A runtime approach to model-independent schema and data translation

Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, Giorgio Gianforme[†]

Università Roma Tre

atzeni@dia.uniroma3.it, bellomarini@yahoo.it, franbugiotti@yahoo.it, giorgio.gianforme@gmail.com

ABSTRACT

A runtime approach to model-generic translation of schema and data is proposed.

It is based on our previous work on MIDST, a platform conceived to perform translations in an off-line fashion. In the original approach, the source database is imported into a dictionary, where it is stored according to a universal model. Then, the translation is applied within the tool as a composition of elementary transformation steps, specified as Datalog programs. Finally, the result is exported into the operational system.

Here we illustrate a new, lightweight approach where the database is not imported. The tool needs only to know the model and the schema of the source database and generates views on the operational system that transform the underlying data (stored in the source schema) according to the corresponding schema in the target model. Views are generated in an almost automatic way, on the basis of the Datalog rules for schema translation.

1. INTRODUCTION

The problem of translating schemas between data models is acquiring progressive significance in heterogeneous environments and has received attention in many works [3, 5, 7, 13, 15, 18]. Applications are usually designed to deal with information represented according to a specific data model, while the evolution of systems (in databases as well as in other technology domains, such as the Web) led to the adoption of many representation paradigms.

For example, many database systems are nowadays object-relational (OR) and so it is reasonable to exploit their full potentialities by adopting such a model while most applications are designed to interact with a relational database. Also, object-relational extensions are often non-standard, and conversions are needed. The explosion of XML, with all its applications (for example, as a format for information

exchange or as the language for the semantic Web), has increased the heterogeneity of representations. In general the presence of several coexisting models introduces the need for runtime translation techniques and tools.

We have recently proposed MIDST [3, 5], a platform for model-independent schema and data translation in order to provide a paradigm to face issues of this kind. MIDST adopts a metalevel approach towards translations by performing them in the context of a universal model (called the *supermodel*), which allows for the management of schemas in many different data models. In fact, the set of models that can be dealt with is large and extensible. The current version handles relational, object-oriented, object-relational, entity-relationship, XML-based, each in many different variants, and new metaconstructs can be added, if needed for handling features not covered by the current ones. Translations in MIDST are organized according to the following pattern. First, the source database is imported into the tool and described according to the metamodel. Then, translations are performed by means of elementary transformation steps; finally, the obtained database is exported into the operational system.[1] MIDST approach provides a general solution to the problem of schema translation, with *model-genericity* (as the approach works in the same way for many models) and *model-awareness* (in the sense that the tool knows models, and can use such a knowledge to produce target schemas and databases that conform to specific target models). However, as pointed out by Bernstein and Melnik [8], this approach is rather inefficient for data exchange. In fact, the necessity to import and export a whole database in order to perform translations is out of step with the current need for interoperability in heterogeneous data environments.

Here we propose a 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 (the target schema) conform with 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 within the supermodel, obtained as composition of elementary ones, each dealing with a specific aspect (construct or feature thereof) to be eliminated or transformed. The main difference is that the import process concerns only the schema of the source database. The rules for schema

---

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

translation are here used as the basis for the generation of views in the operational system. In such a way data is managed only within the operational system itself. In fact, our main contribution is the definition of an algorithm that generates executable data level statements (view definitions) out of schema translation rules.

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

In order to cope with the heterogeneity of the involved languages, we propose an approach that, after a preliminary abstract representation, first generates views organized according to the constructs in 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 and of the relational one. As a running example, we will see how relational views can be generated to access an object-relational schema with references and inheritance.

Section 2 is an overview of the work and the organization of the rest of the paper is described at the end of it.

2. OVERVIEW

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

The starting point for this work is MIDST [3, 5], a platform for model-independent schema and data translation based on a metalevel 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 refer to the basic constructs and are performed within the supermodel. In the current implementation, they are specified in Datalog.
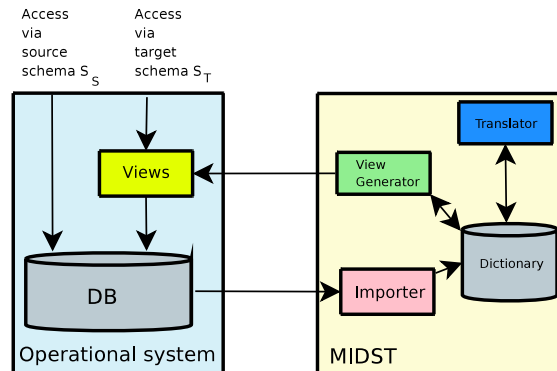

Figure 1: The runtime translation procedure

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

Let us illustrate our approach, by following the main steps it involves, with the help of Figure 1:

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

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

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

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

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

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

As a running example, consider the following. Assume we have an environment where application programs are designed to interact with relational databases while we have an actual database on the operational system based on the object-relational (OR) model, with the following features:[2]

---

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

tables, typed tables, references between typed tables and generalizations over typed tables. In this scenario, our tool generates relational views over the object-relational schema, which can be directly used by application programs.

A concrete case for this example involves the OR schema sketched in Figure 2. The boxes are typed tables: employee (EMP) is a generalization for engineer (ENG) and department (DEPT) is referenced by employee.
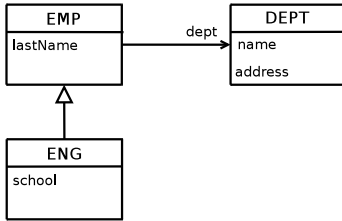

Figure 2: A simple object-relational schema

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:[3]

EMP (<u>EMP_OID</u>, lastname, DEPT_OID)
DEPT (<u>DEPT_OID</u>, name, address)
ENG (<u>ENG_OID</u>, school, EMP_OID)

Given the schema in Figure 2, our tool first imports it in its dictionary. Then, given the specification of the target model (the relational one), it selects an appropriate schema-level translation, which is a sequence of basic translations, each specified by means of a Datalog program. In this case, the schema-level translation should perform the following tasks: it first eliminates the generalizations (in the example, the one between ENG and EMP) and then transforms the typed tables (all tables in the source) into value-based tables. As we will see in the next section, 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, preceded by two auxiliary ones, for the introduction of keys and the replacement of references with foreign keys. The major task of our new version of the tool is the generation of a set of view statements for each of these Datalog programs.

The following is a sketch of a view definition generated in the first step.

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

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

The remainder of the paper is organized as follows. In Section 3 we briefly present the original version of MIDST,

---

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

specifically we introduce some useful details on how schema-level translations are performed within the platform. In Section 4 we illustrate the whole process of runtime translation, explaining how views are generated out of schema-level rules by describing concrete cases of application. Then, in Section 5 we formally present the algorithm for the generation of views from schema-level translations. In Section 6 we discuss some related work; finally in Section 7 we draw our conclusions.

3. TRANSLATIONS IN MIDST

Let us now introduce some of MIDST main features, as needed for the subsequent discussion. MIDST 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, up to renaming, all the constructs of interest. Each model, then, is a specialization of the supermodel, and each schema in any model is also a schema in the supermodel. Therefore, translations from a model to another can be performed within the supermodel.

Figure 3 (taken, with minor variations, from [5]) reports a list of MIDST generic constructs and shows examples of their use in most common models. 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 one another). Each construct is provided with a unique identifier and references are then based on these identifiers.

Let us illustrate the main constructs by means of the running example, the object-relational schema of Figure 2. Each of the typed tables (EMP, ENG, and DEPT) is seen as an Abstract in the supermodel. Then, each column of the typed tables (school for ENG, lastName for EMP, name and address for DEPT) is a Lexical and is related to the corresponding Abstract. Something similar happens for reference fields, so dept is an AbstractAttribute of EMP, and for generalizations, where we say that ENG is a child of EMP.

Each translation in MIDST is encoded as a Datalog program, which is a set of Datalog rules. 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 adoption of the Skolem functor, denoted by "SK" in the example, which, given the OID of an Abstract, yields 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. 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, for a given construct the ranges of the Skolem functions defined over it are disjoint.

Indeed, complex translations require several Datalog programs to specify the transformation of each construct. In

Metaconstruct	Relational	Object-relational	ER	XSD
Abstract	-	typed table	entity	root element
Lexical	column	column	attribute	simple element
BinaryAggregationOfAbstracts	-	-	binary relationship	-
AbstractAttribute	-	reference	-	-
Generalization	-	generalization	generalization	-
Aggregation	table	table	-	-
ForeignKey	foreign key	foreign key	-	foreign key
StructOfAttributes	-	structured column	-	complex element

Figure 3: Simplified representation of MIDST metamodel

MIDST we provide translations among every model of Figure 3 [5].

We adopt a modular approach and decompose translations into simple steps. MIDST includes an inference engine that, given a source and a target model, detects the needed translation steps.

Each step is encoded by a Datalog program and represents an elementary transformation that must be performed. This keeps the level of detail relatively simple and supports significant reuse of rules. Therefore we can concentrate on simple steps. It is important to point out that each step returns a coherent schema as a result; that schema is then used by other steps to perform further phases of the whole translation.

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 [3, 5] this could be done as a process in four main phases: elimination of hierarchies (step A), generation of identifiers for the typed tables without them (step B), elimination of reference columns (step C) and transformation of typed tables into tables (step D).

The general approach to translation is that of copying, with a simple "copy rule," all the constructs that are not modified. For example, in order to copy an Abstract, we need to specify the *copy-abstract* rule (R_1):

```
R1    Abstract ( OID: SK0(oid),
          Name: name )
      <- Abstract ( OID: oid,
          Name: name );
```

When actual transformations are needed, rules are more complex.

As for step A, there are various ways to eliminate generalizations. Let us refer to the one that maintains both the parent and the child typed tables and connects them with a reference. This requires that we copy all typed tables with their columns and then add a new column for each child typed table with a reference to the respective parent typed table.

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

```
R4    AbstractAttribute (
        OID: SK2(genOID, parentOID, childOID),
        Name: name,
```

```
        isNullable: "false",
        abstractOID: SK0(childOID),
        abstractToOID: SK0(parentOID) )
    <- Generalization ( OID: genOID,
        parentAbstractOID: parentOID,
        childAbstractOID: childOID ),
      Abstract ( OID: parentOID,
      Name: name );
```

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).

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

```
R5    Lexical ( OID: SK3(absOID),
          Name: name + "_OID",
          IsNullable: "false",
          IsIdentifier: "true",
          type: "integer",
          abstractOID: SK0(absOID) )
      <- Abstract ( OID: absOID,
          Name: name ),
        ! Lexical (
          IsIdentifier: "true",
          abstractOID: absOID );
```

As in the previous step, we need copy rules for the various 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 specifies this: for each AbstractAttribute (reference) it replicates the key Lexicals of the referred typed table into the referring one (R_6).

```
R6    Lexical ( OID: SK4(oid,lexOID),
          Name: lexName,
          isIdentifier: "false",
          Type: type,
          abstractOID: SK0(absOID) )
      <- AbstractAttribute ( OID: oid,
          abstractOID: absOID,
```

```
            abstractToOID: absToOID),
    Lexical ( OID: lexOID,
            Name: lexName,
            abstractOID: absToOID,
            IsIdentifier: "true",
            Type: type );
```

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 two Datalog rules. The first translates Abstracts into Aggregations (R_7), the second transforms LexicalOfTypedTables into LexicalOfTables (R_8).

With respect to the running example of Figure 2, we have the following: step A eliminates the hierarchies, hence connects ENG to EMP with a reference. Step B creates an identifier for each of the typed tables, EMP, ENG and DEPT. In step C references are translated into value-based correspondences: the identifier of EMP is copied into ENG and the identifier of DEPT is copied into EMP. Finally step D performs the actual translation of EMP, ENG and DEPT into tables. The final result is indeed the relational schema we have already seen in Section 2:

EMP (<u>EMP_OID</u>, lastname, DEPT_OID)
DEPT (<u>DEPT_OID</u>, name, address)
ENG (<u>ENG_OID</u>, school, EMP_OID)

4. GENERATING VIEWS

In this section, we discuss the major ideas of our approach to the generation of views for runtime translations. Then, in the next section we will discuss the details in terms of a complete algorithm.

4.1 The general approach

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

A key idea in the procedure is a classification of MIDST metaconstructs according to the role they play. There are three categories: *container constructs, content constructs* and *support constructs*. Containers are the constructs that correspond to sets of structured objects in the operational system (i.e. Aggregation and Abstract corresponding to tables and typed tables respectively). Content constructs represent elements of more complex constructs,[4] such as columns, attributes or references: usually a field of a record (i.e. Lexical and AbstractAttribute) in the operational system. Support constructs do not refer to data-memorizing structures in the system, but are used to model relationships and constraints between them in a model-independent way. Examples are Generalizations (used to model hierarchies) and ForeignKeys (used to specify referential integrity constraints).

Our Datalog translation rules, in turn, can be classified according to the construct their head predicate refers to.

Therefore we have *container-* (for example, rules R_1 and R_7 in Section 3), *content-* (all other rules in Section 3) and *support-generating* rules.

Exploiting the above observations, the procedure defines a view for each container construct, with fields that derive from the corresponding content constructs. Instead, as support constructs do not store data, they are not used to generate view elements (while they are kept in the schemas). More precisely, given a Datalog schema rule $H \leftarrow B$, if H refers to a container construct, we will generate one view for each instantiation of the body of the rule. If H refers to a content, we need to define a field of a certain view. The head predicates of content-generating rules handle one OID, while the heads of container-generating ones deal with more OIDs. In fact, the role corresponds to an intrinsic structural difference between constructs. While containers have only one OID (which identifies the construct), contents have at least two OIDs: one identifying the field itself and one relating it to the owner container (other OIDs may be needed when fields refer to complex construct, as it happens for AbstractAttribute).

Two major issues in the procedure are the provenance of data (that is, where to derive the values from or how to generate them) for the single field and the appropriate combination of the source constructs (which, from a relational point of view, is equivalent to a join). In Subsection 4.2 we describe possible approaches to the former and conclude the illustration of the informal procedure by presenting examples of SQL statements in case we only deal with one source construct (all attributes refer to it); then we will abandon this assumption and comment on more general cases in which several source constructs must be combined (Subsection 4.3).

We will show the procedure informally with reference to the running example while technical details will be dealt with in Section 5.

Let us discuss step A in the example. 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 and DEPT_A.[5]

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 goal of step A is the elimination of hierarchies; rule R_4 maintains both the parent and the child and connects the constructs 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.

Let us now extend the same reasonings 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.[6] Rule R_6 recalls 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, 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. Many systems distinguish between *views* and *typed views*, then all we need is to handle this distinction.

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

4.2 The provenance of field values

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

In case it has only one parameter, the OID of another field, then the value comes from the instance of the construct having that OID. This is what happens in steps A, B and C whenever a Lexical is copied (rule R_2). Similarly, if the Skolem functor has more than one parameter and one of them refers to a field, then a source construct can be individuated as well.

Instead, if none of the functor parameters refers to a content (it only deals with container or support constructs), the result value has to be generated somehow. This is exactly what happens in steps A, B and C with rules R_4, R_5 and R_6 respectively.

These cases can be handled automatically as well. We introduced solutions that are based on annotations which specify value generation techniques. Here we present an informal description of this approach to give an intuition of

the adopted strategy while technical details will be pursued in Section 5. In rule R_4, the functor generates the OID for a reference field (AbstractAttribute) from the OID of a Generalization (a support construct)

In order to obtain a reference from the child table to the parent it is possible to use the tuple OID[7] as value for the reference field. A reason for this choice is the fact that every instance of a child typed table is an instance of the parent table too. Then for each tuple of the child container there is a corresponding tuple in the parent one with a restricted set of attributes, but with the same tuple OID. Therefore the reference can be made by means of an appropriate casting of this OID.

The following system-generic SQL-like statement is generated for the elimination of hierarchies (step A) in the running example. 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 OID.

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

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

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

4.3 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 need not be the case. If two fields can be accessed from the same container it is wise to do it and to avoid joins. For instance if a construct C has a reference to a construct D and the fields c of C and d of D must be fetched, one can use that reference to get both the values from C without using the join operator. Moreover, a simpler variant is possible if all the fields of a given view derive from the same container, as it happens in all the steps of our example.

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

---

[6]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.

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

constructs, the collection of all the used functors encodes the join conditions.

For instance, consider another way of eliminating generalizations: copying the child attributes into the parent and deleting the child; obviously the parent will preserve its original attributes as well. We would have a content-generating rule for the parent, copying Lexicals from the child to the parent itself with the Skolem functor *SK2.1(genOID, parentOID, childOID, lexOID)*. Conversely, Lexicals from the source parent will be copied to the target one by means of the functor *SK5(lexOID)*. SK2.1 relates a Generalization (support construct) and two Abstracts (containers) and generates a new OID for the Lexical whose OID is lexOID. More simply, SK5 generates OIDs for Lexicals given the OID of another Lexical.

The specific set of content-generating functors ({SK2.1, SK5}) encodes the fact that we have a left join on OID basis in such a way that all the instances of the parent that are also instances of the child, appear in the result view as a single tuple. Moreover the left join guarantees the inclusion of the tuples coding instances of the parent that do not belong to the child.

In the running example we have EMP; according to the lastly mentioned strategy, it has to be merged with its child ENG. The general procedure establishes the presence of a join and the specific pattern for the set {SK2.1, SK5} encodes the conditions.

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

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

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

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

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

5. THE VIEW-GENERATION ALGORITHM

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

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

We will illustrate the technical details of the three parts in Subsections 5.1, 5.2 and 5.3 respectively.

5.1 Procedural analysis

Let us consider an elementary translation $\mathcal{T}$. It is a set of Datalog rules $R_1, R_2, \ldots, R_n$, where each rule R_i has a body B_i and a head H_i whose identifier (OID) is generated by means of a Skolem functor $S\kappa_i$.

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

For example, consider functor $S\kappa4$ of Section 3, used in the implementation of rule R_6 (which eliminates the references). It has the structure:

$$S\kappa4 : AbstractAttribute \times Lexical \rightarrow Lexical$$

meaning that it takes in input the OID of an AbstractAttribute and the OID of a Lexical and generates a unique OID for another Lexical, as it can be inferred from the head literal in which it is used. Also, $type(S\kappa4) = Lexical$.

Let us now investigate the relationship between the role of constructs and the Skolem functors used to generate their OIDs. A container construct has a single OID, which identifies it. Whenever a container is created in a head H_i, the functor $S\kappa_i$ is responsible for the creation and for the uniqueness of that OID. Conversely, a content construct is characterized at least by two OIDs: one that identifies the construct itself and another one referring to its container construct. The former plays the same role as in the containers and it is determined by the application of the Skolem functor $S\kappa_i$; the latter denotes the container construct to which the content belongs and it is calculated by another functor $S\kappa_i^p$. Symbols $S\kappa_i$ and $S\kappa_i^p$ will be used throughout the whole explanation of the procedure to denote the two functors for a content construct.

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

```
Abstract ( OID: SK0(oid),
       Name: name )
```

and it is evident that it is characterized only by its OID. Conversely, a content construct, such as Lexical as mentioned in the head of rule R_2 (which copies Lexicals), has at least two functors (one for each characterizing OID):

```
Lexical ( OID: SK5(lexOID),
         ...
         abstractOID: SK0(absOID) )
```

SK5 (SK_i) is the one used to generate unique values for instances of Lexical from OIDs of other Lexicals; SK0 (SK_i^p) is the one used to connect each instance of Lexical (content) to the proper Abstract (container) by retrieving the OID of the target Abstract (abstractOID) from the one of the source (absOID).

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

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

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

Notice that the body is evaluated only against MIDST supermodel, where the preliminary import phase has generated a representation for the schema of the operational system in terms of MIDST constructs. For example, consider the body of rule R_4 (eliminating Generalizations), an instantiated version of it is the following one:

```
<-
Generalization ( OID: 101,
         parentAbstractOID: 1,
         childAbstractOID: 2 );
```

In the running example, it expresses the fact that the Abstract representing the typed table EMP (with OID 1) is the parent of the Abstract representing the typed table ENG (with OID 2). As it is possible to infer from the example, the conditions expressed in the bodies of Datalog rules (which are evaluated within MIDST supermodel) may refer to container and content constructs as well as to support ones.

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 IB of B. Again with reference to R_4, we have the following instantiated head:

```
AbstractAttribute ( OID: SK2(101, 1, 2),
         isNullable: "false",
```

```
         abstractOID: SK0(2),
         abstractToOID: SK0(1) )
```

This head defines a new AbstractAttribute for a given Generalization involving two Abstracts and, with respect to the operational database system, defines a reference column for a typed table referring to another typed table.

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

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

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

- Given a translation $\mathcal{T}$ and a container-generating rule R in $\mathcal{T}$, we denote as $content(R, \mathcal{T})$ the set of rules generating contents for R. In symbols, $content(R, \mathcal{T}) = \{R_j \in Contents(\mathcal{T}) \mid type(SK_j^p) = type(SK)\}$.

- For each $R \in Containers(\mathcal{T})$ (that is, for each container generating rule) we define an *abstract view*, as a pair $Av = (R, content(R, \mathcal{T}))$, composed of the rule itself and of a set of rules, those that define contents for its container. An abstract view is generic in the sense that it is written with respect to types of constructs. The same argument can be applied to contents: in abstract views, the content-generating rules define types of columns (or attributes, or references, etc) and not specific instances of them.

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

Let us now exemplify these concepts in our running example. Let $\mathcal{T}$ be the translation of step A. It follows that $Containers(\mathcal{T}) = \{R_1\}$ and $Contents(\mathcal{T}) = \{R_2, R_3, R_4\}$. Consequently we can determine the following abstract view: $Av_1 = (R_1, \{R_2, R_3, R_4\})$. Finally, it is possible to instantiate the abstract view Av_1 according to the constructs of the operational system. Then the instances[8] are:

$V_1 = (EMP \rightarrow_{\text{copy-abstract}} EMP$,
$\quad \{ EMP(lastName) \rightarrow_{\text{copy-lexical}} EMP(lastName)$,
$\quad EMP(dept) \rightarrow_{\text{copy-abstractAttribute}} EMP(dept)\})$

$V_2 = (DEPT \rightarrow_{\text{copy-abstract}} DEPT$,
$\quad \{ DEPT(name) \rightarrow_{\text{copy-lexical}} DEPT(name)$,
$\quad DEPT(address) \rightarrow_{\text{copy-lexical}} DEPT(address)\})$

$V_3 = (ENG \rightarrow_{\text{copy-abstract}} ENG$,
$\quad \{ ENG(school) \rightarrow_{\text{copy-lexical}} ENG(school)$,
$\quad Gen(EMP, ENG) \rightarrow_{\text{elim-gen}} ENG(EMP)\})$

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

---

[8]Notice that we do not change the name of the constructs after the application of the rules; anyway no ambiguity arises since they refer to different schemas.

5.2 View-generating statements

The second part of the procedure involves the translation of each instantiation V of every abstract view Av into a view-generating statement with the SQL structure:

```
CREATE VIEW name(col₁, col₂, ...)
AS (SELECT a₁(s₁.col₁), a₂(s₂.col₂), ...
    FROM source(col₁) s₁ cond source(col₂) s₂ cond ...
    )
```

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

Now we have to face two major issues to characterize the general statement: (a) the determination of the source container (in the statement denoted by $source(col_i)$) and (b) the actual value for each content (indicated with the functional symbols a_i); this problem has been informally discussed in Section 4.2 and consists in establishing a way of computing the values for fields in the result views, hence assigning a semantics to the functional symbols a_i in the statement. Problem (b) consists in the determination of the appropriate form of combination needed for the source containers of the various contents (indicated with the symbol $cond$). This problem has been discussed in Section 4.3 and consists in replacing $cond$ with appropriate join conditions in the statement. We will discuss issues (a) and (b) separately in the remainder of the section.

As for point (a), given a content-generating rule R_i consider its functor, Sk_i^p, that is, the one that links the head construct to the parent. The parameters of the functor are instantiated as a consequence of the instantiation of B_i and link the generated content with its source container (the one the functor Sk_i^p is applied on). Sk_i conveys information about the provenance of data (that is the content to derive the value from) for the content under examination.

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

More precisely:

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

- if Sk_i is annotated (case a.2) with a query a, then a is applied in order to calculate the needed value. Notice that the query can be written referring to all the literals in the instantiated content-generating rule. Generally, these queries are very simple and use a small number of parameters. In the SQL statement above,

the functional symbols a denote the application of the query associated to an annotation (in the default cases this query has no effect).

As an example of case (a.1), consider the rule R_6 of step C, presented in Section 3, which replaces the references of typed tables with simple fields (in order to allow for the definition of value-based correspondences). The functor Sk4 is not annotated and takes in input the OID of the AbstractAttribute and the OID of the Lexical referred by it. This implies that values for the new field (generated to represent a reference) have to be directly derived (namely, copied) from values of the source Lexical.

On the other hand, as an example of case (a.2), consider the rule R_4 of step A, presented in Section 3, which replaces the generalizations between two typed tables by adding a specific reference field (AbstractAttribute) in the child table. The functor Sk2 takes in input the OID of the Generalization and the ones of the two involved Abstracts. In this case it is correct to annotate the functor to specify how the values for the field have to be calculated.

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

```
SELECT INTERNAL_OID FROM childOID;
```

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

A similar strategy should be followed to cope with rule R_5 of 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 functor $\text{Sk3}(absOID)$ takes an Abstract as input parameter, meaning that there are no valid sources for the values. A possible annotation could be the following one:

```
SELECT INTERNAL_OID FROM absOID;
```

It implies the adoption of the values of internal tuple identifiers (INTERNAL_OID) as elements for the key of the typed table as explained in Section 4.

As for point (b) two cases are indeed possible for an instantiated view V_i in dependence on the instantiation of the functor Sk_i^p: (b.1) there are sets of contents deriving from the same container, let us call them *sibling contents*. This corresponds to instantiations of rules (be they the same or different ones) where the values of the parameters of the functors Sk_i^p are the same; (b.2) there are contents deriving from different containers, let us call them *non-sibling contents*; this corresponds to instantiations of rules where the assigned values for the functor Sk_i^p are different.

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

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

On the other hand, in (b.2) the $cond$s in the SQL statement must be translated into appropriate join conditions. It is clear that there are several variants for the joins, according to the semantics of the schema-level translation. The

key is that Skolem functors allow to specify this semantics at schema level in such a way that it can be translated into join conditions at data level. We define a *schema-join correspondence* SJ such that SJ : $S^n \rightarrow cond$, where S^n is a tuple of Skolem functors, *cond* is a join condition, expressed as a statement at schema level. The correspondence SJ then assigns a join condition to a specific tuple of functors, which are the ones that generate the OIDs for the contents in the container under examination. Then, for example, if a container has three contents: two sibling contents and a non-sibling one, then the tuple will be composed of two functors, one for the siblings and another one for the single content. Then the correspondence SJ will specify how to combine the two associated source containers in terms of join conditions. As for annotations, join conditions must be written at schema level (for example directly with a pseudo-SQL formalism) and, when omitted, the Cartesian product between the source containers is implied.

Case (b.1) is rather simple and an example of it is the overall translation of step A where, for each typed table, the values are directly derived from one source table and no joins are needed.

Conversely, as seen in Section 4, an occurrence of (b.2) arises in the elimination of generalizations consisting in copying the contents of child Abstracts into the parent. Obviously, since the parent maintains its contents, there are contents coming from the child typed table and others from the parent one. The involved functors are SK2.1, the one responsible for the OIDs of Lexicals copied from the children to the parents (school from ENG to EMP in the example), and SK5, responsible for the OIDs of the parent Lexicals (lastName of ENG in the example). Here we define the schema-join correspondence $f : (SK2.1(genOID, parentOID, childOID, lexOID), SK5(lexOID)) \rightarrow cond_1$, where $cond_1$ can be defined according to a pseudo-SQL formalism as follows:

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

This pseudo-SQL condition, together with the schema-join correspondence definition, specifies that, whenever two non-sibling set of contents derive from the combination of the functors SK2.1 and SK5, then the source containers have to be combined with a left join on the basis of the internal OID. The left join guarantees that instances of the parent that are not also instances of the child are preserved in the result. It is clear that different correspondences, in association with different join conditions, can be defined to cover a wide range of cases.

5.3 Executable statements

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

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

The following SQL statements exemplify the elimination of hierarchies (rule R4) 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 EMP2 and ENG2 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. Examples are the use of type constructors, the various casting functions or explicit scope modifiers.

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

CREATE TYPE ENG2_t as (
     toEMP REF(EMP2_t),
     school varchar(50))
...;

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

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

5.4 Discussion

The proposed algorithm represents the core step in translating schemas from a model to another at runtime since it allows the translation of Datalog rules into actually executable SQL statements on data. In previous works [3, 4, 5] we argued that MIDST is a model-independent implementation of the MODELGEN operator. Here we argue that the proposed algorithm extends the platform to a runtime context and allows for the interaction with heterogeneous database systems, without affecting that property. In fact the initial import of information about the schema of the operational database supports the definition of system-independent and model-independent translations. We manage to decouple the technical details of the operational system and its model from translation rules, by means of suitable import modules that allow to translate the internal representations of the systems in terms of the constructs of the supermodel.

Then the schema-level rules are actually applied on the supermodel in order to obtain schema information about the translated database, in such a way that further operations are possible. What the algorithm performs is a procedural analysis of translation on the basis of a generic whole-part (container-content) classification of supermodel constructs and on the basis of the *model-awareness* principle we foster in MIDST. It means that, although MIDST is model-generic, in the sense that translations can be applied independently of target and source models, we handle specific metadata about models by adopting typed constructs which differ from one another, and strongly typed Skolem functors, which can be applied on and return only specific types

of construct OIDs. Hence, as evident from the formal illustration of the algorithm, model-awareness allows to evaluate the relationships between constructs and their instances in the operational system without affecting the model-generic approach of the whole process.

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

The presented approach solves performance issues that affected MIDST due to the necessity to import into the supermodel and export back the whole database. Schema metadata are obviously much lighter than the actual data and the time spent in importing them has no relevance in the performance of the translation. Furthermore, the computation of the SQL-generating statements is performed only once (and in advance) for each translation; then the optimization of the query and the performance issues are entirely devoted to the operational database system. From our point of view, in other papers [3, 4, 5] we showed that although MIDST is model-independent, hence it handles translations between any pair of models, the number of the needed steps is bounded and small. Moreover, the number of the generated queries is minimal. In fact, due to the detection of the appropriate join conditions, we generate one query for each view needed in the operational system and do not need to unite results from different statements.

6. 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 [8] present the current state of the art in this field and, indirectly, outline an overview of the major approaches and achievements.

The approach towards runtime translation illustrated in this paper is based on MIDST [3, 4, 5], a platform allowing for model-independent schema and data translation. Its theoretical basis are laid in [3, 5, 6, 7] and provide a framework to perform model-independent schema and data translation. In this paper we provide the framework with a runtime design and go beyond the limitations expressed in [8, Sec.3.1].

The problem of translating schemas between models has been pursued by various other authors, including [13, 15], with approaches that are either focussed on the schema level or on abstract models and languages.

In this paper we have tackled the problem of enhancing MIDST with the possibility of applying runtime translations in such a way that data exchange queries are computed out of schema translation rules and are used to generate views. Our approach towards data exchange is not formal, what we are interested in is the set of statements solving the data exchange problem between the source schema and the wanted view; however we share many ideas with characterizations by Fagin et al.[11, 10].

Mork et al. [17] also adopt a runtime approach (based on [3, 7]) to solve the specific problem of deriving a relational schema from an extended entity-relationship model. They use a rule-driven approach and write transformations that are then translated into the native mapping language. However, although they face many issues such as schema update propagation and inheritance, indeed they solve a specific subset of problems and provide an object-relational mapping tool similar to [14]. In [9], Bernstein et al. adopt a runtime approach to allow a developer to interact with XML or relational data in an object-oriented fashion. On the one hand their perspective is different since they only deal with a specific kind of heterogeneity; in addition they address the problem by translating the queries while we aim at generating views on which the original queries can be directly applied.

Our approach is aimed at providing a runtime support to the whole range of translations allowed by MIDST that is not limited to 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 [10, 11, 12, 16, 19] too. Its aim is building a completely defined mapping between two schemas, given a set of user-defined correspondences. As for our translations, these mappings could be translated into directly executable SQL, XQuery or XSLT transformations. However, in the perspective of adopting Clio in order to exchange data between two heterogeneous schemas, the needed mappings should be defined manually; moreover, there is no kind of model-awareness in Clio, which operates on a generalized nested relational model. Although it can be shown to subsume a considerable amount of models, in a real application scenario a preliminary translation and adaptation of the operational system should be performed, leading to the problems of the initial MIDST approach.

The presented runtime extension of MIDST is a significant step with respect to the process of turning the platform into a complete model management system [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 [2].

7. CONCLUSIONS

The main contribution of this paper is a runtime version of MIDST. We have showed 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.

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, some situations are more complex and need further investigation. Examples are the ones involving translations from object-relational to XML and viceversa

The concrete examples we have shown in this paper are based on SQL, which has the advantage of supporting different models, in particular the object-relational (which has many variants) as well as the relational one. However, the approach we have shown has a significant language independent step that can be the basis for further experimentation, especially in the XML world, possibly in conjunction with SQL itself.

The runtime approach 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.

8. 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 platform for model-independent solutions to model management problems (extended abstract). In *SEBD*, pages 310–317, 2008.

[3] P. Atzeni, P. Cappellari, and P. A. Bernstein. Model-independent schema and data translation. In *EDBT Conference, LNCS 3896*, pages 368–385. Springer, 2006.

[4] P. Atzeni, P. Cappellari, and G. Gianforme. MIDST: model independent schema and data translation. In *SIGMOD Conference*, 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, G. Gianforme, and P. Cappellari. Reasoning on data models in schema translation. In *FOIKS Symposium, LNCS 4932*, pages 158–177. Springer, 2008.

[7] 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.

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

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

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

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

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

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

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

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

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

[17] 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.

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

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