based on the supermodel. 28
2.12 The four parts of the dictionary. 29
2.13 Constructs and models. 30
2.14 The relational model. 31
2.15 The binary entity-relationship model. 32
2.16 The n-ary entity-relationship model. 34

2.17 The object-oriented model. 35
2.18 The object-relational model. 36
2.19 The XSD language. 38
2.20 The Supermodel. 39

3.1 The formal system. 47
3.2 The universe of constructs for the examples. 50

5.1 A simplified object-relational model. 78
5.2 Generalizations of the object-relational model of Figure 5.1. 80
5.3 Experimental results. 86

6.1 The runtime translation procedure. 91
6.2 A simple object-relational schema. 93

A.1 Translations between families. 121

Chapter 1

Introduction

1.1 Motivations

Informatics is the science of information. In detail, according to the French Academy, it is the science of rational management of information, also intended as support to human knowledge and communication. Models are needed in order to manage information, that is to represent a specific domain with the needed level of abstraction. Hence the management of models and their descriptions by means of metadata is a key point for every information system and is a fertile research area.

The first use of metadata for data processing was reported in [McG59]. Since then, metadata-related tasks and applications have become truly pervasive and metadata management plays a major role in today's information systems. In fact, the majority of information system problems involve the design, integration, and maintenance of complex application artifacts, such as application programs, databases, web sites, workflow scripts, object diagrams, and user interfaces. These artifacts are represented by means of formal descriptions, or models, and, consequently, metadata on models. Indeed, to solve these problems we have to deal with metadata, but it is well known that applications solving metadata manipulation are complex and hard to build, because of heterogeneity and impedance mismatch. Heterogeneity arises because data sources are independently developed by different people and for different purposes and subsequently need to be integrated. The data sources may use different data models, different schemas, and different value encodings. Impedance mismatch arises because the logical schemas required by applications are different from

the physical ones exposed by data sources. The manipulation includes designing mappings (which describe how two models are related to each other) between the models, generating a model from another model along with a mapping between them, modifying a model or mapping, interpreting a mapping, and generating code from a mapping.

In the past, these difficulties have always been tackled in practical settings by means of ad-hoc solutions, for example by writing a program for each specific application. This is clearly very expensive, as it is laborious and hard to maintain. In order to simplify such manipulation, Bernstein et al. [BHP00, Ber03, Mel04] proposed a model management system. Its goal is to factor out the similarities of the metadata problems studied in the literature and develop a set of high-level operators that can be utilized in various scenarios. Within this system, we can treat models and mappings as abstractions that can be manipulated by such model-at-a-time and mapping-at-a-time generic operators; these operators are meant to be generic in the sense that a single implementation of them is applicable to all of the data models. We want to remark that in this dissertation we use the terms schema and data model as common in the database literature, though a recent trend in model management follows a different terminology (and uses model instead of schema and metamodel instead of data model).

The proposed model management operators are just five and they are demonstrated to be useful to solve three well known meta data problems like schema integration, schema evolution, and round-trip engineering. A description of these operators follows.

Match takes two schemas as input and returns a mapping between them.

Compose takes a mapping between schemas A and B and a mapping between schemas B and C, and returns a mapping between A and C.

Diff takes a schema A and a mapping between A and some schema B, and returns the sub-schema of A that does not participate in the mapping.

ModelGen takes a schema A, and returns a new schema B based on A (typically in a different model than A's) and a mapping between A and B.

Merge takes two schemas A and B and a mapping between them, and returns the union C of A and B along with mappings between C and A, and C and B.

In this dissertation we mainly focus on the ModelGen operator. The problem of translating schemas between data models is acquiring progressive significance in heterogeneous environments. Applications are usually designed to deal with information represented according to a specific data model, while the evolution of systems (in databases as well as in other technology domains, such as the Web) led to the adoption of many representation paradigms.

For example, many database systems are nowadays object-relational (OR) and so it is reasonable to exploit their full potentialities by adopting such a model while most applications are designed to interact with a relational database. Also, object-relational extensions are often non-standard, and conversions are needed. The explosion of the eXtensible Markup Language (XML), with all its applications (for example, as a format for information exchange or as the language for the semantic Web), has increased the heterogeneity of representations. In general the presence of several coexisting models introduces the need for translation techniques and tools.

1.2 Overview

The management of heterogeneous databases, in integrated or collaborative contexts, always involves the need for solutions to *data programmability* issues. In general, data programmability addresses problems dealing with evolving scenarios: changes in a database which collaborates in a heterogeneous environment often imply a sequence of propagating changes in related databases at any level, model, schema and data [BM07, Haa07, HAB+05]. Heterogeneity means that on the one hand systems are developed by different people, fostering different data models and technologies; on the other hand it recalls problems involving different software components using shared and interoperating data.

In this scenario there is the need to translate data and their descriptions from one model (i.e. data model) to another. Even small variations of models are often enough to create difficulties. For example, while most database systems are now object-relational, the actual features offered by different systems rarely coincide, so data migration requires a conversion. Every new database technology introduces more heterogeneity and thus more need for translations. For example, the growth of XML has led to such issues, including the need to have object-oriented wrappers for XML data, the translation from nested XML documents into flat relational databases and vice versa, and the conversion from one company standard to another, such as using attributes for simple values and sub-elements for nesting versus representing all data in sub-elements.

Other popular models lead to similar issues, such as Web site descriptions, data warehouses, and forms. In all these settings, there is the need for translations from one model to another.

According to the model management proposal, these problems can be solved conveniently applying the ModelGen operator, that can be defined as follows using our terminology: given two models M_1 and M_2 and a schema S_1 of M_1, ModelGen translates S_1 into a schema S_2 of M_2 that properly represents S_1.

As there are many different models, what we need is an approach that is generic across models and can handle the idiosyncrasies of each model. Ideally, one implementation should work for a wide range of models, rather than implementing a custom solution for each pair of models.

The first step toward a uniform solution is the adoption of a general model to properly represent all the considered data models (e.g. entity-relationship, object-oriented, relational). The proposed general model is based on the idea of construct: a construct represents a "structural" concept of a data model. We find out a construct for each "structural" concept of every considered data model and, hence, a data model is completely represented by the set of its constructs. Let us consider two popular data models, entity-relationship (ER) and object-oriented (OO). Indeed, each of them is not "a model", but "a family of models"[1], as there are many different proposals for each of them: OO with or without keys, binary and n-ary ER models, OO and ER with or without inheritance, and so on. "Structural" concepts for these data models are, for example, entity, attribute of entity, and binary relationship for the ER and class, field, and reference for the OO.

Then, in order to define a translation, we have to discover correspondences between constructs of the two models involved in the transformation (i.e. source and target model). For example, if we have an ER schema and we want to translate it into an OO schema, we essentially map entities to classes and replace relationships with references. In detail, there is a one-to-one mapping between entities and classes, while things are more complex for relationships. As usually relationships are more sophisticated (they can be n-ary, many-to-many, or have attributes) and the introduction of new classes (besides those corresponding to entities) may be needed. So, if for example we want to translate the ER schema in Figure 1.1, which involves a many-to-many relationship, then we have to introduce a new class to properly represent such a relationship in the OO model (see Figure 1.2).

[1]The notion of *family* of models is intuitive here and will be made more precise in Chapter 3.

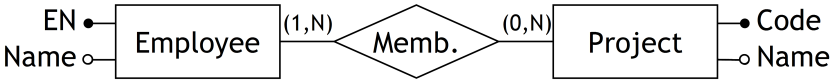


Figure 1.1: A simple entity-relationship schema.

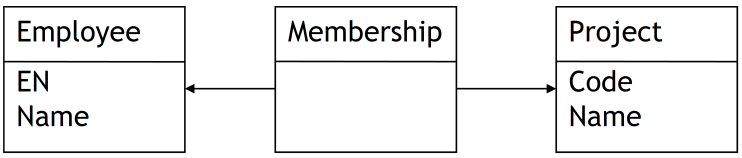


Figure 1.2: A simple object-oriented version of the schema in Figure 1.1.

In this way, we have uniform representations of models (in terms of constructs) and, consequently, of transformations, but these representations are not general. Indeed, the weakness of this (partial) solution is that we should define a transformation for each "source-target" pair of models. This is unfeasible as the number of (variants of) models grows. A dozen of constructs, each with some variations and many possible combinations, is enough to reach hundreds or thousands of different models. To overcome this limit, we exploit an observation of Hull and King [HK87], drawn on later by Atzeni and Torlone [AT93]: most known models have constructs that can be classified according to a rather small set of *metaconstructs*. Recalling our example, entities are mapped to classes (and vice versa in the reverse translation) because the two types of constructs play the same role (or, in other terms, "they have the same meaning"), then we can define a generic metaconstruct to represent both these concepts; the same happens for attributes of entities and fields of classes. Conversely, relationships and references do not have the same meaning and hence one metaconstruct is not enough to properly represent both concepts. From this new point of view, based on metaconstructs, in order to perform a translation, we essentially copy the metaconstructs allowed in the source model and in the target one and eliminate the metaconstructs not allowed in the target model, representing them by means of the allowed ones. With reference to the example, we copy the metaconstructs representing entities and their attributes (since they are allowed in the target model to represent classes and their fields) and transform the metaconstruct (not allowed in the target model) representing relationships, replacing it with the metaconstruct representing references.

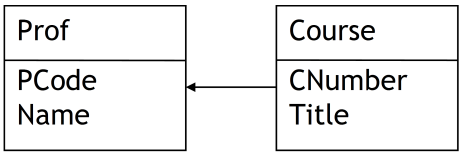


Figure 1.3: A simple object-oriented schema.

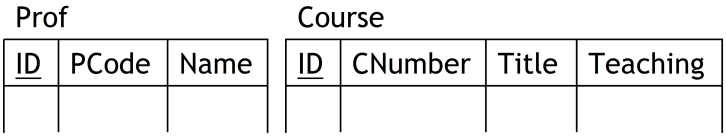


Figure 1.4: A relational version of the schema in Figure 1.3 with new key attributes.

Hence, to define a transformation, we still have to discover correspondences between source and target model, but with reference to the metaconstructs of the involved models.

An interesting issue is that the actual translation can depend on the details of the source and target model. For example, if we go from an object-oriented model to a relational one, then we have to implement different translations depending on whether the source model requires the specification of identifiers or not (ignoring the internal OIDs of objects). Let us refer to the schema in Figure 1.3. If the object-oriented model does not allow (or it allows but does not require) the specification of identifiers, then, in the translation toward the relational model, we have to add new key attributes in the tables generated for each class. If instead identifiers are required (and, in the example, the fields *PCode* and *CNumber* are specified as identifiers of the classes *Prof* and *Course*, respectively), no new attributes are needed. Figures 1.4 and 1.5 show the two possibilities in the translated schema. In both cases, referential integrity is needed, in the former case involving the new attributes and in the latter over the existing identifiers.

The example shows that we need to be able to deal with the specific aspects of models, and that translations need to take them into account. We have shown two versions of the object-oriented model, one that has visible keys (besides the system managed identifiers) and one that does not. The translation has to be different as well.

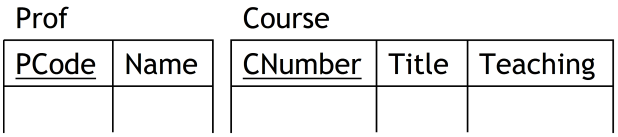


Figure 1.5: A relational version of the schema in Figure 1.3 without new key attributes.

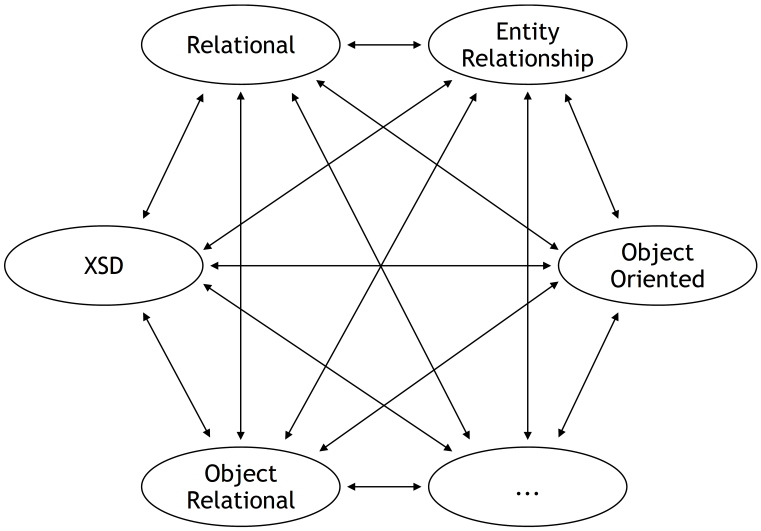


Figure 1.6: Models and transformations.

The goal of the research project at "Roma Tre" University we have contributed to is the development of a platform that allows the specification of the source and target models of interest (including OO, OR, ER, UML, XML Schema, and so on), with all relevant details, and to generate the translation of their schemas (and instances of those schemas) from one model to another. This scenario is sketched in Figure 1.6, where the ovals represent families of models and each bidirectional arrows represent the set of transformations between variants of models of the connected families; the oval with the dots inside represents any other data model that could be represented with the same metamodel approach by means of constructs.

In particular our contribution, detailed in the next section, affected the schema-related component of MIDST (Model Independent Data and Schema Translation), a framework for the development of an effective implementation of a generic (i.e. model independent) platform for schema and data translation. With respect to data translation, we adopt an approach that differs from the proposal sketched in a preliminary report by Atzeni et al. [ACB06]. We propose a new approach where data translations are performed directly on the operational system.

1.3 Outline and Contributions

The rest of the thesis is organized as follows. In Chapter 2, we present the overall approach and the dictionary that handles models and schemas. Then we illustrate our first contribution: the extension of the supermodel that is now capable of modeling essential concepts to represent recent complex data models like the object-relational and the XML Schema (XSD). Example of the newly introduced concepts are nesting relationships, complex structured elements, collections, and substructures. The obtained supermodel is actually used in the MIDST framework. Detailed descriptions of the managed data models are provided. For each model, we explain how its concepts are represented by means of metaconstructs and relationships between them; then we illustrate the complete supermodel, obtained as union of the metaconstructs involved in the considered data models. The chapter ends with a discussion on criticalities and problems arising with the enlargement of the supermodel that motivate and justify the definition of a formal system presented in Chapter 3.

A formal system is conceived to automatically reason on models. Chapter 3 starts with the formalization of such formal system, and then proceeds with the proofs of its soundness and completeness. We associate a concise description with each model, by means of propositional formulas, and a signature with each basic translation. Then, we define the application of a signature to a model description. We prove that the model we derive by means of the application of signatures is exactly the model that allows the set of schemas that can be obtained by means of the Datalog programs. The chapter ends with the presentation of the algorithm, based on the formal system, that allows to find automatically a sequence of translations, out of a library of many basic translations, to perform the transformation requested by the user. A new module of the MIDST tool exploits these results. It is capable to extract automatically descriptions of models (and schemas) and signatures of rules, to apply such

signatures to model (and schema) descriptions; moreover it implements the aforementioned algorithm.

In Chapter 4, we illustrate the refactoring undergone by our supermodel. The goal of such refactoring is the definition of a more compact and cohesive supermodel. In the last part of the chapter, we show how the formal system of Chapter 3 has to be modified in order to work properly with the new supermodel. Such refactoring leads us to solve the problem of a combinatorial explosion of the number of Datalog rules needed.

In Chapter 5, we propose an extension of Datalog. This extension is aimed at introducing some kind of inheritance and polymorphism in Datalog in order to exploit the new features of the refactored supermodel. The idea is to consider every variant of a construct (identified by certain references) like a child of a generalization rooted in a "generic" construct that has only mandatory fields (i.e. without references). Then, extending ad-hoc Datalog, we can define a polymorphic rule for a root construct that can be automatically "instantiated", obtaining specific rules for each variant of that construct. The rule engine of the MIDST tool has been updated to properly "compile" polymorphic rules. In the last part of the chapter we present our experimental results.

In Chapter 6, we present our ongoing project: a new, lightweight, runtime approach to the translation problem, where data is not moved from the operational system and translations are performed directly on it. The new version of the MIDST tool needs only to know the model and the schema of the source database and generates views on the operational system that transform the underlying data (stored in the source schema) according to the corresponding schema in the target model. Views are generated in an almost automatic way, on the basis of the Datalog rules for schema translation.

Chapter 2

A Metamodel Approach

2.1 Overview

Constructs and Models

As we saw in the previous chapter, we adopt a uniform representation for data models. Our approach is based on the idea of a *metamodel*, defined as a set of constructs that can be used to define models, which are instances of the metamodel. This is based on Hull and King's observation [HK87] that the constructs used in most known models can be expressed by a limited set of generic (i.e. model-independent) *metaconstructs*: lexical, abstract, aggregation, generalization, and function. In fact, we define a metamodel by means of a set of generic metaconstructs. Each model is defined by its constructs and the metaconstructs they refer to. Simple versions of the models in the examples in Chapter 1 could be defined as follows:

- an entity-relationship model involves abstracts (the entities), aggregations of abstracts (relationships) and lexicals (attributes of entities and, in most versions of the model, of relationships);
- an object-oriented model involves abstracts (classes), reference attributes for abstracts, which are essentially functions from abstracts to abstracts, and lexicals (fields or properties of classes);
- a relational model involves aggregations of lexicals (tables), components of aggregations (columns), which can participate in keys and foreign keys defined over aggregations and lexicals.

The various constructs are related to one another by means of references (for example, each attribute of an abstract has a reference to the abstract it belongs to) and have properties that specify details of interest (for example, for each attribute we specify whether it is part of the identifier and for each aggregation of abstracts we specify its cardinalities). We will see these aspects in detail in the following sections.

All the information about models and schemas is maintained in a dictionary. We will discuss the dictionary in some detail later in the dissertation; here we just mention that it has a relational implementation, which is exploited by the specification of translations, written in Datalog.

The Supermodel and the Translations

A major concept in our approach is that of *supermodel*, a model that has constructs corresponding to all the metaconstructs known to the system. Thus, each model is a specialization of the supermodel and a schema in any model is also a schema in the supermodel, apart from the specific names used for constructs.

It is worth mentioning that while we say that we follow a "metamodel" approach, what we actually implement in our dictionary is the supermodel, as we will see in Section 2.3.

The supermodel gives us two interesting benefits. First, it acts as a "pivot" model, so that it is sufficient to have translations from each model to and from the supermodel, rather than translations for every pair of models. Thus, a linear and not a quadratic number of translations is needed. Indeed, since every schema in any model is also a schema of the supermodel (modulo construct renaming), the only needed translations are those within the supermodel with the target model in mind. A translation is composed of: (a) a "copy" (with construct renaming) from the source model into the supermodel; (b) an actual transformation within the supermodel, whose output includes only constructs allowed in the target model; (c) another copy (again with renaming) into the target model, as depicted in Figure 2.1. The second advantage is related to the fact that the supermodel emphasizes the common features of models. So, if two source models share a construct, then their translations toward similar target models could share a portion of the translation as well. In our approach, we follow this observation by defining elementary (or *basic*) translations that refer to single constructs (or even specific variants thereof). Then, actual translations are specified as compositions of basic ones, with significant reuse of them.

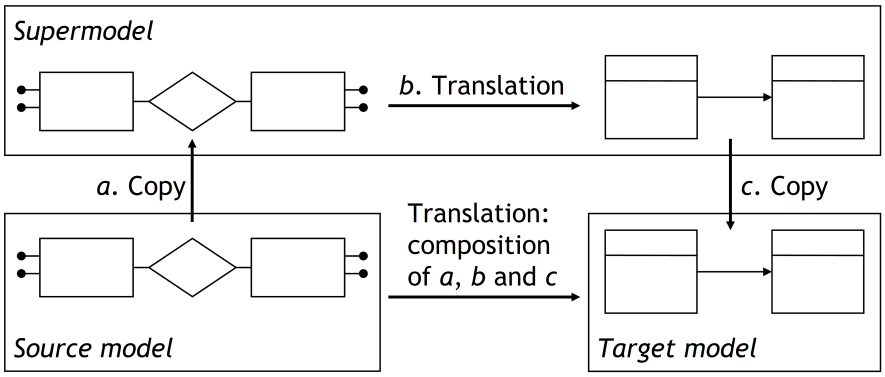


Figure 2.1: The translation process.

For example, assume we have a source ER model with binary relationships (with attributes) and no generalizations and simple OO model as target. To perform the task, we would first translate the source schema by renaming constructs according to their corresponding homologous elements (i.e. abstracts, lexicals, and binary aggregations of abstracts) in the supermodel and then apply the following steps (sketched in Figure 2.2, where the dashed boxes highlight the portion of schema affected by the next transformation):

P$_1$ eliminate attributes of aggregations, by introducing new lexicals or new abstracts and one-to-many aggregations;

P$_2$ eliminate many-to-many aggregations, by introducing new abstracts and one-to-many aggregations;

P$_3$ replace one-to-many aggregations with references between abstracts.

Finally, we would translate the output schema by renaming constructs according to the target model nomenclature (i.e. objects, fields, and references).

Besides reuse of basic translations, the major advantage of the specification of translations as composition of basic ones is the possibility to add or remove steps on the basis of the source and target models. Let us consider two simple examples, just variants of the previous one. If we have a source ER model with no attributes on relationships (still binary), then we can apply steps **P**$_2$ and **P**$_3$ above only. If instead we have a source ER model with generalizations, the

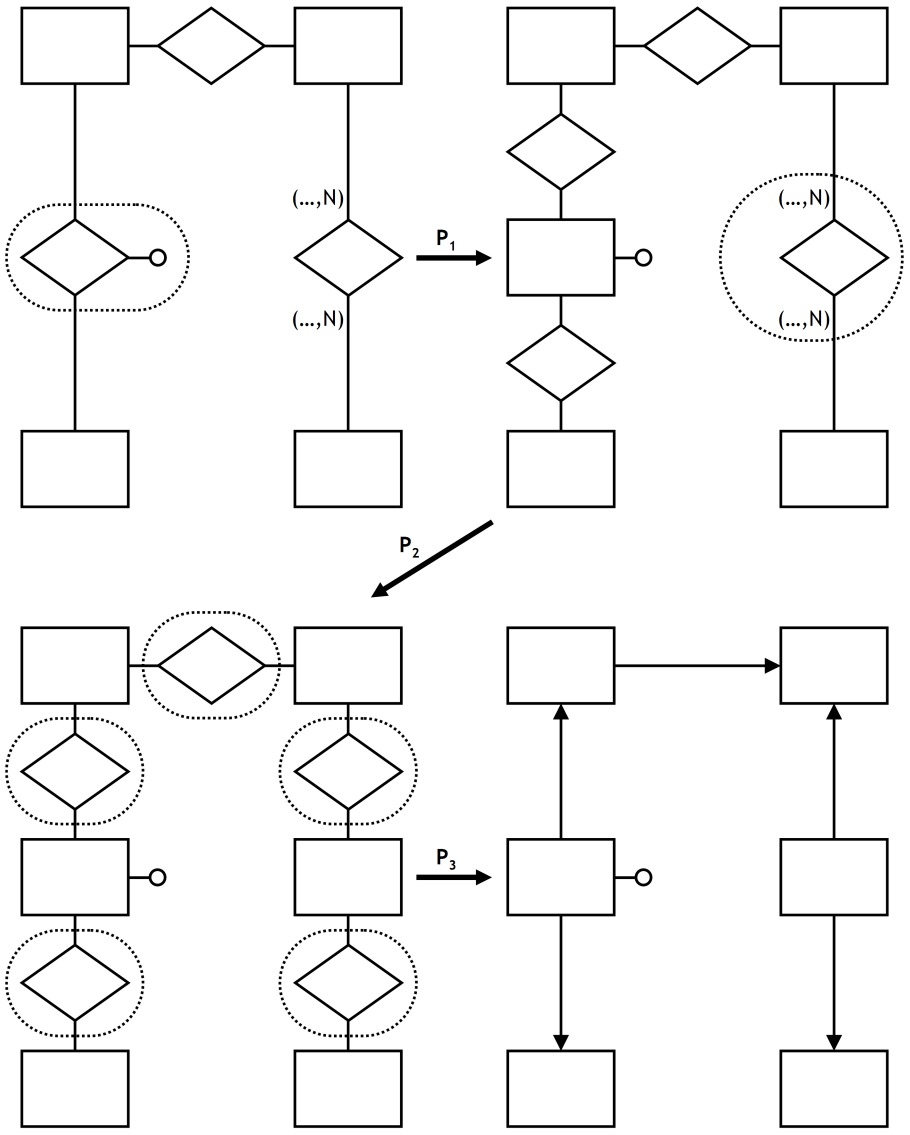


Figure 2.2: A translation composed of three steps.

three steps above are not enough; they have to be followed by another step that takes care of generalizations:

$\mathbf{P}_4$ eliminate generalizations (replacing them with references).

It is important to note that the basic steps are highly reusable. Let us comment on this issue with the help of Figure 2.3 (where we can recognize the three source models of the previous examples, marked with ER2, ER5, and ER3, respectively, and the target model marked with OO2).

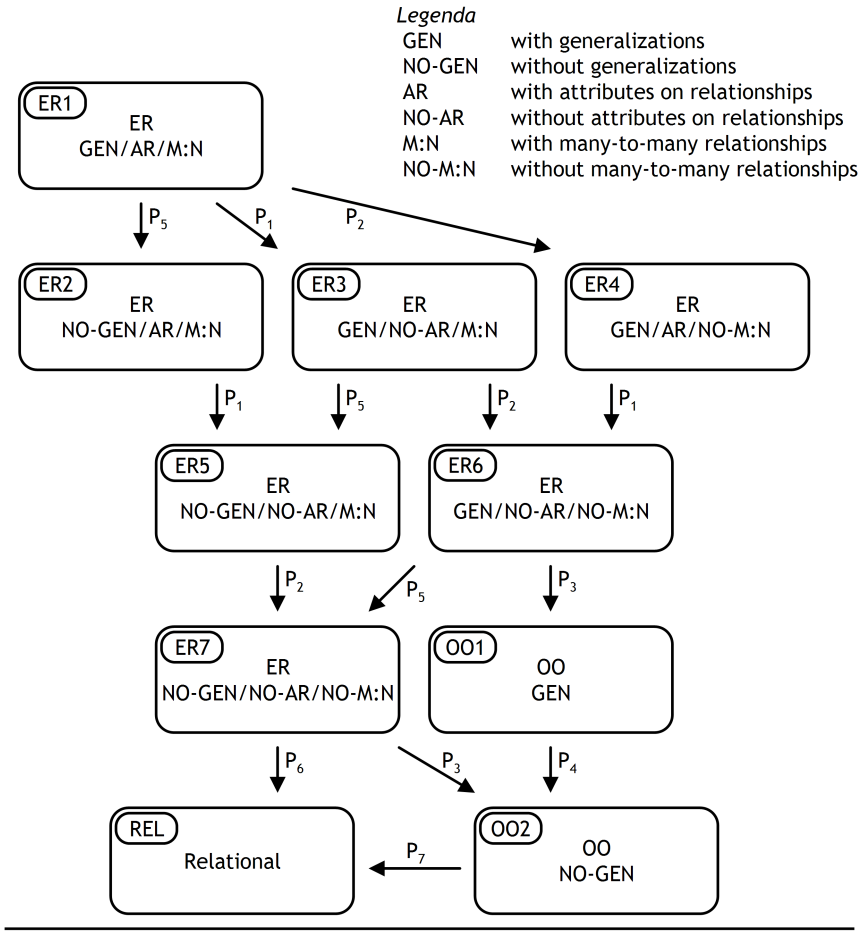
The figure shows several models, that can be obtained by combining the constructs seen in this chapter, and some translations between them. Indeed, this is just a subset of the considered models, but it is sufficient to make the point. In the figure, we have also omitted various translations, including the identity one, which would be useful to go from a model to a more complex version of it, for example from ER2 or ER3 to ER1.

Reuse arises in various ways. First, entire translations can be used in various contexts: the translation composed of steps $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$, which we have mentioned for the translation from ER2 to OO2, can be used also to go from the most complex ER model in the picture (the top one, indicated with ER1) to the OO model with generalizations (OO1) in the picture. Second, individual steps can be composed in different ways: for example, if we want to go from ER1 to the relational model (in the bottom left corner), then we could use basic translation $\mathbf{P}_5$ to eliminate generalizations, then $\mathbf{P}_1$ and $\mathbf{P}_2$ to reach a simple ER model (ER7), and finally $\mathbf{P}_6$ to get to the relational one.

Specification of Basic Translations

Translations are implemented as programs[1] in a Datalog variant with OID-invention (with ideas from ILOG [HY90]), where the latter feature is obtained by means of the use of Skolem functors. Each translation is usually concerned with a very specific task, such as eliminating a certain variant of a construct (possibly introducing another construct), with most of the constructs left unchanged. Therefore, in our programs only a few of their rules concern real translations, whereas most of them just copy constructs from the source schema to the target one. For example, the translation that performs step $\mathbf{P}_2$ in Figures 2.2 and 2.3 (in an ER model, the elimination of many-to-many aggregations of abstracts, by introducing new abstracts and one-to-many aggregations) would involve the rules for the following tasks:

[1] This justifies the symbol $\mathbf{P}$ used to denote basic translations



Basic Translations

- P_1 eliminate attributes from aggregations of abstracts
- P_2 eliminate many-to-many aggregations of abstracts
- P_3 replace aggregations of abstracts with references
- P_4 eliminate generalizations (introducing new references)
- P_5 eliminate generalizations (introducing new aggregations of abstracts)
- P_6 replace abstracts and their aggregations with aggregations of lexicals
- P_7 replace abstracts and references with aggregations of lexicals

Figure 2.3: Some models and translations between them.

$R_{2,1}$ copy abstracts;

$R_{2,2}$ copy lexical attributes of abstracts;

$R_{2,3}$ copy one-to-one and one-to-many aggregations;

$R_{2,4}$ copy attributes of one-to-one and one-to-many aggregations;

$R_{2,5}$ generate an entity for each many-to-many aggregation;

$R_{2,6}$ generate, for each abstract generated by $R_{2,5}$, an aggregation between such abstract and the copy of the first abstract involved in the original many-to-many aggregation;

$R_{2,7}$ generate, for each abstract generated by $R_{2,5}$, an aggregation between such abstract and the copy of the second abstract involved in the original many-to-many aggregation;

$R_{2,8}$ generate, for each attribute of each many-to-many aggregation, an attribute for the abstract generated by $R_{2,5}$.

We will discuss rules in detail in Section 2.5. Here we just make some high-level comments. Rules refer directly to our dictionary, and this is facilitated by the relational implementation: the predicates in the rule correspond to the tables in the dictionary. The body of each rule includes conditions for its applicability. Rule $R_{2,1}$ has no condition, and so it copies all abstracts. Instead, Rule $R_{2,5}$ is applied only to many-to-many binary aggregations of abstracts (denoted by suitable boolean values for its properties). Rule $R_{2,6}$ is more complex because it involves more constructs but again it is applied only to many-to-many binary aggregations of abstracts together with one of the two entities linked by each of these aggregations. Each rule "generates" a new construct instance in the dictionary with a new identifier generated by a Skolem functor[2]; rule $R_{2,1}$ generates a new abstract for each abstract in the source schema (and it is a copy, except for the internal identifier) whereas rule $R_{2,5}$ and $R_{2,6}$ generate a new abstract and a new binary aggregation, respectively, for each binary aggregation, with suitable features.

It is worth noting that the specification of rules in Datalog allows for another kind of reuse. In fact, a basic translation is a program made of several Datalog rules, and it is often the case that a rule is used in various programs.

[2]We will comment on Skolem functors, which may appear in both heads and bodies of rules, in Section 2.5.

This happens for all "copy" rules, such as $R_{2,1}$, and for many other rules; for example, the two programs that implement steps $\mathbf{P}_6$ and $\mathbf{P}_7$ in Figure 2.3 (which translate into the relational model from a simple ER and a simple OO, respectively) would share a rule that transforms abstracts into aggregations of lexicals (entities or classes into tables) and a rule that transforms attributes of abstracts into components of aggregations (attributes or fields into columns).

2.2 Related Work

Various proposals exist that consider schema and data translation. However, most of them only consider specific data models. We comment here on related pieces of work that address the problem of model-independent translations.

The term *ModelGen* was coined by Bernstein in [Ber03] where he argues (as in [BHP00]) for the development of model management systems, consisting of generic operators for solving problems involving metadata and schemas, and provides an example of using ModelGen to solve a schema evolution problem.

An early approach to ModelGen (even before the term was coined) was MDM, proposed by Atzeni and Torlone [AT96]. The basic idea behind MDM and the similar approaches (Claypool and Rundensteiner et al. [CR03, CRZ+01] Song et al. [SZK04], and Bézivin et al. [BBDV03]) is useful but offers only a partial solution to our problem. Their representation of models and transformations is hidden within the tool's imperative source code, not exposed as more declarative, user-comprehensible rules. This leads to several difficulties. First, only the designers of the tool can extend the models and define the transformations. Thus, instance level transformations would have to be recoded in a similar way. Moreover, correctness of the rules has to be accepted by users as a dogma, since their only expression is in complex imperative code. Also, any customization would require changes in the tool's source code. All of these problems are overcome by our approach.

There are two concurrent projects on ModelGen. The approach of Papotti and Torlone [PT05] is not rule-based. Rather, their transformations are imperative programs, with the weaknesses described above. Their translation is done by translating the source data into XML, performing an XML-to-XML translation expressed in XQuery to reshape it to be compatible with the target schema, and then translating the XML into the target model. This is similar to our use of a relational database (i.e. the relational implementation of the supermodel) as the "pivot" between the source and target databases. The approach of Bernstein, Melnik, and Mork [BMM05, MBM07] is rule-based,

like ours. However, unlike ours, it is not driven by a relational dictionary of schemas, models and translation rules. Instead, they focus on flexible mapping of inheritance hierarchies and the incremental regeneration of mappings after the source schema is modified. They also propose view generation and so instance translation.

Bowers and Delcambre [BD03] present Uni-Level Description (UDL) as a metamodel in which models and translations can be described and managed, with a uniform treatment of models, schemas, and instances. They use it to express specific model-to-model translations of both schemas and instances. Like our approach, their rules are expressed in Datalog. Unlike ours, they are expressed for particular pairs of models.

Other approaches to schema translation based on some form of metamodel, thus sharing features with ours, were proposed by Hainaut [Hai96, Hai06] and Boyd, Poulovassilis and McBrien [BM05, MP99, PM98].

Data exchange is a different but related problem, the development of user-defined custom translations from a given source schema to a given target one. It is an old database problem, going back at least to the 1970's [SHT+77], that received constant attention during the last decade (Cluet et al. [CDSS98], Milo and Zohar [MZ98], Popa et al. [PVM+02], and Fagin et al. [FKP05, FKMP03]) and is still relevant (Gottlob and Nash [GN08] and Libkin and Sirangelo [LS08]).

2.3 Models, Schemas, and the Dictionary

Description of Models

As we observed in the previous section, the starting point of our approach is the idea that a *metamodel* is a set of constructs (called *metaconstructs*) that can be used to define models, which are instances of the metamodel. Therefore, we actually define a model as a set of constructs, each of which corresponds to a metaconstruct. An even more important notion, also mentioned in the previous section, is that of *supermodel*: it is a model that has a construct for each metaconstruct, in the most general version. Therefore, each model can be seen as a specialization of the supermodel, except for renaming of constructs.

A conceptual view of the essentials of this idea is shown in Figure 2.4: the supermodel portion is predefined, but can be extended (and we will present our recent extension later in this chapter), whereas models are defined by specifying their respective constructs, each of which refers to a construct of the supermodel (SM-Construct) and so to a metaconstruct. It is important to observe that our

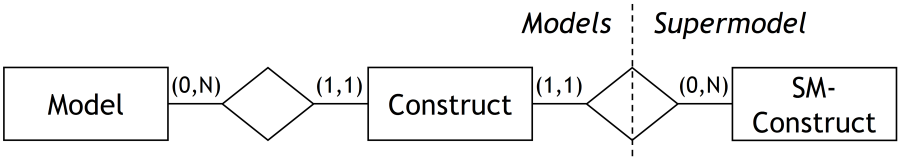


Figure 2.4: A simplified conceptual view of models and constructs.

approach is independent of the specific supermodel that is adopted, as new metaconstructs and so SM-Constructs can be added. This allows us to show simplified examples for the set of constructs, without losing the generality of the approach.

In order to make things concrete and to comment on some details, we show in Figures 2.5 and 2.6 the relational implementation of a portion of the dictionary, as we defined it in our tool. The actual implementation has more tables and more columns for each of them. We concentrate on the principles and omit marginal details.

The SM-CONSTRUCT table shows (a subset of) the generic constructs of the supermodel (which correspond to the metaconstructs) we have: ABSTRACT, ATTRIBUTEOFABSTRACT, BINARYAGGREGATIONOFABSTRACTS, and so on; these are the categories according to which the constructs of interest can be classified. Each construct in the CONSTRUCT table refers to an SM-Construct (by means of the SM-Construct column) and to a model (by means of the Model column). For example, the first row in CONSTRUCT has value "*mc1*" for the SM-Construct column in order to specify that "*Entity*" is a construct (of the "*ER*" model, as indicated by value "*m1*" in the Model column) that refers to the "*Abstract*" SM-Construct. It is worth noting (fourth row of the same table) that "*Class*" is a construct belonging to another model ("*OODB*") but referring to the same SM-Construct.

Tables SM-PROPERTY and SM-REFERENCE describe, at the supermodel level, the main features of constructs, properties and relationships among them. We discuss each of them in turn. Each SM-Construct has some associated properties, described by SM-PROPERTY, which will then require values for each instance of each construct corresponding to it. For example, the first row of SM-PROPERTY tells us that each "*Abstract*" (construct "*mc1*") has an "*Abstract-Name*", whereas the third says that for each "*AttributeOfAbstract*" ("*mc2*") we can specify whether it is part of the identifier of the "*Abstract*" or not (property "*IsIdentifier*"). Correspondingly, at the model level, we have

SM-Construct

OID	SM-Construct-Name	isLexical
mc1	Abstract	false
mc2	AttributeOfAbstract	true
mc3	BinaryAggregationOfAbstracts	false
mc4	AbtsractAttribute	false
...	...	...

SM-Property

OID	SM-Property-Name	SM-Construct	Type
mp1	AbstractName	mc1	string
mp2	AttributeName	mc2	string
mp3	isIdentifier	mc2	boolean
mp4	isFunctional1	mc3	boolean
mp5	isFunctional2	mc3	boolean
...	...	...	...

SM-Reference

OID	SM-Reference-Name	SM-Construct	SM-ConstructTo
mr1	Abstract	mc2	mc1
mr2	Abstract1	mc3	mc1
mr3	Abstract2	mc3	mc1
...	...	...	...

Figure 2.5: The relational implementation of a portion of the supermodel part of the dictionary.

that each "*Entity*" has a name (first row in table PROPERTY) and that for each "*AttributeOfEntity*" we can tell whether it is part of the key (third row in PROPERTY). In the latter case the property has a different name ("*IsKey*" rather than "*IsIdentifier*"). It is worth observing that "*Class*" and "*Field*" have the same features as "*Entity*" and "*AttributeOfEntity*", respectively, because they correspond to the same pair of SM-Constructs, namely "*Abstract*" and "*AttributeOfAbstract*". Other interesting properties are specified in the fourth and fifth rows of SM-PROPERTY. They allow for the specification of cardinalities of binary aggregations by saying whether the participation of an abstract is "functional" or not: a many-to-many relationship will have two *false* values, a one-to-one two *true* ones, and a one-to-many one *true* and one *false*.

Model

OID	ModelName
m1	ER
m2	OODB
...	...

Construct

OID	ConstructName	Model	SM-Construct
co1	Entity	m1	mc1
co2	AttributeOfEntity	m1	mc2
co3	BinaryRelationship	m1	mc3
co4	Class	m2	mc1
co5	Field	m2	mc2
co6	ReferenceField	m2	mc4
...	...	...	...

Property

OID	PropertyName	Construct	SM-Property
pr1	EntityName	co1	mp1
pr2	AttributeName	co2	mp2
pr3	isKey	co2	mp3
pr4	isFunctional1	co3	mp4
pr5	isFunctional2	co3	mp5
...	...	...	...
pr7	ClassName	co4	mp1
pr8	FieldName	co5	mp2
pr9	isId	co5	mp3
...	...	...	...

Reference

OID	ReferenceName	Construct	ConstructTo	SM-Reference
ref1	Entity	co2	co1	mr1
ref2	Entity1	co3	co1	mr2
ref3	Entity2	co3	co1	mr3
...	...	...	...	...
ref6	Class	co5	co4	mr1
...	...	...	...	...

Figure 2.6: The relational implementation of a portion of the models part of the dictionary.

Table SM-Reference describes how SM-Constructs are related to one another by specifying the references between them. For example, the first row of SM-Reference says that each "*AttributeOfAbstract*" (construct "*mc2*") refers to an "*Abstract*" ("*mc1*"); this reference is named "*Abstract*" because its value for each attribute will be the identifier of the abstract it belongs to. Again, we have the information repeated at the model level as well: the first row in table Reference specifies that "*AttributeOfEntity*" (construct "*co2*", corresponding to the "*AttributeOfAbstract*" SM-Construct) has a reference to "*Entity*" ("*co1*"). The same holds for "*Class*" and "*Field*", again, as they have the same respective SM-Constructs. The second and third rows of SM-Reference describe the fact that each binary aggregation of abstracts involves two abstracts and the homologous happens for binary relationships in the second and third rows of Reference.

The close correspondence between the two parts of our dictionary is a consequence of the way it is managed. The supermodel part (Figure 2.5) is its core; it is predefined (but can be extended) and it is used as the basis for the definition of specific models. Essentially, the dictionary is initialized with the available SM-Constructs, together with their properties and references. Initially, the model-specific part of the dictionary is empty and then individual models can be defined by specifying the constructs they include by referring to the SM-Constructs. In this way, the model part (Figure 2.6) is populated with rows that correspond to those in the supermodel part, except for the specific names, such as "*Entity*" or "*AttributeOfEntity*", which are model specific names for the SM-Constructs "*Abstract*" and "*AttributeOfAbstract*", respectively. This structure causes some redundancy between the two portions of the dictionary, but this is not a great problem, as the model part is generated automatically: the definition of a model can be seen as a list of supermodel constructs, each with a specific name.

An additional feature we have is the possibility of specifying conditions on the properties for a construct, in order to put restrictions on a model. For example, to define an object-oriented model that does not allow the specification of identifying fields, we could add a condition that says that the property "*IsId*" associated with "*Field*" is identically "*false*". These restrictions can be expressed as propositional formulas over the properties of constructs. We will make this observation more precise in Chapter 3.

Description of Schemas

The model portion of our dictionary is a metadictionary in the sense that it can be the basis for the description of model-specific dictionaries, with one table for each construct, with their respective properties and references. For example, the schema of a dictionary for the binary ER model mentioned above includes tables ENTITY, ATTRIBUTEOFENTITY, and BINARYRELATIONSHIP, as shown in Figure 2.7. The content of the dictionary describes two schemas, shown in Figure 2.8. The dictionary for a model has a structure that can be automatically generated once the model is defined. At the initialization of the tool, this portion of the dictionary is empty, and suitable tables are created whenever a model is defined.

Figure 2.9 shows a similar dictionary, for an object-oriented data model, with very simple constructs, namely class, field (as discussed above), and reference field. The content of the dictionary describes a schema, shown in Figure 2.10.

In the same way as the supermodel gives a unified view of all the constructs of interest, it is useful to have an integrated view of the schemas in the various models, describing them in terms of their SM-Constructs rather than constructs as we have done so far. As we anticipated in Section 2.1, this gives great benefits to the translation process, allowing an uniform definition of transformations. The supermodel portion of our dictionary has tables whose names are those of the SM-Constructs and whose columns correspond to their properties and references. We show an excerpt of the dictionary in Figure 2.11.

Its content is obtained by putting together the information contained in the model-specific dictionaries. For example, Figure 2.11 shows the portion of the supermodel that suffices for the descriptions of schemas of ER and OO models shown in Figures 2.7 and 2.9. In particular, the table ABSTRACT is the union (modulo suitable renaming) of tables ENTITY and CLASS in Figures 2.7 and 2.9, respectively[3]. Similarly, ATTRIBUTEOFABSTRACT is the union of tables ATTRIBUTEOFENTITY and FIELD.

We summarize our approach to the description of schemas and models by means of Figure 2.12. We have a dictionary composed of four parts, with two coordinates: schemas (lower portion) vs. models (upper portion) and model-specific (left portion) vs. supermodel (right portion).

[3]In the figures, for the sake of understandability, we have used, for each construct and schema, the same identifier in the supermodel dictionary and in the model specific one. So, for example "*s1*" is used both for a schema in the ER model and for the corresponding one in the supermodel.

Schema

OID	SchemaName
s1	SchemaER1
s2	SchemaER2

Entity

OID	EntityName	Schema
e1	School	s1
e2	Professor	s1
e3	Course	s1
e4	Employee	s2
e5	Project	s2

AttributeOfEntity

OID	Entity	AttributeName	isKey	Schema
a1	e1	Code	true	s1
a2	e1	SName	false	s1
a3	e1	City	false	s1
a4	e2	SSN	true	s1
a5	e2	Name	false	s1
a6	e3	CN	true	s1
a7	e3	Title	false	s1
a8	e4	EN	true	s2
...	...	...	...	...

BinaryRelationship

OID	RelationshipName	Entity1	isOpt1	isFunct1	Entity2	...	Schema
b1	Membership	e2	false	true	e1	...	s1
b2	Teaching	e3	true	true	e2	...	s1
b3	Participation	e4	false	false	e5	...	s2

Figure 2.7: The dictionary for a simple entity-relationship model.

Generality of the Approach

The above discussion suggests that it is indeed possible to describe many models and variations thereof by means of just a few more constructs. In fact our first contribution is represented by an extension of the previous supermodel, that is now capable of modeling essential concepts to represent recent complex data models like the object-relational and the XSD. Examples of the newly introduced concepts are nesting relationships, complex structured elements, collections, and substructures.

In the current version of our supermodel we have nine main constructs, the four shown in Figure 2.11 (with AttributeOfAbstract replaced by the

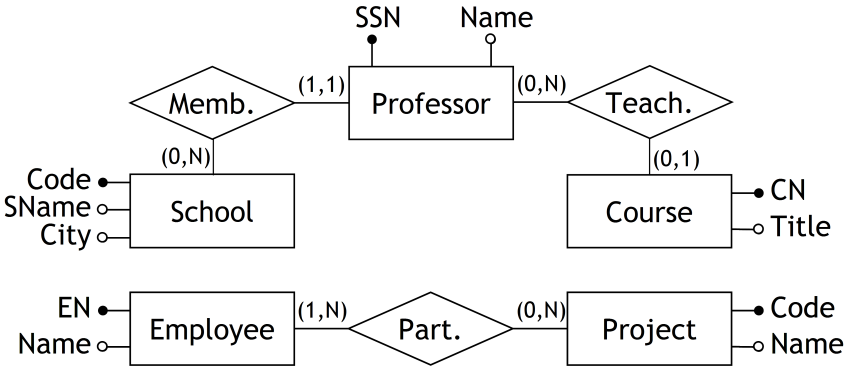


Figure 2.8: The entity-relationship schemas described in Figure 2.7.

Schema

OID	SchemaName
s3	SchemaOO1

Class

OID	ClassName	Schema
c1	School	s3
c2	Professor	s3
c3	Course	s3

Field

OID	Class	FieldName	isId	Schema
f1	c1	Code	true	s3
f2	c1	SName	false	s3
f3	c1	City	false	s3
f4	c2	SSN	true	s3
...	...	...	...	...

ReferenceField

OID	ReferenceName	Class	ClassTo	isOptional	Schema
r1	Membership	c2	c1	false	s3
r2	Teaching	c3	c2	true	s3

Figure 2.9: The dictionary for a simple object-oriented model.

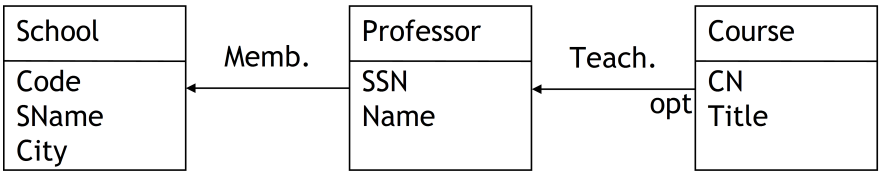


Figure 2.10: The object-oriented schema described in Figure 2.9.

more general LEXICAL, used for all value-based simple constructs) plus additional ones for representing n-ary aggregations of abstracts, generalizations, aggregations of lexicals, foreign keys, and structured (and possibly nested and/or multivalued) attributes. We said "main" constructs, since we have several specifications for them, each one with a particular set of attributes and references. For example, we have many "specific" value-based simple constructs (e.g. one referring to abstract, one to aggregation, and one to binary aggregation of abstracts), but here, in order to present the idea without details, we can represent them by means of a single more general construct (e.g. LEXICAL). We devote the next section to a complete discussion about the supermodel and the specific data models. The relational implementation has a few additional tables, as some constructs require two tables, because of normalization. For example, n-ary aggregations require two tables, one for the aggregations and one for the components of each of them.

We summarize constructs and (families of) models in Figure 2.13, where we show a matrix, whose rows correspond to the constructs and columns to the families we have experimented with. In the cells, we use the specific construct name for the family (for example, Abstract is called Entity in the ER model). The various models within a family differ from one another (i) on the basis of the presence or absence of specific constructs and (ii) on the basis of details of (constraints on) them. To give an example for (i) let us recall that versions of the ER model could have generalizations, or not, and the OR model could have structured columns or just simple ones. For (ii) we can just mention again the various restrictions on relationships in the binary ER model (general vs. one-to-many), which can be specified by means of constraints on the properties. It is also worth mentioning that a given construct can be used in different ways (again, on the basis of conditions on the properties) in different families: for example, a structured attribute could be multivalued, or not, on the basis of the value of a property isSet.

Schema

OID	SchemaName	Model
s1	SchemaER1	m1
s2	SchemaER2	m1
s3	SchemaOO1	m2

Abstract

OID	AbstractName	Schema
e1	School	s1
e2	Professor	s1
e3	Course	s1
e4	Employee	s2
e5	Project	s2
c1	School	s3
...	...	...

AttributeOfAbstract

OID	Abstract	AttributeName	isIdentifier	Schema
a1	e1	Code	true	s1
a2	e1	SName	false	s1
a3	e1	City	false	s1
a4	e2	SSN	true	s1
a5	e2	Name	false	s1
...	...	...	...	...
a8	e4	EN	true	s2
...	...	...	...	...
f1	c1	Code	true	s3
...	...	...	...	...

AbstractAttribute

OID	AttributeName	Class	ClassTo	isOptional	Schema
r1	Membership	c2	c1	false	s3
r2	Teaching	c3	c2	true	s3

BinaryAggregationOfAbstracts

OID	AggregationName	Abstract1	isOpt1	isFunct1	Abstract2	...	Schema
b1	Membership	e2	false	true	e1	...	s1
b2	Teaching	e3	true	true	e2	...	s1
b3	Participation	e4	false	false	e5	...	s2

Figure 2.11: A model-generic dictionary, based on the supermodel.

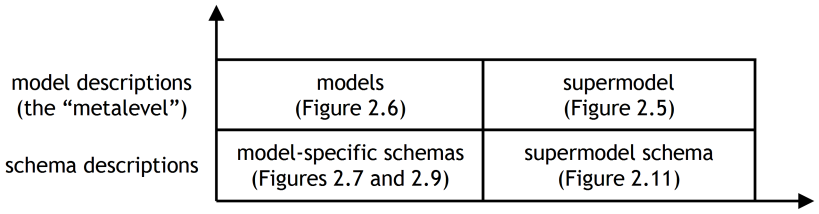


Figure 2.12: The four parts of the dictionary.

An interesting issue to consider here is "How universal is this approach?" or, in other words, "How can we guarantee that we can deal with every possible model?". A major point is that the metamodel is extensible, which is both a weakness and a strength. It is a weakness because it confirms that it is impossible to say you have a complete metamodel. However, it is a strength because it allows the addition of features when needed. This applies both to the details of the models of interest and to the families of models. With respect to the first issue, let us give an example: if one wants to handle XSD in full detail, then the metamodel and the supermodel need to be complex at least as the XSD language is. In fact, the level of detail can vary greatly and it can be chosen on the basis of the context of interest. With respect to the second issue it is worth mentioning that the approach can be used to handle metamodels in other contexts, with the same techniques. Indeed, we have noticed preliminary experiences with semantic Web models [AD06], with the management of annotations [PA07], and with adaptive systems [DT06]: for each of them, we defined a new set of constructs (and so different metamodel and supermodel) and new basic translations, but we used the same framework and the same engine.

In summary, the point is that the approach is independent of the specific supermodel. The supermodel we have mainly experimented with so far is a supermodel for database models and covers a reasonable family of them. If models were more detailed (as is the case for a fully-fledged XSD model) then the supermodel would be more complex. Moreover, other supermodels can be used in different contexts.

	Entity-Relationship	Binary Entity-Relationship	Object (UML Class Diagram)	Object-Relational	Relational	XSD
Abstract	Entity	Entity	Class	TypedTable		Root-Element
Lexical	Attribute	Attribute	Field	Column	Column	Simple-Element
Binary Aggregation OfAbstracts		Binary-Relationship				
Abstract Attribute			ReferenceField	Reference		
Aggregation OfAbstracts	Relationship					
Generalization	Generalization	Generalization	Generalization	Generalization		
Aggregation				Table	Table	
ForeignKey				ForeignKey	ForeignKey	ForeignKey
Structure OfAbstracts			Structured-Field	Structured-Field		Complex-Element

Figure 2.13: Constructs and models.

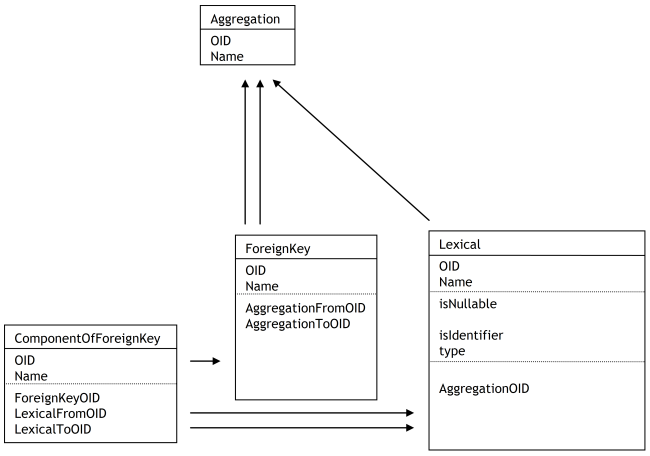


Figure 2.14: The relational model.

2.4 Supermodel Explained

In this section we show our representations of (families of) data models by means of constructs, then we "merge" them all together and present our complete supermodel.

Relational

We consider a relational model with tables constituted by columns of a specified type; each column could allow null value or be part of the primary key of the table. Moreover we can specify foreign keys between tables involving one or more columns.

The Figure 2.14 shows an UML-style diagram of the constructs allowed in the relational model with the following correspondences:

Table - AGGREGATION.

Column - LEXICAL. We can specify the data type of the column (type) and whether it is part of the primary key (isIdentifier) or it allows null value (isNullable). It has a reference toward an AGGREGATION.

Foreign Key - FOREIGNKEY and COMPONENTOFFOREIGNKEY. With the first construct (referencing two AGGREGATIONs) we specify the existence

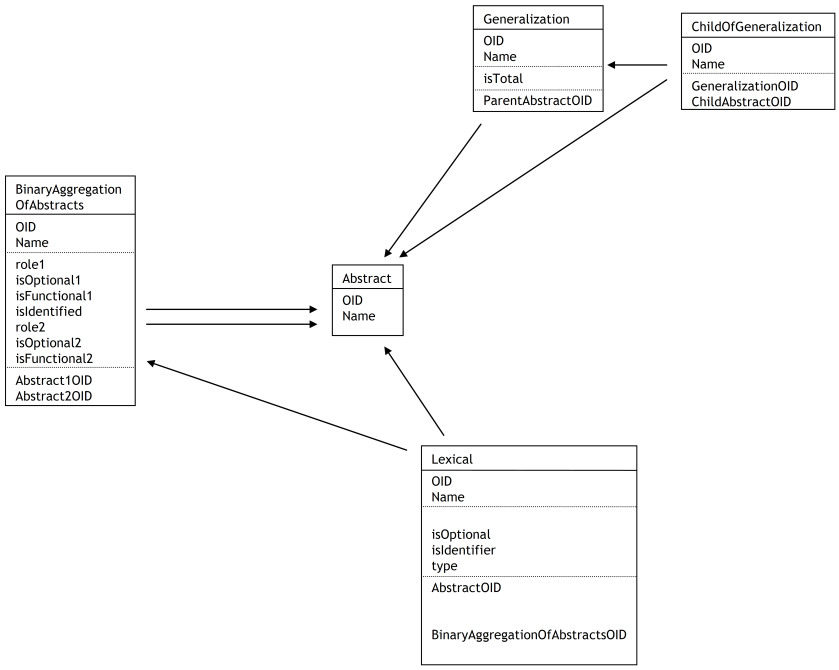


Figure 2.15: The binary entity-relationship model.

of a foreign key between two tables; with the second construct (referencing one ForeignKey and two Lexicals) we specify the columns involved in a foreign key.

Binary entity-relationship

We consider an entity-relationship model with entities and binary relationships together with their attributes and generalizations (total or not); each attribute could be optional or part of the identifier of an entity; for each relationship we specify minimum and maximum cardinality and whether an entity is externally identified by it.

The Figure 2.15 shows an UML-style diagram of the constructs allowed in the model with the following correspondences:

Entity - Abstract.

Attribute of Entity - LEXICAL. We can specify the data type of the attribute (type) and whether it is part of the identifier (isIdentifier) or it is optional (isOptional). It refers to an ABSTRACT.

Relationship - BINARYAGGREGATIONOFABSTRACTS. We can specify minimum (0 or 1 with the property isOptional) and maximum (1 or N with the property isFunctional) cardinality of the involved entities (referenced by the construct). Moreover we can specify the role (role) of the involved entities and whether the first entity is externally identified by the relationship (IsIdentified).

Attribute of Relationship - LEXICAL. We can specify the data type of the attribute (type) and whether it is optional (isOptional). It refers to a BINARYAGGREGATIONOFABSTRACTS.

Generalization - GENERALIZATION and CHILDOFGENERALIZATION. With the first construct (referencing an ABSTRACT) we specify the existence of a generalization rooted in the referenced Entity; with the second construct (referencing one GENERALIZATION and one ABSTRACT) we specify the children of the generalization. We can specify whether the generalization is total or not (isTotal).

We want to remark that we used in this presentation the "main" construct LEXICAL with two mutually exclusive references, instead of two specific construct, each one with a mandatory reference toward ABSTRACT and BINARYAGGREGATIONOFABSTRACTS, respectively.

N-ary entity-relationship

We consider an entity-relationship model with the same features of the aforementioned binary ER except for n-ary relationships instead of binary ones.

The Figure 2.16 shows an UML-style diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

Entity - ABSTRACT.

Attribute of Entity - LEXICAL.

Relationship - AGGREGATIONOFABSTRACTS and COMPONENTOFAGGREGATIONOFABSTRACTS. With the first construct we specify the existence

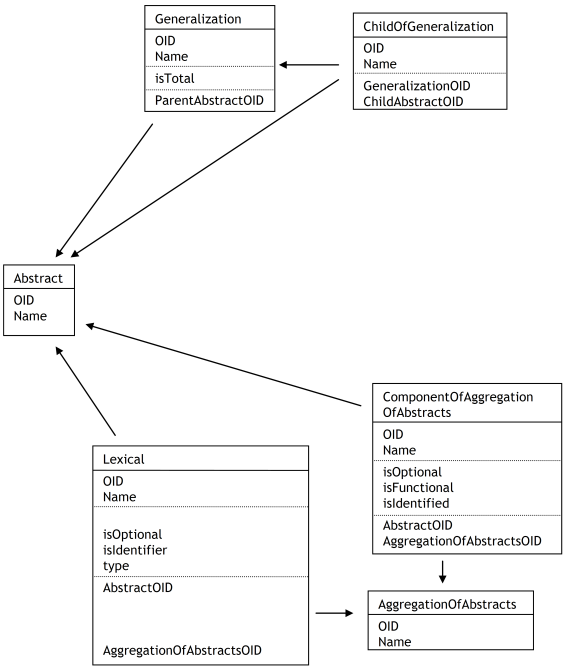


Figure 2.16: The n-ary entity-relationship model.

of a relationship; with the second construct (referencing an AggregationOfAbstracts and an Abstract) we specify the entities involved in such relationship. We can specify minimum and maximum cardinality of the involved entities (0 or 1 with the property isOptional and 1 or N with the property isFunctional, respectively). Moreover we can specify whether an entity is externally identified by the relationship (IsIdentified).

Attribute of Relationship - Lexical. It refers to an AggregationOfAbstracts.

Generalization - Generalization and ChildOfGeneralization.

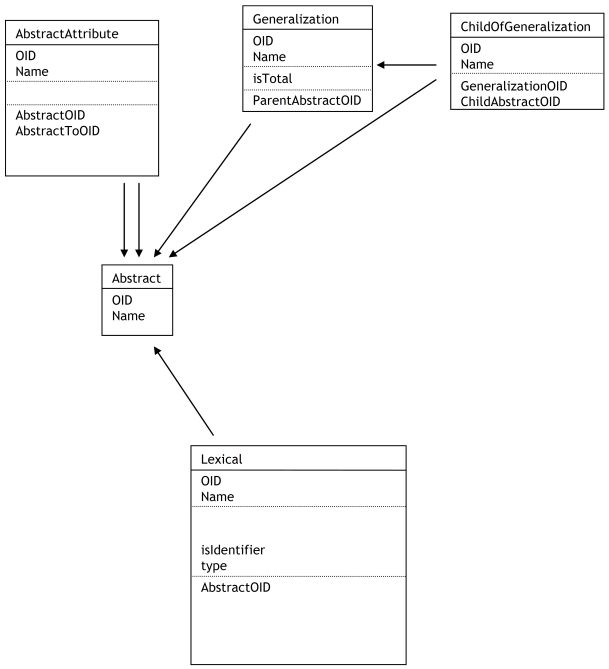


Figure 2.17: The object-oriented model.

Object-oriented

We consider an object-oriented model with classes, simple fields, and reference fields. We can also specify generalizations of classes.

The Figure 2.17 shows an UML-style diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

Class - ABSTRACT.

Field - LEXICAL.

Reference Field - ABSTRACTATTRIBUTE. It has two references toward the referencing ABSTRACT and the referenced one.

Generalization - GENERALIZATION and CHILDOFGENERALIZATION.

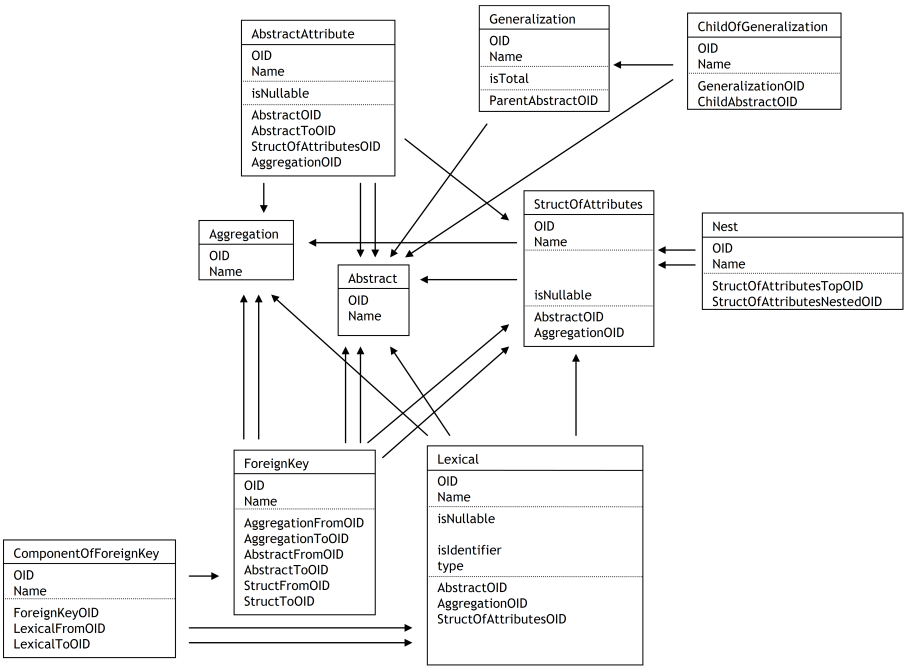


Figure 2.18: The object-relational model.

Object-relational

We consider a simplified version of the object-relational model. We merge the constructs of our relational and object-oriented model, where we have typed-tables rather than classes. Moreover we consider structured columns of tables (typed or not) that can be nested. Reference columns must be toward a typed table but can be part of a table (typed or not) or of a structured column. Foreign keys can involve also typed tables and structured columns. Finally, we can specify generalizations that can only involve typed tables.

The Figure 2.18 shows an UML-style diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

Table - AGGREGATION.

Typed Table - Abstract.

Structured Column - StructOfAttributes and Nest. The structured column, represented by a StructOfAttributes can allow null values or not (isNullable) and can be part of a simple table or of a typed table (this is specified by its references toward Abstract and Aggregation). We can specify nesting relationships between structured columns by means of Nest, that has two references toward the top StructOfAttributes and the nested one.

Column - Lexical. It can be part of (i.e. refer to) a simple table, a typed table or a structured column.

Reference Column - AbstractAttribute. It may be part of a table (typed or not) and of a structured column (specified by a reference) and must refer to a typed table (i.e. it has a reference toward an Abstract).

Foreign Key - ForeignKey and ComponentOfForeignKey. With the first construct (referencing two tables, typed or not, and a structured column) we specify the existence of a foreign key between tables (typed or not) and structured columns; with the second construct (referencing one ForeignKey and two Lexicals) we specify the columns involved in a foreign key.

Generalization - Generalization and ChildOfGeneralization.

Again we remark the use of "main" constructs (StructOfAttributes, Nest, Lexical, AbstractAttribute, ForeignKey, and ComponentOfForeignKey) in this presentation rather than consider a specific one for each allowed combination of references.

XSD

We consider a simplified version of the XSD language. We are only interested in XML documents that can be used to store large amount of data. Indeed we consider XSD documents with at least one top element unbounded (i.e. with *maxOccurs = "unbounded"* according to the syntax and the nomenclature of XSD). Then we deal with elements that can be simple or complex (i.e. structured). For these elements we can specify whether they are optional or whether they can be null (i.e. *nillable* according to the syntax and the nomenclature of XSD). Simple elements could be part of the key of the element they belong to

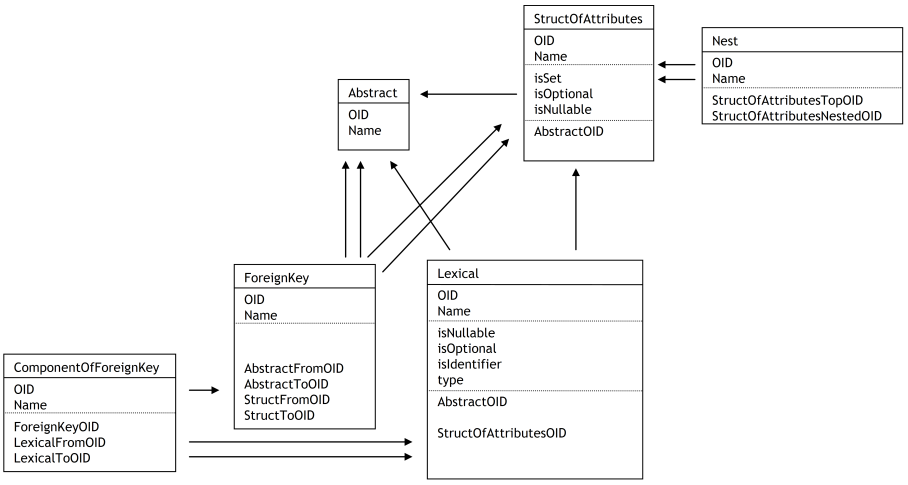


Figure 2.19: The XSD language.

and has an associated type. Moreover we allow the definition of foreign keys (i.e. *key* and *keyref* according to the syntax and the nomenclature of XSD).

The Figure 2.19 shows an UML-style diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

Root Element - Abstract.

Complex Element - StructOfAttributes and Nest. The first construct represents structured elements that can be unbounded or not (isSet), can allow null values or not (isNullable) and can be optional (isOptional). We can specify nesting relationships between complex elements by means of Nest, that has two references toward the top StructOfAttributes and the nested one.

Simple Element - Lexical. It can be part of (i.e. refer to) a root element or a complex one.

Foreign Key - ForeignKey and ComponentOfForeignKey.

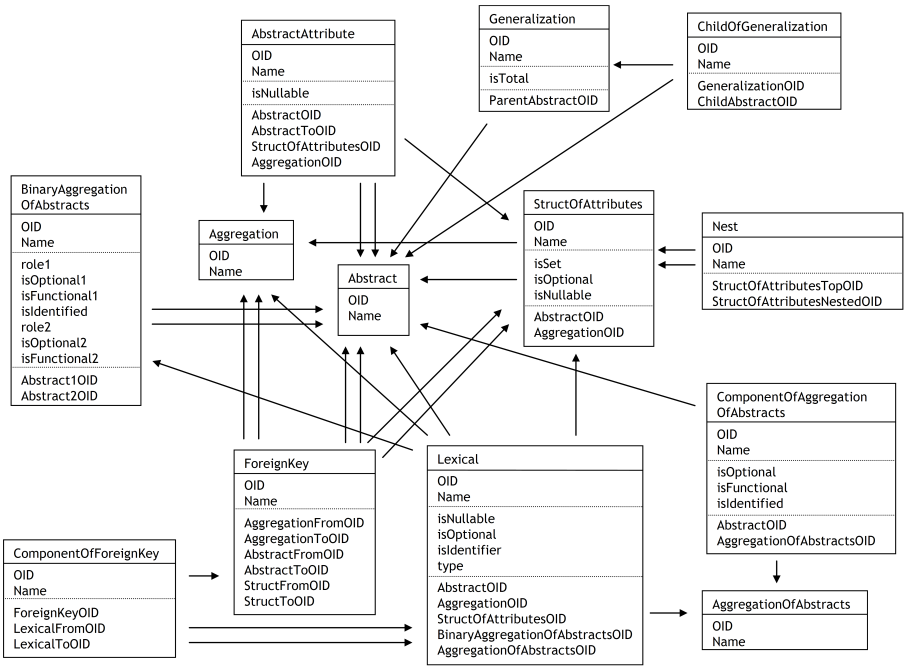


Figure 2.20: The Supermodel.

The Supermodel

The Figure 2.20 shows an UML-style diagram of the constructs allowed in the supermodel, obtained as the merge of the constructs of the previous models.

2.5 The Translations

Section 2.1 gave a general idea of the specification of translations in our proposal as a composition of basic steps. In this section we give some more details on basic translations and their implementation in Datalog. We close the section with a discussion about some issues related to the specification of complex translations.

Basic Translations

We have already shown the sequence of rules useful to eliminate many-to-many relationships in an ER model. We recall that, in our programs, only a few of their rules concern real translations, whereas most of them just copy constructs from the source schema to the target one. In our example the rules concerning transformation are those involving many-to-many relationships. The other relationships (i.e. one-to-one and one-to-many) together with the other constructs need just to be copied.

Let us now go into more detail about our Datalog rules, with reference to three of those involved in the aforementioned transformation. As an example of a copy rule we show rule $R_{2,1}$ that copies entities. The sample rules performing translations are rules $R_{2,5}$ and $R_{2,6}$. The first replaces a many-to-many relationship with a new entity; the second adds a new relationship in order to suitably "connect" the added entity with one of the entities originally involved in the many-to-many relationship (we recall that there is a "twin" rule to connect the other entity).

$R_{2,1}$:
ABSTRACT (OID: #abstract_0(absOID),
sOID: tgt,
Name: n)
←
ABSTRACT (OID: absOID,
sOID: src,
Name: n)

$R_{2,5}$:
ABSTRACT (OID: #abstract_1(aggOid),
sOID: tgt,
Name: n)
←
BINARYAGGREGATIONOFABSTRACTS(
OID: aggOid,
sOID: src,
Name: n,
isFunctional1: `false`,
isFunctional2: `false`)

$R_{2,6}$:
BinaryAggregationOfAbstracts(
OID: #binaryAggregationOfAbstracts_1(absOid,aggOid),
sOID: tgt,
Name: absName+aggName,
Abstract1: #abstract_1(aggOid),
isOptional1: `false`,
isFunctional1: `true`,
isIdentified: `true`,
Abstract2: #abstract_0(absOid),
isOptional2: isOpt,
isFunctional2: `false`)
←
BinaryAggregationOfAbstracts(
OID: aggOid,
sOID: src,
Name: aggName,
Abstract1: absOid,
isOptional1: isOpt,
isFunctional1: `false`,
isFunctional2: `false`),
Abstract (OID: absOID,
sOID: src,
Name: absName)

We first comment on our syntax. We use a non-positional notation for rules, so we indicate the names of the fields and omit those that are not needed (rather than using anonymous variables). Our rules generate constructs for a target schema (tgt) from those in a source schema (src)[4]. We note that, in order to simplify the writing of rules, it is possible to refer to a construct of the target schema also in the body of a rule (specifying that its sOID is equal to tgt). We may assume that variables tgt and src are bound to constants when the rule is executed. Each predicate has an OID argument, as we saw in the examples. When a construct is produced by a rule, it has to have a "new" identifier. It is generated by means of a Skolem functor, denoted by the # sign in the rule.

[4] We can also use temporary literals in the rule's head, in order to perform a simple selection of constructs for convenience, but it is just a technicality and we do not consider it in this dissertation.

We have the following restrictions on our rules. First, we have the standard "safety" requirements [UW97]: the literal in the head must have all fields, and each of them with a constant or a variable that appears in the body (in a positive literal) or a Skolem term. Similarly, all Skolem terms in the head or in the body have arguments that are constants or variables that appear in the body. Moreover, our Datalog programs are assumed to be coherent with respect to referential constraints: if there is a rule that produces a construct C that refers to a construct C', then there is another rule that generates a suitable C' that guarantees the satisfaction of the constraint. In the examples, rule $R_{2,6}$ is acceptable because there are rule $R_{2,1}$, that copies abstracts, and rule $R_{2,5}$, that generates new abstracts, thus guaranteeing that references to abstracts by $R_{2,6}$ are not dangling.

The body of rule $R_{2,5}$ unifies only with binary aggregations that have `false` as a value for IsFunctional1 and IsFunctional2: this is the condition that holds for all many-to-many relationships. Also the body of rule $R_{2,6}$ unifies only with many-to-many relationships. For each of them it generates a binary aggregation between the abstract generated by rule $R_{2,5}$ and one of the abstract originally involved in the many-to-many aggregation; the abstract generated by rule $R_{2,5}$ is externally identified by such new aggregation (IsIdentified: `true`) and, obviously, has minimum and maximum cardinality fixed to one (IsOptional1: `false` and IsFunctional1: `true`); the abstract originally involved in the many-to-many aggregation participates with unbounded maximum cardinality (IsFunctional2: `false`) while the minimum cardinality depends on the corresponding value of the original many-to-many aggregation (this is specified by means of the repeated variable isOpt).

These rules (together with rule $R_{2,7}$) are designed to properly transform many-to-many relationships. Conversely, the basic translation $\mathbf{P}_3$ of the example of Section 2.1 is designed for models that do not have many-to-many relationships: it replaces aggregations with references between abstracts and this substitution is only feasible for one-to-many aggregations. If we apply this rule to a model with many-to-many relationships, without applying a step that removes them before (program $\mathbf{P}_2$ in the examples above and in Figure 2.3), then we would lose the information carried by those relationships. We will formalize this point later in Chapter 3 in such a way that we could say that step $\mathbf{P}_3$ ignores many-to-many relationships.

Let us comment more on the presented rules. Rule $R_{2,1}$ generates a new abstract (belonging to the target schema) for each abstract in the source schema. The Skolem functor #abstract_0 is responsible for the generation of a new identifier. Skolem functors produce injective functions, with the additional

constraint that different functions have disjoint ranges, so that a value is generated only by the same functor with the same argument values. For the sake of readability (and also for some implementation issues omitted here), we include the name of the target construct (abstract in this case) in the name of the functor, and use a suffix to distinguish the various functors associated with a construct. The _0 suffix always denotes the "copy" functor.

Rule $R_{2,6}$ replaces each binary many-to-many relationship (that is, in supermodel terminology, a BinaryAggregationOfAbstracts) with another relationship. The rule has a variety of Skolem functors. The head has three Skolem terms, which make use of three different functors. The first Skolem term (#binaryAggregationOfAbstracts_1) generates a new value for the construct being created, as we saw for the previous rule. Indeed, this is the case for all the Skolem functors appearing in the OID field of the head of a rule. The other two terms correlate the element being created with elements created by rules $R_{2,1}$ and $R_{2,5}$, respectively. In fact, we use the same Skolem terms (#abstract_0 and #abstract_1) previously used in those rules. The new BinaryAggregationOfAbstracts being generated involves the Abstract in the target schema, generated as a copy of the Abstract (denoted by variable absOid in the source schema and involved in a many-to-many aggregation), and the Abstract in the target schema, generated to properly "substitute" the source many-to-many BinaryAggregationOfAbstracts (denoted by aggOid in the source schema).

Our approach to rules allows for a lot of reusability, at various levels. First of all, we have already seen that individual basic translations can be used in different contexts; in the space of models in Figure 2.3, each translation can be used in many simple steps. For example, translation $\mathbf{P}_2$ can be used to eliminate many-to-many relationships in every variant of the entity-relationship model; with reference to Figure 2.3, it can be used to go from er1 to er4, from er3 to er6, or from er5 to er7. Moreover, translation $\mathbf{P}_3$ can be used to transform relationships into references, for going from different variants of the entity-relationship model to homologous variants of the object-oriented model; in Figure 2.3, it can be used to go from er6 to oo1 or from er7 to oo2.

Second, as each translation step is composed of a number of Datalog rules, some of which are just "copy" rules, they can be used in many basic translations. This is easily the case for plain copy rules, but can be applied also to "conditional" ones, that is, copy rules that are applied only to a subset of the constructs. For example, the rule that eliminates many-to-many relationships copies all the relationships that are not many-to-many; this can be done with a copy rule extended with an additional condition in the body.

In some cases, basic translations can be written with respect to a "core" set of Datalog rules, with copy rules added automatically, given the set of constructs in the supermodel. In this way, the approach would become partially independent of the current supermodel, especially with respect to its extensions. For example, in our case, we assumed that initially the supermodel does not include generalizations; in fact, in the specification of the basic translation $\mathbf{P}_2$, which eliminates many-to-many binary aggregations, there are no rules to deal with them. In this scenario, our basic translation $\mathbf{P}_2$ could be defined by means of rules $R_{2,5}$, $R_{2,6}$, and $R_{2,7}$, responsible of properly transforming many-to-many aggregations, with rule $R_{2,1}$, which copies abstracts, added automatically because of referential integrity constraint in the supermodel, and rule $R_{2,2}$, which copies attributes of abstract, added because attributes of abstract are "compatible" with abstracts. Then, if the supermodel were extended with generalizations, the basic translation would be extended with rules $R_{2,9}$ and $R_{2,10}$, which copy generalizations and child of generalizations, respectively.

Most of our rules, such two of those we saw (i.e. $R_{2,1}$ and $R_{2,6}$), are recursive according to the standard definition. However, recursion is only "apparent". The same literal occurs in both the head and the body, but the construct generated by an application of the rule belongs to the target schema, so the rule cannot be applied to it again, as the body refers to a construct of the source schema. A really recursive application may occur only for rules that have atoms that refer to the target schema also in their body. In the following, we will use the term *strongly recursive* for these rules.

In our experiments, we have developed a set of basic translations to handle the models that can be defined with our current metamodel. They are listed in Appendix Basic Translations. In the next chapter we will discuss arguments to confirm the adequacy of this set of rules.

Complex Translations

With many possible models and many basic translations, it becomes important to understand how to find a suitable transformation given a source and a target model. Intuitively, we could think of using a graph such as that in Figure 2.3, with models as nodes and basic translations as edges. Here there are two difficulties. The first one is how to verify what target model is generated by applying a basic translation to a source model. For example, with reference to Section 2.1, to verify that transformation $\mathbf{P}_1$ indeed generates schemas of model ER3 from schemas of ER1 and that it generates schemas of ER5 from schemas of ER2). The second problem is related to the size of the graph:

due to the number of constructs and properties, we have too many models (a combinatorial explosion of them, if the variants of constructs grow) and it would be inefficient to find all associations between basic translations and pairs of models.

We propose a complete solution to the first issue, as follows. We associate a concise *description* with each model, by indicating the constructs it involves with the associated properties (described in terms of propositional formulas), and a *signature* with each basic translation. Then, a notion of *application* of a signature to a model description allows us to obtain the description of the target model. With our basic translations written in a Datalog dialect with OID-invention, as we will see shortly, it turns out that signatures can be automatically generated and the application of signature gives an exact description of the target model.

With respect to the second issue, the complexity of the problem cannot be completely circumvented, but we have devised algorithms that, under reasonable hypotheses, efficiently find a complex translation given a pair of models (source and target). So, for example, again with reference to Section 2.1, given ER2 and OO2, our algorithm properly finds the translation composed of steps $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$, out of a library of many basic translations.

We have devoted Chapter 3 to a detailed description of our solutions.

Chapter 3

A Formal System

3.1 Overview

In this chapter we give formal grounds to the notion of data model and to the management of translations of schemas with reference to our approach to this problem illustrated in Chapter 2. Such formal grounds are useful to answer questions like the following ones: given a set of basic translations, how do we build the actual translations we need? How do we verify that a given sequence of basic translations produces the model we are interested in? And that it does not "forget" any construct?

Let us give an idea of the result. First, we associate with each model a *description* that specifies its constructs. We introduce a similar notion for our Datalog programs, called *signature*: given a program, we can derive its signature, and we have defined the application of signatures of programs to descriptions of models. Then the result is that signatures completely describe the behavior of programs on models, in the sense that the application of signatures provides a "derivation" of models that is sound and complete with respect to the schemas generated by programs. Let us describe the main results with the help of Figure 3.1.

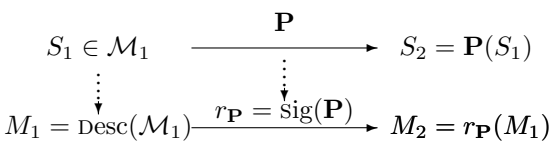


Figure 3.1: The formal system.

Let S_1 be a schema for a model $\mathcal{M}_1$, $\mathbf{P}$ a Datalog program implementing a basic translation, and S_2 the schema obtained by applying $\mathbf{P}$ to S_1. Then, let M_1 be the description of $\mathcal{M}_1$ and $r_{\mathbf{P}}$ the signature of $\mathbf{P}$, which corresponds to a function (the *application* of $r_{\mathbf{P}}$) from descriptions of models to descriptions of models. Soundness and completeness of signatures with respect to Datalog programs can be claimed as follows, respectively:

1. the application of $\mathbf{P}$ to a schema S_1 of $\mathcal{M}_1$ produces a schema $\mathbf{P}(S_1)$ that belongs to the model whose description $r_{\mathbf{P}}(M_1)$ is obtained by applying the signature $r_{\mathbf{P}}$ of program $\mathbf{P}$ to the description M_1 of $\mathcal{M}_1$;

2. among the possible schemas of $\mathcal{M}_1$, there exists a schema S^* such that the application of $\mathbf{P}$ to S^* produces a schema $\mathbf{P}(S^*)$ that (besides belonging to $r_{\mathbf{P}}(M_1)$, as stated by the previous claim) does not belong to any model that is "strictly more restricted" than $r_{\mathbf{P}}(M_1)$.

The two claims together say that the model we derive by means of the application of signatures is exactly the model that allows the set of schemas that can be obtained by means of the Datalog programs. Claim (1) says that the derived model is liberal enough (*soundness*) and claim (2) says that it is restricted enough (*completeness*).

This can be seen as a "syntactic" aspect of the correctness of rules to be complemented by a "semantic" one.

3.2 Related Work

To the best of our knowledge, there is no approach in the literature that tackles the problem we are considering here. There are pieces of work that consider the translation of schemas in heterogeneous frameworks [ACB06, AT96, BMM05, PT05], but none has techniques for inferring high level descriptions of translations from their specification. They all propose some way of generating a complex translation plan but they either handle very simple descriptions of models or have to rely on a hard coding of knowledge of behavior of transformations in terms of pattern of constructs removed and introduced. Bernstein et al. [BMM05] and Papotti and Torlone [PT05] use some form of signature to implement an algorithm based on a heuristic function (as A^* [DP85, RN03]) that produces the shortest transformation plan (if it exists) in terms of number of transformations between the source and the target model.

Translations of schemas by means of Datalog variants have been proposed by various authors [ACM02, BD03, DK97], but no explicit reference to models and to the possibility of reasoning on models has been proposed. The latter work includes some reasoning on constraints, but without reference to the features of models.

Various works exist on the correctness of transformations of schemas, with reference to the well known notion of information capacity dominance and equivalence [AH88, Hul86, MIR93]. Here we are not studying the correctness of the individual translation steps, but the correctness of complex translations, assumed that the elementary steps are correct, following an "axiomatic" approach [AT96].

3.3 Descriptions of Constructs and Models

In this section we formalize the description of models. We define models in terms of their descriptions, blurring the distinction between a model and its description, as descriptions are sufficient for the purpose of the formal system.

Let us consider a subset of the actual universe of constructs we will use in the running examples of this chapter to explain the basic idea. We recall that references are required to build schemas for meaningful models (for example, a relationship without references to entities would make no sense), while properties could be restricted in some way (for example, we can think of models where all cardinalities for relationships are allowed and models where many-to-many relationships are not allowed). Formally, given a **universe** of constructs, each with a set of associated properties:

$$\mathcal{U} = \{C_1(P_1), C_2(P_2), \ldots, C_u(P_u)\}$$

we can define **(the description of) a model** as a mapping that associates a proposition with each construct in the universe:

$$M = \{C_1(f_1), C_2(f_2), \ldots, C_u(f_u)\}$$

where an f_i is a proposition that can involve literals (possibly negated) corresponding to the properties of C_i. The values of the properties of a construct C_i of a schema in a model M, must satisfy the proposition f_i associated with C_i in M. If there are no constraints on the properties of a construct, then the corresponding proposition is *true*. If a construct is not allowed, then the corresponding proposition is *false*.

Construct	*References*	*Properties*	*Abbreviation*
ENTITY			E()
ATTRIBUTEOFENTITY	Entity	isKey isNullable	A(K, N)
RELATIONSHIP	Entity1 Entity2	isOptional isFunctional1 isIdentified isOptional2 isFunctional2	R(O_1, F_1, I, O_2, F_2)
ATTRIBUTEOF RELATIONSHIP	Relationship	isNullable	AR(N)
TABLE			T()
COLUMN	Table	isKey isNullable	C(K, N)

Figure 3.2: The universe of constructs for the examples.

The subset of interest is depicted in Figure 3.2, where in the last column we indicate the abbreviated form we will use in the following[1].

With this universe of constructs we can define, for example, the following set of models to be used in the discussion:

- $\mathcal{M}_{\text{REL}}$: a relational model, with tables and columns (and no restrictions);
- $\mathcal{M}_{\text{RELNON}}$: a relational model with no null values: all columns must have a value *false* for property isNullable;
- $\mathcal{M}_{\text{ER}}$: an ER model with all the available features;
- $\mathcal{M}_{\text{ERSIMPLE}}$: an ER model with no null values on attributes (all attributes have a value *false* for isNullable) and no attributes on relationships;
- $\mathcal{M}_{\text{ERNOM2N}}$: an ER model with no many-to-many relationships (all relationships have a value *true* for isFunct1 or isFunct2).

[1]For the sake of clarity, in this chapter, we use the specific names, such as "entity", instead of the generic ones, such as "abstract".

The corresponding descriptions of these models would be as follows:

- $M_{\text{REL}} = \{\text{E}(\mathit{false}), \text{A}(\mathit{false}), \text{R}(\mathit{false}), \text{AR}(\mathit{false}), \text{T}(\mathit{true}), \text{C}(\mathit{true})\}$
- $M_{\text{RELNON}} = \{\text{E}(\mathit{false}), \text{A}(\mathit{false}), \text{R}(\mathit{false}), \text{AR}(\mathit{false}), \text{T}(\mathit{true}), \text{C}(\neg\text{N})\}$
- $M_{\text{ER}} = \{\text{E}(\mathit{true}), \text{A}(\mathit{true}), \text{R}(\text{F}_1 \vee \neg\text{F}_2), \text{AR}(\mathit{true}), \text{T}(\mathit{false}), \text{C}(\mathit{false})\}$[2]
- $M_{\text{ERSIMPLE}} = \{\text{E}(\mathit{true}), \text{A}(\neg\text{N}), \text{R}(\text{F}_1 \vee \neg\text{F}_2), \text{AR}(\mathit{false}), \text{T}(\mathit{false}), \text{C}(\mathit{false})\}$
- $M_{\text{ERNOM2N}} = \{\text{E}(\mathit{true}), \text{A}(\mathit{true}), \text{R}(\text{F}_1), \text{AR}(\mathit{true}), \text{T}(\mathit{false}), \text{C}(\mathit{false})\}$

In the definition above, we have that all constructs are mentioned in every model, possibly with a *false* proposition (meaning that the construct does not belong to the model). In practice, to simplify the notation, we describe a model by listing only the constructs that really belong to it (i.e. those that have a satisfiable proposition); in this way, the descriptions would be as follows:

- $M_{\text{REL}} = \{\text{T}(\mathit{true}), \text{C}(\mathit{true})\}$
- $M_{\text{RELNON}} = \{\text{T}(\mathit{true}), \text{C}(\neg\text{N})\}$
- $M_{\text{ER}} = \{\text{E}(\mathit{true}), \text{A}(\mathit{true}), \text{R}(\text{F}_1 \vee \neg\text{F}_2), \text{AR}(\mathit{true})\}$
- $M_{\text{ERSIMPLE}} = \{\text{E}(\mathit{true}), \text{A}(\neg\text{N}), \text{R}(\text{F}_1 \vee \neg\text{F}_2)\}$
- $M_{\text{ERNOM2N}} = \{\text{E}(\mathit{true}), \text{A}(\mathit{true}), \text{R}(\text{F}_1), \text{AR}(\mathit{true})\}$

We can define a partial order on models, as follows:

$M_1 \sqsubseteq M_2$ (read M_1 is **more restricted** than M_2) if for every $C \in \mathcal{U}$ it is the case that $f_1 \wedge f_2$ is equivalent to f_1 (that is, f_1 implies f_2), where $C(f_1) \in M_1$ and $C(f_2) \in M_2$

It can be shown that $\sqsubseteq$ is a partial order (modulo equivalence of propositions), as it is reflexive, antisymmetric, and transitive.

If models are described only in terms of the constructs that have satisfiable properties, then the partial order can be rewritten as:

$M_1 \sqsubseteq M_2$ if for every $C(f_1) \in M_1$ there exist $C(f_2) \in M_2$ such that $f_1 \wedge f_2$ is equivalent to f_1

[2]Without loss of generality, we assume that in a one-to-many relationship, it is the first entity that has a functional role, and so $\text{F}_1 = \mathit{true}$ and $\text{F}_2 = \mathit{false}$.

In plain words, $M_1 \sqsubseteq M_2$ means that M_2 has at least the constructs of M_1 and, for those in M_1, it allows at least the same variants. For the example models:

- $M_{\text{RELNON}} \sqsubseteq M_{\text{REL}}$ (and $M_{\text{REL}} \not\sqsubseteq M_{\text{RELNON}}$): they have the same constructs, but M_{RELNON} has a more restrictive condition on construct C than M_{REL};
- $M_{\text{ERNOM2N}} \sqsubseteq M_{\text{ER}}$ (and $M_{\text{ER}} \not\sqsubseteq M_{\text{ERNOM2N}}$): they have the same constructs, but M_{ERNOM2N} has a more restrictive condition on construct R than M_{ER};
- $M_{\text{ERSIMPLE}} \sqsubseteq M_{\text{ER}}$ (and $M_{\text{ER}} \not\sqsubseteq M_{\text{ERSIMPLE}}$): the constructs in M_{ERSIMPLE} are a proper subset of those in M_{ER} and, for each of them, the condition in M_{ERSIMPLE} is at least as restrictive as the respective one in M_{ER};
- $M_{\text{ERNOM2N}} \not\sqsubseteq M_{\text{ERSIMPLE}}$, $M_{\text{ERSIMPLE}} \not\sqsubseteq M_{\text{ERNOM2N}}$: in fact M_{ERNOM2N} has AR which is not in M_{ERSIMPLE}, but has a more restrictive conditionon R.

We can define two binary operators on the space of models as follows:

$$\begin{aligned} M_1 \sqcup M_2 &= \{C(f_1 \vee f_2) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\} \\ M_1 \sqcap M_2 &= \{C(f_1 \wedge f_2) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\} \end{aligned}$$

If models are described only in terms of the constructs that have satisfiable properties, then the operators can be rewritten as:

$$\begin{aligned} M_1 \sqcup M_2 &= \{C(f_1) \mid C(f_1) \in M_1 \text{ and there is no } C(f_2) \in M_2\} \cup \\ &\quad \{C(f_2) \mid C(f_2) \in M_2 \text{ and there is no } C(f_1) \in M_1\} \cup \\ &\quad \{C(f_1 \vee f_2) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\} \\ M_1 \sqcap M_2 &= \{C(f_1 \wedge f_2) \mid C(f_1) \in M_1, C(f_2) \in M_2 \text{ and } f_1 \wedge f_2 \neq \mathit{false}\} \end{aligned}$$

It can be shown that the space of models forms a lattice with respect to these two operators (modulo equivalence of propositions). The proofs of the claims that guarantee the lattice structure follow the definitions and the fact that the boolean operators in propositional logic form a lattice. The supermodel (the fictitious most general model mentioned in Chapter 1) is the top element of the lattice (with the *true* proposition for every construct). It is worth noting that models obtained as the result of these operators, especially the $\sqcap$, could have, in some extreme cases, little practical meaning. For example, the bottom element

of the lattice is the (degenerate) empty model, which has the *false* proposition for every construct (or, in other words, no constructs).

We can also define a **difference** operator on models as:

$$M_2 - M_1 \quad = \quad \{C(f_2 \wedge \neg f_1) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\}$$

If models are described only in terms of the constructs that have satisfiable properties, then the difference operator can be rewritten as:

$$\begin{aligned} M_2 - M_1 \quad = \quad & \{C(f_2 \wedge \neg f_1) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\} \cup \\ & \{C(f_2) \mid C(f_2) \in M_2 \text{ and there is no } C(f_1) \in M_1\} \end{aligned}$$

This operator can also generate models with meaningless conditions, and indeed we will see in Section 3.6 that we use it only for technical steps during the search for translations.

3.4 Signatures of Rules and Their Application

In order to handle rules and to reason on them, in an effective way, we introduce the notion of *signature* of a Datalog rule. The definition gives a unique construction, so the signature can be automatically computed for each rule.

As a preliminary step, let us define the **description of an atom** in a Datalog rule. Given an atom $C(\text{ARGS})$, consider the fields in ARGS that correspond to properties (ignoring the others); let them be $p_1 : v_1, \ldots, p_k : v_k$; each v_i is either a variable or a boolean constant *true* or *false*. Then, the description of $C(\text{ARGS})$ is a construct description $C(f)$, where the proposition f is the conjunction of literals corresponding to the properties in $p_1, \ldots, p_k$ that are associated with a constant; each of them is positive if the constant is *true* and negated if it is *false*. If there are no constants, then the proposition is *true*.

Let us see an example, recalling one of the rules we have seen in detail in Chapter 2, modulo renaming of constructs. For the sake of clarity, we report here rule $R_{2,6}$ using the specific constructs names of the ER model.

$R_{2,6}$:
RELATIONSHIP (
OID: #relationship_1(eOid,relOid),
sOID: tgt,
Name: entityName+relName,
Entity1: #entity_1(relOid),
isOptional1: `false`,
isFunctional1: `true`,
isIdentified: `true`,
Entity2: #entity_0(eOid),
isOptional2: isOpt,
isFunctional2: `false`)
←
RELATIONSHIP (
OID: relOid,
sOID: src,
Name: relName,
Entity1: eOid,
isOptional1: isOpt,
isFunctional1: `false`,
isFunctional2: `false`),
ENTITY (OID: eOID,
sOID: src,
Name: entityName)

The description of the atom in the head of the rule is $\text{R}(\neg \text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg \text{F}_2)$ because the properties isFunctional1 and isIdentified are bounded to `true` and the properties isOptional1 and isFunctional2 are bounded to `false` in the atom. Analogously, the description of the RELATIONSHIP atom in the body of the rule is $\text{R}(\neg \text{F}_1 \wedge \neg \text{F}_2)$. Conversely, the description of the ENTITY atom in the body of the rule is $\text{E}(\mathit{true})$ because there are no properties bounded to constant in the atom.

Let us now define the **signature of a Datalog rule**. Let R be a rule, with a head $C(\text{ARGS})$ and a body with a list of atoms referring to constructs which need not be distinct $\langle C_{j_1}(\text{ARGS}_1), C_{j_2}(\text{ARGS}_2), \ldots, C_{j_h}(\text{ARGS}_h)\rangle$; comparison terms (with inequalities, according to our hypotheses) do not affect the signature, and so we can ignore them. The signature r_R of R is composed of three parts (B, H, MAP):

- B (the **body of** r_R) describes the applicability of the rule, by referring to the constructs in the body of R; B is a list of descriptions of atoms, $\langle C_{j_1}(f_1), C_{j_2}(f_2), \ldots, C_{j_h}(f_h)\rangle$, where $C_{j_i}(f_i)$ is the description of the atom $C_{j_i}(\text{ARGS}_i)$.

- H (the **head of** r_R) indicates the conditions that definitely hold on the result of the application of R, because of constants in its head; H is defined as the description $C(f)$ of the atom $C(\text{ARGS})$ in the head.

- MAP (the **mapping of** r_R) is a partial function that describes where values of properties in the head originate from. It is defined as follows. Its domain is the set of properties of the construct in the head; MAP is defined for the properties that are associated, in the head, with a variable. For our assumptions, each variable in the head appears also in the body, and only once. If a variable appears for a property p' in the head and for a property p of a construct C_{j_k} in the body, then MAP is defined on p' as $\text{MAP}(p') = C_{j_k}(p)$.

The body signature of rule $R_{2,6}$ is $B_{2,6} = \langle \text{R}(\neg \text{F}_1 \wedge \neg \text{F}_2), \text{E}(\textit{true})\rangle$; indeed, the rule is applicable only to many-to-many relationships, that is, if both F_1 and F_2 are *false*. Similarly, for $R_{2,3}$ we have $B_{2,3} = \langle \text{R}(\text{F}_1)\rangle$, as the rule copies one-to-many relationships.

The head signature of rule $R_{2,6}$ is $H_{2,6} = \text{R}(\neg \text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg \text{F}_2)$: all the relationships produced by the rule have O_1 and F_2 equal to *false* and F_1 and I equal to *true*.

The mapping for rule $R_{2,6}$ is $\text{MAP}_{2,6} = \langle \text{O}_2 : \text{R}(\text{O}_1)\rangle$ (we denote the function as a list of pairs, including only the properties on which it is defined). The name of the construct in the head is not mentioned, because it is known, but let us note that it might be different from the one in the body; this is the case for $R_{2,8}$ (in the Appendix Rules) where $\text{MAP} = \langle \text{N} : \text{AR}(\text{N})\rangle$ and the first N is a property of A, the construct in the head.

Before defining the *application* of the signature of a rule to a model, we need two preliminary notions. First, we say that the signature $r_R = (B, H, \text{MAP})$ of a rule R is **applicable** to a model M if, for each $C_{j_i}(f_i)$ in B, there is $C_{j_i}(f_{j_i}^M) \in M$ such that $f_{j_i}^M \wedge f_i$ is satisfiable. In plain words, each construct in the body has to appear in the model, and the two propositions must not contradict one another. For example, $R_{2,6}$ is not applicable to M_{ERNoM2N} because we have $\text{R}(\neg \text{F}_1 \wedge \neg \text{F}_2)$ in the body of the rule and $\text{R}(\text{F}_1)$ in the model: the conjunction of $\neg \text{F}_1 \wedge \neg \text{F}_2$ and F_1 is not satisfiable. Second, let us define the **transformation** μ_{MAP} induced by mapping MAP on literals. In plain words,

we use μ_{MAP} to "transfer" constraints on literals over properties in the body to literals over properties in the head according to the MAP of the rule. Let l be a literal for a property p of an atom $C_{j_i}(\ldots)$ in the body of a rule R. Then, if $C_{j_i}(p)$ belongs to the range of MAP, with $\text{MAP}(p') = C_{j_i}(p)$, we have that $\mu_{\text{MAP}}(l)$ is a literal for the property p' with the same sign as l; if $C_{j_i}(p)$ does not belong to the range of MAP, then $\mu_{\text{MAP}}(l) = \mathit{true}$. Let us define μ_{MAP} also on constants: $\mu_{\text{MAP}}(\mathit{true}) = \mathit{true}$ and $\mu_{\text{MAP}}(\mathit{false}) = \mathit{false}$. The notion can be extended to conjunctions and disjunctions of propositions as follows: (i) $\mu_{\text{MAP}}(f_1 \wedge f_2) = \mu_{\text{MAP}}(f_1) \wedge \mu_{\text{MAP}}(f_2)$ if $f_1 \wedge f_2$ is satisfiable, otherwise $\mu_{\text{MAP}}(f_1 \wedge f_2) = \mathit{false}$; (ii) $\mu_{\text{MAP}}(f_1 \vee f_2) = \mu_{\text{MAP}}(f_1) \vee \mu_{\text{MAP}}(f_2)$.

We are now ready for the definition of the **application** $r_R(M)$ of the signature of a rule R to a model M. In practice, such function has to combine constraints expressed in the head of the rule, with constraints of the source model that could be "transferred" to the output by means of repeated variables in the head and the body of the rule. If r_R is not applicable to M, then we define $r_R(M) = \{\}$. The interesting case is when r_R is applicable to M. Let the signatures of the body and of the head of R be $B = \langle C_{j_1}(f_1), C_{j_2}(f_2), \ldots, C_{j_h}(f_h)\rangle$ and $H = C(f)$, respectively. For every atom $C_{j_i}(f_i)$ in the body, let $f_{j_i}^M$ be the proposition associated with C_{j_i} in the model M.

Let us first give the definition in the special case where all the constructs in the source model M have propositions that are just conjunctions of literals. In this case:

$$r_R(M) = \{C(f')\} \text{ where } f' = f \wedge (\bigwedge_{i=1}^{h} \mu_{\text{MAP}}(f_{j_i}^M \wedge f_i))$$

Let us note that $(f_{j_i}^M \wedge f_i)$ is satisfiable, since the rule is applicable, and that it is just a conjunction of literals, because this is the case for f_i, by construction, and for $f_{j_i}^M$, by hypothesis. In plain words, the condition in the result is obtained as the conjunction of the proposition in the head, f, with those obtained, by means of MAP, from those in the source model (the $f_{j_i}^M$'s) and those in the body of the rule (the f_i's).

If the $f_{j_i}^M$'s include disjunctions, then let us rewrite $f_{j_i}^M \wedge f_i$ in disjunctive normal form $g_{i,1} \vee \ldots \vee g_{i,q_i}$. Then f' is built as the conjunction of the disjunctions of the applications of μ_{MAP} to the disjuncts:

$$r_R(M) = \{C(f')\} \text{ where } f' = f \wedge (\bigwedge_{i=1}^{h} (\bigvee_{t=1}^{q_i} \mu_{\text{MAP}}(g_{i,t})))$$

Let us see some examples. First, we have that $r_{R_{2,6}}(M_{\text{ERNoM2N}}) = \{\}$, as the rule is not applicable to the model (as we already saw).

Second, we have $r_{R_{2,6}}(M_{\text{ER}}) = \{\text{R}(f')\} = \{\text{R}(\neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2)\}$. The rule is applicable since only construct R in the body of the rule has an associated proposition and $(\text{F}_1 \vee \neg\text{F}_2) \wedge (\neg\text{F}_1 \wedge \neg\text{F}_2)$ is satisfiable, as it is equivalent to $\neg\text{F}_1 \wedge \neg\text{F}_2$. Then, applying the definition, we have that the conjunction of the disjunctions $\mu_{\text{MAP}}(\ldots)$ is *true*, since the only property in the body mapped to the head is O_1, which does not appear in the argument of μ_{MAP}. Therefore, f' equals the condition f in the head of the signature: $f' = f = \neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2$.

As a third example, to see MAP and μ_{MAP} really in action, let us apply $R_{2,6}$ to model $M = \{\text{E}(\mathit{true}), \text{R}((\text{F}_1 \vee \neg\text{F}_2) \wedge \neg\text{O}_1)\}$. The rule is applicable and we have $r_{R_{2,6}}(M) = \{\text{R}(f')\} = \{\text{R}(\neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2 \wedge \neg\text{O}_2)\}$, as:

$$\begin{aligned}
f' &= f \wedge \mu_{\text{MAP}}(((\text{F}_1 \vee \neg\text{F}_2) \wedge \neg\text{O}_1) \wedge (\neg\text{F}_1 \wedge \neg\text{F}_2)) \wedge \mu_{\text{MAP}}(\mathit{true} \wedge \mathit{true}) \\
&= f \wedge \mu_{\text{MAP}}((\text{F}_1 \wedge \neg\text{O}_1 \wedge \neg\text{F}_1 \wedge \neg\text{F}_2) \vee (\neg\text{F}_2 \wedge \neg\text{O}_1 \wedge \neg\text{F}_1 \wedge \neg\text{F}_2)) \wedge \mathit{true} \\
&= f \wedge (\mu_{\text{MAP}}(\text{F}_1 \wedge \neg\text{O}_1 \wedge \neg\text{F}_1 \wedge \neg\text{F}_2) \vee \mu_{\text{MAP}}(\neg\text{F}_2 \wedge \neg\text{O}_1 \wedge \neg\text{F}_1 \wedge \neg\text{F}_2)) \\
&= f \wedge (\mu_{\text{MAP}}(\mathit{false}) \vee (\mu_{\text{MAP}}(\neg\text{F}_2) \wedge \mu_{\text{MAP}}(\neg\text{O}_1) \wedge \mu_{\text{MAP}}(\neg\text{F}_1))) \\
&= f \wedge (\mathit{false} \vee (\mathit{true} \wedge \neg\text{O}_2 \wedge \mathit{true})) \;=\; \neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2 \wedge \neg\text{O}_2.
\end{aligned}$$

Let us define the **signature of a Datalog program**. Given a program $\mathbf{P}$ consisting of a set of Datalog rules $R_1, R_2, \ldots, R_n$, its signature $r_{\mathbf{P}}$ is the set of the rule signatures r_{R_i}'s, with i in $[1, n]$.

We are now ready for the definition of the **application** $r_{\mathbf{P}}(M)$ of the signature of a program $\mathbf{P}$ to a model M. Let us first consider programs with no strongly recursive rules; we will remove this assumption before the end of the section. In this case the application is the least upper bound of the applications of the rule signatures r_{R_i}'s to M: $r_{\mathbf{P}}(M) = \bigsqcup_{i=1}^{n} r_{R_i}(M)$. In this way, we have a construct for each applicable rule and, if a construct is generated by more than one rule, the associated proposition is the disjunction of the propositions associated to it by the various rules.

If we apply the program $\mathbf{P}_2$ in our running example to the ER model M_{ER}, then all constructs get copied and maintain the *true* proposition, except relationships, for which rules $R_{2,3}$, $R_{2,6}$, and $R_{2,7}$ generate, respectively, $\text{R}(\text{F}_1)$, $\text{R}(\neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2)$ (as we saw above) and $\text{R}(\neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2)$. Therefore, as the disjunction of the three formulas is F_1, the target model will have $\text{R}(\text{F}_1)$ and so we can say that the application of the signature of the program to M_{ER} produces M_{ERNoM2N}. The results in Section 3.5 will tell us that, as a consequence, the application of $\mathbf{P}_2$ to schemas of M_{ER} produce schemas of M_{ERNoM2N}.

Let us now consider also strongly recursive rules, that is, according to our definition, rules whose body includes atoms referring to the target schema. These rules may be applied on the basis of constructs generated by previous applications of other rules. As a consequence, the application of signatures is also defined recursively, as a minimum fixpoint. We can redefine the application of $r_{\mathbf{P}}$ to have two arguments, the source model and the target one, and its recursive application to a model M_0 is the fixpoint of the recursive expression $M = r_{\mathbf{P}}(M_0, M)$. Since it turns out that the application of $r_{\mathbf{P}}$ is monotonic, then, by Tarski's theorem [Tar55], we have that the minimum fixpoint exists and can be obtained by computing $M_1 = r_{\mathbf{P}}(M_0, \perp)$ (where $\perp$ is the empty model), $M_{i+1} = r_{\mathbf{P}}(M_0, M_i)$ and stopping when $M_{i+1} = M_i$.

3.5 Soundness and Completeness

In this section we demonstrate the usefulness of the formal system, proving that we can characterize the models obtained by applying Datalog rules by means of the notion of application of the signature of a rule to a model.

We need a few preliminary concepts. A **pseudoschema** is a set of ground atoms (called **ground constructs** hereinafter) each of which has the form $C(\text{oid} : o,\ p_1 : v_1,\ p_2 : v_2,\ \ldots,\ p_k : v_k,\ r_1 : o_1,\ r_2 : o_2,\ \ldots,\ r_z : o_z)$, where C is the name of a construct that has exactly the properties $p_1, p_2, \ldots, p_k$ and the references $r_1, r_2, \ldots, r_z$, each v_i is a boolean constant and each o_i is an identifier. A **schema** is a pseudoschema that satisfies the referential constraints defined over constructs. That is, if a schema includes a ground construct $C(\ldots)$ with a reference $r_i : o_i$ and r_i is subject to a referential constraint to a construct C', then the schema has to include a ground atom of the form $C'(\text{oid} : o_i, \ldots)$.

Given a pseudoschema S_0, a **closure** of S_0 is a schema that contains all the ground constructs in S_0. It can be shown that a closure of a pseudoschema can be obtained by applying a procedure similar to the chase for inclusion dependencies [CK86], which has a finite result in our case as we have acyclic referential constraints, and then replacing variables with "new", distinct values.

A schema **belongs to a model** if its predicate symbols (that is, its constructs) belong to the model and, for each ground atom $C(\ldots)$, the boolean values for its properties satisfy the proposition associated with C in the model.

Given a ground construct c with properties $p_1 : v_1,\ p_2 : v_2,\ \ldots,\ p_k : v_k$, we define the **description** of c, denoted with $\text{sig}(c)$, as $C(f)$ where $f = l_1 \wedge \ldots \wedge l_k$, and each l_j is a literal with the symbol p_j, positive if $v_j = \mathit{true}$ and negated if $v_j = \mathit{false}$.

As an example, R(Oid : *o*, O1: *false*, F1: *false*, I: *false*, O2: *true*, F2: *false*, ...) is a ground construct (with references omitted as not relevant) describing a many-to-many relationship, optional on one side and not optional on the other and without external identification. Its description is R($\neg$O1$\wedge\neg$F1$\wedge\neg$I$\wedge$O2$\wedge\neg$F2).

The notion of description can be extended to schemas: given a schema S, we define $\mathrm{sig}(S) = \bigsqcup_{c \in S}\{\mathrm{sig}(c)\}$. It is interesting to note (even if we will not use this property) that $\mathrm{sig}(S) = \sqcap\{M | S \text{ belongs to } M\}$ (that is, $\mathrm{sig}(S)$ is the greatest lower bound of the models to which S belongs). Therefore, $\mathrm{sig}(S) \sqsubseteq M$ if and only if S belongs to M.

Lemma 1
Let M be a model and R a Datalog rule. For each ground construct c in the pseudoschema $R(S)$ produced by the application of R to a schema S of M, it is the case that $\{\mathrm{sig}(c)\} \sqsubseteq r_R(M)$.

Proof
The proof is trivial if R is not applicable to S, because in this case the pseudoschema $\mathbf{P}(S)$ is empty and there is no ground construct c.
If R is applicable to S, let c be a ground construct generated by the application of R to S with properties $p_1 : v_1,\ p_2 : v_2,\ \ldots,\ p_k : v_k$.
The proof proceeds by first showing that r_R is applicable to M: the idea is that if the rule generates a new ground construct, then, for each of its body literals there is some ground construct in the schema that unifies with it; therefore the schema element satisfies both the condition in the body and that in the model and so their conjunction is satisfiable.
Let be $C(\ldots) \leftarrow C_{j_1}(\ldots), C_{j_2}(\ldots), \ldots, C_{j_h}(\ldots)$ the specification of R and let $\langle C_{j_1}(f_1), C_{j_2}(f_2),\ \ldots, C_{j_h}(f_h)\rangle$ be the body B of R. As c was generated, then rule R was applicable, hence S contains at least h ground constructs $c_i = C_{j_i}(\ldots)$ such that each of them unifies with an atom of the body of R.
Let be φ_i the assignment of values to properties corresponding to c_i, for each i in $[1, h]$. We have:

- φ_i satisfies f_i because c_i unifies with the ground constructs $C_{j_i}(\ldots)$;
- φ_i satisfies $f_{j_i}^M$ (remember that $f_{j_i}^M$ is the formula associated to C_{j_i} in the model M) because the schema S belongs to M and c_i belongs to S.

So we have an assignment of values to properties that satisfy both f_i and $f_{j_i}^M$. Indeed r_R is applicable to M and following the definition of application it results that $r_R(M) = C(f')$, with $f' = f \wedge (\bigwedge_{i=1}^{h} \mu_{\mathrm{MAP}}(f_{j_i}^M \wedge f_i))$.

Then, the proof shows that the assignment $\varphi : (p_1 = v_1, p_2 = v_2, \ldots, p_k = v_k)$ (that is, the one with the constants in c) satisfies the proposition f', by showing that it satisfies both f and $\bigwedge_{i=1}^{h} \mu_{\text{MAP}}(f_{j_i}^M \wedge f_i)$.

- φ satisfies f because constants in the head of rule R are present also in ground construct generated c by construction (in fact f is generated using such constants of the head) and therefore in φ;

- φ satisfies $\bigwedge_{i=1}^{h} \mu_{\text{MAP}}(f_{j_i}^M \wedge f_i)$; we show this by proving that φ satisfies $\mu_{\text{MAP}}(f_{j_i}^M \wedge f_i)$, for $i = 1, 2, \ldots, h$.
  Since R is applicable to S, there exist h assignments φ_1, φ_2, ..., φ_h to properties of constructs C_{j_1}, C_{j_2}, ..., C_{j_h} that satisfy formulas $f_{j_1}^M \wedge f_1$, $f_{j_2}^M \wedge f_2$, ..., $f_{j_h}^M \wedge f_h$, respectively.
  Let us consider an assignment φ_i satisfying $f_{j_i}^M \wedge f_i$ and rewrite this formula in disjunctive normal form (if it is not the case), obtaining the formula $g_{i,1} \vee g_{i,2} \vee \ldots \vee g_{i,q_i}$; so we have $f_{j_i}^M \wedge f_i \equiv \bigvee_{t=1}^{q_i} g_{i,t}$, hence $\mu_{\text{MAP}}(f_{j_i}^M \wedge f_i) \equiv \bigvee_{t=1}^{q_i} \mu_{\text{MAP}}(g_{i,t})$ and there exists at least one term $g_{i,\tau}$ (with τ in $[1, q_i]$) satisfied by φ_i.
  We prove that φ satisfies $\mu_{\text{MAP}}(g_{i,\tau}) = \mu_{\text{MAP}}(l_1) \wedge \mu_{\text{MAP}}(l_2) \wedge \ldots \wedge \mu_{\text{MAP}}(l_w)$ because it satisfies all terms $\mu_{\text{MAP}}(l_v)$ for $v = 1, 2, \ldots, w$. We have the following two cases:

  1. if $C_{j_i}(p_v)$ (where p_v is the property associated with literal l_v) does not belong to the codomain of the map of the rule, following the definition of transformation it results that $\mu_{\text{MAP}}(l_v) = \texttt{true}$ and so φ satisfies it;
  2. if $C_{j_i}(p_v)$ belongs to the codomain of the map of the rule, φ assigns to the property p^* associated with p_v via the map ($\text{MAP}(p^*) = C_{j_i}(p_v)$), by means of repeated variables in the head and the body of the rule, the same value that φ_i assigns to p_v; φ_i satisfies l_v (because it satisfies $g_{i,\tau}$) and following the definition of transformation $\mu_{\text{MAP}}(l_v)$ has the same sign of l_v, therefore φ satisfies $\mu_{\text{MAP}}(l_v)$.

□

Lemma 2
Let M be a model and R a Datalog rule. If s is a construct description such that $\{s\} \sqsubseteq r_R(M)$ then there is a schema S of M such that the application of R to S produces a pseudoschema $R(S)$ that contains exactly one construct c such that $\text{Sig}(c) = s$.

Proof
The proof proceeds by considering a construct c with description $s = \text{sig}(c)$ and showing that there is a set of ground constructs belonging to S corresponding to the atoms in the body of R out of which c can be produced.
Let be $R = C(\ldots) \leftarrow C_{j_1}(\ldots), C_{j_2}(\ldots), \ldots, C_{j_h}(\ldots)$ and consider a pseudoschema S_0 that contains h ground constructs, with repeated OIDs for repeated variables in the body of R and distinct ones elsewhere.
Some values of boolean properties of these ground constructs are copied (extracted) from constants in the body of R or traced back from the formula in s, by means of MAP (which gives no ambiguity, as there are no repeated boolean values in the head); any other property can have an arbitrary value.
The values of properties induced by the body are needed to guarantee that R is applicable to S_0. So, given an atom $C_{j_i}(f_i)$ of the body of r_R, we use f_i to initialize values of properties of ground construct c_i corresponding to such atom. In particular we assign a *true* value to properties corresponding to positive literals in f_i and *false* value to those corresponding to negated literals.
Let us consider values induced by MAP; the assignment φ of values to properties induced by s satisfies the formula f' of $\{C(f')\} = r_R(M)$ and, in particular, it satisfies the conjunction $\bigwedge_{i=1}^{h} \mu_{\text{MAP}}(f_{j_i}^M \wedge f_i)$. Each formula $f_{j_i}^M \wedge f_i$ of the conjunction corresponds to an element of the body of the rule and so it is associated with a ground construct c_i among those h constructs of S_0 previously introduced. Let us consider a formula $f_{j_i}^M \wedge f_i$ of the conjunction and rewrite it in disjunctive normal form (if it is not the case), obtaining the formula $g_{i,1} \vee g_{i,2} \vee \ldots \vee g_{i,q_i}$. Hence $\mu_{\text{MAP}}(f_{j_i}^M \wedge f_i) \equiv \bigvee_{t=1}^{q_i} \mu_{\text{MAP}}(g_{i,t})$ and φ, obviously, satisfies at least one element $\mu_{\text{MAP}}(g_{i,\tau})$ (with τ in $[1, q_i]$), where $g_{i,\tau} = l_1 \wedge l_2 \wedge \ldots \wedge l_w$; so φ satisfies all the terms $\mu_{\text{MAP}}(l_v)$ for $v = 1, 2, \ldots, w$ and we can use value assigned to literal $\mu_{\text{MAP}}(l_v)$ by φ to initialize value of property of ground construct c_i corresponding to literal l_v using ${\mu_{\text{MAP}}}^{-1}$.
Let us assign arbitrary values to other properties of c_i not initialized yet: this is possible because it means these properties are not directly involved in R.
Therefore we found h ground constructs c_i, and each of them satisfies a formula $f_{j_i}^M \wedge f_i$ (because they satisfy at least one element $g_{i,\tau}$ by construction). Consequently, pseudoschema S_0, made of these ground constructs, belongs to the model M (because of the $f_{j_i}^M$'s) and rule R is applicable to S_0.
Then, consider a closure S of S_0, which is a schema. Applying R to S, we obtain exactly a construct c' such that $s = SIG(c') = SIG(c)$, that is the only element of the pseudoschema $R(S)$.

□

Let us briefly comment on the latter lemma. Given our example model M_{ER} and rule $R_{2,6}$, we have that (as we saw) $r_{R_{2,6}}(M_{\mathrm{ER}}) = \{\mathrm{R}(\neg \mathrm{O}_1 \wedge \mathrm{F}_1 \wedge \mathrm{I} \wedge \neg \mathrm{F}_2)\}$. The lemma says that all construct descriptions $s \sqsubseteq r_{R_{2,6}}(M_{\mathrm{ER}})$ can be obtained as a result of the application of $R_{2,6}$ to some schema of M_{ER}. For example, description $s = \{\mathrm{R}(\neg \mathrm{O}_1 \wedge \mathrm{F}_1 \wedge \mathrm{I} \wedge \neg \mathrm{O}_2 \wedge \neg \mathrm{F}_2)\}$, which satisfies $s \sqsubseteq r_{R_{2,6}}(M_{\mathrm{ER}})$ can be obtained by applying rule $R_{2,6}$ to a schema which includes (together with other constructs) at least a many-to-many relationship, and all of them with $\mathrm{O}_1 = \mathit{false}$ (this follows from $\mathrm{MAP}_{2,6} = \langle \mathrm{O}_2 : \mathrm{R}(\mathrm{O}_1) \rangle$).

Lemmas 1 and 2 can be synthesized as the following theorem, which describes the behavior of individual Datalog rules with respect to models and schemas. It states that using the signature of a rule we can characterize the descriptions of the constructs that can be generated by that rule out of a given model.

Theorem 1
Let M be a model and R a Datalog rule. Then $\{s\} \sqsubseteq r_R(M)$ if and only if there is a schema S of M such that $R(S)$ contains exactly one construct c such that $\mathrm{sig}(c) = s$.

Let us now extend the results to Datalog programs.

Lemma 3
Let M be a model and $\mathbf{P}$ a Datalog program. The application of $\mathbf{P}$ to a schema S of M produces a schema $\mathbf{P}(S)$ that belongs to $r_{\mathbf{P}}(M)$.

Proof
The result of the application of $\mathbf{P}$ to a schema S produces a pseudoschema that is indeed a schema because the program is coherent with respect to referential integrity constraints by hypothesis.
Then, this schema belongs to $r_{\mathbf{P}}(M)$ because of: (i) Lemma 1; (ii) the definition of application $r_{\mathbf{P}}(M)$ as $\bigsqcup_{i=1}^{n} r_{R_i}(M)$, where n is the number of rules R_i's composing $\mathbf{P}$; (iii) the observation that, obviously, $r_{R_j}(M) \sqsubseteq \bigsqcup_{i=1}^{n} r_{R_i}(M)$ for each j in $[1, n]$.
If a ground construct c is generated by a rule R_j of the program $\mathbf{P}$ applied to M, then $\{\mathrm{sig}(c)\} \sqsubseteq r_{R_j}(M)$, by Lemma 1. Moreover, since $r_{\mathbf{P}}(M) = \bigsqcup_{i=1}^{n} r_{R_i}(M)$ and $r_{R_j}(M) \sqsubseteq \bigsqcup_{i=1}^{n} r_{R_i}(M)$, the assignment of values to properties induced by the formula f of $C_j(f) = r_{R_j}(M)$ satisfies $r_{\mathbf{P}}(M)$. Therefore all constructs generated by rules of $\mathbf{P}$ satisfy $r_{\mathbf{P}}(M)$ and so $\mathbf{P}(S)$ belongs to $r_{\mathbf{P}}(M)$.

□

Lemma 4
Let M be a model and $\mathbf{P}$ a Datalog program. If S' is a schema that belongs to $r_{\mathbf{P}}(M)$, then there is a schema S of M such that $\text{sig}(S') \sqsubseteq \text{sig}(\mathbf{P}(S))$.

Proof
Let us first consider non-recursive programs. The proof is essentially based on Lemma 2: for every ground construct c in S', we have (partly by hypothesis and partly by definition) that $\{\text{sig}(c)\} \sqsubseteq \text{sig}(S') \sqsubseteq r_{\mathbf{P}}(M)$; so, there is a construct description $C(f)$ in $r_{\mathbf{P}}(M)$ such that c has the predicate symbol C and its assignment of values to properties satisfies f.
Now, by definition of $r_{\mathbf{P}}(M)$, we have that f is obtained as the disjunction of the formulas associated with the various rules that have C in the head and therefore c (since it satisfies f) has to satisfy one of them. If R is such a rule, it turns out that $\{\text{sig}(c)\} \sqsubseteq r_R(M) \sqsubseteq r_{\mathbf{P}}(M)$ and so, by Lemma 2 we have that there is a schema S_c of M such that $R(S_c)$ contains a construct c' such that $\text{sig}(c) = \text{sig}(c')$.
Then, for each construct c in S', let us consider the corresponding schema S_c of M claimed by Lemma 2 and consider the "union" of such schemas for the various constructs, $S = \bigsqcup_{c \in S'}(S_c)$. A closure S^+ of S is a schema for model M by construction because obtained as the closure of union of schemas belonging to M and $\text{sig}(S') \sqsubseteq \text{sig}(\mathbf{P}(S^+))$ by construction of S^+ (i.e. by the definition of closure of a schema of Section 3.5).
The proof for recursive rules proceeds by induction on the number of steps needed to reach the fixpoint, with the induction step based on the arguments above.

□

Again, Lemmas 3 and 4 can be synthesized as the following theorem, which describes the behavior of Datalog programs with respect to models and schemas. It states that using the signature of the rules composing a program we can characterize the descriptions of the constructs that can be generated by that program out of a given model.

Theorem 2
Let M be a model and $\mathbf{P}$ a Datalog program. Then a schema S' belongs to $r_{\mathbf{P}}(M)$ if and only if there is a schema S of M such that $\text{sig}(S') \sqsubseteq \text{sig}(\mathbf{P}(S))$ modulo equivalence of propositions.

Theorem 2 synthesizes Lemmas 3 and 4 in a direct way. However, there is another point of view, which is more interesting, stated by the following theorem.

Theorem 3
Let M be a model and $\mathbf{P}$ a Datalog program. Then:

1. for every schema S of M, it is the case that $\text{sig}(\mathbf{P}(S)) \sqsubseteq r_{\mathbf{P}}(M)$;
2. there is a schema S of M such that $\text{sig}(\mathbf{P}(S)) = r_{\mathbf{P}}(M)$.

Proof
Claim 1 is essentially Lemma 3.
Claim 2 follows from the application of Lemma 4 to the extreme case of a schema S' whose description $\text{sig}(S')$ is exactly $r_{\mathbf{P}}(M)$ (this is possible because the constraints we have on our schemas are only referential integrity ones): by Lemma 4 we have that there is a schema S such that $\text{sig}(S') \sqsubseteq \text{sig}(\mathbf{P}(S))$, that is, $r_{\mathbf{P}}(M) \sqsubseteq \text{sig}(\mathbf{P}(S))$ and by Lemma 3 we have that $\text{sig}(\mathbf{P}(S)) \sqsubseteq r_{\mathbf{P}}(M)$.
□

Theorem 3 is our main result. It states that the derivation of model descriptions by means of the application of the signatures of Datalog programs is sound and complete with respect to the models generated by the program: a Datalog program can generate schemas with all and only the descriptions generated by the application of the signature of rules. In other words, descriptions completely characterize the models that can be generated by means of a Datalog program.

3.6 Applications of the Results

The technical development of the previous sections can be used in various ways to support the activities of an actual tool for schema translation, such as the MIDST tool [ACB06, ACG07, ACT+08] we have developed.

A simple use of the result is the possibility offered to check which is the model obtained as the result of the application of a program: the results in Section 3.5 allow the "rule designer" to know the output model without running (or inspecting) Datalog rules, but simply generating the signatures of rules and the description of a schema (model) and applying those signatures to that description.

A related use, still in rule specification, is the possibility to check whether a Datalog program takes into consideration all the constructs of a given source model, that is whether the application of a Datalog program to a given source model causes a loss of information. Let us say that the **domain** of a rule with respect to a model is the set of constructs of such model that are considered by the rule; formally, given a rule R and a model M, if R is applicable to M, the domain $\textsc{dom}(r_R, M)$ of R with respect to M is the set of the constructs of M that unify with the atoms of body B of the signature of R; if R is not applicable to M, then $\textsc{dom}(r_R, M)$ is the empty set. We can extend this notion to Datalog programs. Given a program $\mathbf{P}$ and a model M, the **domain** of $\mathbf{P}$ with respect to M, $\textsc{dom}(r_{\mathbf{P}}, M)$ is $\bigsqcup_{R \in \mathbf{P}'} \textsc{dom}(r_R, M)$ where $\mathbf{P}'$ denotes the program that includes only the rules in $\mathbf{P}$ that are applicable to M. Now, constructs (or variants of them) **ignored** by a program $\mathbf{P}$ when applied to a model M are those that do not unify with any atom of the body of any Datalog rule of $\mathbf{P}$, that is, those in the difference between M and $\textsc{dom}(r_{\mathbf{P}}, M)$.

A more ambitious goal would involve the automatic selection of rules for the generation of complex translations out of a library. A general approach for this, followed also by other authors in similar contexts [BMM05, PT05], would be based on the generation of a search tree and on the adoption of heuristics (for example based on A^*-type algorithms) that, under certain hypothesis, is optimal and complete. The formal system introduced allows the automatic generation of concise description of translation steps, and then could be the basis for the application of algorithms based on heuristics but it has several drawbacks:

- it could be computationally unfeasible, because the number of basic steps needed can be high;
- its termination in general need not be guaranteed, as multiple application of rules could arise, with no bounds (i.e. a program could introduce and eliminate the same constructs in turn, producing a loop);
- it could produce a translation plan driven for a specific criterion (it depends on the heuristics adopted) and hence the result plan may differ from the optimal one.

In the remainder of the section, we propose a different algorithm that, under suitable assumptions, is effective and more efficient. The preliminary assumptions are formalizations of some observations derived by our experience with the tool, in terms of both definition of models and specification of rules.

It can be observed that most translations, when applied to certain models, return a schema that is more restricted than the input, because they just eliminate a feature. Eliminations can be performed by dropping a construct or reducing its variants. Other translations, instead, introduce new constructs, besides eliminations of some constructs. So, we have two types of translations, *reduction* and *transformation*. With reference to the examples of Chapter 2, the elimination of generalizations is a reduction, performed by dropping the constructs devoted to represent generalizations, substituting them with new references or relationships (depending on the target model); the elimination of many-to-many relationships is performed adding constraints to the formula of the construct devoted to represent relationships; the replacing of relationships with references is an example of the latter category of programs.

A second observation is that we have few "families" of models, such as ER, OO and relational, and we manage many variations for each family. With respect to the previous observation, reductions allow one to move within a family (i.e. they return a schema or model of the same family), while transformations allow to move toward another family. Again with reference to Chapter 2, the elimination of generalizations is a reduction within the OO and ER families; the elimination of many-to-many relationships is a reduction within the ER family; the replacing of relationships with references is a transformation from the ER to the OO family.

Formally, a family of models $\mathcal{F}$ is a set of models defined by means of a model M^* (the **progenitor** of $\mathcal{F}$) and a set of models $M_{*,1}, M_{*,2}, \ldots, M_{*,k}$ (the **minimal** models of $\mathcal{F}$). It contains all models that are subsumed by M^* and subsumes at least one of the $M_{*,i}$'s:

$$\mathcal{F} = \{M \mid M \sqsubseteq M^* \text{ and } M_{*,i} \sqsubseteq M, \text{ for some } 1 \leq i \leq k\}$$

For example, the model M_{ER} of our previous examples should be the progenitor of the ER family, and a model with only entities and relationships should be one of the minimal models of such family.

Let us now formalize the notions of **reduction** and **transformation**. A translation $\mathbf{P}$ is a reduction for a family $\mathcal{F}$ if, when applied to a schema S of a model $M \in \mathcal{F}$, it generates a schema that is subsumed by the input ($\mathbf{P}(S) \sqsubseteq S$). The translations that are not reductions for a certain family are indeed transformations for such family (i.e. they tipically eliminate one or more constructs of the input and introduce new ones).

For example, translations $\mathbf{P}_2$ and $\mathbf{P}_3$ of Chapter 2 are a reduction and a transformation for the ER family, respectively.

Now we can present our assumptions on the set of basic translations.

Assumption 1
For each pair of families $\mathcal{F}_1$, $\mathcal{F}_2$ there are a model M_1 in $\mathcal{F}_1$ and a translation $\mathcal{T}$ such that, for each schema S_1 of M_1:

1. $\mathcal{T}$ does not ignore any construct of S_1;

2. $\mathcal{T}$ produces a schema that belongs to the progenitor M_2^* of $\mathcal{F}_2$.

This hypothesis requires the existence of f^2 translations, where f is the number of different families. This is not a real problem, as f is reasonably small and, whatever the approach, these rules would be needed in order to allow transformations between any pair of families. In general there might be pairs of families with more than one translation, but we ignore this issue, as it would not add much to the discussion.

Assumption 2
For each family $\mathcal{F}$, for each minimal model $M_{*,i}$ of $\mathcal{F}$, there is a translation from the progenitor M^* of $\mathcal{F}$ to $M_{*,i}$, entirely composed of reductions that do not ignore constructs.

The satisfaction of this assumption can be verified by considering all the reductions for a family (that is, the basic translations that are reductions for the progenitor of that family) and performing an exhaustive search on them. In principle, this may be inefficient, but in practice it can be done in a fast way, as the number of reductions in a family is small, and most of them are commutative.

We can prove that, if the set of basic translations satisfies Assumption 1 and Assumption 2, for each family and each pair of models M_1, M_2 of the same family, there is a translation from M_1 to M_2 that does not ignore constructs.

Since each model in the family is subsumed by the progenitor M^*, we have that $M_1 \sqsubseteq M^*$. Also, since the family has a set of minimal models, we have that for each model M in the family there is a minimal model M_* that is subsumed by it: either M_1 is minimal, in which case the statement is trivial, or there is another model M' such that $M' \sqsubseteq M_1$, and we can recursively apply the same argument, at most a finite number of times, as the set of models is finite. By Assumption 2, there is a translation from M^* to M_* that does not ignore any construct. This translation can be applied to every schema of M_1, producing a schema that belongs to M_* (since $M_1 \sqsubseteq M^*$, we have that every schema of M_1 is also a schema of M^*).

If M_* is a minimal model subsumed by M_2 then $M_* \sqsubseteq M_2$ and the claim is proved; otherwise, we can repeat the same procedure for M_2, identifying another translation from the progenitor M^* to a minimal model M'_*. This translation can be applied to every schema of M_*, producing a schema that belongs to M'_* (since $M_* \sqsubseteq M^*$, we have that every schema of M_* is also a schema of M^*) and so (as $M'_* \sqsubseteq M_2$) also to M_2.

Exploiting these assumptions, we designed an algorithm, based on practical application of the theoretical results provided in Chapter 3, that always find a complete transformation from a source schema to a target model with the following structure:

1. a reduction (composed of a sequence of translations that are reductions) within the source family (i.e. the family of the source model);

2. a transformation from the source family to the target family (i.e. the family of the target model);

3. a reduction (composed of a sequence of translations that are reductions) within the target family (i.e. the family of the target model).

On the basis of Assumption 1, the transformation of step 2 always exists. The first set of reductions (step 1) is needed to transform the source schema into another schema belonging to the model M_1 claimed in Assumption 1. Applying the translations of steps 1 and 2 to the source schema, we obtain a resulting schema that needs not belong to the target model, hence the second set of reductions (step3) is needed to guarantee it.

The previous assumptions and arguments justify the following algorithm, whose input is composed of a source schema S_1 and a target model M_2, and refers to a given set of families and a given set of Datalog programs.

findCompleteTranslation(S_1, M_2)
1 $\mathcal{F}_1 =$ family(S_1)
2 $\mathcal{F}_2 =$ family(M_2)
3 $\mathcal{T} =$ getTransformation$(\mathcal{F}_1, \mathcal{F}_2)$
4 $M'_1 =$ getSource$(\mathcal{T})$
5 $\mathcal{T}_1 =$ getReduction$(\mathcal{F}_1, M'_1)$
6 $\mathcal{T}_2 =$ getReduction$(\mathcal{F}_2, M_2)$
7 **return** $\mathcal{T}_1 \circ \mathcal{T} \circ \mathcal{T}_2$

Lines 1 and 2 find the families to which the source schema and the target model belong, respectively. In order to find the family to which a schema (model) belongs, it is enough to test the "inclusion" of such schema (model) against the progenitors of the families. This is feasible since the number of families is small. Let S be a schema and M^* the progenitor of a family $\mathcal{F}$, if $S \sqsubseteq M^*$ than $S \in \mathcal{F}$.

Line 3 finds the transformation $\mathcal{T}$ between the families $\mathcal{F}_1$ and $\mathcal{F}_2$, whose existence is guaranteed by Assumption 1. A Datalog program $\mathbf{P}$ is a transformation between families $\mathcal{F}_a$ and $\mathcal{F}_b$, whose progenitors are M_a^* and M_b^*, respectively, if $\bigsqcup_{R \in \mathbf{P}} B_R \sqsubseteq M_a^*$ (where B_R represents the body signature B of rule R) and $r_{\mathbf{P}}(M_a) \sqsubseteq M_b^*$. It is interesting to note that this operation can be performed off-line for each pair of families and not during a transformation process.

Then, line 4 computes the source model M_1' for transformation $\mathcal{T}$. Here we simply give a name to the already computed "union" of body signatures of rules of the selected transformation, $\bigsqcup_{R \in \mathcal{T}} B_R$.

Next, line 5 finds the sequence of reductions $\mathcal{T}_1$ needed to go from the progenitor of $\mathcal{F}_1$ to M_1' (on the basis of Assumption 2) and line 6 does the same within the target family (leading to $\mathcal{T}_2$). The first step to find a reduction (that is, a sequence of reduction programs) toward a model within a family is the search for reductions for that family, testing if the application of a program to the progenitor of such family returns a model subsumed by the progenitor. Then we have to order the reductions to avoid that a program could introduce a construct eliminated by a previous program. Given two Datalog programs $\mathbf{P}_1$ and $\mathbf{P}_2$, $\mathbf{P}_1$ precedes $\mathbf{P}_2$ if $\bigsqcup_{R \in \mathbf{P}_1} H_R \sqcap \bigsqcup_{R \in \mathbf{P}_2} B_R \neq \{\}$ (where H_R represents the head signature H of rule R). Obviously we have also to check that, at each step, no information gets lost. Finally, after we found a reduction, we can optimize it with respect to the actual input, which need not be the progenitor of a family. Again we note that the first two steps of this procedure can be executed off-line for each family, thus performing just the optimization step at run-time during the transformation.

Finally, the algorithm returns the needed translation that is the concatenation of $\mathcal{T}_1$, $\mathcal{T}$, and $\mathcal{T}_2$.

The algorithm is indeed used in our tool for the automatic generation of translations. In the Appendix Basic Translations, we show that the list of basic translations we used satisfies Assumptions 1 and 2 and so the translation can be always generated.

Chapter 4

Refactoring the Supermodel

4.1 Coherent and Cohesive

The usefulness of the MIDST proposal depends on the expressive power of its supermodel, that is the set of models handled together with accuracy and precision of their representations. In order to improve the expressive power of the supermodel, it has been necessary to introduce new constructs, often just variants of preexisting ones, leading to a growth in the number of constructs.

A key observation is the following: many constructs, despite differences in their syntactical structures, are semantically similar or identical. For example, in the ER model, attributes of entities and relationships show some similarity: even if they have different references (toward entities and relationships, respectively), and the first ones have an extra property (isKey), they both represent a lexical value. Also attributes in the ER model and columns in the relational model are very similar: they have different references (because first belong to entities or relationships and latter to tables) but have the same properties. In these cases, two or more constructs can be collapsed into a unique construct with their common semantics and a structure obtained by the union of the structures of the involved constructs. Clearly, constructs thus obtained have some optional references, together with some mandatory ones. This observation leads us to obtain a more compact (i.e. smaller number of constructs) and cohesive (i.e. one construct to represent all concepts with same semantics) supermodel. This is what we implicitly have already done in Chapter 2 for the sake of simplicity of presentation and figures, talking about "main" constructs. Here we fully exploit such observation, refactoring again the supermodel.

4.2 Refactoring the Formal System

Abstracting from the semantics of the refactoring of the supermodel, technically such refactoring causes the introduction of optional references. This implies the necessity to refactor the formal system introduced in Chapter 3 in order to keep on reasoning on data models, because now we can not abstract from considering references.

In this section we briefly illustrate how the concepts previously introduced need to be changed. The intuition is that, now, for every construct, we have to represent a formula over its properties and references, hence the universe is:

$$\mathcal{U} = \{C_1(P_1 \cup Ref_1), C_2(P_2 \cup Ref_2), \ldots, C_u(P_u \cup Ref_u)\}$$

For example, collapsing attributes of entities and relationships of the ER model and columns of the relational model, the definition of attribute, in the new approach, should be:

$$Attribute(isKey, isNullable, Entity, Relationship, Table)$$

The description of a model is still a set of constructs, each with an associated formula. Without loss of generality, let us assume that the formula associated with a construct C is a conjunction of literals:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge ref_1 \wedge ref_2 \wedge \ldots \wedge ref_m$$

where the p_i's are literals for properties and the ref_j's are literals for references. It states that construct C has non-null values exactly for the references corresponding to positive ref$_j$'s, and its properties must satisfy the constraints expressed by the conjunctions of the p_i's. We want to remark that the meaning of a literal for a reference in a formula is different from that of a literal for a property: a positive literal for a reference states that such reference must be non-null; a negated one states that such reference must be null.

Now formulas are more complex, but it is due to the increased complexity of the structure of the constructs. On the other hand they are more expressive, since with a single formula now we can express constraints not only on properties but also on references. Moreover we can also express constraints on the relationships between references of a construct, in the sense that, for each construct, we can force the presence of specific combinations of references and avoid other ones. For example we can state that both the references toward entities of a relationship must be valued at the same time, while just one of the three references of an attribute can be valued.

Some of the models discussed in Chapter 3, would be as follows (using the abbreviations E, R, T in the propositions for references toward entities, relationships, and tables, respectively):

- $M_{\text{ER}} = \{\text{E}(\mathit{true}), \text{R}((\text{F}_1 \vee \neg\text{F}_2) \wedge \text{E}_1 \wedge \text{E}_2), \text{A}(\text{E} \wedge \neg\text{R} \wedge \neg\text{T} \vee \neg\text{K} \wedge \neg\text{E} \wedge \text{R} \wedge \neg\text{T})\}$
- $M_{\text{ERSIMPLE}} = \{\text{E}(\mathit{true}), \text{R}((\text{F}_1 \vee \neg\text{F}_2) \wedge \text{E}_1 \wedge \text{E}_2), \text{A}(\neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T})\}$

In order to simplify the notation, in the following we omit the negated references; hence, the previous model descriptions would be as follows:

- $M_{\text{ER}} = \{\text{E}(\mathit{true}), \text{R}((\text{F}_1 \vee \neg\text{F}_2) \wedge \text{E}_1 \wedge \text{E}_2), \text{A}(\text{E} \vee \neg\text{K} \wedge \text{R})\}$
- $M_{\text{ERSIMPLE}} = \{\text{E}(\mathit{true}), \text{R}((\text{F}_1 \vee \neg\text{F}_2) \wedge \text{E}_1 \wedge \text{E}_2), \text{A}(\neg\text{N} \wedge \text{E})\}$

The notions of $\sqsubseteq$, $\sqcup$, $\sqcap$ and $-$ go unchanged. For example, we can check that also with this new formalism $M_{\text{ERSIMPLE}} \sqsubseteq M_{\text{ER}}$. In fact, it results that for E and R the condition is trivially verified, since they have the same formula associated in the two models. For A we have to consider the complete formulas (i.e. with also the negated atom for references); simplifying the conjunction of the two formulas in the two models, it results $\text{A}(\neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T})$, since:

$$\begin{aligned} f &= (\neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T}) \wedge (\text{E} \wedge \neg\text{R} \wedge \neg\text{T} \vee \neg\text{K} \wedge \neg\text{E} \wedge \text{R} \wedge \neg\text{T}) \\ &= \neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T} \vee \neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T} \wedge \neg\text{K} \wedge \neg\text{E} \wedge \text{R} \wedge \neg\text{T} \\ &= \neg\text{N} \wedge \text{E} \wedge \neg\text{R} \wedge \neg\text{T} \end{aligned}$$

Regarding the signature of Datalog rules, we redefine the description of an atom considering also the references. Given an atom $C(\text{ARGS})$, the corresponding construct description $C(f)$ is computed as the conjunction of the formula obtained on the basis of properties associated with a constant (as described in Section 3.4) with one literal for each reference in ARGS. For example, the signatures of the two atoms in the body of rule $R_{2,6}$ are $\text{R}((\neg\text{F}_1 \wedge \neg\text{F}_2) \wedge \text{E}_1 \wedge \text{E}_2)$ and $\text{E}(\mathit{true})$, respectively.

Then, the definitions of the three parts (B, H, MAP) of the signature r_R of a Datalog rule R go unchanged. For example, recalling rule $R_{2,6}$: the body signature is $B_{2,6} = \langle \text{R}(\neg\text{F}_1 \wedge \neg\text{F}_2 \wedge \text{E}_1 \wedge \text{E}_2), \text{E}(\mathit{true}) \rangle$; the head signature is $H_{2,6} = \text{R}(\neg\text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg\text{F}_2 \wedge \text{E}_1 \wedge \text{E}_2)$; the mapping is $\text{MAP}_{2,6} = \langle \text{O}_2 : \text{R}(\text{O}_1) \rangle$. We remark that references are not involved in the MAP and, hence, the application of μ_{MAP} to a literal for a reference always returns *true*. Also the application of the signature of a rule R and of a program $\mathbf{P}$ to a model M goes unchanged.

The most important thing is that all the theorems can be still proved.

Chapter 5

PolyDatalog

5.1 Overview

The main drawback of the refactoring of the supermodel is a general complication of Datalog rules. In fact now, for each predicate C in the body of a rule we have to specify which variant of it (i.e. which subset of its references we consider in the rule) we are interested in. This is done by adding a predicate for each of the references of the variant of the predicate we consider. As a consequence, it causes also a growth of the specificity of the single Datalog rules, thus contrasting with one of the main features of the approach that is genericity.

In this chapter, we report on our experience in extending Datalog with features that handle inheritance and some form of polymorphism, and show that in this way it is possible to increase the effectiveness of the language and the degree of reuse of the individual rules.

We introduce hierarchies in the supermodel, based on structural similarities of constructs, and extend Datalog in order to exploit such hierarchies: with our extension (based on directives for the rule engine and on the use of polymorphic variables) it is no longer necessary to write a specific rule for each variant of each construct, but it is possible to write just one polymorphic rule for each root construct of a generalization; it will be the rule engine that will compile Datalog rules, substituting polymorphic variables, obtaining specific rules for each variant of each root construct.

5.2 Related Work

The idea of extending logics and rule based systems with concepts like polymorphism, typing, and inheritance goes back to the beginning of 80's [MO84]. Recent approaches [DT93, DT94, ALUW93, AKM98, Jam97, LDL02, KLW95] adapt theories and methodologies of object-oriented programming and systems, proposing several techniques to deal with methods, typing, overriding, and multiple inheritance.

Gulog [DT93, DT94] is a deductive object-oriented logic (or, alternatively, according to its creators, a deductive object-oriented database programming language) with hierarchies, inheritance, overriding, and late binding; every Gulog program can be translated in an equivalent Datalog program with negation ($Datalog^{neg}$), where negated predicates are used to discern applicability of a rule to a class or subclass. Many works proposed extensions of Datalog and provided algorithms to translate their custom Datalog programs in "classic" Datalog with negation. In $Datalog^{meth}$ [ALUW93], a deductive object oriented database query language, Datalog is extended with classes and methods; its programs can be translated in Datalog with negation as well. Selflog is a modular logic programming with non-monotonic inheritance. In [AKM98], moving from SelfLog and $Datalog^{meth}$, Datalog is extended with inheritance (there are explicit precedence rules between classes) with or without overriding; programs can be rewritten in Datalog with an extra-predicate to mark rules and make them applicable only for a certain class or subclass; they propose also a fine grained form of inheritance for Datalog systems, where specialization of method definitions in subclasses is allowed and, when a local definition is not applicable, a class hierarchy is traversed bottom-up (from subclass to superclass) until a class with an applicable method is reached. $Datalog^{++}$ [Jam97] is an extension of Datalog with classes, objects, signatures, is-a relationships, methods, and inheritance with overriding; $Datalog^{++}$ programs can be rewritten in Datalog with negation. A language with encapsulation of rule-based methods in classes and non-monotonic behavioral inheritance with overriding, conflict resolution, and blocking (two features missing in other languages, according to the authors) is presented in [LDL02]. In f-logic [KLW95], limiting to topics of interest, there are polymorphic types, classes, and subclasses; it is possible to distinguish between two kinds of inheritance: structural, where subclasses inherit attributes of super-classes, and behavioral, where subclasses inherit methods of super-classes; three methodologies (pointwise, global-method, user-controlled) to manage the overriding with behavioral inheritance are provided.

Our approach differs from the aforementioned proposals. They introduce concepts of object-oriented programming and, in particular, propose overriding of methods for sub-classes, where needed; we have a different goal, we do not need overriding and do not define anything for sub-classes (sub-predicates, in our case). Instead, using object-oriented programming terminology, we define a method (the rule) for the super-class (the polymorphic construct) and, moving from it, generate specific methods (other rules) for the sub-classes (child constructs). From this point of view, our work has something in common with [BI97] where reusing and modification of rules is allowed by defining "ad hoc" rules to substitute name of predicates involved in other rules.

5.3 Extending Datalog

In this section we illustrate PolyDatalog, an extension of Datalog with concepts of polymorphism and inheritance, that exploits structural similarities of predicates and rules. In order to define a more expressive supermodel capable of properly representing a large number of models, we introduced new constructs, often just variants of preexisting ones. Despite the constructs are apparently few in number, we remark (as we said in Chapter 4) that in our supermodel we collapsed semantically identical constructs even if they are syntactically different because of various references; hence the global number of "real" constructs is high. Moreover, the need to represent complex concepts, like structured elements or nested elements, triggered a higher structural complexity of the supermodel. The growing number of predicates (i.e. constructs in our scenario) and the increasing structural complexity of the supermodel triggered two problems: hard scalability and low reuse of Datalog rules.

Changing perspective and using nomenclature of software analysis and design, it is possible to consider every variant of a construct as a child of a generalization rooted in a generic construct with no references. The idea is that to define an analogous transformation of all variants of the constructs it is no more necessary to write a specific rule for each of them, but it is possible to write just one polymorphic rule for each root construct of a generalization; it will be the rule engine that will compile Datalog rules, analyzing the generalizations defined in the supermodel and substituting polymorphic variables, obtaining one specific rule for each variant of each root construct. In the general case, a polymorphic rule designed for the transformation of a root construct C, with n child constructs, when compiled, will be instantiated n times, producing a specific Datalog rule for each child of the generalization rooted in C.

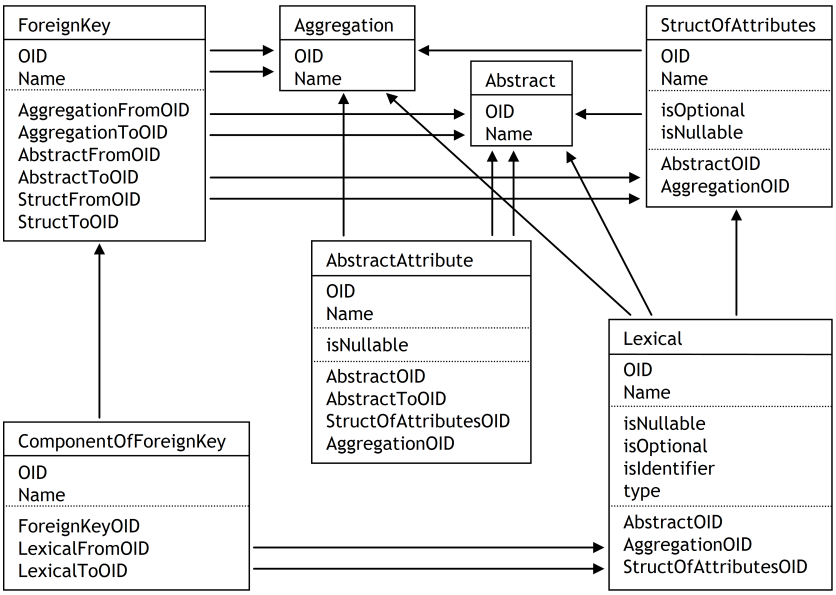


Figure 5.1: A simplified object-relational model.

In the reminder of this section we introduce the main ideas with an example involving a simplified version of the object-relational model (depicted in Figure 5.1), where most of the constructs have optional references since they were obtained by collapsing semantically similar constructs. In particular:

- Lexical has all its references (toward Abstract, Aggregation, and StructOfAttributes) mutually exclusive;
- StructOfAttributes has all its references (toward Abstract and Aggregation) mutually exclusive;
- AbstractAttribute has one mandatory reference (AbstractToOID, toward Abstract) and the others (toward Abstract, Aggregation, and StructOfAttributes, respectively) mutually exclusive;
- ForeignKey has two triples of references (toward Abstract, Aggregation, and StructOfAttributes) ending with "FromOID" and with "ToOID", respectively, whose elements are mutually exclusive.

Hence instances of Lexical and StructOfAttributes have just one valued reference while instances of AbstractAttribute and ForeignKey have two valued references. It results in a huge number of rules despite many groups of them are very similar by syntactical and semantic point of view.

These constructs can therefore be "generalized", each resulting in a hierarchy whose parent is the generic construct with properties and no references and whose children are the specific constructs, each with a set of mandatory references (among the allowed ones) and no properties. This is shown in Figure 5.2.

In the following we show the benefits of using inheritance in this scenario with an example: the transformation of a schema of the simplified object-relational model of Figure 5.1 into a relational model with tables, columns and foreign keys. This transformation could be composed by two steps:

1. eliminate typed tables;
2. eliminate structured columns.

A large number of sub-steps is required in order to perform the first step, as the transformation of typed tables into simple tables causes many other transformations, because all concepts (constructs) related in some way to a typed table have to be transformed too. Some examples follow. Reference columns have to be transformed into foreign keys and this can require the introduction of new key columns besides the creation of new columns to define foreign keys on them. Structured columns of typed tables have to be transformed in structured columns of tables and all the foreign keys involving typed tables as source or destination have to be transformed conveniently.

In terms of constructs and rules, something else has to be done because also components of foreign keys, columns of typed tables, and structured columns need to be explicitly transformed. In detail, in order to eliminate typed tables, we have to copy all elements not linked in any way with typed tables, like tables and their columns, structured columns of tables, foreign keys involving tables and their structured columns, and we have to properly transform typed tables and elements linked to them, like structured columns of typed table and foreign keys involving typed tables and their structured columns. The first step is quite simple but the latter has some criticalities. The only way to transform typed tables in the object-relational model without loss of information is to transform them in tables and elements linked to them accordingly, and to generate key columns for these new tables (to define foreign keys on them). All these steps require about thirty Datalog rules.

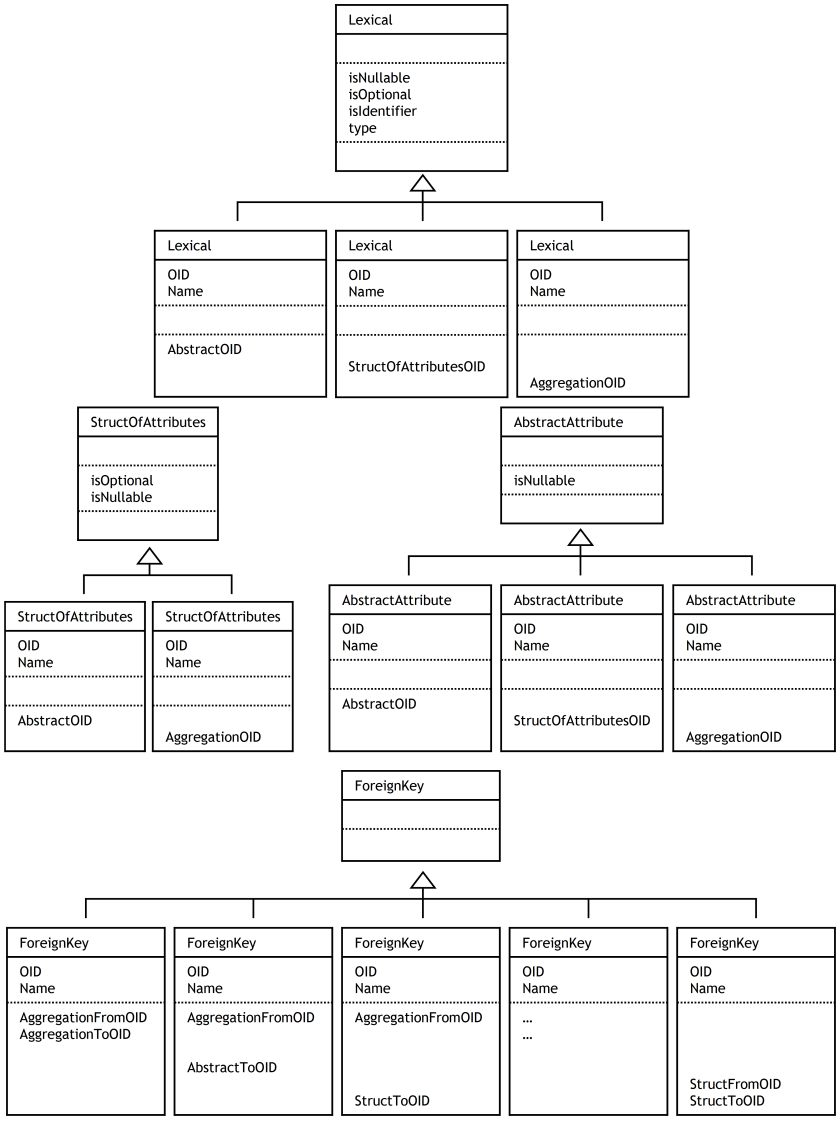


Figure 5.2: Generalizations of the object-relational model of Figure 5.1.

It is a pretty obvious and correct guess that many of these rules are syntactically very similar to one another and semantically identical. Two examples follow. First, the semantics of rules involving lexicals of "something" is always the same whichever is that "something": transport the values of various elements to the target schema, according to their belonging elements. Second, rules involving foreign keys have a unique goal: transport foreign keys to the target schema, according to the transformations undergone by elements linked by the keys themselves. Let us focus on the first example and show our corresponding Datalog rules. Rules involving lexicals are three:

R_a copy lexicals of aggregations;

R_b transform lexicals of abstracts in lexicals of aggregations;

R_c copy lexicals of structures of attributes.

The rules are as follows, where we omit some non-relevant details:

R_a:
LEXICAL (OID: #lexical_0(lexOID),
aggregationOID: #aggregation_0(aggOID),
isIdentifier: isId,
...,
type: type)
←
LEXICAL (OID: lexOID,
aggregationOID: aggOID,
isIdentifier: isId,
...,
type: type),
AGGREGATION (OID: aggOID);

R_b:
LEXICAL (OID: #lexical_0(lexOID),
aggregationOID: #aggregation_1(absOID),
...)
←
LEXICAL (OID: lexOID,
abstractOID: absOID,
...),
ABSTRACT (OID: absOID);

R_c:
LEXICAL (OID: #lexical_0(lexOID),
structOfAttributesOID: #structOfAttributes_0(structOID),
...)
←
LEXICAL (OID: lexOID,
structOfAttributesOID: structOID,
...),
STRUCTOFATTRIBUTES (OID: structOID);

Reasoning on these rules, we note that, despite some syntactical difference, they have the same semantics. In fact, they are isomorphic modulo renaming of constructs and Skolem functors and can be rewritten as:

LEXICAL (OID: #lexical_0(lexOID),
⟨construct⟩OID: ⟨SkolemForConstruct⟩(constructOID),
...)
←
LEXICAL (OID: lexOID,
⟨construct⟩OID: constructOID,
...),
⟨CONSTRUCT⟩ (OID: constructOID);

Let us comment on the syntax. This polymorphic rule states exactly the semantics we are interested in for transformation of lexicals. In the body, it abstracts from the specific construct referenced by LEXICAL, using the polymorphic variable ⟨CONSTRUCT⟩ as a predicate name and referring to such predicate by means of the polymorphic variable ⟨construct⟩ used as part of the field name of the LEXICAL reference. Also in the head, it abstracts from the construct referenced by LEXICAL (⟨Construct⟩)), but, in order to preserve correspondences of the source schema, it uses the polymorphic variable ⟨SkolemForConstruct⟩ as functor name. The rule engine can compile this rule, exploiting the polimorphic variables, thus producing the "standard" Datalog rules, R_a, R_b, and R_c).

Applying polymorphism to the whole program to eliminate typed tables, it is possible to compress many other rules (two rules involving structures of attributes, three rules to create lexicals to define foreign keys, three rules to create foreign keys, three rules to create component of foreign keys, and nine rules involving foreign keys) thus obtaining an abatement of number of rules from twenty seven to ten.

5.4 Implementation

There is a certain amount of theoretical and practical issues we have to deal with when considering the semantics of a PolyDatalog rule. We devote the following subsections to them.

Identifying the Translated Construct

First and foremost, the major critical issue is strictly bound to the PolyDatalog semantics. In order to instantiate a polymorphic rule, we need to know how the constructs referenced by a polymorphic construct have been translated within previous rules in the translation process. In general, in a Datalog rule, we therefore need to identify the "main", or "translated", construct within the rule's body, that is, establish which construct featured as a literal in its body is the actually translated construct into the construct expressed by the literal in its head. We found out three conditions a construct must satisfy in order to be recognized as the "main" construct within a rule:

- it must be featured in the rule's body;
- its own OID argument must be found as the argument of the Skolem functor which is used to generate the destination construct's OID;
- its own OID must not appear as an argument of a Skolem functor used to reference some destination construct in the head literal. More generally, any destination construct (i.e. with sOID equal to tgt) explicitly appearing in the rule's body, will never be the translated construct of that rule.

Let us refer to the examples of section 5.3. Using the aforementioned conditions, we can discover that the three constructs referenced by Lexical in the considered model (i.e. Abstract, Aggregation, and StructOfAttributes) have been turned into Aggregation, Aggregation, and StructOfAttributes, respectively, during the elimination of typed tables.

It is worth noting that here we give just a quick glimpse at the various Datalog rules in the simplest cases, but when dealing with more complex rules, with many constructs in their bodies and a wider range of syntax elements involved, things might get far more complicated.

Multiple Translations for a Single Construct

The second issue is strictly bound to the identification process for the main construct of a Datalog rule. As it results from our previous assertions, when we proceed to check whether a given construct is actually the translated construct of a rule, we are considering it in terms of its name. In other words, we check whether a construct, having a certain name, is the main construct within a set of scanned rules. At this high level of abstraction, though, a seemingly critical scenario may occur: throughout a whole translation process, a given kind of construct bearing a specific name may have multiple translations (i.e. may result as the main construct of multiple rules). As an example, let us consider the elimination of many-to-many relationships within the ER family: many-to-many relationships have to be translated into entities, while others relationships have to be copied. When compiling a polymorphic rule for the copy of attributes of relationships, we have to consider both the translations undergone by the relationships.

The key point is that multiple translations depend on the specific features of the involved constructs (i.e. on some constraints on its properties). Hence, in order to deal with this particular situation, the parameters used to discriminate between the different instances of a construct must be included when instantiating the polymorphic rule involving such construct (i.e. the construct is referenced by the polymorphic one).

Multiple Polymorphic References within a Single Literal

The third issue is related to polymorphic constructs that allow for multiple mandatory references among their optional ones (e.g. foreign keys).

From a superficial point of view, we could think it is enough for these mandatory references to be handled separately. Actually, this is not the case: since their simultaneous presence is a constraint for a construct, we will have to generate every possible combination for the allowed references, when instantiating the PolyDatalog rule involving that construct. The compiling of such rules produce $\prod_{i=1}^{k} n_i$ rules, where n_i is the number of translations for the construct C_i pointed by a polymorphic reference and k is the number of polymorphic references of the construct under examination.

It is rather evident that these particular cases allow for the most effective use of the PolyDatalog rules, whereas a single polymorphic rule succeeds in replacing several tens of classic Datalog rules.

PolyDatalog Interpreter

The previous discussions and arguments justify the following algorithm, which takes as input a basic translation **P**, and refers to a given set of generalizations defined over the metaconstructs:

```
polyDatalogInterpreter(P)
 1  for each R of P
 2     if isPolymorphic(R) then {
 3           rootC = getPolymorphicConstruct(R)
 4           cList = getChildren(C)
 5           for each C_i in cList {
 6              ruleSet_i = findRules(C_i)
 7              IR_i = instantiateRules(C_i, ruleSet_i)
 8              P = P − R + IR_i
 9           }
10     }
```

It analyzes every rule of the program (line 1). If a polymorphic variable is found (line 2), it finds out the polymorphic construct involved (line 3) and then obtains the list of children of such construct (line 4), by analyzing the generalizations. For each child of the generalization C_i (line 5), it looks out for non polymorphic rules in **P** whose main construct is one of those referred by C_i (line 6). Now it can produce the needed non polymorphic rules (line 7) and substitute the polymorphic rule in **P** with them (line 8).

The rule engine of the MIDST tool has been updated according to the presented algorithm. It is now capable of properly "compiling" polymorphic rules, finding out polymorphic variables and recovering the needed information, in order to produce a suitable set of standard Datalog rules.

5.5 Experimental results

In this section we present the experimental results obtained comparing the old library of basic translations with the new one featuring polymorphic rules.

We have already seen in Section 5.3 the power of the proposed Datalog extension with respect to the specific translation from the object-relational to the relational model. By extending this approach to the whole set of schema translations handled by our system, we have obtained significant advantages:

	Number of Initial Rules	Number of Resulting Rules	Number of Replaced Rules	% of Replaced Rules
Entity-Relationship	21	18	3	14.2
Binary Entity-Relationship	83	79	4	4
Object (UML Class Diagram)	8	8	0	0
Object-Relational	480	325	155	32.3
Relational	7	7	0	0
XSD	138	88	50	36.2
Supermodel	29	13	16	55

Figure 5.3: Experimental results.

an average 30% of Datalog rules (with peaks of 55%) have been replaced by a handful of PolyDatalog rules. The summary of these results is shown in Figure 5.3, where for each data model we indicate the number of rules before and after the introduction of polymorphism, the number of replaced rules and the percentage of such replacement. We collect in the supermodel row the results for copy rules.

We want to remark that this is not just about a reduction in the amount of rules necessary for each translation. For starters, easiness of developing is increased, for a PolyDatalog rule is relatively easier to write than a classic Datalog rule (where we should pay attention to the particular references for the featured constructs). Besides, as we have early said, correctness and completeness are assured, because the instances resulting from a polymorphic rule are exactly identical to those rules we would have previously specified ourselves. Furthermore, PolyDatalog's inner parametricity in terms of the polymorphic references greatly enhances scalability, reusability, and maintainability of rules and whole translations: first, if the translation for a specific construct referenced by a polymorphic one changed over time, the corresponding PolyDatalog rule would not change at all; second, only a handful of polymorphic rules have succeeded in removing hundreds of original Datalog rules within all the trans-

lation processes handled by our system; and finally, we could anytime define more PolyDatalog rules should the need arise, for instance when newer and more complex hierarchies are introduced in the data dictionary, all the while getting even larger benefits from our Datalog extension.

Chapter 6

Toward an On-line Operator

6.1 Overview

In this chapter we illustrate a runtime (or on-line) approach to model-generic translation of schema and data. The proposal is based on our previous work on the MIDST platform, originally conceived to perform translations in an off-line fashion. As seen in Chapter 2, in the original approach, the source database is imported into a dictionary, where it is stored according to a universal model. Then, the translation is applied within the tool as a composition of elementary transformation steps, specified as Datalog programs. Finally, the result is exported into the operational system[1].

MIDST approach provides a general solution to the problem of schema translation, with *model-genericity* (as the approach works in the same way for many models) and *model-awareness* (in the sense that the tool knows models, and can use such a knowledge to produce target schemas and databases that conform to specific target models). However, as pointed out by Bernstein and Melnik [BM07], this approach is rather inefficient for data exchange. In fact, the necessity to import and export a whole database in order to perform translations is out of step with the current need for interoperability in heterogeneous data environments.

Here we propose a new, lightweight, runtime approach to the translation problem, where data is not moved from the operational system and translations are performed directly on it. The tool needs only to know the model and the

[1]We use the term *operational system* to refer to the system that is actually used by applications to handle their data.

schema of the source database and generates views on the operational system that transform the underlying data (stored in the source schema) according to the corresponding schema in the target model. Views are generated in an almost automatic way, on the basis of the Datalog rules for schema translation. The approach is model-generic and model-aware, as it was the case with MIDST, because we leverage on MIDST dictionary for the description of models and schemas and also on its key idea of having translations within the supermodel, obtained as composition of elementary ones, each dealing with a specific aspect (construct or feature thereof) to be eliminated or transformed. The main difference is that the import process concerns only the schema of the source database. The rules for schema translation are used here as the basis for the generation of views in the operational system. In such a way data is managed only within the operational system itself. In fact, our main contribution is the definition of an algorithm that generates executable data level statements (view definitions) out of schema translation rules.

A major difference between an off-line and a runtime approach to translation is the following. For an off-line approach, as translations are performed within the translation tool (MIDST in our case), the language for expressing translations can be chosen once, for all models. A significant difficulty is in the import/export components, which have to mediate between the operational systems and the tool repository, in terms of both schemas and data. In fact, in the development of MIDST, a lot of effort was devoted to import/export modules, whereas all translations were developed in Datalog. In a runtime approach, the difficulties with import/export are minor, because only schemas have to be moved, but the translation language depends on the actual operational systems. In fact, if there is significant heterogeneity, then stacks of languages may be needed (e.g. SQL, SQL/XML, and XQuery), possibly with different dialects, and our techniques need to cope with them.

In order to cope with the heterogeneity of the involved languages, we propose an approach based on two steps: first it generates views organized according to the constructs in the target model, but independent of the specific languages, and then actually concretises them into executable statements on the basis of the specific language supported by the operational system.

In this chapter we provide a general solution to the language independent step, whereas for the final one we concentrate on SQL, with respect to a set of models that include many variations of the object-relational and of the relational one. As a running example, we will see how relational views can be generated to access an object-relational schema with references and inheritance.

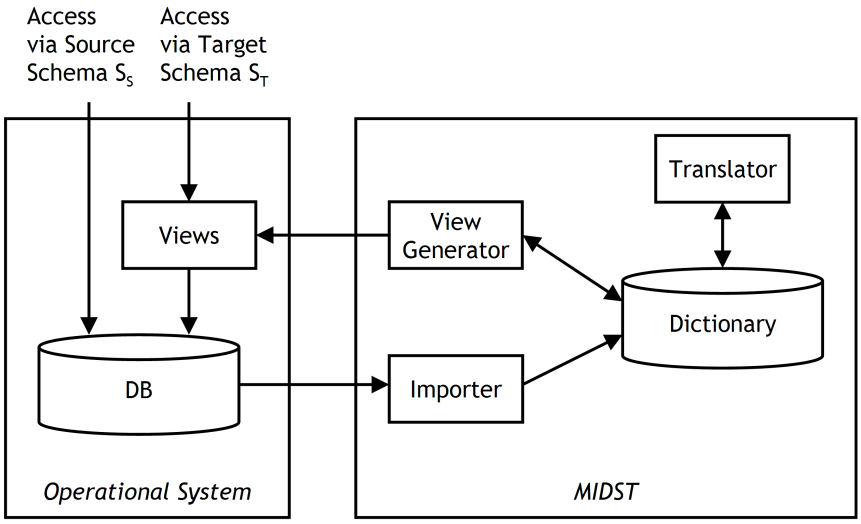


Figure 6.1: The runtime translation procedure.

6.2 Overall Approach

The goal of a tool for schema and data translation is to provide support to the adoption of a wide family of heterogeneous data models. In a runtime perspective, this means that application programs, designed to interact with a specific data model M_t, would be allowed to work with another data model M_s in a transparent way. The tool we propose supports this feature by translating the schemas of M_s (which actually contain the data of interest for the programs) in terms of views of model M_t. Then, the application programs would use these views to access data organized according to M_s.

As we said, in our original approach translations are dealt with in an off-line fashion, meaning that the import of both schema and data into MIDST is needed as well as an export of the result. In this chapter we describe an enhanced version of our platform that enables the creation of executable statements generating views in the operational system.

Let us illustrate our approach, by following the main steps it involves, with the help of Figure 6.1:

1. given a schema S_s (of a *source* model M_s) in the operational system, the user (or the application program) specifies a *target* model M_t;

2. schema S_s (but not the actual data) is imported into MIDST, and specifically in its dictionary, where it is described in supermodel terms;

3. MIDST selects the appropriate translation $\mathcal{T}$ for the pair of models (M_s, M_t), as a sequence of basic translations available in its library;

4. the schema-level translation $\mathcal{T}$ is applied to S_s to obtain the target schema S_t (according to the target model M_t);

5. on the basis of the schema-level translation rules in $\mathcal{T}$, the tool generates views over the operational system, in three phases: first it generates an abstract description of views that specify schema S_t (and so conform to model M_t) in terms of the elements of the source schema in S_s; then, it translates these abstract descriptions into system-generic SQL-like view definitions; finally, it compiles statements that define the actual views in the specific language available in the operational system.

Let us observe that steps 1-4 appear also in the previous version of MIDST, whereas 5 is completely new, in all its phases, and clearly significant.

As a running example, consider the following. Assume we have an environment where application programs are designed to interact with relational databases while we have an actual database on the operational system based on the object-relational model, with the following features: tables, typed tables, references between typed tables, and generalizations over typed tables. In this scenario, our tool generates relational views over the object-relational schema, which can be directly used by application programs.

A concrete case for this example involves the OR schema sketched in Figure 6.2. The boxes are typed tables: employee (*EMP*) is a generalization for engineer (*ENG*) and department (*DEPT*) is referenced by employee.

The goal of the runtime application of MIDST is to obtain a relational database for this, such as the one that involves the following tables[2]:

EMP (EMP_OID, lastname, DEPT_OID)
DEPT (DEPT_OID, name, address)
ENG (ENG_OID, school, EMP_OID)

[2]As it is well known, there are various ways to map generalizations to tables, and this is one of them.

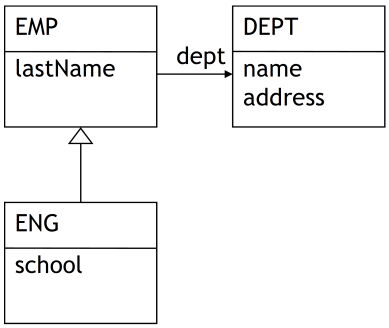


Figure 6.2: A simple object-relational schema.

Given the schema in Figure 6.2, our tool first imports it in its dictionary. Then, given the specification of the target model, it selects an appropriate schema-level translation, which is a sequence of basic translations. In this case, the schema-level translation should perform the following tasks:

$\mathbf{P}_A$ eliminate generalizations (replacing them with references);

$\mathbf{P}_B$ eliminate reference columns (replacing them with foreign keys and generating new identifiers for the typed table, if needed);

$\mathbf{P}_C$ transform typed tables into tables.

The major task of our new version of the tool is the generation of a set of view statements for each of these Datalog programs. The following is a sketch of a view definition generated in the first step.

```
CREATE VIEW ENG_A ... AS (
   SELECT ... SCHOOL, ... EMP_OID
   FROM ENG
);
```

It extends *ENG* (denoted as *ENG_A* to distinguish the new version from the original one) with a supplementary attribute, *EMP_OID*. It implements a strategy for the elimination of generalizations, where both the parent and child typed tables are kept, with a reference from the child to the parent. In the technical sections of this chapter we will see how we produce views of this kind and we will show the missing details.

6.3 Generating Views

In this section, we discuss the major ideas of our approach to the generation of views for runtime translations. Then, in Section 6.4 we will discuss the technical details and present a complete algorithm.

The General Approach

The core goal of the procedure is to generate executable statements defining views. This is obtained by means of an analysis of the Datalog schema rules. The analysis gives a system-generic statement. A system-generic statement is a view defined by means of an SQL-like language that could be translated into another language (e.g. SQL, SQL/XML, XQuery) in order to be executed by the operational system.

A key idea in the procedure is a classification of MIDST metaconstructs according to the role they play. There are three categories: *container*, *content*, and *support constructs*. Container constructs are those corresponding to structured objects in the operational system (i.e. AGGREGATION and ABSTRACT corresponding to tables and typed tables respectively). Content constructs represent elements of more complex constructs[3], such as columns, attributes or references: usually a field of a record (i.e. LEXICAL and ABSTRACTATTRIBUTE) in the operational system. Support constructs do not refer to data-memorizing structures in the system, but are used to model relationships and constraints between them in a model-independent way. Examples are GENERALIZATION (used to model hierarchies) and FOREIGNKEY (used to specify referential integrity constraints).

Our Datalog rules, in turn, can be classified according to the construct featured as head predicate. Therefore we have *container-*, *content-*, and *support-generating* rules (e.g. generating ABSTRACT, LEXICAL, and FOREIGNKEY, respectively).

Exploiting the above observations, the procedure defines a view for each container construct, with fields that derive from the corresponding content constructs. Instead, as support constructs do not store data, they are not used to generate view elements (while they are kept in the schemas). More precisely, given a Datalog schema rule $H \leftarrow B$, if H refers to a container construct, we will generate one view for each instantiation of the body of the rule. If H refers to a content, we need to define a field of a certain view. The head predicates

[3]For the sake of simplicity in the examples we will refer only to flat models and hence we do not consider contents of contents.

of container-generating rules handle one OID, while the head predicates of content-generating rules deal with more OIDs. In fact, the role corresponds to an intrinsic structural difference between constructs. While containers have only one OID (which identifies the construct), contents have at least two OIDs: one identifying the field itself and one relating it to the owner container (other OIDs may be needed when fields refer to complex construct, as it happens for AbstractAttribute).

Two major issues in the procedure are the provenance of data (that is, where to derive the values from or how to generate them) for the single field and the appropriate combination of the source constructs (which is equivalent to a join, from a relational point of view). In the next subsection we describe possible approaches to the former issue and conclude the illustration of the informal procedure by presenting examples of SQL statements in case we only deal with one source construct (all attributes refer to it); then we will abandon this assumption and comment on more general cases in which several source constructs must be combined, facing the latter issue.

Let us discuss program $\mathbf{P}_A$ in the example. There are various ways to eliminate generalizations. Let us refer to the one that maintains both the parent and the child typed tables and connects them with a reference. This requires that we copy all typed tables with their columns and then add a new column for each child typed table with a reference to the respective parent typed table. The only container-generating rule is the one that copies the typed tables (R_1) and hence we generate a view for each typed table of the operational system: *EMP_A*, *ENG_A* and *DEPT_A*[4].

The other rules are content-generating. From rules copying Lexical (R_2) the procedure infers owner view, name, and type for each field. For the copy of AbstractAttribute (R_3), the procedure works likewise with the addition that it has to handle the values encoding the references between constructs in an object-oriented fashion. Finally, for the rule that handles Generalization together with ChildOfGeneralization (R_4), maintaining both the parent and the child and connecting the constructs with a reference, the problem of data provenance for fields is evident: while in rules R_2 and R_3 the values are copied from the source fields, in rule R_4 an appropriate value that links the child table with the parent one has to be generated.

In program $\mathbf{P}_B$ we use a rule (R_5) to generate a key attribute for each typed table without identifier. It is a content-generating rule since it generates a key Lexical for every Abstract without an identifier. Hence we add

[4]We use the suffix to distinguish the versions of tables and views in the various steps.

another field to the views that correspond to those ABSTRACTs. Then we translate references into value-based correspondences (foreign-key)[5]. Before defining such foreign keys, we need a rule (R_6) that copies the identifier values of the referred construct into the referring one in order to allow for the definition of value-based correspondences. It implies the addition of a new field to the view that corresponds to the referring ABSTRACT.

Program $\mathbf{P}_C$ is simpler. The only transformation involved turns typed tables into tables once they do not have any generalizations nor references and the presence of identifiers is guaranteed. The issue is then limited to the internal representation of views handled by the operational system. Many systems distinguish between *views* and *typed views*, then all we need is to handle this distinction.

This procedure does not depend on the specific constructs nor on the operational system or language. It is not related to constructs because we only rely on the concepts of container and content to generate statements. Other constructs may be added to MIDST supermodel without affecting the procedure: it would be sufficient to classify them according to the role they play (i.e. container, content, or support). Moreover, it is not related to the operational system constructs or languages since the statements are designed as system-generic. A specification step, exploiting the information coming from a negotiation between MIDST and the operational system, will be then needed to generate system-specific statements. Furthermore, this approach is extensible because we might also consider (as we will see shortly) adding *annotations* to functors whenever conditions get more complex and in order to handle specific cases. The procedure is not bound to a single language and the generation of statements could involve the integration of several dialects fetching data from heterogeneous sources. This would not increase the complexity of the analysis nor the system-generic statements.

The Provenance of Field Values

In this subsection we consider the problem of the data provenance of the single field. It means that the procedure needs to know either a source field to derive a value from or a generation technique. We devised an automatic procedure that, for a given rule, collects information about the provenance of values by analyzing the parameters of the Skolem functor used in the head of the rule.

[5]Notice that we refer to foreign-key values, as we use them, but not to foreign-key constraints because they are not usually meaningful in views.

In case it has only one parameter, the OID of another field, then the value comes from the instance of the construct having that OID. This is what happens in programs $\mathbf{P}_A$ and $\mathbf{P}_B$ whenever a LEXICAL is copied (rule R_2). Similarly, if the Skolem functor has more than one parameter and one of them refers to a field, then a source construct can be individuated as well. Instead, if none of the functor parameters refers to a content (it only deals with container or support constructs), the result value has to be generated somehow. This is what happens in programs $\mathbf{P}_A$ and $\mathbf{P}_B$ with rules R_4, R_5, and R_6, respectively.

These cases can be handled automatically as well. We introduce solutions that are based on annotations which specify value generation techniques. Here we present an informal description of this approach to give an intuition of the adopted strategy while technical details will be pursued in Section 6.4.

In rule R_4, the functor generates the OID for an ABSTRACTATTRIBUTE (a content construct) from the OID of a GENERALIZATION (a support construct). In order to obtain a reference from the child table to the parent it is possible to use the tuple OID[6] as value for the reference field. A reason for this choice is the fact that every instance of a child typed table is an instance of the parent table too. Then for each tuple of the child container there is a corresponding tuple in the parent one with a restricted set of attributes, but with the same tuple OID. Therefore the reference can be made by means of an appropriate casting of this OID.

The following system-generic SQL-like statement is generated for the elimination of hierarchies (program $\mathbf{P}_A$) in the running example. The *ENG* typed table is a specialization of *EMP*, so the rule copies its attributes and adds the values for the field referencing the parent *EMP* by casting the tuple OID.

```
CREATE VIEW ENG_A ... AS (
   SELECT ... SCHOOL, REF(ENG_OID) AS EMP_OID
   FROM ENG
);
```

In rule R_5, the functor generates the OID for a LEXICAL from the OID of an ABSTRACT therefore it conveys the fact that the value of the field corresponding to that LEXICAL derives from a container. A possible strategy would involve the transformation of the tuple OID into a value for this field. This solution would guarantee the presence of a unique identifier.

[6] In OR systems, every typed table usually has a supplementary field, OID, treated as a unique identifier which can be used to base reference mechanisms on. Notice that this OID is not related to the OID used in MIDST which identifies the constructs.

In rule R_6, the functor indicates that the value of the field derives both from the AbstractAttribute and the Lexical. Whenever a Lexical is involved in the provenance of a value, such value comes from it independently of the other involved constructs.

Combining Source Constructs

On the basis of the discussion in the previous subsection, it turns out that for each field in a view, we have either a provenance or a generation. Provenance can refer to different source constructs, in which case it is needed to correlate them. In database terms, a correlation intuitively corresponds to a join. However, in practice, this need not be the case. If two fields can be accessed from the same container it is wise to do it and to avoid joins. For instance if a construct C has a reference toward a construct C' and the fields c_i of C and c'_j of C' must be fetched, one can use that reference to get both the values from C without using the join operator. Moreover, a simpler variant is possible if all the fields of a given view derive from the same container, as it happens in all the steps of our example.

In our paradigm, the information about the join conditions are encoded in the Skolem functors. In fact we handle typed functors that generate OIDs for specific constructs given the OIDs of a fixed set of constructs. Therefore we may state that for a given set of contents, each of which is derived through the application of a Skolem functor on other constructs, the collection of all the used functors encodes the join conditions.

For instance, consider another way of eliminating generalizations: copying the child attributes into the parent and deleting the child. In this case, we have a content-generating rule for the parent, copying Lexicals from the child to the parent itself. The Skolem functor #lexical_2.1 (genOID, parentOID, childOID, lexOID) creates OIDs for those Lexicals, relating one Generalization (genOID), two Abstracts (parentOID, childOID) and one Lexical (lexOID). Obviously the parent preserves its original Lexicals (i.e. attributes) as well. The functor #lexical_5(lexOID) creates OIDs for those Lexical, using the OID of another Lexical (lexOID).

The specific set of content-generating functors ({#lexical_2.1, #lexical_5}) encodes (by means of their parameters) the fact that we have a left join on OID basis in such a way that all the instances of the parent that are also instances of the child, appear in the result view as a single tuple. Moreover the left join guarantees the inclusion of all the instances of parents that do not belong to any child.

In the running example we have *EMP*; according to the lastly mentioned strategy, it has to be merged with its child *ENG*. The general procedure establishes the presence of a join whose condition is encoded by the specific set {#lexical_2.1, #lexical_5}.

```
CREATE VIEW EMP_A (..., LASTNAME, SCHOOL) ... AS (
   SELECT ... EMP.LASTNAME, ENG.SCHOOL
   FROM EMP LEFTJOIN ENG
   ON (CAST (EMP.OID AS INTEGER) = CAST (ENG.OID AS INTEGER)
);
```

Notice that, in this statement, the pattern bases joins on the sharing of tuple OIDs which takes place between parent and child instances.

As mentioned before, there might be cases in which fields of different containers can be accessed by just referring to a single container by means of references. This is what happens in program $\mathbf{P}_B$ where the values for the fields in the referring typed table, must be derived from the key fields in the referred one (rule R_6). The following statement is among those generated for program $\mathbf{P}_B$: *EMP* has references toward *DEPT* (which does not appear in the statement) via the field *dept* and *DEPT_OID* is the identifier for *DEPT* added in rule R_5. Then, we need to copy *DEPT_OID* values into a field of *EMP* according to the semantics of the rule. It is clear that there are two sources: *EMP* and *DEPT*. However *DEPT_OID* can be accessed via *dept*, therefore the join between the two containers is not needed.

```
CREATE VIEW EMP_B ... AS (
   SELECT ... LASTNAME, dept->DEPT_OID AS DEPT_OID
   FROM EMP_A
);
```

So, source constructs are handled in a lightweight way: joins are avoided by exploiting *dereferencing* (as in the example) when such a feature is supported by the operational system. Otherwise, when they are necessary, their treatment is globally encapsulated in Skolem functors that relate constructs in a strongly-typed fashion. In general, we can provide a different combination of Skolem functors for each needed join condition. The concept is that we exploit functor expressivity and strong typedness to understand how to combine the containers of the different fields.

6.4 The View-Generation Algorithm

Let us now discuss with some detail the algorithm we adopt to generate views at runtime from Datalog rules encoding schema-level translations.

The algorithm takes as input a schema-level translation expressed as a set of Datalog rules, a classification for the involved constructs (i.e. container, content, and support), and generates SQL statements defining views on the basis of the translation. The algorithm is composed of three parts: an abstract specification of the views; the generation of system-generic SQL-like statements corresponding to those views; translation of the system-generic statements into statements that are actually executable on the operational system.

We devote one subsection to each of the three parts of the algorithm to illustrate their technical details.

Procedural Analysis

In our context, each Skolem functor Sk is associated with a given construct, to which we refer as the type *type*(Sk) of the functor. Each functor always appears with the same arity and with arguments that have each a fixed type. The associated function is injective and function ranges are pairwise disjoint.

For example, let us consider rule R_6 which replaces the references of typed tables with simple fields, in order to allow for the definition of value-based correspondences; in detail, for each reference (AbstractAttribute) it replicates the key fields (Lexicals) of the referred typed table (Abstract) into the referring one. The functor has the following structure:

$$\text{Sk6} : \text{AbstractAttribute} \times \text{Lexical} \rightarrow \text{Lexical}$$

meaning that it takes in input the OID of an AbstractAttribute and the OID of a Lexical and generates a unique OID for another Lexical, as it can be inferred from the head literal in which it is used. Moreover, it results that *type*(Sk6) = Lexical.

Let us now investigate the relationship between the role of constructs and the Skolem functors used to generate their OIDs. A container construct has a single OID, which identifies it. Whenever a container is featured in a head H_i, the functor Sk_i is responsible for the creation and for the uniqueness of that OID. Conversely, a content construct is characterized at least by two OIDs: one that identifies the construct itself and another one referring to its container construct. The former plays the same role as in the containers and it is determined by the application of the Skolem functor Sk_i; the latter denotes

the container construct to which the content belongs and it is calculated by another functor SK_i^p. Symbols SK_i and SK_i^p will be used throughout the whole explanation of the procedure to denote the two functors for a content construct.

For example, the head of rule R_1 (which copies abstracts) has the form:

ABSTRACT (OID: #abstract_0(absOID),
sOID: tgt,
Name: n)

It is evident that an ABSTRACT (i.e. a container) is characterized only by its OID. Conversely, a LEXICAL (i.e. a content), as confirmed by the head of rule R_2 (which copies LEXICALs of ABSTRACTs), is characterized by two OIDs:

LEXICAL (OID: #lexical_0(lexOID),
...
abstractOID:#abstract_0(absOID))

The functor #lexical_0 (i.e. SK_i) is the one used to generate a unique identifier for a LEXICAL from the OID (lexOID) of another LEXICAL; the functor #abstract_0 (i.e. SK_i^p) is the one used to connect each instance of a LEXICAL (i.e. content) to the proper ABSTRACT (i.e. container) by generating the OID of the target ABSTRACT (abstractOID) from the one of the source (absOID).

In order to formalize a classification of constructs on the basis of the number of OIDs they have, similar considerations should be necessary also for support constructs. Indeed, in a complex system there might be support constructs as well as both containers and contents. However, this classification aims at providing a mechanism to detect content- and container-generating rules on the basis of the head predicate. Since support constructs do not contribute to the generation of structures that handle actual data, we can limit our discussion to the illustrated cases.

Therefore a container construct is a construct where only one OID (the identifier), and the respective Skolem functor are meaningful, while a content one needs at least two OIDs. Consequently, we distinguish between container- and content-generating rules on the basis of the number of OIDs in the head predicate.

Given a Datalog rule R we define an *instantiated body* IB as a specific assignment of values for the construct attributes appearing in it. It means that for each construct in the body we have values for name, properties, references, and OID that satisfy the predicates in the body of the rule itself with respect to the considered schema.

We define an *instantiated head* IH for a given instantiated body IB, as a construct whose name, properties, references, and OID are instantiated as a consequence of instantiation IB of B.

Finally an *instantiated Datalog rule* IR is a pair (IH, IB) where IH is an instantiated head for the instantiated body IB of R.

Let us see an example considering rule R_4, that replaces generalizations with references. Notice that the rule is evaluated only against the MIDST supermodel, where the preliminary import phase has generated a representation for the schema of the operational system in terms of MIDST constructs. A possible instantiated body is the following one:

```
GENERALIZATION (OID: 101,
                ...
                parentAbstractOID: 1),
CHILDOFGENERALIZATION (
                OID: 102,
                ...
                generalizationOID: 101,
                childAbstractOID: 2)
```

It states that the ABSTRACT representing the typed table *EMP* (with OID equal to 1 and used as value for the attribute parentAbstractOID) is the parent of the ABSTRACT representing the typed table *ENG* (with OID equal to 2 and used as value for the attribute childAbstractOID). As it is possible to infer from the example, the conditions expressed in the bodies of Datalog rules (which are evaluated within MIDST supermodel) may refer to container and content constructs as well as to support ones.

With reference to R_4 and to the sample instantiated body, we have the following instantiated head:

```
ABSTRACTATTRIBUTE (
                OID: #abstractAttribute_4(102),
                ...
                AbstractOID: #abstract_0(2),
                AbstractToOID: #abstract_0(1))
```

This head defines a new reference column for a typed table aimed at substituting a parent-child relationship; using supermodel terminology, it generates an ABSTRACTATTRIBUTE for a given CHILDOFGENERALIZATION (involving two ABSTRACTS and one GENERALIZATION).

Let us introduce some notation and definitions that are useful to explain how views are generated.

- Given a translation $\mathbf{P}$, we denote the sets of content- and container-generating rules in it as $\mathit{ContentGen}(\mathbf{P})$ and $\mathit{ContainerGen}(\mathbf{P})$, respectively.
- Given a translation $\mathbf{P}$ and a container-generating rule R in $\mathbf{P}$, we define $\mathit{content}(R_i, \mathbf{P}) = \{R_j \in \mathit{ContentGen}(\mathbf{P}) \mid \mathit{type}(\textsc{Sk}_j^p) = \mathit{type}(\textsc{Sk}_i)\}$. In plain words $\mathit{content}(R_i, \mathbf{P})$ is the set of rules generating contents for the construct generated by R_i.
- For each $R \in \mathit{ContainerGen}(\mathbf{P})$ (that is, for each container generating rule) we define an *abstract view*, as a pair $\textsc{Av} = (R, \mathit{content}(R, \mathbf{P}))$, composed of the rule itself and a set of rules defining contents for the container generated by rule R. An abstract view is generic in the sense that it is written with respect to types of constructs. The focus is on the type of container construct generated and the same argument can be applied to contents: the content-generating rules define types of contents (e.g. columns, attributes, or references) and not specific instances of them.

Given an abstract view Av, we compute *instantiated views* over it. Each of them is defined as $V = (\textsc{Ir}, \{col_1, col_2, ..., col_n\})$. They are pairs composed of an instantiation Ir of the container-generating rule R and of the set of all the possible instantiations of rules in $\mathit{content}(R, \mathbf{P})$ that are coherent with Ir.

Let us see an example considering program $\mathbf{P}_A$ of Section 6.2. It follows that $\mathit{ContainerGen}(\mathbf{P}_A) = \{R_1\}$ and $\mathit{ContentGen}(\mathbf{P}_A) = \{R_2, R_3, R_4\}$. Consequently we can determine the following abstract view: $\textsc{Av}_1 = (R_1, \{R_2, R_3, R_4\})$. Finally, it is possible to instantiate the abstract view $\textsc{Av}_1$ according to the constructs of the operational system. Then the instances may be represented as:

$V_1 = (EMP_{R_1}, \{EMP(lastName)_{R_2}, EMP(dept)_{R_3}\})$

$V_2 = (DEPT_{R_1}, \{DEPT(name)_{R_2}, DEPT(address)_{R_2}\})$

$V_3 = (ENG_{R_1}, \{ENG(school)_{R_2}, ENG(EMP)_{R_4}\})$

Let us comment the first instantiated view. It states that container *EMP*, generated by rule R_1, will have two contents (i.e. *lastName* and *dept* originally belonging to *EMP*) generated by rules R_2 and R_3, respectively.

An abstract view describes all the views that must be generated from a container-generating rule and its instantiations correspond to the views that will be generated.

View-generating Statements

The second part of the procedure involves the translation of each instantiation V of every abstract view $\text{Av} = (R, \mathit{content}(R, \mathbf{P}))$ into a view-generating statement with the SQL structure:

```
CREATE VIEW name(col_1, col_2, ...) AS (
   SELECT a_1(s_1.col_1), a_2(s_2.col_2), ...
   FROM source(col_1)s_1 cond source(col_2)s_2 cond ...
);
```

In the statement, *name* is the instantiated name of the head construct of R, that denotes a container in the operational system. Then, col_1, col_2, ..., col_n are the names of the constructs generated by all the possible instantiations of the body of the rules in $\mathit{content}(R, \mathbf{P})$, and so some of them may derive from different instantiations of the same rule, while others may derive from different rules.

Now we have to face two major issues to characterize the general statement:

(a) the determination of the source container (in the statement denoted by $\mathit{source}(\mathit{col}_i)$) and the actual value for each content (indicated with the functional symbols a_i); this problem has been informally discussed in Section 6.3 and consists in establishing a way of computing the values for fields in the result views, hence assigning a semantics to the functional symbols a_i in the statement;

(b) the determination of the appropriate form of combination needed for the source containers of the various contents (indicated with the symbol *cond*); this problem has been discussed in Section 6.3 and consists in replacing *cond* with appropriate join conditions in the statement.

As for point (a), consider the Skolem functors of a given content-generating rule R_i. Functor Sk_i^p links the generated content with its source container and its parameters are instantiated as a consequence of the instantiation of body B_i of R_i. Functor Sk_i conveys information about the provenance of data (i.e. the content to derive the value from) for the content under examination.

The strategy we follow relies on a default case in which the functor has a parameter whose type is content. If this happens, that container is the source for the values. Otherwise, it is possible to specify annotations to force a specific behavior. An annotation is a query (for example an SQL statement) that specifies how to calculate the value for a field. Annotations must be written at schema level, expressing transformations to be applied for each different instantiation, as it happens for Datalog rules.

More precisely:

(a.1) if SK_i is not annotated, at least one of its parameters must refer to a content construct (a real one in the operational system, since the functor is instantiated). Therefore, the value for the container instance is derived from it without any further computation.

(a.2) if SK_i is annotated with a query a, then a is applied in order to calculate the needed value. Notice that the query can be written referring to all the literals in the instantiated content-generating rule. Generally, these queries are very simple and use a small number of parameters. In the SQL statement above, a functional symbols a_i denotes the application of a query associated to an annotation (in the default cases this query has no effect).

As an example of case (a.1), consider rule R_6 of program $\mathbf{P}_B$ which replaces the references of typed tables with simple fields (in order to allow for the definition of value-based correspondences). The functor #lexical_6 is not annotated and takes as input the OID of the AbstractAttribute and the OID of a Lexical referred by it. This implies that values for the new field (generated to represent a reference) have to be directly derived (namely, copied) from values of the source Lexical.

On the other hand, as an example of case (a.2), consider rule R_4 of program $\mathbf{P}_A$ which replaces the generalizations between two typed tables by adding a specific reference field (AbstractAttribute) in the child table. The functor #abstractAttribute_4 takes in input the OID of a ChildOfGeneralization (childOID) and hence, in this case, it is correct to annotate the functor to specify how the values for the field have to be calculated.

The following pseudo-SQL statement is an example of annotation defined at schema level that helps calculate the value for the field:

```
SELECT INTERNAL_OID FROM childOID;
```

It specifies that the value of the reference must coincide with the OID of the tuple under examination for the childOID (which refers to the view that is being populated).

A similar strategy should be followed to cope with rule R_5 of program $\mathbf{P}_B$. As we have seen, such a rule generates a key field for every typed table without an identifier: thus the problem of generating a unique value at data level arises. In the head of the rule, the functor #lexical_5 takes the OID of an ABSTRACT (absOID) as input parameter, meaning that there are no valid sources for the values. A possible annotation could be the following one:

```
SELECT INTERNAL_OID FROM absOID;
```

It implies the adoption of the internal tuple identifiers (INTERNAL_OID) as values for the keys of typed tables.

As for point (b) two cases are indeed possible for an instantiated view V_i depending on the instantiation of the functors SK_i^p's:

(b.1) there are contents deriving from the same container, let us call them *sibling contents*. This corresponds to instantiated rules (not necessarily referring to the same rule) where the values of the parameters of the functors SK_i^p's are the same;

(b.2) there are contents deriving from different containers, let us call them *non-sibling contents*. This corresponds to instantiated rules (not necessarily referring to the same rule) where the values of the parameters of the functors SK_i^p's are different.

Case (b.1) can be thought of as a default case, in which no further definitions are necessary and the translation can be performed directly; conversely, case (b.2) requires some decision and needs the definition of strategies to combine the sources.

In fact, in (b.1) it is sufficient to copy the contents from the container SK_i^p refers to. Thus, for each set of sibling contents, we have the specification of a container in the `FROM` part of the SQL statement.

On the other hand, in (b.2) the *cond*s in the SQL statement must be translated into appropriate join conditions. It is clear that there are several variants for the joins, according to the semantics of the schema-level translation. The key is that Skolem functors allow to specify this semantics at schema level in such a way that it can be translated into join conditions at data level. We define a *schema-join correspondence* SJ such that $\text{SJ} : S^n \rightarrow cond$, where S^n is a tuple of Skolem functors and *cond* is a join condition, expressed as a statement

at schema level. The correspondence SJ assigns a join condition to a specific tuple of functors, which are the ones that generate the OIDs for the contents in the container under examination. Then, for example, if a container has three contents: two sibling contents and a non-sibling one, then the tuple will be composed of two functors, one for the siblings and another one for the single content. Then the correspondence SJ will specify how to combine the two associated source containers in terms of join conditions. As for annotations in point (a), join conditions must be written at schema level (for example directly with a pseudo-SQL formalism) and, when omitted, the Cartesian product between the source containers is implied.

Case (b.1) is rather simple and an example of it is the overall translation of program $\mathbf{P}_A$ where, for each typed table, the values are directly derived from one source table and no joins are needed.

Conversely, as seen in Section 6.3, an occurrence of case (b.2) arises in the elimination of generalizations consisting in copying the contents of child typed tables into the parent. Obviously, since the parent maintains its contents, there are contents coming from the child typed table and others from the parent one. The involved functors are #lexical_2.1, the one responsible for the OIDs of LEXICALs copied from the children to the parents (*school* from *ENG* to *EMP* in the example), and #lexical_5, responsible for the OIDs of the parent LEXICALs (*lastName* of *ENG* in the example). Here we define the schema-join correspondence SJ : $(\#\text{lexical\_2.1}(\ldots), \#\text{lexical\_5}(\ldots)) \rightarrow cond_1$, where $cond_1$ can be defined according to a pseudo-SQL formalism as follows:

```
parentOID LEFT JOIN childOID ON INTERNAL_OID;
```

This pseudo-SQL condition, together with the schema-join correspondence definition, specifies that, whenever two non-sibling set of contents derive from the combination of the functors #lexical_2.1 and #lexical_5, then the source containers have to be combined with a left join on the basis of the internal OID. The left join guarantees that instances of the parent that are not also instances of the child are preserved in the result. It is clear that different correspondences, in association with different join conditions, can be defined to cover a wide range of cases.

Executable Statements

After a system-generic SQL statement has been generated for a Datalog translation, it is customized according to the specific language and structures of the operational database system in order to be finally applied.

With respect to a complex translation involving more than one program, each system-generic SQL statement encoding an elementary step is translated in terms of a system-specific and executable one. Then the views generated by one step are used by the following one and all the statements represent a pipeline of transformations yielding the desired output view.

The following SQL statements exemplify the elimination of hierarchies (rule R_4) which takes place in program $\mathbf{P}_A$, with reference to IBM DB2. This DBMS adopts the concept of *typed view*, which is a view whose type has to be defined explicitly. This motivates the presence of the two initial statements defining the types *EMP2* and *ENG2* in the result schema. The statements below implement the strategy consisting in using the internal OID to relate a child with its parent. It is evident that a lot of technical details depending on the operational system (i.e. DB2 in the example) are introduced in this last phase (e.g. the use of type constructors, the various casting functions, or explicit scope modifiers).

```
CREATE TYPE EMP2_t AS (
   lastname varchar(50)
)
NOT FINAL INSTANTIABLE
MODE DB2SQL WITH FUNCTION ACCESS REF USING INTEGER;

CREATE TYPE ENG2_t AS (
   toEMP REF(EMP2_t), school varchar(50)
)
NOT FINAL INSTANTIABLE
MODE DB2SQL WITH FUNCTION ACCESS REF USING INTEGER;

CREATE VIEW EMP2 of EMP2_t
   MODE DB2SQL (REF is EMP2OID USER GENERATED) AS
   SELECT EMP2_t(INTEGER(EMPOID)), lastname
   FROM EMP;

CREATE VIEW ENG2 of ENG2_t
   MODE DB2SQL (REF is ENG2OID USER GENERATED,
      toEMP WITH OPTIONS SCOPE EMP2) AS
   SELECT ENG2_t(INTEGER(ENGOID)),
      EMP2_t(INTEGER(EMPOID)), school
   FROM ENG;
```

Discussion

The proposed algorithm represents the core step in translating schemas from a model to another at runtime since it allows the translation of Datalog rules into actually executable SQL statements on data. The original MIDST framework is a model-independent implementation of the ModelGen operator. Here we argue that the proposed algorithm extends the platform to a runtime context and allows for the interaction with heterogeneous database systems, without affecting model-independence feature. In fact the initial import of information about the schema of the operational database supports the definition of system-independent and model-independent translations. We manage to decouple the technical details of the operational system and its model from translation rules, by means of suitable import modules that allow to translate the internal representations of the systems in terms of the constructs of the supermodel.

Then the schema-level rules are actually applied on the supermodel in order to obtain schema information about the translated database, in such a way that further operations are possible. What the algorithm performs is a procedural analysis of translation on the basis of a generic whole-part (container-content) classification of supermodel constructs and on the basis of the model-awareness principle we foster in MIDST. It means that, although MIDST is model-generic, in the sense that translations can be applied independently of target and source models, we handle specific metadata about models by adopting typed constructs which differ from one another, and strongly typed Skolem functors, which can be applied on and return only specific types of construct OIDs. Hence, as evident from the informal illustration of the algorithm, model-awareness allows to evaluate the relationships between constructs and their instances in the operational system without affecting the model-generic approach of the whole process.

The whole-part classification of the constructs of the supermodel is not a limitation because it is the essential relationship in most common data models. Moreover, more complex structures of target systems (e.g. nested tables and generalizations) are indeed treated by means of support constructs that can be even used in translations to specify schema-level conditions.

The presented approach solves performance issues that affected MIDST due to the necessity to import into the supermodel and export back the whole database. Schema metadata are obviously much lighter than the actual data and the time spent in importing them has no relevance in the performance of the translation. Furthermore, the computation of the SQL-generating statements is performed only once (and in advance) for each translation; then the

optimization of the query and the performance issues are entirely devoted to the operational database system. From our point of view, we showed that although MIDST is model-independent, hence it handles translations between any pair of models, the number of the needed steps is bounded and small. Moreover, the number of the generated queries is minimal. In fact, due to the detection of the appropriate join conditions, we generate one query for each view needed in the operational system and do not need to unite results from different statements.

6.5 Related work

In this chapter we have tackled the problem of enhancing MIDST with the possibility of applying runtime translations in such a way that data exchange queries are computed out of schema translation rules and are used to generate views. Our approach toward data exchange is not formal, what we are interested in is the set of statements solving the data exchange problem between the source schema and the wanted view; however we share many ideas with characterizations by Fagin et al. [FKP05, FKMP03].

Mork et al. [MBM07] also adopt a runtime approach (based on [ACB06, AT96]) to solve the specific problem of deriving a relational schema from an extended entity-relationship model. They use a rule-driven approach and write transformations that are then translated into the native mapping language. However, although they face many issues such as schema update propagation and inheritance, indeed they solve a specific subset of problems and provide an object-relational mapping tool similar to Hibernate [Hib]. Terwilliger et al. [TMB08] adopt a runtime approach to allow a developer to interact with XML or relational data in an object-oriented fashion. On the one hand their perspective is different since they only deal with a specific kind of heterogeneity; in addition they address the problem by translating the queries while we aim at generating views on which the original queries can be directly applied.

Our approach is aimed at providing a runtime support to the whole range of translations allowed by MIDST that is not limited to OO-to-relational or XML-to-OO, but involves any possible transformation between a pair of models in our supermodel (i.e ER, OR, OO, XSD, and Relational, actually).

Our approach shares some analogies with Clio [FKMP03, FKP05, HHH$^+$05, MHH00, VMP03] too. It is aimed at building a completely defined mapping between two schemas, given a set of user-defined correspondences. As for our translations, these mappings could be translated into directly executable SQL,

XQuery or XSLT transformations. However, in the perspective of adopting Clio in order to exchange data between two heterogeneous schemas, the needed mappings should be defined manually; moreover, there is no kind of model-awareness in Clio, which operates on a generalized nested relational model. Although it can be shown to subsume a considerable amount of models, in a real application scenario a preliminary translation and adaptation of the operational system should be performed, leading to the problems of the initial MIDST approach.

The presented runtime extension of MIDST is a significant step with respect to the process of turning the platform into a complete model management system [ABBG08a]. In such a perspective, Datalog rules can be used to specify model-to-model translations as well as more general transformations that implement schema evolution and model management operators. Therefore the possibility of applying translation, hence operators, at runtime allows for the runtime solution to model management problems with model-independent approaches like those illustrated in [ABBG08b].

Conclusions

Many problems facing data management and other areas of computer-aided engineering involve the manipulation of models. Applications that manipulate models are complicated and hard to build. The goal of generic model management is to support the productivity of developers by providing high-level operators defined on schemas and mappings over them, thus reducing the cost of developing such applications.

In this dissertation we focused on model-generic translation of schemas, discussing our recent results and our contributions to the development of the MIDST platform that allows the specification of the (data) models of interest, with all relevant details, and the generation of translations of their schemas from one model to another.

The usefulness of the MIDST proposal depends on the expressive power of its supermodel, that is the set of models handled together with accuracy and precision of their representations. In order to improve the expressive power of the supermodel, we extended it with concepts like nesting relationships, complex structured elements, collections, and substructures, in order to represent recent complex data models like the object-relational and the XSD.

With many possible models and many basic translations, it becomes important to understand how to find a suitable translation given a source and a target model. Here there are two difficulties. The first one is how to verify what target model is generated by applying a basic step to a source model. The second one is related to the dimension of the search space.

In order to solve these problems, we conceived a formal system to automatically reason on models, based on the notions of description of a model and signature of a basic translation. It allows to infer the model (description) $r_{\mathbf{P}}(M)$ obtained out of a model M by applying the signature $r_{\mathbf{P}}$ of a Datalog program $\mathbf{P}$. We have proved that such derivation is sound and complete: the application of $\mathbf{P}$ to schemas of M produces only schemas that belongs to $r_{\mathbf{P}}(M)$

and potentially all of them. Moreover, the formal system has been the theoretical basis for the development of an algorithm for the automatic generation of translations, that has been implemented within the MIDST framework.

Another drawback of the growth of the supermodel is a general complication of the specification of translations, by means of Datalog rules. We have proposed an extension of Datalog, called PolyDatalog, based on the use of hierarchies and a sort of polymorphism, that provides a significant simplification in the definition of complete translations (Datalog programs) and a higher level of reuse in the specification of elementary translations (Datalog rules) in scenarios with structural similarities of predicates and syntactical and semantical similarities of rules. We have integrated such extension in our MIDST project and the successful experiments have supported our claim, since PolyDatalog's inner parametricity in terms of the polymorphic variables greatly enhanced scalability, reusability, and maintainability of rules and whole translation, besides leading to an abatement of the number of Datalog rules constituting complete programs.

The MIDST approach provides a general solution to the problem of schema translation, with model-genericity (as the approach works in the same way for many models) and model-awareness (in the sense that the tool knows models, and can use such a knowledge to produce target schemas and databases that conform to specific target models). Although, in its original version, it was rather inefficient for data exchange. In fact, the necessity to import and export a whole database in order to perform translations is out of step with the current need for interoperability in heterogeneous data environments.

We presented a new, lightweight, runtime approach to the translation problem, where data is not moved from the operational system and translations are performed directly on it. The new version of the MIDST tool needs only to know the model and the schema of the source database and generates views on the operational system that transform the underlying data (stored in the source schema) according to the corresponding schema in the target model. Views are generated in an almost automatic way, on the basis of the Datalog rules for schema translation.

As stated by Melnik and Bernstein in [BM07], a midterm target of the model management studies is the development of a reusable component (a model management system) to be embedded in higher level applications in order to provide runtime solutions to data programmability problems, transparent to the user. We are following this direction and our next challenge is to provide contributions toward the development of a model-independent but model-aware approach to model management.

The ambitious idea is to develop a general model management platform that extends MIDST and is based on its principles. The metamodel approach is binding to achieve model-independance and model-awareness with all the advantages highlighted throughout the dissertation. The second key point is represented by Datalog. In the MIDST proposal we used it in order to define rules conceived to perform transformations of schemas. Exploiting completely its expressive power, we can implement also model management operators in Datalog. The intuition is that these operators could be generated (semi)automatically exploiting the conceptual structure of the supermodel and its corresponding fixed relational implementation, in such a way that if the supermodel is extended, the operators can be extended as well.

Appendices

A. Basic Translations

This appendix lists the set of basic translations used in our experiments, for the supermodel illustrated in Section 2.3 (and specifically in Figure 2.13).

1. eliminate multivalued structures of attributes in an abstract, by introducing new abstracts and foreign keys
2. eliminate (nested, monovalued) structures of attributes in an abstract, by flattening them
3. eliminate (nested, monovalued) structures of attributes in an aggregation, by flattening them
4. eliminate foreign keys involving abstracts, by introducing abstract attributes
5. eliminate abstract attributes, replacing them with foreign keys involving abstracts
6. eliminate lexicals of aggregations of abstracts, by moving them to abstracts (possibly new)
7. eliminate lexicals of binary aggregations of abstracts, by moving them to abstracts (possibly new)
8. eliminate many-to-many binary aggregations of abstracts, by introducing new abstracts and binary aggregations of abstracts
9. eliminate n-ary aggregations of abstracts, by introducing new abstracts and binary aggregations of abstracts
10. replace binary aggregations of abstracts with aggregations of abstracts

11. nest abstracts and abstract attributes within referencing abstracts
12. eliminate generalizations, by keeping the leaf abstracts and merging the other abstracts into them
13. eliminate generalizations, by keeping the root abstracts and merging the other abstracts into them
14. eliminate generalizations, by keeping all abstracts and relating them by means of (n-ary) aggregations of abstracts (that involve two abstracts)
15. eliminate generalizations, by keeping all abstracts and relating them by means of binary aggregations of abstracts
16. eliminate generalizations, by keeping all abstracts and relating them by means of abstract attributes
17. replace (one-to-many) binary aggregations with abstract attributes
18. replace abstract attributes with (one-to-many) binary aggregations
19. replace abstracts and binary (one-to-many) aggregations of them with aggregations (of lexicals) and foreign keys
20. replace abstracts and abstract attributes with aggregations (of lexicals) and foreign keys
21. replace aggregations (of lexicals) and foreign keys with abstracts and binary aggregations of them
22. replace aggregations (of lexicals) and foreign keys with abstracts and foreign keys over them
23. replace aggregations (of lexicals) and foreign keys with abstracts and abstract attributes

We briefly comment on the completeness of this set of rules with respect to the models used in our experiments showing that Assumptions 1 and 2 of Chapter 3 are satisfied.

With respect to Assumption 1, we need to show that for each pair of families we have a translation from one to the other, and viceversa. This is shown in the table in Figure A.1, where each cell indicates the translation or the sequence of translations needed to go from the model associated with the row to the

	ER	B-ER	OO	OR	Rel	XSD
ER	-	9	9,17	9,17	9,19	9,17,11
B-ER	10	-	17	17	19	17,11
OO	18,10	18	-	-	20	5,11
OR	18,10	18	23	-	20	11
Rel	21,10	21	23	-	-	22
XSD	4,18,10	4,18	4	1	20	-

Figure A.1: Translations between families.

model associated with the column. It is worth noting that, in some cases, the cell contains two or even three basic translations; then, with reference to the discussion we made in Section 3.6 after Assumption 1, we can think that the system has a composition of these translations defined as a basic one.

With respect to Assumption 2, it suffices to list the minimal models in each family and the sequences of reductions that form a translation from the progenitor of the family to them, as follows:

Entity-relationship - one minimal model with no generalizations and no attributes on aggregations; the reduction is 6, 14 (or 12 or 13);

Binary entity-relationship - one minimal model again, with no generalizations, no lexicals of aggregations and no many-to-many aggregations; the reduction is 7, 15 (or 12 or 13), 8;

Object-oriented - one minimal model, with no generalizations and no structures of attributes: the reduction is 2, 16 (or 12 or 13);

Object-relational - three minimal models, the progenitors of the relational model (with a reduction 2, 3, 16 (or 12 or 13), 20) and of the OO model (4, 23) and a model with no structures of attributes in abstracts and no generalizations, for which the reduction is 2, 16 (or 12 or 13);

Relational - the progenitor coincides with the minimal model;

XSD - one minimal model, where structures of attributes are only monovalued; the reduction is just translation 1.

B. Rules

This appendix lists the complete set of rules for the running example used in Chapter 3. Apart from the names of constructs, it is the same set of rules of step $\mathbf{P}_2$ of Section 2.1 and Figure 2.13.

Rule $R_{2,1}$: *copy entities*

ENTITY(OID: #entity_0(eOid), sOID: tgt, Name: n)
←
ENTITY(OID: eOid, sOID: src, Name: n)

Signature:

$H_{2,1} = \langle \text{E}(true) \rangle$
$B_{2,1} = \langle \text{E}(true) \rangle$
$\text{MAP}_{2,1} = \langle \rangle$

Rule $R_{2,2}$: *copy attributes of entities*

ATTRIBUTEOFENTITY(OID: #attributeOfEntity_0(aOid), sOID: tgt,
Name: n, isKey: isK, isNullable: isN, EntityOID: #entity_0(eOid))
←
ATTRIBUTEOFENTITY(OID: aOid, sOID: src,
Name: n, isKey: isK, isNullable: isN, EntityOID: eOid)

Signature:

$H_{2,2} = \langle \text{A}(true) \rangle$
$B_{2,2} = \langle \text{A}(true) \rangle$
$\text{MAP}_{2,2} = \langle \text{K} : \text{A}(\text{K}), \text{N} : \text{A}(\text{N}) \rangle$

Rule $R_{2,3}$: *copy one-to-one and one-to-many relationships* [1]

RELATIONSHIP(OID: #relationship_0(rOid), sOID: tgt, Name: n,
role1: r1, isOptional1: isOpt1, isFunctional1: true, isIdentified: isId,
role2: r2, isOptional2: isOpt2, isFunctional2: isFunct2,
Entity1: #entity_0(eOid1), Entity2: #entity_0(eOid2))
←
RELATIONSHIP(OID: rOid, sOID: src, Name: n,
role1: r1, isOptional1: isOpt1, isFunctional1: true, isIdentified: isId,
role2: r2, isOptional2: isOpt2, isFunctional2: isFunct2,
Entity1: eOid1, Entity2: eOid2)

Signature:

$H_{2,3} = \langle \text{R}(\text{F}_1) \rangle$
$B_{2,3} = \langle \text{R}(\text{F}_1) \rangle$
$\text{MAP}_{2,3} = \langle \text{O}_1 : \text{R}(\text{O}_1), \text{I} : \text{R}(\text{I}), \text{O}_2 : \text{R}(\text{O}_2), \text{F}_2 : \text{R}(\text{F}_2) \rangle$

Rule $R_{2,4}$: *copy attributes of one-to-one and one-to-many relationships*

ATTRIBUTEOFRELATIONSHIP(OID: #attributeOfRelationship_0(arOid),
sOID: tgt, Name: n, isNullable: isN,
RelationshipOID: #relationship_0(rOid))
←
ATTRIBUTEOFRELATIONSHIP(OID: arOid,
sOID: src, Name: n, isNullable: isN, RelationshipOID: rOid),
RELATIONSHIP(OID: rOid, sOID: src, isFunctional1: true)

Signature:

$H_{2,4} = \langle \text{AR}(\mathit{true}) \rangle$
$B_{2,4} = \langle \text{AR}(\mathit{true}), \text{R}(\text{F}_1) \rangle$
$\text{MAP}_{2,4} = \langle \text{N} : \text{AR}(\text{N}) \rangle$

[1] We recall that, without loss of generality, we assume that in a one-to-many relationship, it is the first entity that has a functional role, and so $\text{F}_1 = \mathit{true}$ and $\text{F}_2 = \mathit{false}$.

Rule $R_{2,5}$: *generate an entity for each many-to-many relationship*

ENTITY(OID: #entity_1(rOid), sOID: tgt, Name: n)
←
RELATIONSHIP(OID: rOid, sOID: src, Name: n,
isFunctional1: `false`, isFunctional2: `false`)

Signature:

$H_{2,5} = \langle \text{E}(true) \rangle$
$B_{2,5} = \langle \text{R}(\neg \text{F}_1 \wedge \neg \text{F}_2) \rangle$
$\text{MAP}_{2,5} = \langle \rangle$

Rule $R_{2,6}$: *for each entity generated by $R_{2,5}$, generate a relationship between it and the copy of the first entity involved in the many-to-many relationship*

RELATIONSHIP(OID: #relationship_1(eOid,rOid), sOID: tgt, Name: eN+rN,
isOptional1: `false`, isFunctional1: `true`, isIdentified: `true`,
isOptional2: isOpt, isFunctional2: `false`,
Entity1: #entity_1(rOid), Entity2: #entity_0(eOid))
←
RELATIONSHIP(OID: rOid, sOID: src, Name: rN,
isOptional1: isOpt, isFunctional1: `false`, isFunctional2: `false`,
Entity1: eOid),
ENTITY(OID: eOid, sOID: src, Name: eN)

Signature:

$H_{2,6} = \langle \text{R}(\neg \text{O}_1 \wedge \text{F}_1 \wedge \text{I} \wedge \neg \text{F}_2) \rangle$
$B_{2,6} = \langle \text{R}(\neg \text{F}_1 \wedge \neg \text{F}_2), \text{E}(true) \rangle$
$\text{MAP}_{2,6} = \langle \text{O}_2 : \text{R}(\text{O}_1) \rangle$

Rule $R_{2,7}$: *for each entity generated by* $R_{2,5}$*, generate a relationship between it and the copy of the second entity involved in the many-to-many relationship*

RELATIONSHIP(OID: #relationship_1(eOid,rOid), sOID: tgt, Name: eN+rN,
isOptional1: `false`, isFunctional1: `true`, isIdentified: `true`,
isOptional2: isOpt, isFunctional2: `false`,
Entity1: #entity_1(rOid), Entity2: #entity_0(eOid))
←
RELATIONSHIP(OID: rOid, sOID: src, Name: rN,
isFunctional1: `false`, isOptional2: isOpt, isFunctional2: `false`,
Entity2: eOid),
ENTITY(OID: eOid, sOID: src, Name: eN)

Signature:

$H_{2,7} = \langle \mathrm{R}(\neg \mathrm{O}_1 \wedge \mathrm{F}_1 \wedge \mathrm{I} \wedge \neg \mathrm{F}_2) \rangle$
$B_{2,7} = \langle \mathrm{R}(\neg \mathrm{F}_1 \wedge \neg \mathrm{F}_2), \mathrm{E}(\mathit{true}) \rangle$
$\mathrm{MAP}_{2,7} = \langle \mathrm{O}_2 : \mathrm{R}(\mathrm{O}_2) \rangle$

Rule $R_{2,8}$: *for each attribute of each many-to-many relationship, generate an attribute for the entity generated by* $R_{2,5}$

ATTRIBUTEOFENTITY(OID: #attributeOfEntity_1(arOid), sOID: tgt,
Name: n, isKey: `false`, isNullable: isN, EnitityOID: #entity_1(rOid))
←
ATTRIBUTEOFRELATIONSHIP(OID: arOid, sOID: src,
Name: n, isNullable: isN, RelationshipOID: rOid),
RELATIONSHIP(OID: rOid, sOID: src,
isFunctional1: `false`, isFunctional2: `false`)

Signature:

$H_{2,8} = \langle \mathrm{A}(\neg \mathrm{K}) \rangle$
$B_{2,8} = \langle \mathrm{AR}(\mathit{true}), \mathrm{R}(\neg \mathrm{F}_1 \wedge \neg \mathrm{F}_2) \rangle$
$\mathrm{MAP}_{2,8} = \langle \mathrm{N} : \mathrm{AR}(\mathrm{N}) \rangle$

Bibliography

[ABBG08a] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, and Giorgio Gianforme. From schema and model translation to a model management system. In *BNCOD*, pages 227–240. Springer, 2008.

[ABBG08b] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, and Giorgio Gianforme. A platform for model-independent solutions to model management problems. In *SEBD*, pages 310–317, 2008.

[ACB06] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Model-independent schema and data translation. In *EDBT*, pages 368–385. Springer, 2006.

[ACG07] Paolo Atzeni, Paolo Cappellari, and Giorgio Gianforme. MIDST: model independent schema and data translation. In *SIGMOD Conference*, pages 1134–1136. ACM, 2007.

[ACM02] Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and translation for heterogeneous data. *Theor. Comput. Sci.*, 275(1-2):179–213, 2002.

[ACT+08] Paolo Atzeni, Paolo Cappellari, Riccardo Torlone, Philip A. Bernstein, and Giorgio Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.

[AD06] Paolo Atzeni and Pierluigi Del Nostro. Management of heterogeneity in the semantic web. In *ICDE Workshops*, pages 60–63. IEEE Computer Society, 2006.

[AH88] Serge Abiteboul and Richard Hull. Restructuring hierarchical database objects. *Theor. Comput. Sci.*, 62(1-2):3–38, 1988.

[AKM98] Foto N. Afrati, Isambo Karali, and Theodoros Mitakos. On inheritance in object oriented datalog. In *IADT*, pages 280–289, 1998.

[ALUW93] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. *SIGMOD Rec.*, 22(2):32–41, 1993.

[AT93] Paolo Atzeni and Riccardo Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Inf. Syst.*, 18(6):349–362, 1993.

[AT96] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *EDBT*, pages 79–95. Springer, 1996.

[BBDV03] J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The ATL transformation-based model management framework. Research Report 03.08, IRIN, Université de Nantes, 2003.

[BD03] Shawn Bowers and Lois M. L. Delcambre. The Uni-Level Description: A uniform framework for representing information in multiple data models. In *ER*, pages 45–58. Springer, 2003.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.

[BHP00] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[BI97] Anthony J. Bonner and Tomasz Imielinski. Reusing and modifying rulebases by predicate substitution. *J. Comput. Syst. Sci.*, 54(1):136–166, 1997.

[BM05] Michael Boyd and Peter McBrien. Comparing and transforming between data models via an intermediate hypergraph data model. *J. Data Semantics IV*, pages 69–109, 2005.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12. ACM, 2007.

[BMM05] Philip A. Bernstein, Sergey Melnik, and Peter Mork. Interactive schema translation with instance-level mappings. In *VLDB*, pages 1283–1286, 2005.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *SIGMOD Conference*, pages 177–188. ACM, 1998.

[CK86] Stavros S. Cosmadakis and Paris C. Kanellakis. Functional and inclusion dependencies. *Advances in Computing Research*, 3:163–184, 1986.

[CR03] Kajal T. Claypool and Elke A. Rundensteiner. Sangam: A transformation modeling framework. In *DASFAA*, pages 47–54, 2003.

[CRZ+01] Kajal T. Claypool, Elke A. Rundensteiner, Xin Zhang, Hong Su, Harumi A. Kuno, Wang-Chien Lee, and Gail Mitchell. Gangam - A solution to support multiple data models, their mappings and maintenance. In *SIGMOD Conference*, page 606. ACM, 2001.

[DK97] Susan B. Davidson and Anthony Kosky. WOL: A language for database transformations and constraints. In *ICDE*, pages 55–65, 1997.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, 1985.

[DT93] Gillian Dobbie and Rodney W. Topor. A model for sets and multiple inheritance in deductive object-oriented systems. In *DOOD*, pages 473–488, 1993.

[DT94] Gillian Dobbie and Rodney W. Topor. Representing inheritance and overriding in datalog. *Computers and Artificial Intelligence*, 13:133–158, 1994.

[DT06] Roberto De Virgilio and Riccardo Torlone. Modeling heterogeneous context information in adaptive web based applications. In *ICWE*, pages 56–63. ACM, 2006.

[FKMP03] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224. Springer, 2003.

[FKP05] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[GN08] Georg Gottlob and Alan Nash. Efficient core computation in data exchange. *J. ACM*, 55(2), 2008.

[Haa07] Laura M. Haas. Beauty and the beast: The theory and practice of information integration. In *ICDT*, pages 28–43. Springer, 2007.

[HAB+05] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael J. Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal, and Vishal Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD Conference*, pages 778–787. ACM, 2005.

[Hai96] Jean-Luc Hainaut. Specification preservation in schema transformations - application to semantics and statistics. *Data Knowl. Eng.*, 19(2):99–134, 1996.

[Hai06] Jean-Luc Hainaut. The transformational approach to database engineering. In *GTTSE*, pages 95–143. Springer, 2006.

[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD Conference*, pages 805–810. ACM, 2005.

[Hib] Hibernate. http://www.hibernate.org/.

[HK87] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.

[Hul86] Richard Hull. Relative information capacity of simple relational database schemata. *SIAM J. Comput.*, 15(3):856–886, 1986.

[HY90] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, 1990.

[Jam97] Hasan M. Jamil. Implementing abstract objects with inheritance in datalog$^{\text{neg}}$. In *VLDB*, pages 56–65, 1997.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[LDL02] Mengchi Liu, Gillian Dobbie, and Tok Wang Ling. A logical foundation for deductive object-oriented databases. *ACM Trans. Database Syst.*, 27(1):117–151, 2002.

[LS08] Leonid Libkin and Cristina Sirangelo. Data exchange and schema mappings in open and closed worlds. In *PODS*, pages 139–148. ACM, 2008.

[MBM07] Peter Mork, Philip A. Bernstein, and Sergey Melnik. Teaching a schema translator to produce O/R views. In *ER*, pages 102–119. Springer, 2007.

[McG59] William C. McGee. Generalization: Key to successful electronic data processing. *J. ACM*, 6(1):1–23, 1959.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. Springer, 2004.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[MIR93] Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB*, pages 120–133, 1993.

[MO84] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for PROLOG. *Artif. Intell.*, 23(3):295–307, 1984.

[MP99] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *CAiSE*, pages 333–348. Springer, 1999.

[MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.

[PA07] Stefano Paolozzi and Paolo Atzeni. Interoperability for semantic annotations. In *DEXA Workshops*, pages 445–449. IEEE Computer Society, 2007.

[PM98] Alexandra Poulovassilis and Peter McBrien. A general formal framework for schema transformation. *Data Knowl. Eng.*, 28(1):47–71, 1998.

[PT05] Paolo Papotti and Riccardo Torlone. Heterogeneous data translation through XML conversion. *J. Web Eng.*, 4(3):189–204, 2005.

[PVM+02] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating Web data. In *VLDB*, pages 598–609, 2002.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[SHT+77] Nan C. Shu, Barron C. Housel, R. W. Taylor, Sakti P. Ghosh, and Vincent Y. Lum. EXPRESS: A data EXtraction, Processing, and REStructuring System. *ACM Trans. Database Syst.*, 2(2):134–174, 1977.

[SZK04] Guanglei Song, Kang Zhang, and Jun Kong. Model management through graph transformations. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 75–82. IEEE Computer Society, 2004.

[Tar55] Alfred Tarski. A lattice-theorethic Fixpoint Theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TMB08] James F. Terwilliger, Sergey Melnik, and Philip A. Bernstein. Language-integrated querying of XML data in SQL Server. *PVLDB*, 1(2):1396–1399, 2008.

[UW97] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall, 1997.

[VMP03] Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.