A runtime approach to model-generic translation of schema and data[☆]

Paolo Atzeni[*], Luigi Bellomarini, Francesca Bugiotti, Fabrizio Celli, Giorgio Gianforme

Dipartimento di Informatica e Automazione, Università Roma Tre, Via della Vasca Navale 79 — 00146 Roma Italy

Abstract

To support heterogeneity is a major requirement in current approaches to integration and transformation of data. This paper proposes a new approach to the translation of schema and data from one data model to another, and we illustrate its implementation in the tool MIDST-RT.

We leverage on our previous work on MIDST, a platform conceived to perform translations in an off-line fashion. In such an approach, the source database (both schema and data) is imported into a repository, where it is stored in 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 (again both schema and data) is exported into the operational system.

Here we illustrate a new, lightweight approach where the database is not imported. MIDST-RT needs only to know the schema of the source database and the model of the target one, and generates views on the operational system that expose the underlying data 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 proposed solution can be applied to different scenarios, which include data and application migration, data interchange, and object-to-relational mapping between applications and databases.

Keywords: model management, data model, schema translation

1. Introduction

The problem of translating schemas between data models is acquiring progressive significance in heterogeneous environments and has received attention in many projects, including MIDST and its predecessors (Atzeni et al. [5, 6]), DBMain (Hainaut [18]), AutoMed and its predecessors (McBrien, Smith and Poulovassilis [21, 27]), Chameleon (Papotti and Torlone [26]), and the work by Mork, Bernstein and Melnik [25]. This is motivated by the fact that 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. Moreover, there is currently a significant adoption of XML repositories that manage native XML data. This fact has increased the heterogeneity of representations.

In general, the presence of several coexisting models introduces the need for translation techniques and tools. In fact, *Model Management* (Bernstein [9]), a high-level approach to meta data management that offers high-level operators to deal with schemas and mappings, includes an operator (called *ModelGen*) for translating schemas from a model to another.

We have recently proposed MIDST [5], a platform for model-independent schema and data translation in order to provide a paradigm to face issues of this kind, and to implement the ModelGen operator. MIDST adopts a metalevel approach towards translations by performing them in the context of a universal model (called

[☆]This paper is an extended version of a conference paper [2]

[*]Corresponding author

Email addresses: atzeni@dia.uniroma3.it (Paolo Atzeni), bellomarini@yahoo.it (Luigi Bellomarini), franbugiotti@yahoo.it (Francesca Bugiotti), fabrizio.celli.1985@gmail.com (Fabrizio Celli), giorgio.gianforme@gmail.com (Giorgio Gianforme)

the *supermodel*[1]), which allows for the management of schemas in many different data models. The tool has been experimented with many models, including relational, object-oriented (OO), object-relational, entity-relationship (ER), XML-based, each in many different variants. Translations are organized according to the following pattern. First, the source database is imported into the tool and described in its dictionary in terms of the supermodel. Then, the translation is performed by means of a sequence of elementary steps (rules), each dealing with a specific aspect to be eliminated or transformed. Finally, the obtained database is exported into the target operational system.[2] This 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 [10], this approach is rather inefficient. 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 platform, MIDST-RT: it is based on a runtime approach to the translation problem, where data is not moved from the operational system and translations are performed directly on it. What the user obtains at runtime is a set of views (defining the target schema) conforming to the target model. 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 based on the supermodel, obtained as composition of elementary ones. However, the management of the involved data is completely different. In fact, 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 the original, off-line version of MIDST, a lot of effort was devoted to import/export modules, whereas all translations were developed in Datalog. In a runtime approach, instead, 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 (involving for example, SQL, SQL/XML, XQuery, and combinations of them). Also, different dialects of the various languages may exist, and our techniques need to cope with them.

In order to handle the heterogeneity of the involved languages, we propose an approach that, after a preliminary abstract representation, first generates views organized according to the target model, but independent of the specific languages, and then actually concretizes them into executable statements on the basis of the specific language supported by the operational system.

In this paper 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 model and of the relational one, and their extension with XML.

The paper is organized as follows. Section 2 is an explanation of scenarios in which our work can be placed. Section 3 is an overview of the work and introduces a running example. Section 4 gives the necessary background on MIDST. Sections 5 and 6 present our approach, by first illustrating its principles and then the technical details. In Section 7 we show how our approach can contribute to the scenarios discussed in Section 2, with actual samples of code. In Section 8 we discuss some related work. Finally in Section 9 we draw our conclusions.

2. Motivating scenarios

The main result of our work is the ability to define views over the operational system, in order to execute a light transformation that needs only to import the source schema in our dictionary. The meaning of “view” depends on the operational system: for an RDBMS or an ORDBMS, a “view” is a stored query leading to a virtual table that shows data in a different way; for an object-oriented language, a “view” is a set of objects that reference each other; for the Web, a “view” can be

[1]The use of a universal model has been adopted, in different forms, by the various projects mentioned above [5, 6, 18, 21, 25, 26, 27].

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

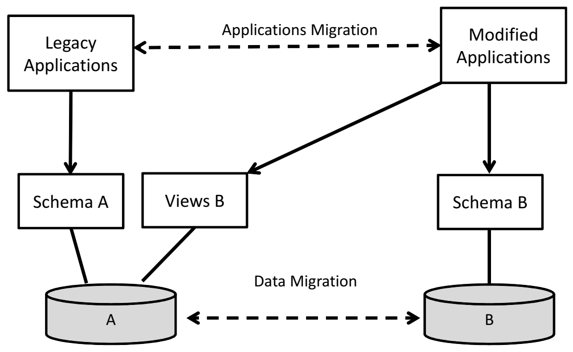


Figure 1: The data migration scenario

an XML document that shows data extracted from a relational database.

In this section we briefly describe some representative problems in this context and explain how MIDST-RT can support them.

2.1. Data and application migration

"Data-migration" is a process of data movement between different storage systems (and different technologies) caused by changes in the technology or in the organization of data. In order to obtain an effective migration, it is important to keep in mind that applications have to be migrated as well. As argued by Brodie and Stonebraker [12], migration needs to be incremental, and some legacy functions should coexist with the newly developed ones. MIDST-RT can support this, by offering two different interfaces to the same database. Let us consider the following practical problem in order to see how to solve it using MIDST-RT:

- let A be an ORDBMS used by some applications of an enterprise;
- the enterprise decides to change its commercial partner, so it will use B, another DBMS (with a different version of the object-relational model or with the relational one);
- given the actual differences between systems A and B, the original schema is not compatible with B and so current applications do not work with it. New applications need to be developed and tested, without interrupting operation.

Figure 1 explains how we can support this problem:

1. MIDST-RT can generate a set of views over system A that show a schema compatible with the specification of system B. In such a way, the enterprise can gradually update its applications: the modified components will use the new schema shown by the set of generated views, while the other ones will use the original schema;
2. then, after all applications have been modified, the data can be actually migrated, by executing the same queries that define the views, this time materializing their results.

It is important to observe that this approach would support the intermediate phase where the old data exist together with the new schema, while off-line or data-exchange approaches [5, 17, 18, 27] would support only the last step.

2.2. XML for data interchange

XML is widely used in the process of data movement between applications or DBMSs, especially via a network. Thus, a user can benefit from the usage of XML formats for different reasons: she can migrate a database using the network, she can allow the communication between incompatible systems, she can use such an XML file as an input for an application, and so on. MIDST-RT is able to create an XML view over a relational or object-relational database. We talk about an "XML view" because we perform a runtime translation: first, we do not import data into MIDST-RT, but only the schema of the source database; then, we produce executable statements, so the XML file is always up to date even when the source database is frequently updated. As an example, we can consider the following simple scenario. A user has a relational database and she wants to send data to a Java application through the network. She needs to produce an XML document that contains such a data, and then she needs a framework for the marshalling/unmarshalling of the document. MIDST-RT helps the creation of the XML document in a flexible way with three interesting features which differentiate this approach from the data exchange one [24]: the dynamic generation, the handling of various source models and the possibility of customizing translations. Such a document will be processed by the destination application, possibly by taking advantage of the document schema (XSD).

In Subsection 7.2 we will show a concrete example for this scenario by using MIDST-RT.

2.3. O/R mapping

The need for mapping object-oriented applications and relational databases arises in many contexts [23, 25], and various technologies have been developed to support it. The problem can be seen in two major forms:

(i) given a relational schema (or an object-relational one) it is convenient to produce object-oriented wrappers, that is, software artifacts that ensure an object-oriented access to a relational database; (ii) given a set of classes that define objects in an object-oriented language it is often useful to obtain the schema of a relational database for the persistent storage of the corresponding data.

As far as the first scenario concerns, MIDST-RT can contribute with the generation of wrappers, which can be seen as a form of views, with benefits in the flexibility both in the source model (many variations of the relational and object-relational ones) and in the target model (different languages and programming conventions), as well as in the mapping (which can be customized, for example for performance issues). We will see in Subsection 7.3 a concrete example for this scenario.

In the second scenario we want to automatically generate a database from a set of classes. Even with respect to existing technologies that support this problem (such as JDO [20], Hibernate [8], or ADO.NET Entity Framework [23]), or in combination with them, MIDST-RT can provide specific benefits:

- flexibility in the database management, thanks to the knowledge of different representations of the object-relational model;
- flexibility in the definition of the target schema, thanks to the possibility of transforming the source schema before the creation of the mapping.

At the moment, we have not implemented this scenario in MIDST-RT, but we are working on it, since it is an important application for our tool. This can be realized by introducing an object-oriented importer that translates a set of Java or C# classes into MIDST-RT internal representation.

3. Overview

As we said in the Introduction, the starting point for this work is MIDST [5], a platform for model-independent schema and data translation based on a meta-level approach over a wide range of data models. In MIDST the various data models are described in terms of a small set of *basic constructs*. Schemas of the various models are described within a common model, the *supermodel*, which generalizes all of them, as it involves all the basic constructs. Translations are performed within the supermodel and are obtained as a composition of elementary steps (which we call *basic translations*). In the current implementation, they are specified in Datalog.

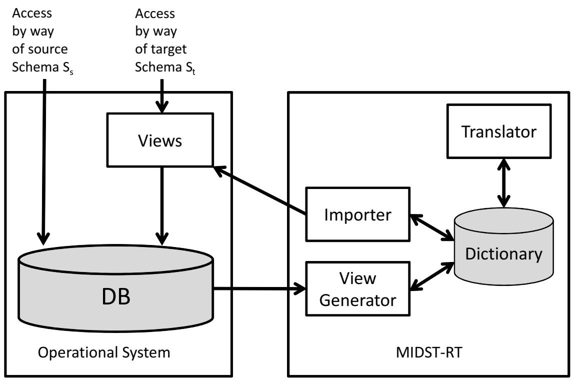


Figure 2: The runtime translation procedure

In this paper we describe a new approach and an enhanced version of our platform (called MIDST-RT) which enables the creation of executable statements generating views in the operational system. Let us illustrate the runtime translation procedure, by following the main steps it involves, with the help of Figure 2:

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-RT, and specifically in its dictionary, where it is described in supermodel terms;
3. MIDST-RT selects the appropriate translation **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 **T** is applied to S_s, still within the tool, 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 **T**, MIDST-RT generates views in the specific language available in the operational system;
6. MIDST-RT exports and executes the produced statements over the operational system, in order to create a set of views that perform the translation.

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. Step 6 is a revised form of the export step of MIDST: it exports and executes the view creation statements, rather than exporting the data.

As a running example, let us consider an environment where some applications interact with an object-relational database. Then, assume we want to write an

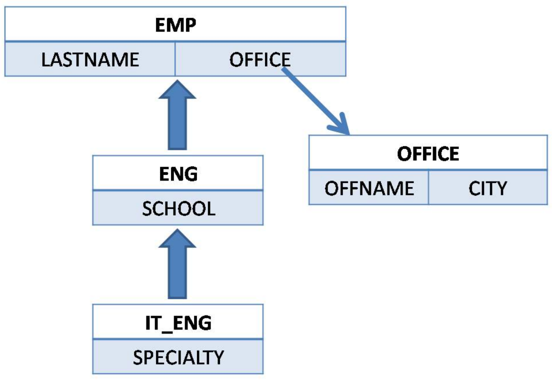


Figure 3: An object-relational schema

application that uses the same data but interacting with a relational data model. We can consider a version of the OR model that has the following features:[3] tables, typed tables (i.e. tables with identifiers), references and foreign keys 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 the new application program.

A concrete case for this example involves the OR schema sketched in Figure 3. The boxes are typed tables: employee (EMP) is a generalization for engineer (ENG), which is in turn a generalization for IT-engineer (IT_ENG); office (OFFICE) 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 with the foreign keys suggested by the names of the attributes (details omitted for the sake of space):[4]

OFFICE (OFFICE_OID, offName, city)
EMP (EMP_OID, lastName, OFFICE_OID_fk)
ENG (ENG_OID, school, EMP_OID_fk)
IT_ENG (IT_ENG_OID, specialty, ENG_OID_fk)

Given the schema in Figure 3, our tool first imports it into its dictionary. Then, given the specification of the target model (the relational one), the tool automatically selects an appropriate schema-level translation, which is a sequence of basic translations, each specified by means of a Datalog program. The user can customize the proposed sequence, in order to execute a personalized translation. The customization may include the addition, removal or reorder of the basic translations of the sequence chosen by the tool.

In the example, the schema-level translation performs the following tasks: it first eliminates the multiple levels of generalizations (in the example, the one between ENG and EMP and the one between IT_ENG and ENG) and then transforms the typed tables (all tables in the source) into value-based tables. In MIDST this would be done in four steps, with a first Datalog program for the elimination of generalizations and a fourth one for the transformation of typed tables into value-based ones with two auxiliary intermediate steps for the introduction of keys and the replacement of references with foreign keys, respectively. The tool we propose here generates a set of view statements for each of these Datalog programs.

The following is a sketch of one of the view definitions generated in the first step:

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

We use the name ENG_A to distinguish the new version from the original one.[5] View ENG_A extends ENG 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 Section 5 and 6, we will see in detail how we produce views of this kind, by showing the principles, the complete description of the algorithm, and the specific details that are needed for the SQL statements. The discussion will need some background on MIDST, which is given in the next section.

4. Translations in MIDST

MIDST (Atzeni et al. [5]) is based on the idea that the constructs of the various models correspond to a limited set of types of constructs and it is therefore possible to define a "universal model", called the *supermodel*, that includes, possibly renamed, all the constructs of interest. Each model, then, is a specialization of the supermodel, and each schema in any model is also a

[3]This is just a possible version of the OR model, and our tool can handle many others.

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

[5]In the technical sections of the paper, we use this convention, with the suffix, when needed for the sake of readability and conciseness. In the tool, names are repeated and distinguished by using different schema names.

schema in the super- model. Therefore, translations from a model to another can be performed within the supermodel.

In order to support the process of generating translations, similar models are grouped into families [5]. Figure 4 (taken, with minor variations, from [5]) reports a list of MIDST generic constructs and shows examples of their use in most common models managed by our tool: rows correspond to constructs, columns to families of models, while each cell shows the name of a construct in a specific family.

In the tool, each construct has a name, a set of properties (which allow the specification of variants and details) and a set of references (which allow constructs to refer to each other). Each construct is provided with a unique identifier (OID) and references are then based on OIDs.

Let us illustrate the main constructs by means of the running example, the object-relational schema of Figure 3. Each of the typed tables (EMP, ENG, IT_ENG and OFFICE) is seen as an Abstract in the supermodel. Then, each column of the typed tables (SPECIALTY for IT_ENG, SCHOOL for ENG, LASTNAME for EMP, OFFNAME and CITY for OFFICE) is a Lexical and is related to the corresponding Abstract. Similarly, reference fields (OFFICE in this case) are modeled as AbstractAttributes (of EMP in the example). Finally, Generalizations appear in the supermodel: ENG is a child of EMP and IT_ENG is a child of ENG.

Each translation step in MIDST is specified as a Datalog program, which is a set of Datalog rules. More precisely, we use a variant of Datalog with "value invention" [13, 19], where values for new OIDs are generated. We use Skolem functors to generate OIDs. For example, the following rule translates an Abstract (a typed table in an OR model) into an Aggregation (a simple table):

```
Aggregation ( OID: SK1(oid),
        Name: name )
<- Abstract ( OID: oid,
        Name: name );
```

Notice the use of a Skolem functor, SK1 in the example, which, given the OID of an Abstract, produces a corresponding OID for an Aggregation. We use Skolem functions to generate new identifiers for constructs, given a set of input parameters, as well as for referring to them whenever needed, given the same set of parameters. Skolem functions are injective. So, in this case SK1 will generate a different OID, and so a different new Aggregation, for each Abstract in the source schema. For a given target construct many functors can be defined (denoted by numeric suffixes in the examples), each taking different parameters in dependence on the source constructs the target one is generated from. As a consequence, in order to guarantee the uniqueness of the OIDs, the ranges of the Skolem functions are disjoint. Other functors for Aggregation (which exist in the tool, but not shown in this paper) generate a different set of OIDs.

Translations taking place in real scenarios require several Datalog programs to specify the transformation of each construct. We pursue a modular approach and decompose translations into simple steps (each returning a coherent schema of a specific model that is then used by the subsequent step). This is done by means of a library of Datalog programs implementing elementary steps and of an inference engine which can determine the appropriate sequence of steps to be applied.

With reference to our running example, let us take into account the translation from the version of the OR model we are considering towards a classical relational model. In MIDST [5] this could be done as a process in four steps:

A elimination of generalizations;
B generation of identifiers for typed tables;
C elimination of reference columns with the introduction of value-based columns and foreign keys;
D transformation of typed tables into tables.

In each translation step we copy, with a simple "copy rule", all the constructs that are not modified. For example, in order to copy an Abstract, we have the *copy-abstract* rule (R_1):

```
R1   Abstract ( OID: SK2(oid),
             Name: name )
     <- Abstract ( OID: oid,
             Name: name );
```

The tool has a copy rule for each construct automatically generated out of the definition of the supermodel.

When actual transformations are needed, rules are more complex. Let us illustrate the main points.

As for Step A, there are various ways to eliminate generalizations. Let us refer to the one that keeps 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. In terms of MIDST constructs, this means that, for each Generalization between two Abstracts, an AbstractAttribute (a reference column) referring to the parent Abstract must be added to the child

Metaconstruct	**relational**	**OR**	**OO**	**ER**
Abstract	-	typed table	class	entity
Lexical	column	column	field	attribute
BinaryAggregationOfAbstracts	-	-	-	binary relationship
AbstractAttribute	-	reference	reference field	-
Generalization	-	generalization	generalization	generalization
Aggregation	table	table	-	-
ForeignKey	foreign key	foreign key	-	-
StructOfAttributes	-	structured column	structured field	-

Figure 4: Simplified representation of MIDST metamodel

Abstract. The Datalog rule implementing this last step is the following (in the following denoted as R_4, or *elim-gen*):

```
R4 AbstractAttribute (
          OID: SK3(genOID, childOID),
          Name: name,
          IsNullable: "false",
          abstractOID: SK2(childOID),
          abstractToOID: SK2(parentOID) )
     <-  Generalization ( OID: genOID,
          parentAbstractOID: parentOID,
          childAbstractOID: childOID ),
        Abstract ( OID: parentOID, Name: name),
        Abstract ( OID: childOID);
```

Let us observe that the Skolem functor SK3 has two arguments, because we need to create a new AbstractAttribute for each Generalization and for each of its child Abstracts: indeed a Generalization may have various children and an Abstract may be a child in various Generalizations.

In order to obtain a coherent schema we also need to copy all the constructs in the schema, other than generalizations. This is done by the *copy-abstract* rule (R_1) we have seen above, together with similar ones for the other constructs, *copy-lexical* (R_2) and *copy-abstractAttribute* (R_3) reported below.

```
R2    Lexical ( OID: SK7(lexOID),
                Name: name,
                IsNullable: isN,
                IsIdentifier: isI,
                abstractOID: SK2(absOID) )
      <- Lexical ( OID: lexOID,
                   Name: name,
                   IsIdentifier: isI,
                   IsNullable: isN,
                   abstractOID: absOID ),
         Abstract( OID:absOID );


R3    AbstractAttribute( OID: SK8(oid),
                   Name: name,
                   isNullable: isN,
                   abstractToOID: SK2(absToOID),
                   abstractOID: SK2(absOID) )
         <- AbstractAttribute (
                   OID: oid,
                   Name: name,
                   isNullable: isN,
                   abstractToOID: absToOID,
                   abstractOID: absOID ),
            Abstract( OID:absOID ),
            Abstract( OID:absToOID );
```

Step B is needed because it is not guaranteed that typed tables (in the OR model) have key attributes, whereas, in order to transform references into value-based correspondences (subsequent Step C), keys are a precondition. The following Datalog rule (R_5), where the “!” character denotes a negation, implements this strategy: for each Abstract without any identifier, it generates a new key Lexical for it.

```
R5   Lexical ( OID: SK4(absOID),
              Name: name + "_OID",
              IsNullable: "false",
              IsIdentifier: "true",
              Type: "integer",
              abstractOID: SK2(absOID) )
     <- Abstract ( OID: absOID,
              Name: name ),
        ! Lexical ( IsIdentifier: "true",
              abstractOID: absOID );
```

As in the previous step, we need copy rules for all the constructs in the model (the same as above, R_1, R_2, R_3).

Step C replaces reference columns with value-based ones and connects them to the target table with a referential integrity constraint. The following rule (R_6) specifies this: for each AbstractAttribute (reference), it replicates the key Lexicals of the referred typed table into the referring one.

```
R6   Lexical ( OID: SK5(oid,lexOID),
              Name: lexName,
              IsIdentifier: "false",
              Type: type,
              abstractOID: SK2(absOID) )
     <- AbstractAttribute ( OID: oid,
              abstractOID: absOID,
              abstractToOID: absToOID),
         Lexical ( OID: lexOID,
              Name: lexName,
              abstractOID: absToOID,
              IsIdentifier: "true",
              Type: type ),
         Abstract( OID:absOID ),
         Abstract( OID:absToOID );
```

Here we just need the application of two copy rules (R_1 and R_2).

Finally, in Step D, typed tables are eliminated and this is simply performed by means of two Datalog rules. The first translates Abstracts into Aggregations (R_7), the second transforms Lexicals referring to Abstracts into Lexicals referring to Aggregations (R_8). We omit R_7 and R_8 for sake of space, as they would not add much to the discussion.

With respect to the running example of Figure 3, we have the following:

- Step A eliminates the hierarchies, hence connects ENG to EMP with a reference and IT_ENG to ENG with another reference;
- Step B creates an identifier for each of the typed tables: EMP_OID for EMP, ENG_OID for ENG, and so on;
- in Step C, references are translated into value-based correspondences: a new Lexical EMP_OID_fk is added to ENG, with foreign key constraint towards the identifier EMP_OID of EMP; similarly OFFICE_OID_fk is added to EMP and ENG_OID_fk to IT_ENG, each with the appropriate foreign key;
- finally, Step D performs the actual translation of EMP, ENG, IT_ENG and OFFICE into tables.

The final result is indeed the relational schema we have already seen in Section 3:

OFFICE (<u>OFFICE_OID</u>, offName, city)
EMP (<u>EMP_OID</u>, lastName, OFFICE_OID_fk)
ENG (<u>ENG_OID</u>, school, EMP_OID_fk)
IT_ENG (<u>IT_ENG_OID</u>, specialty, ENG_OID_fk)

5. Generating views

As we said, the core goal of the runtime translation procedure is to generate executable statements defining views. This is obtained by means of an analysis of the Datalog programs used to translate schemas as discussed in Section 4. In this section we discuss the major ideas of how views are constructed: which views, which components for them, where values come from and how they have to be correlated if needed (in the relational case, in SQL terms: which views, and, for each of them, which columns, which sources in the FROM clause and which join conditions). Then, in the next section, we will discuss the details in terms of a complete algorithm.

5.1. The general approach

The first issue to be considered is how to find which views are needed in a translation step, on the basis of the Datalog program that implements it. A key idea in this respect is a classification of MIDST metaconstructs (those in Figure 4) according to the role they play. There are three categories: *container*, *content*, and *support* constructs.[6] Containers are the constructs that correspond to sets of structured objects in the operational system (i.e. Aggregations and Abstracts corresponding to tables and typed tables, respectively). Content constructs represent elements of more complex constructs, 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 structures where data are logically stored in the system (for example relations), but are used to model relationships and constraints between them in a model-independent way. Examples are Generalizations (used to model hierarchies) and ForeignKeys (used to specify referential integrity constraints). This sharp distinction is not sufficient in practice, since there are some constructs that can be content and container at the same time: we call them *dual* constructs. For example, we model nested structures using the metaconstruct StructOfAttributes: a StructOfAttributes is a content for the construct in which it is contained (an Abstract or another StructOfAttributes) and a container for the constructs it aggregates.

In turn, Datalog translation rules can be classified according to the construct their head predicate refers to. Therefore, we have *container*- (for example, rules R_1

[6]This classification shares some similarity with that proposed by McBrien and Poulovassilis [21], which has however a different goal.

and R_7 in Section 4), *content-* (all other rules in Section 4) , *support-* and *dual-generating* rules.

The introduction of this classification is motivated by the observation that in all models we have constructs that have an independent existence (and are used to organize data or to represent real-world concepts), other constructs that exist only as components of independent constructs (and maintain component information), constructs that play both these roles, and finally constructs that describe properties of constructs of the previous two categories. These are the four categories we have just illustrated: container, content, dual, and support, respectively.

Exploiting the above observations, the procedure defines a view for each container construct, with fields that derive from the corresponding content (and dual) 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 or a dual, then we define a field of a certain view.

We will present the translation procedure with its technical details in Section 6. In the rest of this section, we first illustrate the procedure with reference to the running example, and then discuss two major issues in the procedure, namely: (i) the provenance of data (that is, where to derive the values from or how to generate them) for the single field (Subsection 5.2) and (ii) the appropriate combination of the source constructs, which, from a relational point of view, corresponds to a join (Subsection 5.3).

Let us consider the running example again. Step A includes rules R_1, R_2, R_3, R_4. The only container-generating rule is R_1, which copies all the typed tables, hence we generate a view for each typed table of the operational system: EMP_A, ENG_A, IT_ENG_A and OFFICE_A.[7]

The other rules are content-generating. Rules R_2 and R_3 copy Lexicals (simple fields) and AbstractAttributes (references), respectively. From rule R_2, the procedure infers the owner view, name, and type for each field. For AbstractAttributes the procedure works likewise (rule R_3) with the addition that it has to handle the values encoding the references between constructs in an object-oriented fashion.

The main rule of Step A is R_4, which eliminates generalizations by maintaining the parent and the child and connecting them with a reference. Here 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. We will discuss this issue in Section 5.2.

Let us now extend the same reasoning to the non-copying rules of the other steps.

In Step B we generate a key attribute for each typed table using rule R_5. It is a content-generating rule since it generates a key Lexical for every Abstract without an identifier. Hence we add another field to the views that correspond to those Abstracts.

Once Step B has guaranteed the presence of a key, in Step C we translate references into value-based (foreign-key) correspondences.[8] Rule R_6 addresses the need to copy 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.

Step D is simpler, since the only transformation involves turning 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. In fact, many systems have both *views* and *typed views*, and so we have to transform the former into the latter, or vice versa, according to the target model.

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 (container, content, support, or dual). Moreover, it is not related to the operational system constructs or languages since the statements are designed as system-generic structures. A specification step, exploiting the information coming from the operational system, will then be needed to generate system-specific statements. Furthermore, this approach is flexible because (as we will see shortly) it allows *annotations* on Datalog programs whenever conditions get more complex and in order to handle specific cases.

[7]As we said, we use the suffix here to distinguish the versions of tables and views in the various steps.

[8]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.

5.2. *The provenance of field values*

In this subsection we consider the problem of the data provenance of the individual fields. We discuss how the procedure finds for every value either a source field to derive the value from or a generation technique for it. Our procedure, for a given rule, collects information about the provenance of values by analyzing the Skolem functor used in the head of the rule.

If the Skolem functor has only one parameter and this parameter is the OID of another content field, then the value comes from the instance of the construct having that OID. In the example, this is what happens whenever a Lexical is copied using rule R_2 (with the functor *SK7(lexOID)* that copies the content construct Lexical from a unique source content). Similarly, if the Skolem functor has more than one parameter and only one of them refers to a field, then a source construct can be determined as well. For example, let us refer to a translation step in our repository, which implements the elimination of hierarchies by removing parent Abstracts and moving their attributes to child Abstracts. Such a step includes the following rule, shown here in simplified form:

```
R18   Lexical( OID: SK15(lexOID, childOID),
         ...
         abstractOID: SK2(childOID) )
      <- Lexical( OID:lexOID,
            ...
            abstractOID:parentOID ),
         Generalization( OID: genOID,
            parentAbstractOID: parentOID,
            childAbstractOID: childOID ),
         Abstract ( OID: childOID ),
         Abstract ( OID: parentOID );
```

In the rule, the functor *SK15* has two parameters: lexOID, referring to a content (the attribute), and childOID, referring to a container (the child Abstract). Clearly, the value is derived from the attribute, in many cases just copied. Concretely, this could have been used to copy the attribute SCHOOL into IT_ENG.

Instead, if more than two or none of the functor parameters refer to a content construct, the result value has to be retrieved in some other way. This is exactly what happens in steps A and B with rules R_4 (functor *SK3(genOID,childOID)*) and R_5 (*SK4(absOID)*) respectively. This case can be handled with the use of annotations, which specify where values come from. Here we present an informal description of this approach to give an intuition of the adopted strategy while technical details will be shown in Subsection 6.2.1. In rule R_4, functor *SK3* generates the OID for a reference field (AbstractAttribute) from the OID of a Generalization (a support construct) and from the OIDs of an Abstracts (container construct). Here an annotation is used to specify that the reference from the child table to the parent can be implemented by means of the tuple ID (TID)[9] used as value for the 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 TID. Therefore, the reference can be made by means of some manipulation of this TID. In detail, the rules of Step A in the running example lead to the following system-generic pseudo-SQL statements:

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

CREATE VIEW IT_ENG_A ... AS
   SELECT ... SPECIALTY,
       REF(IT_ENG_OID) AS ENG_OID
   FROM IT_ENG ;
```

ENG participates in a Generalization with EMP, so the rule copies its attributes and adds the values for the field referencing the parent EMP by casting the tuple TID. A similar thing happens for IT_ENG, which participates in a Generalization with ENG.

Similarly, 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. Our strategy involves the transformation of the TID into a value for this field. This solution would guarantee the presence of a unique identifier.

5.3. *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 is not always necessary. If two fields can be accessed from the same container, then the join can be avoided. For instance, considering an object-relational schema, if a construct C has a reference to a construct D, then we

[9] In OR systems, every typed table usually has a supplementary field, which we call TID, a system-managed identifier which can be used to base reference mechanisms on.

can use that reference to derive the values c of C and d of D, without any join.

In our paradigm we associate join conditions to Datalog rules and Skolem functors whenever necessary. In fact we handle typed functors, in the sense that they generate OIDs for specific constructs given the OIDs of a fixed set of constructs.

Let us see an example with an application of this technique. Consider another way of eliminating generalizations: moving the child attributes into the parent and deleting the child; obviously the parent will preserve its original attributes as well. In a multilevel case, this means that only the "top ancestor" is maintained, and attributes of all the "descendants" are moved to it. This requires, as a preliminary step (handled by a recursive rule), to detect the top ancestor for each child. These pairs are maintained in an auxiliary table and the Lexicals are copied from a child to the corresponding ancestor by means of a rule that uses a Skolem functor *SK6(ancestorOID, childOID, lexOID)*. Conversely, Lexicals from the source ancestor would be copied to the target one by means of the simpler functor *SK7(lexOID)*. Functor *SK6* relates two Abstracts (containers) and generates a new OID for the Lexical whose OID is *lexOID*. Instead, *SK7* generates OIDs for Lexicals given the OID of another Lexical.

The adopted combination of content-generating functors {*SK6*, *SK7*} encodes the sourcing of data as follows: as we will clarify in Section 6.2, it is a left join on TIDs between the ancestor and the child; in such a way, all the instances of the ancestor that are also instances of the child, appear in the result view as a single tuple. Moreover, the left join guarantees the inclusion of the tuples that represent instances of the ancestor that do not belong to the child.

In the running example, we have a two level generalization and so Lexicals have to be copied (if they exist) from two different child tables, thus leading to two left joins:

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

Notice that, in this statement, the pattern bases joins on the sharing of TIDs that takes place between parent and child instances. Moreover, consider that it is not always necessary to perform a join operation. In fact, there are some ORDBMSs (like DB2) that allow to perform our translation by accessing only the top level table of the hierarchy. For example, our query in DB2 will be characterized by the use of the `OUTER` keyword in the `FROM` clause, which exposes all the columns of the parameter table and of its subtables:

```
CREATE VIEW EMP_A
       (..., LASTNAME, SCHOOL, SPECIALTY) AS
   SELECT ...EMP.LASTNAME,
        EMP.SCHOOL, EMP.SPECIALTY
   FROM OUTER(EMP) ;
```

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 Step C where the values for the fields in the referring typed table can be derived from the key fields in the referred one (rule R_6).

The following statement is among the ones generated for Step C:

```
CREATE VIEW EMP_C ... AS
   SELECT ... LASTNAME,
           OFFICE->OFFICE_OID AS OFFICE_OID
   FROM EMP_B ;
```

Indeed, EMP has references towards OFFICE (which does not appear in the statement) via the field OFFICE and OFFICE_OID is the identifier for OFFICE added by rule R_5. Then, we need to copy OFFICE_OID values into a field of EMP according to the semantics of the rule. It is clear that there are two sources: EMP and OFFICE. However OFFICE_OID can be accessed via OFFICE, therefore the join between the two containers is not needed.

In this way, joins are avoided when possible, 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 typing to understand how to combine the containers of the different fields.

6. The view-generation algorithm

We now illustrate our algorithm for generating views at runtime from Datalog rules encoding schema-level

```
    Procedure $translateAsView(source\_schema, target\_model, targetDBMS)$

    Input: a source schema, a target model and a target DBMS
    Output: SQL statements that perform the runtime translation
0     schema := importFromTargetSystem(schema_name);
1     translation := findAutomaticTranslation();
      for each translation_step in translation do
2.a     view_generators := computeViewGenerators(translation_step);
        for each view_generator in view_generators do
2.b       l_independent_views.add(instantiateViewGenerator(view_generator, source_schema));
        for each l_independent_view in l_independent_views do
3         pseudoSQL_statements.add(computePseudoSQLstatement(l_independent_view));
        for each pseudoSQL_statement in pseudoSQL_statements do
4         executable_statements.add(computeExecutableStatement(pseudoSQL_statement, targetDBMS));
5     return executable_statements
```

Figure 5: The view-generation algorithm

translations. The procedure is shown in Figure 5 and includes tasks from the previous version of MIDST (Tasks 0 and 1) as well as new ones (all the others). Let us comment on them.

The algorithm takes as input the name of the source schema and the indication of the desired target model and of the target DBMS. The "import" subprocedure (Task 0 in the figure) is a function already in the previous version of MIDST, adopted here in order to build an internal representation of the source schema. It maps each construct of the source schema in terms of supermodel constructs. Then (Task 1) we use the target model parameter to invoke another existing MIDST function: findAutomaticTranslation. It produces a translation (for translating the source schema into the target model), which is composed of a sequence of elementary steps. Each step is, in turn, a set of Datalog rules. The rest of the procedure generates the views, on the basis of the Datalog rules in the translation steps. This is done in various tasks, with a process that finds general features first and then specializes them to the actual target context. Specifically, Task 2 produces language-independent views, and this is done in two subtasks: we first produce "view-generators" (Subtask 2.a), which depend only on the model at hand, and then instantiates them to (language-independent) views, which refer to the schema elements of interest (Subtask 2.b). Then, Task 3 transforms these views into statements in pseudo-SQL.[10] Finally, Task 4 compiles executable statements in the specific language (e.g. SQL, SQL/XML, XQuery) of the target operational system.

We describe the technical details of the procedure in the next subsections, as follows: the generation of language-independent views (Task 2) in Subsection 6.1, their conversion to pseudo-SQL views (Task 3) in Subsection 6.2, and finally the compilation of the executable view-creation statements (Task 4) in Subsection 6.3.

6.1. Language-independent views

As we said, language-independent views are built in two subtasks. The first of them, which produces "view-generators" (Subtask 2.a), is performed by means of the algorithm shown in Figure 6. Its input is an elementary translation step $\mathcal{T}$, which is a set of Datalog rules. As we said in Section 5, our goal is to produce a view for each container construct in the head of rules in $\mathcal{T}$ with components (columns in relational terms) for each of its content constructs. The classification of constructs is part of our supermodel, and so it is immediate for our procedure to discover which schema elements have to become views and which components thereof. In fact, line 1 in the algorithm finds container-generating rules by means of a simple inspection. Then, the loop in lines 2-6 builds a view-generator for each container rule. The most delicate step is to associate components with views, that is, to establish, for each component, which is the view it belongs to. This is done by finding the content-generating rules associated with the container rule at hand (line 3) and then building a view generator for the container rule and the associated content rules (line 4).

Let us introduce a bit of notation. Given a translation $\mathcal{T}$, we denote the set of content-generating and

[10]As we will clarify later, this is essentially a simplified version of SQL, which has the goal of generalizing in a declarative syntax the various languages of the commercial DBMS's.

```
     Procedure computeViewGenerators(translation_step)

     Input: a set of Datalog rules of a translation step
     Output: the view-generators corresponding to the translation step
1.     containerRules := findContainerRules(translation_step);
2.     for each containerRule in containerRules do;
3.        contentRules := findContentRules(containerRule, translation_step);
4.        view_generator := createViewGenerator(containerRule, contentRules);
5.        view_generators.add(view_generator);
6.     return view_generators;
```

Figure 6: The algorithm for finding view-generators

dual-generating rules in it as *Contents*($\mathcal{T}$) and the set of container-generating rules as *Containers*($\mathcal{T}$). Given $\mathcal{T}$ and a container-generating rule R in $\mathcal{T}$, we denote as *content*($R, \mathcal{T}$) the set of rules in *Contents*($\mathcal{T}$) generating content (and dual) constructs for R.

So, line 3 in the algorithm computes the association between container and content rules: given a container rule in a translation step, it finds the corresponding content rules. This is is determined by analyzing the Skolem functors in the rules in $\mathcal{T}$. In our context, each Skolem functor *SK* is associated with a given construct, the one which it generates OIDs for. Each functor always appears with the same arity and arguments, each one having a fixed type. The associated function is injective and function ranges are pairwise disjoint. For example, consider functor *SK5* of Section 4, used in rule R_6 (which eliminates the references). As it can be seen from the rule, and especially its head, *SK5* takes as input the OID of an AbstractAttribute and the OID of a Lexical and generates a unique OID for another Lexical:

$$SK5 : AbstractAttribute \times Lexical \rightarrow Lexical$$

The relationship between content and container constructs is determined by the OIDs. Container constructs have one main OID whose uniqueness is guaranteed by a *primary* Skolem functor (the one that generates the OID in the head). On the other hand, content constructs have more than one OID: one of them identifies the content itself while the others relate it to other constructs such as the container. This second category of OIDs is generated by a family of *secondary* Skolem functors. Our procedure includes in *content*($R, \mathcal{T}$) the content rules in $\mathcal{T}$ that involve, as secondary functor, the primary functor of the container rule R.

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

```
Abstract ( OID: SK2(oid),
           Name: name )
```

and it is apparent that it is only characterized by its OID, the one that identifies it. Conversely, a content construct has at least two functors (one for each characterizing OID). This is the case for example for Lexical as mentioned in the head of rule R_2 (repeated here for the sake of convenience):

```
R2    Lexical ( OID: SK7(lexOID),
                Name: name,
                IsNullable: isN,
                IsIdentifier: isI,
                abstractOID: SK2(absOID) )
      <- Lexical ( OID: lexOID,
                   Name: name,
                   IsIdentifier: isI,
                   IsNullable: isN,
                   abstractOID: absOID ),
         Abstract( OID:absOID );
```

Here, *SK7* is the primary functor, used to generate unique OIDs for instances of Lexical from OIDs of other Lexicals; *SK2* is a secondary one, used to connect each instance of Lexical (content) to the appropriate Abstract (container) by retrieving the OID of the target Abstract (abstractOID) from the one of the source (absOID).

Therefore, in our running example, if $\mathcal{T}$ is the translation of Step A, we have that *Containers*($\mathcal{T}$) = {R_1} and *Contents*($\mathcal{T}$) ={R_2, R_3, R_4} and *content*($R_1, \mathcal{T}$) = {R_2, R_3, R_4}. In fact, each of the rules R_2, R_3, R_4 has *SK2* (the primary functor of R_1) as a secondary functor.

This completes the discussion of line 3 of the algorithm in Figure 6. The rest of the algorithm is pretty easy. Line 4 is based on a definition, as follows. For each $R \in$ *Containers*($\mathcal{T}$) (that is, for each container generating rule) we define a *view-generator* as a pair $VG = (R, content(R, \mathcal{T}))$, composed of the rule itself and of a set of rules, those that define contents for its container. Essentially, a view-generator tells which rules define containers (and so will lead to views in the target schema) and which are the rules that define the respective contents (and so will lead to fields of the corresponding views). Finally, line 5 just prepares the result to be returned by the algorithm.

In the example, our algorithm will determine, for Step A, the following view-generator: $VG_1 = (R_1, \{R_2, R_3, R_4\})$. Intuitively, this view-generator says that in the target schema we have container constructs as generated by rule R_1 (and so, Abstracts), each with content constructs generated by R_2, R_3, and R_4 (Lexicals and AbstractAttributes).

Let us now move to the actual construction of language-independent views, Subtask 2.b in the main algorithm in Figure 5. This does not require procedural details, and is based on some definitions.

Given a Datalog rule R, we define an *instantiated body* IB as a specific assignment of values for the constructs appearing in the body of R. 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. For example, given the body of rule R_2 (copy-lexical), an instantiated version of it is the following one:

```
Lexical ( OID: 100,
          Name: "lastName",
          IsIdentifier: "false",
          IsNullable: "false",
          abstractOID: 3 ),
Abstract( OID: 3 );
```

In the running example, it expresses the fact that we are copying the Lexical "lastName" (with OID 100) from the Abstract "EMP" (with OID 3). We remark that in general the conditions expressed in the bodies of Datalog rules (which are evaluated within MIDST-RT supermodel) may refer to container, content, and dual constructs as well as to support ones.

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 the instantiation of variables in IB. Again with reference to R_2, we have the following instantiated head:

```
Lexical ( OID: SK7(100),
          Name: "lastName",
          IsNullable: "false",
          IsIdentifier: "false",
          abstractOID: SK2(3) )
```

This head defines a new Lexical for a given Abstract (with OID obtained applying the functor *SK2* to the argument 3) that is a copy of the original Lexical of the Abstract with OID 3.

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

Then, Subtask 2.b in the algorithm in Figure 5 computes a set of *language-independent views* for a view-generator VG, where each of them is defined as $V = (IR, \{c_1, c_2, \ldots\ c_n\})$, and is composed of an instantiation IR of rule R and of the set of all the possible instantiations of rules in $content(R, \mathcal{T})$ that are coherent with IR.

In the example, the language-independent views for VG_1 are:[11]

$$
\begin{aligned}
V_1 = \ & (EMP \rightarrow_{\text{copy-abstract}} EMP\ , \\
& \quad \{\ EMP(lastName) \rightarrow_{\text{copy-lexical}} EMP(lastName), \\
& \quad EMP(office) \rightarrow_{\text{copy-abstractAttribute}} EMP(office)\}) \\
V_2 = \ & (OFFICE \rightarrow_{\text{copy-abstract}} OFFICE\ , \\
& \quad \{\ OFFICE(offName) \rightarrow_{\text{copy-lexical}} OFFICE(offName), \\
& \quad OFFICE(city) \rightarrow_{\text{copy-lexical}} OFFICE(city)\}) \\
V_3 = \ & (ENG \rightarrow_{\text{copy-abstract}} ENG\ , \\
& \quad \{\ ENG(school) \rightarrow_{\text{copy-lexical}} ENG(school), \\
& \quad Gen(EMP, ENG) \rightarrow_{\text{elim-gen}} ENG(EMP)\}) \\
V_4 = \ & (IT\_ENG \rightarrow_{\text{copy-abstract}} IT\_ENG\ , \\
& \quad \{\ IT\_ENG(specialty) \rightarrow_{\text{copy-lexical}} IT\_ENG(specialty), \\
& \quad Gen(ENG, IT\_ENG) \rightarrow_{\text{elim-gen}} IT\_ENG(ENG)\})
\end{aligned}
$$

In plain words, this means that we will have to produce four views, each with the associated components. For example, V_1 says that there will be a view *EMP*, with columns *lastName* and *office*.

It is worth noting that in our tool language-independent views contain additional information besides the one shown above. In particular, a language-independent view is a map of actual values assigned to the variables of the rules (content- and container-generating) that belong to the view-generator. As a concrete example, the language-independent view V_4 is represented in our tool as:

```
CONTAINER: [oid=75; name = IT_ENG;
      internal_oid = IT_ENG_OID ]
CONTENTS: {
     [oid=332; name = SPECIALTY;
     absOID=75; isN = false; isId = false],
     [oid=6; parentOID=74; childOID=75;
     genOID=6; parentName = ENG ]
   };
```

[11] The descriptive names of the rules are inserted for the sake of readability of the example. Notice that we have omitted the suffix _A as no ambiguity arises.

where “CONTAINER” represents the instantiation of the container-generating rule that copies the Abstracts (in the example the typed table IT_ENG), while “CONTENTS” represent the useful instantiations of the content-generating rules that copy Lexicals and remove Generalizations.

6.2. *Pseudo-SQL view creation statements*

Let us now devote our attention to Task 3 of the procedure in Figure 5. It performs the translation of a language independent view into a pseudo-SQL view-creation statement and it follows the algorithm shown in Figure 7. Its input is a language-independent view, $V = (\mathit{IR}, \{c_1, c_2, \ldots c_n\})$ instantiation of a view-generator $\mathit{VG} = (R, \mathit{content}(R, \mathcal{T}))$, for a container rule R. The resulting pseudo-SQL statement has the following structure:

```
CREATE VIEW $name(col_1, col_2, \ldots, col_n)$ AS
      SELECT $a_1(s_{j_1}.col_1), a_2(s_{j_2}.col_2), \ldots, a_n(s_{j_n}.col_n)$
      FROM $sources$;
```

Line 1 of the algorithm in Figure 7 obtains *name* from V (the variable named *l_independent_view*) by retrieving the name of the head construct of the instantiation IR of the container-generating rule R. This is the name of the actual view to be created.

Next, line 2 derives the names for the columns of the view, $col_1, col_2, \ldots, col_n$, by getting the names of the constructs generated by the heads of the instantiated rules $\{c_1, c_2, \ldots c_n\}$ in V, and so each of them is a content (or dual) construct.

In line 3, the algorithm identifies, on the basis of the primary Skolem functor of the container rule R, the *main source containers* for the view: essentially, these are the containers (usually just one[12]) in the source schema that are transformed in to the view being constructed here. In the example, we will have that EMP_A is the source container for EMP_B and so on.

Then the algorithm proceeds by producing the details for the SELECT statement in the view:

(a) the identification of the source container (let us call it $source(s_{j_i}.col_i)$) for the provenance of each content element col_i and of the respective actual value for it (indicated with the functional symbols a_i); this is done in lines 4-5, discussed in detail in Subsection 6.2.1

(b) the actual construction of *sources* in the FROM clause, with a refinement and the merge of the various elements $source(s_{j_i}.col_i)$, with the possible use of suitable join conditions (lines 6-10, illustrated in Subsection 6.2.2).

At the end the procedure creates and returns the pseudo-SQL statement putting together the elements produced in the previous steps (lines 11-12).

Let us consider the aspects, (a) and (b) above, in turn.

6.2.1. *Finding value provenance*

Let us now discuss how the algorithm identifies, in lines 4-5, the sources of each content col_i. Specifically, this involves the decision on whether the value can be copied (if so, from where) or has to be generated (if so, how). This is done on the basis for the information given by the Skolem functors of the rules that generate col_i and the annotations possibly specified on them. Let us provide some detail. Given a content-generating rule R', its secondary functor links the generated content to its source container (the one the functor is applied to). The parameters of the functor are instantiated as a consequence of the instantiation of the body of R'. The primary functor conveys information about the provenance of data (that is, the content to derive the value from) for the content under examination. In general, the joint instantiation of both primary and secondary functors indicates where to retrieve the values from. Specifically, if the primary functor can link the head content to a source construct, then the secondary functor allows to determine the corresponding container construct. It may happen that it is not possible to associate the primary functor to a source content (and thus to a source container) uniquely. In such cases the strategy we follow relies on the possibility of using of *annotations*, fragments of pseudo-SQL code that can be associated with Datalog rules, and more precisely to functors in them. Specifically it is possible to associate the primary functor with a generation technique for the value. This is essential for the functors that have two or more content arguments (or no content arguments at all). For example in Rule R$_4$ we have the primary functor *SK3* that has no content argument. As we will shortly see, an annotation is needed here.

Then, our algorithm proceeds as follows.

(a.1) Default case: there is no annotation on the primary functor; this is possible when (i) the functor has exactly one parameter, a content, or (ii) it has more parameters, with at least a content one and at most a container one. In case (i) the column of the view comes from the container in the

[12]In the sequel, in order to simplify the discussion, we will assume that there is only one main source container for each view. The more general case is intricate but straightforward.

```
     Procedure computePseudoSQLstatement(l_independent_view)

     Input: a language-independent view
     Output: the pseudo-SQL view creation statement
 1.    name := l_independent_view.getContainerName();
 2.    columns := l_independent_view.getContentNames();
 3.    sourceContainers := instantiated_container_rule.getSource();
 4.    for each instantiated_content_rule in l_independent_view do
 5.       targetList.add(instantiated_content_rule.calculateSource());
 6.    fromClause.add(sourceContainers);
 7.    for each instantiated_content_rule in l_independent_view do
 8.       if instantiated_content_rule.getSourceContainer ∉ sourceContainers then
 9.          fromClause.add(instantiated_container_rule.getJoinCondition());
10.    fromClause.simplify();
11.    pseudo-SQL_statement := create pseudo-SQL_statement(name, columns, targetList, fromClause);
12.    return pseudo-SQL_statement;
```

Figure 7: The creation of pseudo-SQL view statement

source indicated by the secondary functor. In case (ii) the column of the view comes from the container mentioned in the functor. In both cases, the algorithm finds a target list element for the SQL statement composed of the names (in the source schema) of the container and of the content element. The algorithm computes also the provenance for such an element (to be used in the subsequent steps to build the `FROM` clause): it is the container mentioned above; if it does not coincide with the main source container for the view, the provenance is defined as a join of the two containers (and possibly others) on the basis of repeated OIDs in the body of the rule.

(a.2) Annotation case: if the primary functor is annotated with a query fragment *a*, then *a* is applied in order to calculate the value. Notice that the query can be written referring to all the literals in the instantiated content-generating rule. Usually, these queries are simple and involve a small number of parameters. The provenance is handled as in case (a.1), on the basis of the containers involved in the rule and in the annotation.

As an example of case (a.1), consider again the rule R$_3$ of Step A (which we partially show here again for convenience), which copies the AbstractAttributes:

```
R3 AbstractAttribute( OID: SK8(oid),
                 Name: name,
                 isNullable: isN,
                 abstractToOID: SK2(absToOID),
                 abstractOID: SK2(absOID) )
   <- ...
```

This rule is not annotated and its functor *SK8* takes in input the OID of the AbstractAttribute. In this case, in the target list of the view we will have an element *s.col*, where *s* is the name of the Abstract and *col* the name of the AbstractAttribute. The provenance of such an element will be the Abstract *s*. In the actual example, we will have, in the construction of the view EMP_A, an element in the target list of the form (EMP.Office) and its provenance would be EMP.

On the other hand, as an example of case (a.2), consider the rule R$_4$ of Step A which replaces the generalizations between two typed tables by adding a specific reference field (AbstractAttribute) in the child table:

```
R4 AbstractAttribute (
        OID: SK3(genOID, childOID),
        Name: name,
        IsNullable: "false",
        abstractOID: SK2(childOID),
        abstractToOID: SK2(parentOID) )
    <- Generalization ( OID: genOID,
         parentAbstractOID: parentOID,
         childAbstractOID: childOID ),
       Abstract ( OID: parentOID, Name: name ),
       Abstract ( OID: childOID );
```

Here, *SK3*, the primary functor, takes in input the OID of the Generalization and the OID of an Abstract. In this case, the functor is annotated with:

```
SELECT INTERNAL_ID FROM ABSTRACT(parentOID)
```

This annotation specifies that the value of the reference (indeed AbstractAttributes represent references) must coincide with the OID of the Abstract that is the parent of the generalization. In this case, in the target list, we will have the parent Abstract (in the sense that, as allowed by most OR systems, we will use the system managed TID as a value). Such an Abstract will also be the provenance for the value. However, as the main container for the view to be generated is the child Abstract, the actual provenance is the join of the two Abstracts. In the actual example, in the elimination of the Generalization between ENG and EMP, we would have the element EMP in the target list for view ENG_A and its provenance would be the join between ENG and EMP.

A similar strategy should be followed to cope with rule R_5 of Step 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 primary functor *SK4* takes an Abstract as input parameter, and so there are no natural sources for the values. A possible annotation could be the following one:

```
SELECT INTERNAL_ID FROM ABSTRACT(absOID)
```

This implies the adoption of the values of internal tuple identifiers (INTERNAL_ID) as elements for the key of the typed table as explained at the end of Subsection 5.2.

6.2.2. Building the FROM clause

Let us now discuss how point (b) above is performed, that is, how sources for the various elements are constructed and combined.

The FROM clause is initialized (line 6 in Figure 7) with the source containers for the language-independent view at hand. Then, the instantiated content rules in the view are examined one at the time (lines 7-9) and if the source container is not a main source container, then the join condition (computed in line 5, as we said above) involving both containers (and additional ones of needed) is added to the FROM clause.

Let us show a result of the application of this step, both to illustrate it and to motivate the next one. In the first example in Subsection 5.3, the algorithm would generate three elements for the source clause, namely the main source container EMP, and the two left joins between EMP and ENG, and between EMP, ENG and IT_ENG.

Finally (line 10), the algorithm examines the elements in the FROM clause that has been initially generated, and performs simplifications by merging the various join conditions, on the basis of common containers and of subsumed expressions. In the example, the simple element EMP and the left join between EMP and ENG would be removed because they are subsumed by the double left join over EMP, ENG, and IT_ENG.

At the end (lines 11-12), the procedure creates the pseudo-SQL statement combining the information retrieved on the previous steps and returns the statement.

6.3. Executable view-creation 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 phase, each system-generic SQL statement encoding an elementary step is translated in terms of a system-specific and executable one.

The following SQL statements exemplify the elimination of hierarchies (rule R_4) which takes place in step 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 EMP_A_t and ENG_A_t in the result schema. The statements below implement the strategy consisting in using the internal OID to make the child refer to its parent. It is apparent that a lot of DB2 technical details are introduced in this last phase (for example, the use of type constructors, the various cast functions and explicit scope modifiers).

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

CREATE TYPE ENG_A_t AS (
     toEMP REF(EMP_A_t),
     school varchar(50))
...;

CREATE VIEW EMP_A of EMP_A_t MODE DB2SQL
         (REF is EMPOID USER GENERATED) AS
     SELECT EMP_A_t(INTEGER(EMPOID)), lastName
     FROM EMP;

CREATE VIEW ENG_A of ENG_A_t MODE DB2SQL
         (REF is ENGOID USER GENERATED,
          toEMP WITH OPTIONS SCOPE EMP_A) AS
     SELECT ENG_A_t(INTEGER(ENGOID)),
       EMP_A_t(INTEGER(EMPOID)), school
     FROM ENG;
```

The produced statements are finally sorted in order to take care of the dependencies between views, so that a view that refers to another one is created later.

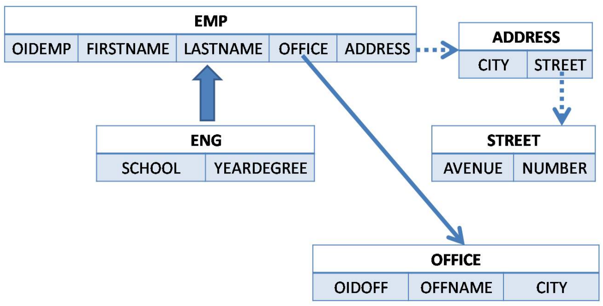


Figure 8: An object-relational schema: OR_DEMO

7. Views for the example scenarios

In this section we consider three scenarios of executable statements, in order to better understand the concept of "view" with respect of the motivating examples proposed at the beginning of this paper.

7.1. Relational views

Let us consider the object-relational schema OR_DEMO shown in Figure 8, where we have three typed tables and two structured types. We have a generalization (ENG is a subtable of EMP) and a reference (from EMP to OFFICE). Structured types are used to build a two-level complex object (the value of ADDRESS comes from the ADDRESS type whose values involve the STREET type). We want a translation that produces a set of relational views with reference to IBM DB2 [14]. MIDST-RT completely supports this activity, with a component whose interface is shown in Figure 9. The user would perform the following sequence of steps, which are highlighted in the figure:

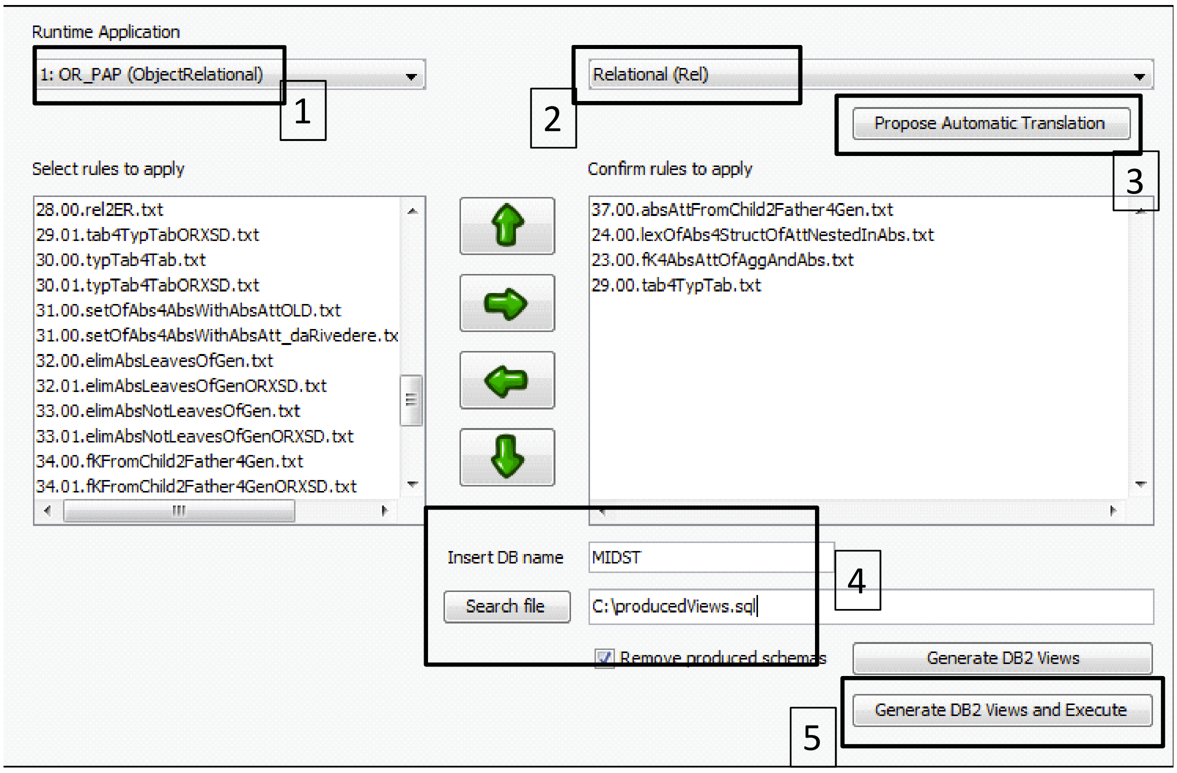
0. Import of the source schema from the operational system into the tool dictionary (this step is not shown in the figure).
1. Selection of the source schema (the one imported in step 0).
2. Selection of the target schema (relational).
3. Automatic selection of the programs to apply. The user can modify the set of selected programs in order to customize some steps of the translation.[13]
4. Insertion of useful information for the generation of the statements, such as the DB name and the path in which the statements will be produced.
5. Generation of the statements in a text file or direct execution over DB2.

Let us comment on the produced statements.[14] For brevity and without loss of generality, we describe only the first step of the translation (that is, the removal of generalizations). DB2 handles object views with the concept of typed view, which is a view whose type has to be defined explicitly. This motivates the presence of the "create type" statements defining the types OFFICE_t, EMP_t and ENG_t in the result schema. The statements below implement the strategy consisting in using the internal OID to make the child refer to its parent.

```
-- ******************************************************
-- STEP 1: removing generalizations
-- ******************************************************
CREATE TYPE OR_DEMO_1.OFFICE_t AS(
   CITY varchar(50),
   OFFNAME varchar(50))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.OFFICE of
         OR_DEMO_1.OFFICE_t MODE DB2SQL
   (REF is OIDOFFICE USER GENERATED) AS
   SELECT
      OR_DEMO_1.OFFICE_t(
        CAST(OR_DEMO.OFFICE.OIDOFF AS INTEGER)),
      OR_DEMO.OFFICE.CITY,
      OR_DEMO.OFFICE.OFFNAME
   FROM OR_DEMO.OFFICE;

CREATE TYPE OR_DEMO_1.EMP_t AS(
   LASTNAME varchar(50),
   FIRSTNAME varchar(50),
   ADDRESS OR_DEMO.ADDRESS_t,
   OFFICE REF(OR_DEMO_1.OFFICE_t))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.EMP of
         OR_DEMO_1.EMP_t MODE DB2SQL
   (REF is OIDEMP USER GENERATED,
    OFFICE WITH OPTIONS SCOPE OR_DEMO_1.OFFICE) AS
   SELECT
      OR_DEMO_1.EMP_t(
       CAST(OR_DEMO.EMP.OIDEMP AS INTEGER)),
      OR_DEMO.EMP.LASTNAME,
      OR_DEMO.EMP.FIRSTNAME,
      OR_DEMO.EMP.ADDRESS,
      OR_DEMO_1.OFFICE_t(
       CAST(OR_DEMO.EMP.OFF AS INTEGER))
   FROM OR_DEMO.EMP;

CREATE TYPE OR_DEMO_1.ENG_t AS(
   SCHOOL varchar(50),
   YEARDEGREE integer,
   to_EMP REF(OR_DEMO_1.EMP_t))
MODE DB2SQL REF USING INTEGER;

CREATE VIEW OR_DEMO_1.ENG of
         OR_DEMO_1.ENG_t MODE DB2SQL
   (REF is OIDENG USER GENERATED,
    to_EMP WITH OPTIONS SCOPE
       OR_DEMO_1.EMP) AS
```

[13]For example, the system proposes to remove generalizations merging the children into the parent, while the user wants to keep both the children and the parent.

[14]Notice that, as we anticipated in Section 3, in the tool the names are distinguished by means of schema names, and so there is no need to use suffixes.



Runtime Application
1: OR_PAP (ObjectRelational)
Relational (Rel)
Propose Automatic Translation
Select rules to apply
28.00.rel2ER.txt
29.01.tab4TypTabORXSD.txt
30.00.typTab4Tab.txt
30.01.typTab4TabORXSD.txt
31.00.setOfAbs4AbsWithAbsAttOLD.txt
31.00.setOfAbs4AbsWithAbsAtt_daRivedere.tx
32.00.elimAbsLeavesOfGen.txt
32.01.elimAbsLeavesOfGenORXSD.txt
33.00.elimAbsNotLeavesOfGen.txt
33.01.elimAbsNotLeavesOfGenORXSD.txt
34.00.fKFromChild2Father4Gen.txt
34.01.fKFromChild2Father4GenORXSD.txt
Confirm rules to apply
37.00.absAttFromChild2Father4Gen.txt
24.00.lexOfAbs4StructOfAttNestedInAbs.txt
23.00.fK4AbsAttOfAggAndAbs.txt
29.00.tab4TypTab.txt
Insert DB name
MIDST
Search file
C:\producedViews.sql
Remove produced schemas
Generate DB2 Views
Generate DB2 Views and Execute
1
2
3
4
5

Figure 9: A screenshot of MIDST-RT

```
SELECT
   OR_DEMO_1.ENG_t(
    CAST(OR_DEMO.ENG.OIDEMP AS INTEGER)),
   OR_DEMO.ENG.SCHOOL,
   OR_DEMO.ENG.YEARDEGREE,
   OR_DEMO_1.EMP_t(
    CAST(OR_DEMO.ENG.OIDEMP AS INTEGER))
FROM OR_DEMO.ENG;
```

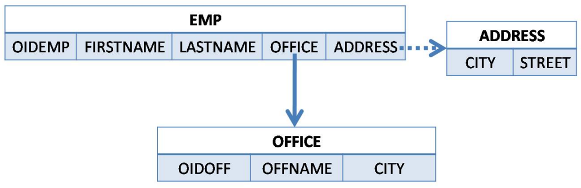


Figure 10: An object-relational schema

The subsequent steps of the translation process will refer to the previous ones. This means that, after the removal of generalizations, we will have a set of views that represents a new schema without generalizations. We call this schema OR_DEMO_1. The next step is the elimination of nested types: we define a new set of views over the views previously defined. Thus, we will have a schema OR_DEMO_2 composed of a set of views defined over OR_DEMO_1. Then we must eliminate all the references (we introduce foreign-keys) and we must transform typed tables into simple tables. At the end, we have four new schemas (because the translation is composed of four steps), but only the last one, OR_DEMO_4, represents our target schema, a relational one.

7.2. XML views

Consider the object-relational schema shown in Figure 10 and suppose we need an XML document that contains all its data in a structured way. We can do this with MIDST-RT by choosing XSD as the target model. In this way, the tool produces a statement that, executed over DB2, will create an XML document with all data directly extracted from the original schema. This can be possible by using an SQL/XML language, specific for the operational system, that contains functions that help the user to create XML elements from relational data. The tool produces the following statement:

```
SELECT XMLELEMENT(
  name "orxml",
    XMLCONCAT(
      XMLAGG(
        XMLELEMENT(
          name "emp",
          XMLELEMENT(name "OIDEMP",e.OIDEMP),
          XMLELEMENT(name "firstName",e.firstName),
          XMLELEMENT(name "lastName",e.lastName),
          XMLELEMENT(name "offref",e.off),
          XMLELEMENT(
            name "address",
            XMLELEMENT(name "city", e.address..city),
            XMLELEMENT(name "street",e.address..street)
          )
        )
      ),
      (SELECT XMLAGG(
         XMLELEMENT(
           name "office",
           XMLELEMENT(name "OIDOFF",d.OIDOFF),
           XMLELEMENT(name "offName",d.offName),
           XMLELEMENT(name "city",d.city)
        )
       )
       FROM OR_XML.OFFICE d)
    )
)
FROM OR_XML.EMP e;
```

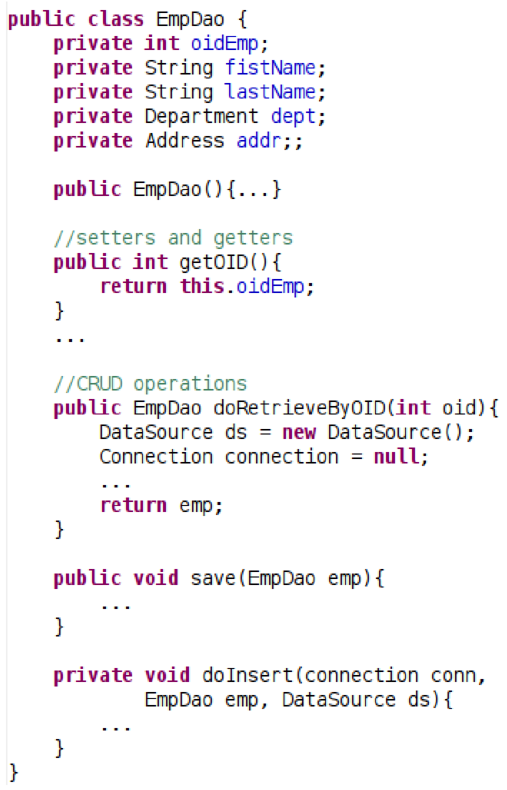
The produced XML document will be:

```
<orxml>
    <emp>
        <OIDEMP>1</OIDEMP>
        <firstName>Mark</firstName>
        <lastName>Brown</lastName>
        <offref>2</offref>
        <address>
            <city>Rome</city>
            <street>Viale Marconi 1</street>
        </address>
    </emp>
    ...
    <office>
        <OIDOFF>2</OIDOFF>
        <offName>ROMA TRE</offName>
        <city>Rome</city>
    </office>
      ...
</orxml>
```

7.3. Object-oriented views

In this last scenario we start from an object-relational schema in order to obtain a set of Java classes that allows an object-oriented access to the database. This example briefly sketches how the generation process of a piece of Java code from relational tables may be performed.

The example we show produces some classes that contain CRUD methods (create, retrieve, update, delete) to access the database. Thus, we are following the DAO (data access object) design pattern. We are also able to produce classes by referring to other technologies, for example using Hibernate annotations. Moreover, thanks to an object-oriented importer, we can import the schema from the Java classes and produce an object-relational database: this is very simple, in fact,



```
public class EmpDao {
    private int oidEmp;
    private String fistName;
    private String lastName;
    private Department dept;
    private Address addr;;

    public EmpDao(){...}

    //setters and getters
    public int getOID(){
        return this.oidEmp;
    }
    ...

    //CRUD operations
    public EmpDao doRetrieveByOID(int oid){
        DataSource ds = new DataSource();
        Connection connection = null;
        ...
        return emp;
    }

    public void save(EmpDao emp){
        ...
    }

    private void doInsert(connection conn,
            EmpDao emp, DataSource ds){
        ...
    }
}
```

Figure 11: The produced Java class

inside our metamodel, the object-oriented model is entirely contained into the object-relational one, so we do not need any translation.

This problem has a lot of solutions in the literature, but MIDST-RT ensures flexibility: in fact, thanks to the internal set of rules, the user can decide to modify the source schema to obtain the preferred translation, or can perform a translation towards a model that presents some non-standard features.

Starting from the object-relational schema shown in Subsection 7.2, one possibility is the creation of three Java classes using the DAO pattern. So we will have the objects Emp, Office and Address. Figure 11 shows the source code of the Java class EmpDAO.

8. Related work

The problem of translating schemas between models has a largely recognized significance and has been pursued in the literature according to several perspectives of model management. Bernstein and Melnik [10] present the recent state of the art in this field and, indirectly, outline an overview of the major approaches and achievements.

The starting point for this paper is our MIDST project (Atzeni et al. [4, 5]), which developed a platform allowing for model-independent schema and data translation, whose initially ideas and theoretical bases were laid by Atzeni and Torlone [6]. While we refer to the above papers for a general discussion on related work, we mention here the work of Hainaut [18] and McBrien, Poulovassilis, and Smith [21, 27], with which we share the use of some form of metamodeling technique. Indeed, we have great similarity with both these approaches, which also adopt a translation process composed of various steps. There are differences in the kind of universal metamodel, which is simpler in [21, 27] and more complex (and so closer to ours) in [18].

This paper has the goal to provide the MIDST framework with a runtime design and so to overcome the limitations mentioned by Bernstein and Melnik [10, Sec.3.1] with respect to the off-line approach. In this sense, this paper goes beyond the work of Hainaut [18] and McBrien, Poulovassilis, and Smith [21, 27], who can transform the database instance, but it in an essentially static way.

Other proposals have recently appeared with the goal of supporting dynamic translations, as follows. Mork et al. [25] also adopt a runtime approach (based on the work by Atzeni and Torlone [6] as well) 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, they indeed solve a specific subset of problems and provide an object-relational mapping tool. Bernstein et al. [11] 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. Instead, our approach is aimed at providing a runtime support to the whole range of translations allowed by MIDST that is not limited to object-to-relational or XML-to-object, but involves any possible transformation between a pair of models in our supermodel (ER, OR, OO, XSD, relational, etc.).

Our approach shares some analogies with Clio [15, 16, 17, 24, 28] too. Clio 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 this model 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 [1]. In such a perspective, Datalog rules are not only seen as model-to-model translations, but encode 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 the ones illustrated in [3].

9. Conclusions

The main contribution of this paper is a runtime approach to data translation, with the development of the MIDST-RT tool. We have shown how we can generate executable statements out of translation rules. The approach aims at being general, in the sense that the final objective is to derive an executable statement for any possible translation. Then, we have also shown some scenarios which may benefit from the usage of MIDST-RT, in order to allow flexibility and customization.

A major issue is the query language. It is necessary to specify a language capable of interacting with all the involved models homogeneously. Although, in some cases, such a single language would be available, other situations are more complex and need further investigation. Examples are the ones involving translations from object-relational to XML and vice versa. We have used here combinations of languages including SQL/XML and XQuery/SQL, over one single platform. In fact, the solution described in this paper actually refers to transformations taking place in a single system, offering the logical support to both models. Indeed, it may be the case that more systems are involved; however the adoption of the appropriate middleware solutions might offer working solutions based, for example, on a common exchange format.

Moreover, in this paper we have shown examples of relational views. These views have an intrinsic problem: in fact, when we define a relational view, it is quite probable that it will not be updatable. A possible solution to this problem (and possible future work) is the introduction of the concept of "reverse mapping" [25], a mapping that keeps trace of the origin of data shown by views in order to modify the source database when the user tries to modify a view.

Let us conclude by discussing a few issues where our approach shows some limitations that we are working to overcome. From the implementation point of view, it is clear that the target system will have some restrictions on how it deals with views (including materialization, persistence, update propagation). However this limitation is related to the specific target system and it comes as a direct consequence of the runtime perspective where no third-party actors interfere. From a theoretical point of view, open issues are related with the generality and correctness of the approach. As for generality of modeling, MIDST metamodel collects all the constructs most commonly used in models and can be extended whenever necessary. Extensions could also go towards a richer representation of semantics, where integrity constraints are described and supported, in the sense that their satisfaction is verified and reasoning on them can be performed. Clearly, this would require a different approach on the management of the supermodel, which would require additional features beside and beyond the relational implementation. In this respect, we are considering approaches based on description logics [7].

Acknowledgement

We would like to thank the anonymous reviewers for their very helpful comments.

References

[1] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. From schema and model translation to a model management system. In *BNCOD*, pages 227–240, 2008.

[2] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. A runtime approach to model-independent schema and data translation. In *EDBT*, pages 275-286, 2009.

[3] P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. MISM: A Platform for Model-Independent Solutions to Model Management Problems. *J. Data Semantics*, 14:133–161, 2009.

[4] P. Atzeni, P. Cappellari, and G. Gianforme. MIDST: model independent schema and data translation. In *SIGMOD*, pages 1134–1136. ACM, 2007.

[5] P. Atzeni, P. Cappellari, R. Torlone, P. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB Journal*, 17:1347–1370, 2008.

[6] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *EDBT Conference, LNCS 1057*, pages 79–95. Springer, 1996.

[7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2nd edition, 2007.

[8] C. Bauer and G. King. Java Persistence with Hibernate. Manning Publications Co., 2006.

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

[10] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, pages 1–12, 2007.

[11] P. A. Bernstein, S. Melnik, and J. F. Terwilliger. Language-integrated querying of xml data in sql server. In *VLDB*, pages 1396–1399, 2008.

[12] M. L. Brodie, M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995.

[13] L. Cabibbo, The expressive power of stratified logic programs with value invention, In *Inf. Comput. 147* pages 22–56, 1998.

[14] M. Carey, S. Rielau, and B. Vance. Object view hierarchies in DB2 UDB. In *EDBT, LNCS 1777*, pages 478–492, 2000.

[15] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

[16] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[17] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, pages 805–810. ACM, 2005.

[18] J.-L. Hainaut. The transformational approach to database engineering. In *GTTSE, LNCS 4143*, pages 95–143. Springer, 2006.

[19] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, 1990.

[20] D. Jordan. Java Data Objects. O'Reilly, 2003.

[21] P. McBrien and A. Poulovassilis. A uniform approach to intermodel transformations. In *CAiSE Conference, LNCS 1626*, pages 333–348, 1999.

[22] B. McLaughin. Java and XML Data Binding. O'Reilly Media, 2002.

[23] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD*, pages 461–472, 2007.

[24] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[25] P. Mork, P. A. Bernstein, and S. Melnik. Teaching a schema translator to produce O/R views. In *ER Conference, LNCS 4801*, pages 102–119. Springer, 2007.

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

[27] A. Smith and P. McBrien. A generic data level implementation of modelgen. In *BNCOD 25, LNCS 5071*, pages 63–74, 2008.

[28] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.