MISM: A Platform for Model-Independent Solutions to Model Management Problems

Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, and Giorgio Gianforme

Dipartimento di informatica e automazione
Università Roma Tre
{atzeni}@dia.uniroma3.it,
{bellomarini,bugiotti}@yahoo.it,
{giorgio.gianforme}@gmail.com

Abstract. Model management is a metadata-based approach to database problems aimed at supporting the productivity of developers by providing schema manipulation operators.

Here we propose MISM (Model Independent Schema Management), a platform for model management offering a set of operators to manipulate schemas, in a manner that is both model-independent (in the sense that operators are generic and apply to schemas of different data models) and model-aware (in the sense that it is possible to say whether a schema is allowed for a data model). This is the first proposal for model management in this direction.

We consider the main operators in model management: merge, diff, and modelgen. These operators play a major role in solving various problems related to schema evolution (such as data integration, data exchange or forward engineering), and we show in detail a solution to a major representative of the class, the round-trip engineering problem.

Keywords: model management, model management operators, round-trip engineering, model-independent schema and data translation.

1 Introduction

The need for complex transformations of data arises in many different contexts, because of the presence of multiple representations for the same data or of multiple sources that need to coexist or to be integrated [11,18,20]. A major goal of technology in the database field is to enhance the productivity of software developers, by offering them high-level features that support repetitive tasks. This has been stressed since the introduction of the relational model, with the emphasis on set-oriented operations [12,13], but it was pursued, at least implicitly, in earlier developments of generalized techniques [22]. The *model management* proposal [7,8] is a recent, significant effort in this direction: its goal is the development of techniques that consider metadata and operations over them. More precisely, a model management system [11] should handle schemas and mappings between them by means of operators supporting operations to discover correspondences between

S. Spaccapietra, L. Delcambre (Eds.): Journal on Data Semantics XIV, LNCS 5880, pp. 133–161, 2009.

schemas (MATCH), performing the most common set-oriented operations (such as union of schemas, MERGE, and difference of schemas, DIFF) and translating them from a data model to another (MODELGEN). These operations should be specified at a high level, on schemas and mappings, and should allow for the (support to the) generation of data-level transformations. Many application areas can benefit from the use of model management techniques, including data integration over heterogeneous databases, data exchange between independent databases, ETL (Extract, Transform, Load) in data warehousing, wrapper generation for the access to relational databases from object-oriented applications, dynamic Web site generation from databases.

Most of the work in model management has considered the need for *model independence*, that is, the fact that the techniques do not refer to individual data models,[1] but are more general. In detail, this requires that a single implementation of the operators should fit (i.e. be applicable) to any schema regardless of the specific data model it belongs to. This has usually been done by adopting some "universal data model," a model that is more general than the various models of interest in a heterogeneous framework. In the literature, such a data model is called *universal metamodel* [11] or *supermodel* [3,6]. If the operations of interest also include translations from a data model to another (the MODELGEN operator), it is important that the individual data models are represented, in such a way that it becomes possible to describe the fact that a schema belongs to a data model. We will call this property *model-awareness*. The various proposals for MODELGEN [3,6,25,26] do include the model independence feature, to a larger or lesser extent. For the other operators, the major efforts in the model management area (as summarized by Bernstein and Melnik [11]) do not handle the explicit representation of data models nor generic definitions of the operators.

The goal of this paper is to show a model independent and model aware approach to model management, thus providing concrete details to Bernstein's original proposal [8] and contributing to support its feasibility.

In the rest of this introductory section we first discuss two motivating examples, then we provide an overview of the approach and finally we state the contribution of the paper and the organization of the rest of it.

1.1 Motivating Examples

In order to have a context for specific examples and a complete solution, we will refer to the "round-trip engineering" problem [8], which can be defined as follows: given two schemas, where the second is somehow obtained from the first (for example, generated in a semiautomatic way, with standard rules partially overridden by human intervention), the problem has the goal of "repairing" the first if the second is modified. This problem is often considered in model management papers [8] as a representative of the "schema evolution" family. These problems arise in all application settings and therefore can be used to

[1] There is some disagreement on terminology in the literature: we use the term *data model* here for what is often called just *model* [3,6] and in other papers *metamodel* [11].

demonstrate the effectiveness of model management, in terms of both individual operators and compositions of them.

Let us consider an example derived from an academic scenario (see Figure 1): a university has various schools and one of them has a relational database with a portion containing all the information of interest about its departments, courses, and professors. Its schema is shown in the box labeled S_1 in Figure 1. It is composed of three tables, *Professor*, *Course*, and *Department*. Apart from the specific attributes, each relation has a key, denoted by the "ID" suffix and underlined in the figure. As each course is offered by a specific department and given by a professor, there are foreign keys from *Course* to the other two tables, denoted by arrows in the figure.

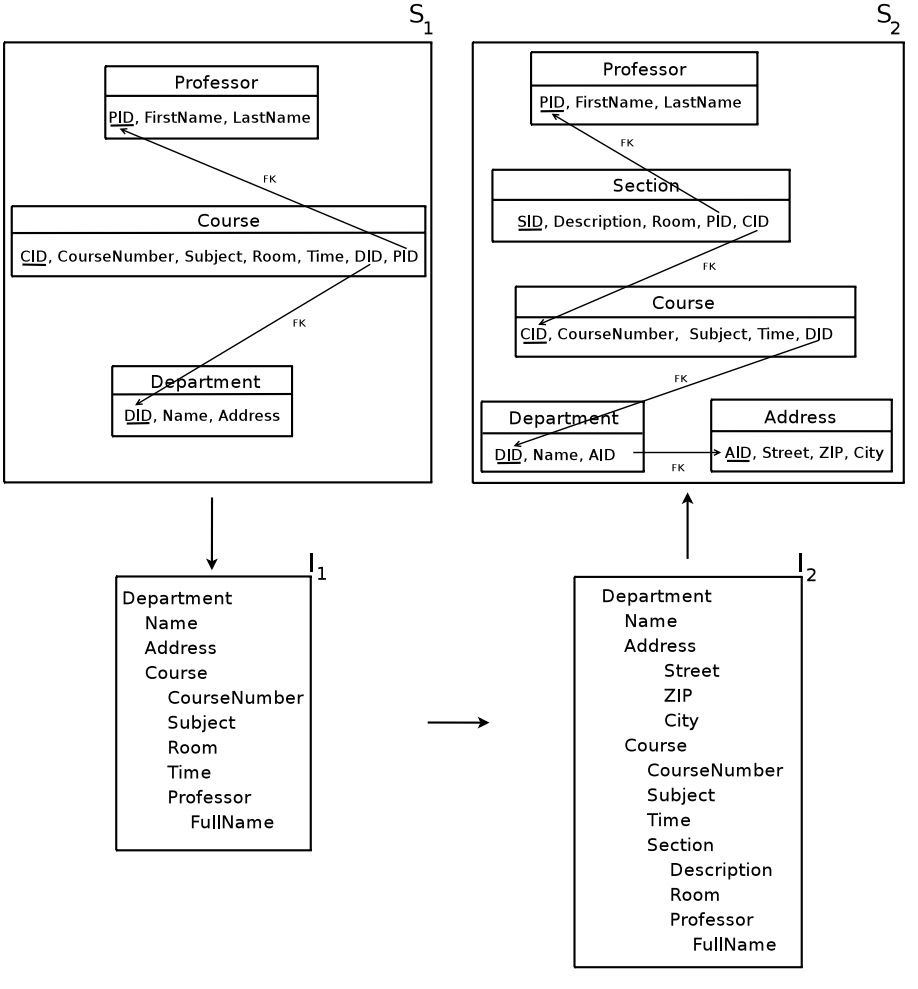

Fig. 1. The round-trip engineering problem

Assume now that this portion of the database is used (together with other goals) as the source to send data on courses to a central office in the university, which gathers data from all schools. This office requires data in an XML format, which is the one sketched in the box labeled I_1 in Figure 1. There is indeed a close correspondence between S_1 and I_1 (possibly because they were designed together). In fact, I_1 can be obtained by means of a nesting operation based on departments, each with the associated set of courses and with the instructor for each course. Clearly, this is one natural way to transform the relational data in S_1 into XML, but not the only one, as there would be other solutions that involve course or professor as the root. In this sense, we can say that this is not the result of an automated translation, but of a customization, that is, a choice among a few standard alternatives. Let us also observe that in S_1 we have attributes *FirstName* and *LastName* for *Professor*, whereas in I_1 we have the element *FullName*. There could be various reasons for this, but the only aspect relevant here is that, again, the transformation has been customized, with the concatenation of the two attributes in S_1 into a single element in I_1.

Then, assume that the exchange format is modified, with a new version, I_2, also shown in Figure 1. There are a few differences between I_2 and I_1. First, we have that *Address* is a simple element in I_1, while it is a complex element in I_2, composed of *Street*, *Zip*, and *City*. The second, and most important, difference is the presence of a complex element *Section* nested in *Course* and containing *Professor*. A course can be composed of various sections. Each section has a single professor, and therefore *Professor*, which in I_1 was directly contained in *Course*, is part of *Section*. Each section of a course takes place in a different room so the element *Room* is now in *Section*.

Now, the goal is to obtain a schema in the relational model (for example the one shown in the box labeled S_2 in Figure 1) that properly corresponds to S_1 as modified by the changes in I_2. It should be clear that S_2 cannot be obtained by applying to I_2 a standard, automatic translation from XML to the relational model (an application of the MODELGEN operator), because we could not keep track of the customizations we mentioned above. The idea for a solution to this problem was proposed by Bernstein [8], in terms of a script of model management operators, using DIFF to compute differences, MODELGEN to translate and MERGE to integrate. Intuitively, we have to detect the actual differences between the original and the modified target schemas I_1 and I_2 respectively. Then we have to translate these differences back to the specification model (in our case the relational one) and finally integrate the translated differences with the original specification S_1 obtaining a revised specification S_2. The requirement is that we should obtain I_2, if we apply to S_2 the sequence of translations and customizations used to obtain I_1 from S_1. With reference to our example, applying the sequence of operators as described in the algorithm, we produce indeed the relational schema illustrated in the box labeled S_2 in Figure 1. Schema S_2 includes new tables *Section* and *Address* corresponding to the new complex elements in I_2. *Department* has a foreign key to *Address* and *Section* to *Course*. Also, attribute *Room* is in *Section* and not anymore in *Course*.

In the existing literature, the proposals for the various operators are not general and accurate enough, as they refer to a rather limited set of models and do not have features that support the description of models, and so the plan proposed by Bernstein has not yet been implemented in a general way.

The goal of this paper is to show that this plan can indeed be made concrete, in a model-independent and model-aware way, which works for many different models but performs the translations knowing the specific features of the models of interest.

With the twofold goal of using a different model and of presenting a simpler example, let us consider another scenario. Let us assume we have a high level specification tool that translates ER schemas into relational tables by generating appropriate SQL DDL, allowing some customization. Again, if changes are made to the SQL implementation, then we want them to be propagated back to the ER specification. This is illustrated in Figure 2, where S_1 represents a specification in the ER model and I_1 represents its relational implementation. The customization in the translation produces two columns *FName* and *LName* in I_1 for the single attribute *Name* in S_1. Then, if I_1 is modified to a new version I_2, the latter is not coherent with S_1. The main difference between I_2 and I_1 is in the key for the

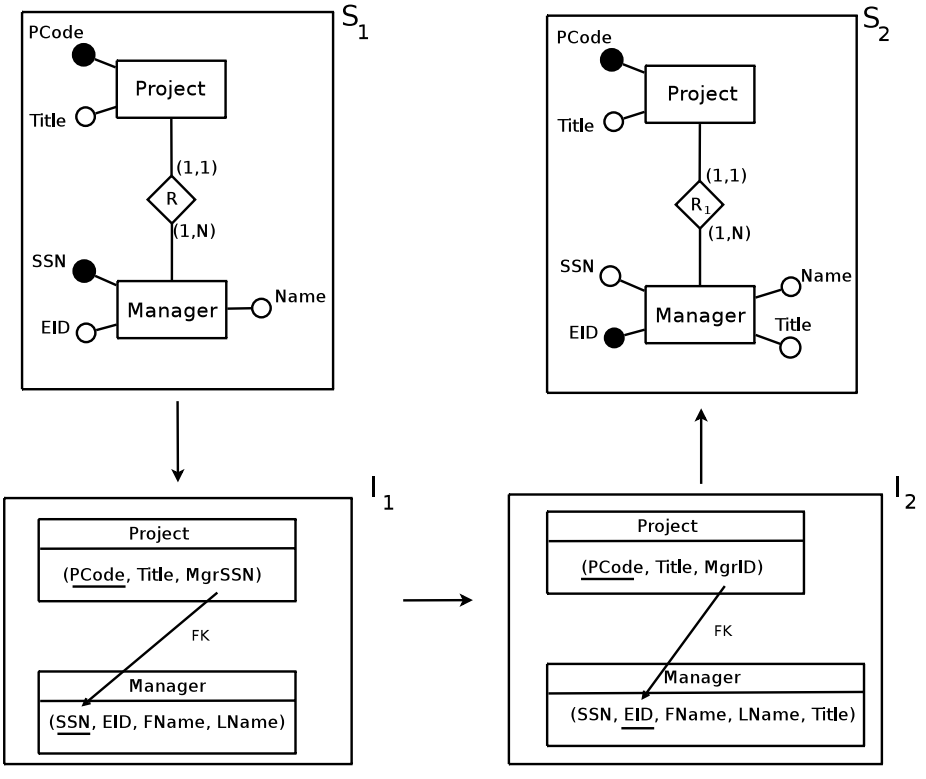

Fig. 2. A simple scenario for the round-trip engineering problem

Manager table and, as a consequence, in the foreign key structure that refers to it. Also, *Manager* has a new attribute, *Title*. The goal is to find a specification S_2 from which I_2 could be generated, in the same semiautomatic way as I_1 was obtained from S_1. Indeed, what we want to obtain is an ER schema S_2, which differs from the original one in the attributes of the entity *Manager*: the identifier is *EID* instead of *SSN* and there is the new attribute *Title*.

In the remainder of the paper we will follow this second example, which will allows us to explain completely our approach, without taking too much space.

1.2 Overview of the Approach

The solution we propose in this paper includes a definition and implementation of the major model management operators (DIFF, MERGE, and MODELGEN). It is based on our experience in the MIDST platform [3,4,5], where a model-independent approach for schema and data translation was introduced (with a generic implementation of the MODELGEN operator). MIDST adopts a meta-level approach in which the artifacts of interest are handled in a repository that represents data models, schemas, and databases in an integrated way, both model-independent and model-aware. This is a fundamental starting point, as stated before, in order to be able to define a model management system. This repository is implemented as a multilevel dictionary. Data models are defined in terms of the constructs they involve. A schema of a specific data model is allowed to use only the constructs that are available for that model. In this framework, the *supermodel* is the model that includes the whole range of constructs, so that every schema in every model is also a schema in the supermodel. Then, all translations are performed within the supermodel, in order to scale with respect to the size of the space of models [5]. In this paper, we show how the dictionary and the supermodel provide grounds for the model-independent definition of the other operators of interest, namely MERGE and DIFF.

In MIDST, translations are obtained as the composition of basic steps each of which is written as a Datalog program. The language was chosen for two reasons: first, it matches in an effective way the structure of our data model and dictionary (which is implemented in relational form); second, its high level of abstraction and the declarative form allow for a clear separation between the translations and the engine that executes them. Moreover, Datalog can be straightly translated into SQL and the original choice was aimed at covering the widest spectrum of application scenarios. However, other syntax or specification formalisms could be adopted as well.

Here we propose a general model management platform, MISM (Model Independent Schema Management), which is based on MIDST but extends it in a significant way. We start from MIDST's representation for data models, schemas, and databases and define model management operators by means of Datalog programs with respect to such representation. Specifically, we leverage on the features of MIDST's dictionary for the uniform representation of models as well as the infrastructure for the definition and the application of schema manipulation operators. MISM offers all the major operators, including MERGE, DIFF, and

a basic version of MATCH, all implemented in a model-generic way. The structure of the dictionary also allows for the automatic generation of Datalog programs implementing the new operators, with respect to the given supermodel, in such a way that, if the supermodel were extended, the operators would be automatically extended as well.

1.3 Contribution

To the best of our knowledge, this is the first proposal for a model-independent platform for model management. Specifically, this paper offers three main contributions:

- The model-independent definition and implementation of important model management operators. In fact, we define them by means of programs with predicates acting on the constructs of the supermodel.
- The automatic generation of the programs implementing the operators only using the supermodel as input. These programs are valid for any schema defined in terms of model-generic constructs.
- A complete solution to the round-trip engineering problem as a representative of the problems that can be solved with this approach. It is based on a script defined in terms of a convenient combination of our operators and allows a walk through of our implementation.

1.4 Organization of the Paper

In Section 2 we describe how models, schemas and translations are dealt with in MIDST. In particular we describe schema representation within MIDST metalevel. We illustrate how model-independent transformations can be performed in the framework.

Then in Section 3 we illustrate in detail model management operators in MISM, the extension of MIDST we propose here, and present their definitions. Discussion on their model-independence and model-awareness is provided. As a consequence, in Section 4 we show possible Datalog implementations for these operators satisfying the specifications of the previous section.

Section 5 presents a solution of the round-trip engineering problem in terms of our operators and shows how MISM can be used to solve this problem. A concrete scenario of solution, addressing the problem introduced in Figure 2 is then provided.

Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Models, Schemas, and Translations in MIDST

This section presents the needed background, with a discussion of the relevant features of our previous project, MIDST [3,5], whose goal was to provide a generic version of the MODELGEN operator, which can be defined as follows:

given a source schema S expressed in a source model, and a target model TM, MODELGEN generates a schema S' expressed in TM that is "equivalent" (according to a suitable definition) to S. MIDST obtains model-independence and model-awareness by means of the adoption of a rich dictionary, which stores models, schemas and data in a uniform and coordinated way. In this paper, we leverage on MIDST from two points of view: first, we show definitions and implementations of additional operators, MERGE and DIFF, and it is again the organization of the dictionary that supports model-independence and model-awareness; second, MODELGEN is used in the scripts we propose, together with the new operators. Hence both MIDST dictionary and its implementation of the MODELGEN operator are part of the new model management platform we propose in this paper.

MIDST adopts a model-generic representation of schemas based on a combination of constructs. Its founding observation is the similarity of features which arises across different data models. This means that all the existing models can be represented with a rather small set of general purpose constructs [21] called *metaconstructs* (or simply *constructs* when no ambiguity arises). Let us briefly illustrate this idea. Consider the concept of entity in the ER model family and that of class in the OO world: they both have a name, a collection of properties and can be in some kind of relationship between one another. To a greater extent, it is easy to generalize this observation to any other construct of the known models and determine a rather small set of general constructs. Therefore models are defined as sets of constructs from a given universe, in which every construct has a specific name (such as "entity" or "object"): for instance a simple version of the ER model may be composed of Abstracts (the entities), Aggregations of Abstracts (the relationships) and Lexicals referring to Abstracts (attributes of entities); instead the relational model could have Aggregations (the tables), Lexicals referring to Aggregations (the columns), and foreign keys specified over finite sets of Lexicals. Thus schemas are collections of actual constructs (schema elements) related to one another. Figure 3 lists the metaconstructs used in the current version of MIDST [5] and the corresponding specific constructs we have in various popular (families of) data models.

As we said in the introduction, the set of all the possible constructs in MIDST forms the *supermodel*, a major concept in our framework. It represents the most general model, such that any other model is a specialization of it (since a subset of its constructs). Hence a schema S of a model M is necessarily a schema of the supermodel as well.

MIDST manages the information of interest in a rich dictionary. Its details have been described elsewhere [2] and are beyond the scope of this paper. Let us summarize its main features. It has two layers, both implemented in the relational model: a *basic* level and a *metalevel*.

The basic layer of the dictionary has a model-specific part (some tables of which are shown in Figure 4 with reference to our running example), where schemas are represented with explicit reference to the various models, and, more important, a model-generic one, where there is a table for each construct in the supermodel:

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

Fig. 3. Simplified representation of MIDST metamodel

ER_ENTITY

OID	Entity-Name	Schema
e1	Project	s1
e2	Manager	s1
...	...	...

ER_ATTRIBUTEOFENTITY

OID	Entity	Att-Name	Type	isKey	Schema
a1	e1	PCode	int	true	s1
a2	e1	Title	string	false	s1
a3	e2	SSN	int	true	s1
a4	e2	EID	int	false	s1
a5	e2	Name	string	false	s1
...	...	...	...	...	...

ER_BINARYRELATIONSHIP

OID	Rel-Name	Entity1	IsOptional1	IsFunctional1	Entity2	...	Schema
b1	R	e1	false	true	e2	...	s1
...	...	...	...	...	...	...	...

REL_TABLE

OID	Table-Name	Schema
t1	Project	i1
t2	Manager	i1
...	...	...

REL_COLUMN

OID	Table	Col-Name	Type	isKey	Schema
c1	t1	PCode	int	true	i1
c2	t1	Title	string	false	i1
...	...	...	...	...	i1
c7	t2	LName	string	false	i1
...	...	...	...	...	...

Fig. 4. A portion of a model-specific representation of schemas S_1 and I_1 of Figure 2

so there is a table for SM_ABSTRACT (the SM_ prefix emphasizes the fact that we are in the supermodel portion of the dictionary), a table for SM_AGGREGATION and so on (with an example in Figure 5). These tables have a column for each property of interest for the construct (for example, a Lexical can be part of the identifier of the corresponding Abstract, or not, and this is described by means of a boolean property). References are used to link constructs to one another, and

SM_ABSTRACT

OID	Abs-Name	Schema
e1	Project	s1
e2	Manager	s1
...	...	...

SM_AGGREGATION

OID	Aggr-Name	Schema
t1	Project	i1
t2	Manager	i1
...	...	...

SM_LEXICAL

OID	Abstract	Aggr	Lex-Name	Type	isId	Schema
a1	e1	-	PCode	int	true	s1
a2	e1	-	Title	string	false	s1
a3	e2	-	SSN	int	true	s1
...	...	...	...	...	...	...
c1	-	t1	PCode	int	true	i1
...	...	...	...	...	...	...
c7	-	t2	LName	string	false	i1
...	...	-	...	...	...	...

SM_BINARYAGGREGATIONOFABSTRACTS

OID	Agg-Name	Abstract1	IsOptional1	IsFunctional1	Abstract2	...	Schema
b1	R	e1	false	true	e2	...	s1
...	...	...	...	...	...	...	...

Fig. 5. A portion of a model-generic representation of the schemas S_1 and I_1 of Figure 2

so the tables in the dictionary have fields with foreign keys connecting them to each other. For example, the SM_LEXICAL table has an attribute that contains references to SM_ABSTRACT, to represent the fact that a Lexical (for example an attribute of entity in the ER model) has to belong to a parent construct, which could be an Abstract (an entity). In both parts, constructs are organized in such a way they guarantee the *acyclicity constraint*, meaning that no cycles of references are allowed between them. This is convenient in situations where a complete navigation through the schemas is necessary and a topological order is helpful.

The two parts of the dictionary play complementary roles in the translation process, which is MIDST's main goal: the model specific part is used to interact with source and target schemas and databases, whereas the supermodel part is used to perform translations, by referring only to constructs, regardless of how they are used in the individual models. This allows for model-independence.

In fact, every translation in MIDST is composed of three phases: first, the source schema, expressed in a specific source model, is copied into the supermodel; second, the actual translation is carried out in the supermodel environment; finally, the result schema, which refers to the supermodel, but is compatible with the target model, is copied into the target model itself. The translation engine exploits a library of elementary translations, each of which is written as a Datalog program, and combines them, on the basis of the specific source and target model of interest.

MIDST dictionary includes a higher layer, a *metalevel*, which gives a characterization of the construct properties and relationships among them [2,5]. It involves few tables, each with few rows, which form the core of the dictionary. A significant portion is shown in Figure 6. Its main table, named MSM_CONSTRUCT (here, the MSM_ prefix denotes that we are in the "metasupermodel" world, as we

MSM-CONSTRUCT

OID	Construct-Name	IsLex
mc1	Abstract	false
mc2	Lexical	true
mc3	BinaryAggregationOfAbstracts	false
mc4	AbstractAttribute	false
...	...	...

MSM-PROPERTY

OID	Prop-Name	Constr	Type
mp1	Abstract-Name	mc1	string
mp2	Att-Name	mc2	string
mp3	IsId	mc2	bool
mp4	IsFunctional1	mc3	bool
mp5	IsFunctional2	mc3	bool
...	...	...	...

MSM-REFERENCE

OID	Ref-Name	Constr	ConstrTo
mr1	Abstract	mc2	mc1
mr2	Abstract1	mc3	mc1
mr3	Abstract2	mc3	mc1
...	...	...	...

Fig. 6. The supermodel part of the metalevel portion of the dictionary of MIDST

are describing the supermodel) stores the name and a unique identifier (OID) for each construct, so this table actually memorizes every allowed construct; indeed, the rows of this table correspond essentially to those in Figure 3. Each construct is also characterized by a set of properties describing the details of interest. There is a table, MSM_PROPERTY, reporting name, type and owner construct for each property. The properties, for example, allow to define whether an entity attribute is identifier or not and to specify the cardinality of relationships. Constructs refer to one another with references, recorded in the table MSM_REFERENCE.

As we have illustrated, the metalevel lays the basis for the definition of constructs which can be then used in defining models and so on the structure of the lower layer of the dictionary: in fact, the model-generic layer (Figure 5) has one table for each row in MSM_CONSTRUCT (and so we have, as we said, tables named SM_ABSTRACT, SM_AGGREGATION, SM_LEXICAL, and so on), with columns corresponding to the properties and references of the construct, as described in MSM_PROPERTY and MSM_REFERENCE, respectively.

The aim of the following sections is to define operators on the basis of constructs in such a way that model-independent solutions to model management problems can then be described. In fact, solutions will be formulated as scripts involving the application of such operators. We will see that the structure of the dictionary, especially with its metalevel, plays a major role in the automatic generation of Datalog programs for the implementation of the operators.

3 Operators

Model management, as we said in the introduction, refers to a wide range of problems, which share the need for high level solutions. Therefore many operators

have been proposed, depending on the family of problems of interest. Here we concentrate on schema evolution, where proposals [8,10] require MATCH, DIFF and MERGE and, if an explicit representation of models is needed, also MODELGEN. In such proposals, the MATCH operator is used to discover mappings between the elements of the involved schemas. In fact, mappings play a major role, as they provide the operators with essential information about the relationships between the involved schemas. For example, an operator that computes the difference between two schemas needs to know the correspondences between constructs in order to subtract them correctly. Likewise, an operator that combines schemas must know those correspondences in order to avoid the generation of duplicates. Here, exploiting our construct-based representation of data models, we can propose definitions of the main operators (DIFF, MERGE, and MODELGEN) that compare constructs on the basis of their names and structures. In fact, we assume that if two constructs have different names or different structures, they should be considered as different. In this way, as we clarify in the next subsection, our approach considers MATCH as complementary.

We already have an implementation for MODELGEN in our MIDST proposal (and hence in MISM as well), and so we have to concentrate on DIFF and MERGE. In the rest of this section we will present specifications for these operators that refer to MIDST dictionary, preceded by the discussion of a preliminary notion, equivalence of schema elements. Then, in Section 4 we will show how to generate Datalog implementations for them.

3.1 Equivalence of Schema Elements

The basic idea behind the DIFF and MERGE operators is the set-theoretical one. In fact, we can consider each schema as composed of a set of *schema elements* (the actual constructs it involves), and then consider DIFF as a set-theoretic difference (the elements that are in the first schema and not in the second) and MERGE as a union (the elements that are at least in one of the two schemas). In general, we might be interested in comparing schemas that represent the concepts of interest by means of different elements. In such a case, a preliminary step would require the identification or specification of the correspondences between them. This is usually done by means of an application of the MATCH operator, which, in general, can produce correspondences of various types (i.e. one-to-one, one-to-many, or even many-to-many) and may require a human intervention in order to disambiguate or to better specify. Besides, in MIDST context, let us recall that each schema element is represented with respect to a specific model-generic construct (i.e. an element refers to an Abstract, another one refers to an Aggregation and so on): in this sense we say that an element is an *instance of* a construct. Consequently, we distinguish between *construct-preserving* correspondences and *non construct-preserving* ones. The first type maps elements, instances of a certain construct, only to elements that are instances of the same model-generic construct; viceversa, correspondences not satisfying this property belong to the second type. For example in the XML schemas of Figure 1 the correspondence between the simple element *Address* and the complex one (again

called *Address*), composed of *Street*, *Zip*, and *City*, is not construct-preserving. In fact the address is represented by a simple element in the first schema (i.e. a Lexical in MIDST), while in the second one it requires a complex element (i.e. a StructOfAttributes in MIDST) with its components (i.e. some Lexicals in MIDST). Clearly, non construct-preserving correspondences denote different ways to organize the data of interest and therefore the involved constructs of the two schemas have to be considered as different. On the other hand, constructs that have different names but the same structure while handling the same data, have to be considered as equivalent. These are one-to-one correspondences, which can be discovered manually or by means of a matching system (among the many existing ones [27]).

The arguments above lead to a notion of renaming of a schema: given a correspondence c, the *renaming* of a schema S with respect to c is a schema where the names of the elements in S are modified according to c. Then, we have a basic idea of equivalence conveyed by the following recursive statement:

two schema elements are equivalent with respect to a renaming if: (i) they are instances of the same model-generic construct; (ii) their names are equal, after the renaming; (iii) their features (names and properties) are equal; and (iv) they refer to equivalent elements.

For the sake of simplicity, we can assume that the renaming is always applied to one of the schemas, in order to guarantee that corresponding constructs with the same type also have the same name. In some sense this would correspond to a *unique name assumption*. Then, equivalence would be simpler, as name equality would be required:

two schema elements are equivalent if their types, names and features are equal and they refer to equivalent elements.

It is important to observe that the definition is recursive, as equivalence of pairs of elements requires the equivalence of the elements they refer to. This is well defined, because the structure of references in our supermodel is acyclic, and therefore recursion is bounded. Let us consider few cases from our running example, namely schemas I_1 and I_2 in Figure 2. We have a column *Title* for a table *Project* in both schemas, and the two are equivalent, as they have the same name, the same properties (they are both non-key), and refer to equivalent elements (the tables named *Project*). Instead, the column *Title* of *Project* in I_1 is not equivalent to *Title* of *Manager* in I_2, because *Project* and *Manager* are not equivalent. Also, the two columns named *SSN* are not equivalent, because the one in I_1 is key and that in I_2 is not.

3.2 Definitions of the Operators

We are now ready to give our definitions and show some examples. According to what we said in the previous section, we assume that suitable renamings have

been applied in such a way that a unique name assumption holds. We start with a preliminary notion, to be revised shortly.

Given two schemas S and S', the difference $\textsc{diff}(S, S')$ *is a schema S'' that contains all the schema elements of S that do not appear in S'.*

This first intuitive idea must be refined, otherwise some inconsistencies could arise. In fact, it may be the case that a schema element appears in the result of a difference while an element it refers to does not. This causes incoherent schemas with "orphan" elements. With respect to the schemas in our running example, this happens for the column *MgrID* in the difference $\textsc{diff}(I_2, I_1)$, which belongs to the result, while the table *Project* does not. Instead we want to have *coherent* schemas, where references are not dangling.

In order to solve this difficulty, we modify our notion of a schema, by introducing *stub elements* (similar to the *support objects* of [8]). Specifically, we extend the notion of *schema element*, by allowing two kinds: *proper elements* (or simply *elements*), those we have seen so far, and *stub elements*, which are essentially fictitious elements, introduced to guarantee that required references exist. We say that a schema is *proper* if all its elements are proper.

According to this technique, the result of $\textsc{diff}(I_2, I_1)$ contains the stub version of *Project* in order to avoid the missing reference of *MgrID*.

The definition of the difference should therefore be modified in order to take into account stub elements both in the source schemas and in the result one.

Given S and S', $\textsc{diff}(S, S')$ *is a schema S'' that contains: (i) all the schema elements of S that do not appear in S'; (ii) stub versions for elements of S that appear also in S' (and so should not be in the difference) but are referred to by other elements in* $\textsc{diff}(S, S')$.

The notion is recursive, but well defined because of the acyclicity of our references.

In the literature [8], the DIFF operator is often used in model management scripts to detect which schema elements have been added to or removed from a schema. Our definition addresses this target. Given an "old" schema S and a "new" one S', the "added" elements (also called the *positive* difference) can be obtained as $\textsc{diff}(S', S)$ whereas the "removed" ones (the *negative* difference) are given by $\textsc{diff}(S, S')$.

With respect to the running example in Figure 2, the negative difference, $\textsc{diff}(I_1, I_2)$, contains the columns *MgrSSN* of *Project* and *SSN* (key) and *EID* (non-key) of *Manager*. Column *MgrSSN* belongs to the difference since it belongs to I_1 and there is no attribute with the same name in I_2. Instead, *EID* and *SSN* belong to $\textsc{diff}(I_1, I_2)$ because the attributes with the same respective names in I_2 have properties that differ from those in I_1: *EID* is key in I_1 and not key in I_2, whereas the converse holds for *SSN*. The negative difference does not contain the two tables as proper elements, because they appear in both schemas, but it needs them as stub elements because the various columns have to refer to

them. The negative difference also includes the foreign key in I_1 since it does not appear in I_2 (the foreign key in I_2 involves different columns).

Similarly, the positive difference includes the columns *MgrID* of *Project* and *SSN* (non-key), *EID* (key) and *Title* of *Manager*, both tables as stub elements, and the foreign key in I_2.

An important observation is that the definition we have given here is model-independent, because it refers to constructs as they are defined in our supermodel. At the same time, it is *model-aware*, because it is always possible to tell whether a schema belongs to a model, on the basis of the types of the involved schema elements. As a consequence, it is possible to introduce a notion of closure: we say that a model management operator O is *closed with respect to a model* M if, whenever O is applied to schemas in M, then the result is a schema in M as well. Given the various definitions, it follows that the difference is a closed operator, because it produces only constructs that appear in its input arguments.

Let us now turn our attention to the second operator of interest, MERGE. We start again with a preliminary definition.

Given S and S', their merge MERGE(S, S') *is a schema S'' that contains the schema elements that appear in at least one of S or S', modulo equivalence.*[2]

It is clear that merge is essentially a set-theoretic union between two schemas, with the avoidance of duplicates managed by means of the notion of equivalence of schema elements.

Since our schemas might involve stub elements, as we saw above, let us consider their impact on this operator. Clearly, the operator cannot introduce new stub elements, as it only copies elements. However, stubs can appear in the input schemas, and the delicate case is when equivalent elements appear in schemas, proper in one and stub in the other.[3]

Given S and S', their merge MERGE(S, S') *is a schema S'' that contains the schema elements that appear in at least one of S or S', modulo equivalence. An element in S'' is proper if it appears as proper in at least one of S and S' and stub otherwise.*

As an example, consider the following schemas, each composed of a single table. S: *Project(PCode, Title)* and S': *Project(PCode, MgrSSN)*. Their merge will be another schema S'' containing the table *Project(PCode, Title, MgrSSN)*. Notice that the table *Project* and the column *PCode* appear both in S and in S'

[2] Technically, both here and in the difference, we should note that schema elements have their identity. Therefore, in all cases we have new elements in the results; so, here, we copy in the result schema the elements of the two input schemas, and "modulo equivalence" means that we collapse the pairs of elements of the two schemas that are equivalent (only pairs, with one element from each schema, as there are no equivalent elements within a single schema).

[3] Equivalence of elements neglects the difference between stub and proper elements, as it is not relevant in this context.

and, since they are recognized as equivalent, there are no duplicates in S''. The column *Title* appears only in S while *MgrSSN* only in S'; therefore one copy of each is present in the result schema S''. We will see a complete example of MERGE in Section 5, while discussing the details of our running example.

For this operator, arguments for model independence and model closure can be made in the same way as we did for DIFF: specifically, only schema elements deriving from schemas S and S' will appear in the result and, consequently, if they belong to a given model, then S'' will belong to that model as well.

For the sake of homogeneity in notation, let us define also the operator that performs translations between models:

Given a schema S of a source model M and a target model M', the translation MODELGEN(S, M') *is a schema S' of M' that corresponds to S.*

We have discussed at length MODELGEN elsewhere [3,5]. Here we just mention that this notation refers to a generic version of it that works for all source and target models (the source model is not needed in the notation as it can be inferred from the source schema), thus avoiding different operators for different pairs of models. Indeed, our MIDST implementation [4,5] of MODELGEN includes a feature that can select the appropriate translation for any given pair of source and target models.

4 Model-Independent Operators in MISM

In this section we show how the definitions of the operators can be made concrete, in a model-independent way, in our tool, leveraging on the structure of its dictionary. The implementation has been carried out in Datalog, and here we concentrate on its main principles, namely the high-level declarative specification, and the possibility of automatic generation of the rules, on the basis of the metalevel description of models.

The Datalog specification of each operator is composed of two parts:[4]

1. equivalence test;
2. procedure application.

The first part tests the equivalence to provide the second part with necessary preliminary information on the elements of the input schemas.

We first illustrate how the equivalence test can be expressed in Datalog, and then proceed with the discussion for the specific aspects of DIFF and MERGE. At the end of the section, we discuss how all these Datalog programs can be automatically generated out of the dictionary.

4.1 Equivalence Test

The first phase involves the implementation of a test for equivalence of constructs, according to the definition we gave in Section 3. Given the definition,

[4] For the sake of readability we describe them in a procedural way, even if the specification is clearly declarative.

all we need is a rule for each model-generic construct that compares the schema elements that are instances of such a construct. It refers to two schemas, denoted by the "schema variables" SOURCE_1 and SOURCE_2, respectively. It generates an intensional predicate (a view, in database terms) that indicates the pairs of OIDs of equivalent constructs. As an example, let us see the Datalog rule that compares Aggregations.

```
EQUIV_Aggregation [DEST] (OID1: oid1, OID2: oid2)
  <- SM_Aggregation [SOURCE_1] (OID: oid1, Name: name),
     SM_Aggregation [SOURCE_2] (OID: oid2, Name: name);
```

Aggregation has no references (and also no properties) and so the comparison is based only on name equality (verified with the variable *name*). If the names of the two Aggregations are equal, then they are equivalent, and so their OIDs are included in the view for equivalent Aggregations. In the running example, tables *Project* and *Manager* of the two schemas are detected as equivalent since they have the same names, respectively.

The situation becomes slightly more complex when constructs involve references. This is the case for Lexicals of Aggregation (in the running example, the various columns of *Project* and *Manager*).

```
EQUIV_Lexical [DEST] (OID1: oid1, OID2: oid2)
  <- SM_Lexical [SOURCE_1] (OID: oid1, Name: name, isIdentifier: isId,
         isNullable: isNull, type: t, aggregationOID: oid3),
     SM_Lexical [SOURCE_2] (OID: oid2, Name: name, isIdentifier: isId,
         isNullable: isNull, type: t, aggregationOID: oid4),
     EQUIV_Aggregation (OID1: oid3, OID2: oid4);
```

The first and the second body predicates compare names and homologous properties of a pair of Lexicals, one belonging to I_1 (SOURCE_1) and the other to I_2 (SOURCE_2). Comparisons are made by means of repeated variables (such as *name*, *isId*, *isNull*, *t*). Moreover, as Lexicals involve references to Aggregations (as no column exists without a table, in the example), we need to compare the elements they refer to. The last predicate in the body performs this task by verifying that the Aggregations (tables) referred to by the Lexicals (columns) of I_1 and I_2 are equivalent (i.e. the corresponding pair of OIDs is in the equivalence view for Aggregation).

If the constructs under examination belonged to a deeper level, there would be a predicate to test the equivalence of ancestors for each step of the hierarchical chain. Each predicate would query the appropriate equivalence view to complete the test. Termination is guaranteed by the acyclicity of the supermodel.

Let us observe that the Datalog program generated in this way is model-aware since it takes into account the type of constructs when performing comparisons. In fact, as it is clear in the examples, Datalog rules are defined with specific respect to the type of the constructs to be compared: a Lexical is compared only with another Lexical and so for an Abstract or other constructs.

The program is model generic as well, since the set of rules contains a rule for each construct in the supermodel. Then a given pair of schemas will really make

use of a subset of the rules, the ones referring to the constructs they actually involve according to their model.

4.2 The DIFF Operator

The DIFF operator is implemented by a Datalog program with the following steps:

1. equivalence test (comparison between the input schemas);
2. selective copy.

The first step is the equivalence test we have described in Section 4.1.

As for the second step, there is a Datalog rule for each construct of the supermodel, hence taking into account each kind of schema element: the rule verifies whether the OID of an element of the first schema belongs to a tuple in the equivalence view. If this happens, this means that there is an equivalent construct in the second schema, implying that the difference must not contain it, otherwise the copy takes place. For example, the rule for Aggregations results as follows:

```
SM_Aggregation [DEST] (OID: #AggregationOID_0(oid), Name: name)
  <- SM_Aggregation [SOURCE_1] (OID: oid, Name: name),
     !EQUIV_Aggregation (OID1: oid);
```

In the rule, the # symbol denotes a Skolem functor, which is used to generate new identifiers (in the same way as we did in MIDST [5]). Indeed, the functor is interpreted as an injective function, in such a way that the rule produces a new construct for each different source construct on which it is applicable. The various functions also have disjoint ranges. The rule copies into the result schema all the Aggregations of SOURCE_1 that are not equivalent to any Aggregation of SOURCE_2. The condition of non-equivalence is tested by the negated predicate (negation is denoted by "!") over the equivalence view; in fact, if the OID of an Aggregation of the first source schema is present in the view, then it has a corresponding Aggregation in the second source schema, and so it must not belong to the difference.

With reference to the running example, let us compute DIFF(I_1, I_2). The rule above represents the computation of the difference with respect to tables. Since in Figure 2 both *Project* and *Manager* in I_1 have an equivalent table in I_2, then the difference does not contain any Aggregation.

Consider now the rule for Lexicals (columns):

```
SM_Lexical [DEST] (OID: #LexicalOID_0(oid), Name: name,
      isIdentifier: isId, isNullable: isNull, type: t,
      aggregationOID: #AggregationOID_0(oid1))
  <- SM_Lexical [SOURCE_1] (OID: oid, Name: name,
       isIdentifier: isId, isNullable: isNull, type: t,
       aggregationOID: oid1),
     !EQUIV_Lexical (OID1: oid);
```

It copies into the result schema all the Lexicals of SOURCE_1 that are not equivalent to any Lexical of SOURCE_2. In the example of Figure 2, the Lexical *MgrSSN* has been removed from *Project*. Also, *SSN* of *Manager* is key in I_1 but not in I_2 and the converse for *EID*. Consequently all of the mentioned Lexicals will belong to the difference DIFF (I_1, I_2).

For the sake of simplicity, we have omitted from the above rules the features that handle stub elements. However the actual implementation of the difference requires them in order to address the consistency issues we have discussed in the previous section. The strategy we adopt is the following: when a non-first level element (that is, one with references) is copied, the procedure copies its referred elements too if they are not copied for another reason. Then, unless they are proper parts of the result, the procedure marks the referred elements as stub. The following rule exemplifies this with respect to Aggregations.

```
SM_Aggregation [DEST] (OID: #AggregationOID_0(oid), Name: name,
       isStub: true)
  <- SM_Aggregation [SOURCE_1] (OID: oid, Name: name),
     EQUIV_Aggregation (OID1: oid, isStub: false);
```

If a Lexical (referring to an Aggregation) belongs to the difference, then the referred Aggregation must be copied into the difference as stub (if it has not been copied directly). The rule above copies from the first schema every Aggregation that would not belong to the difference since it has an equivalent (non stub) element in the second schema (which is verified by the predicate over the view, which also contains information on whether the equivalence involves stub elements) and marks it as stub. As for the input, we must subtract schemas with stub elements properly. Thus the selective copy in step 2 must be adapted: it should copy (into the result schema) a non-stub element in the first schema only if the second schema does not contain a non-stub equivalent element. This last condition is tested by a predicate over an equivalence view like the one in the above Datalog rule.

The techniques described refer to the rules for the specification of the difference of schemas. Indeed, as our dictionary includes also a data level (as illustrated in a previous paper of ours [2]), which lists all data items that instantiate a given construct, it is interesting to see how the operator could be specified in such a way that the result is a schema, as we saw above, together with the associated data. While working at MODELGEN, we tackled the same issue, and we developed a technique that generated data level Datalog programs out of schema level ones [3]. In such a context, correctness was a delicate issue, as each translation has its own specific features, and the tool administrator has the responsibility of verifying the correctness. Here we are interested in a general program, that implements difference, and therefore we cannot rely upon the approval of a human. However, things are indeed easier, as the difference needs to include all instances of the constructs that appear in the result schema: for example, if the result of DIFF includes table *Manager*, then we need all its instances in the result database, but this is just a copy, as *Manager* is a table in the source schema as well. So, data level rules for DIFF could be produced as rules that copy all instances of constructs, with the condition that the construct appears in the

result schema, which is easy to express, as it is indeed the condition in the body of the schema rule. Therefore, while we omit the details for the sake of space, we can safely claim that we can generate correct rules that operate on data from those that operate on schemas.

4.3 The MERGE Operator

The approach we follow for MERGE is based on the same ideas as the one for DIFF. We code it in terms of Datalog rules defined over the constructs of MIDST supermodel. Rules copy elements of one type to elements of the same type and we guarantee the needed model closure.

The MERGE operator, as defined in Section 3.2, is represented by a Datalog program with the following tasks:

1. equivalence test (comparison between the input schemas);
2. selective copy from the first argument;
3. selective copy from the second argument.

The first step involves the computation of an equivalence view containing the correspondences between the elements of the input schemas.

Assume we are computing $S'' = \text{MERGE}(S, S')$. In step 2 the procedure copies into the destination schema S'' all the elements in S, except those that are stub in S and non-stub in S'. In step 3 the procedure copies all the elements of S' that are not present in S and those that are non-stub in S' and stub in S.

The combination of these two steps implies that in S'' there will not be duplicates of any element. If an element is present both in S and S', in S as a stub and in S' as a non-stub, it will be present in S'' as a non-stub. A stub element will appear in the result as stub as well, if an element is present only in S or S' as a stub or both in S and S' as stub.

In such an implementation of the MERGE, a thorough handling of references is important and we achieve this by means of Skolem functions, which are injective as we said in the previous section. In fact, it may happen for an element of the result schema to have a stub parent in the first source schema and a non-stub parent coming from the second source schema: let E be an element of S which is copied into the result schema. E has a stub parent P in S and there is another element P' which is the equivalent non-stub element of P in S'. P will not be copied from S, but there will be its equivalent P' coming from S'. As a consequence, the reference of E to P must use an OID that is derived from the OID of P' in S' and not from the OID of P in S. As we have seen for the difference, this logic can be implemented in Datalog on the basis of a predicate over the equivalence views.

By following arguments similar to those for DIFF, we can claim that, from the schema level Datalog programs for MERGE, we can generate programs that implement the operator on data, thus performing the merge of the actual databases (in the internal representation in our dictionary). The reason is that the operator is again a sort of selective copy.

4.4 Automatic Generation of Datalog Programs for the Operators

The implementations of both the phases of the operators are based on comparisons and copies of schema elements considered in terms of constructs of the supermodel. We have seen in Section 2 that MIDST handles the descriptions of these constructs in a dictionary, defining their names, features and references to one another. An automatic generation of the Datalog programs we have presented is possible and indeed represents a key point of the approach we propose here. Concretely, we propose a new module of MISM, *OpGen*, that automatically generates the rules according to the supermodel constructs. OpGen reads the information in the dictionary about constructs, their references, and their properties, and uses it to produce appropriate Datalog rules in the right order, according to the structure of constructs. As we said in the respective sections, for each operator we can generate data level rules that perform the selective copy of the instances of the involved constructs.

Automatically generated operators are not only model-independent but also supermodel-independent. In fact, in case of extensions to and modifications of the supermodel, all we need is to use OpGen to generate an updated version of the operators.

It is worth noting that our model-generic operators are scalable, since their internal complexity does not depend on the size of the input schemas nor on the number of modifications. In fact, they are generated by OpGen once and work for every possible set of input schemas defined in terms of constructs of MIDST supermodel. Moreover, although more efficient implementations of them could be designed, their application is entirely devoted to the database system which addresses, as a consequence, all the optimization issues.

5 A Model-Independent Solution to the Round-Trip Engineering Problem

In the previous sections we described the most common model management operators. We have shown that since they are defined over the constructs of MIDST supermodel, they are model-independent; moreover we have shown that it is possible to exploit their model awareness in order to satisfy the model closure property. This implies that solutions to model management problems, given in terms of these operators, are model-independent.

Here we show how our approach can be used to provide a model-independent solution to the round-trip engineering problem, illustrated in the introduction as one of the most representative ones in the model management area.

5.1 The General Procedure

Consider Figure 7: S_1 is the *specification schema* and I_1 the *implementation* schema obtained from S_1 with the application of the transformation (a translation and, possibly, some customizations) map_1. Let I_2 be a modified version of I_1. The goal is to determine a specification S_2 from which I_2 could be derived.

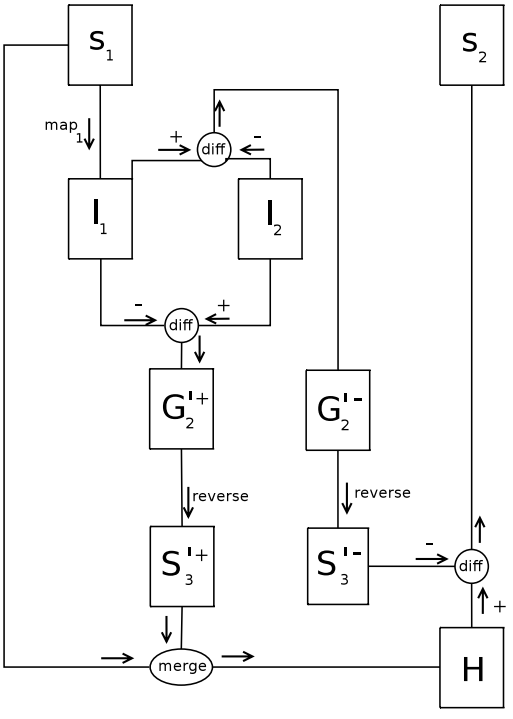

Fig. 7. A procedure for the round-trip engineering problem

Operationally, we assume that I_1 has been generated from the specification schema by the MODELGEN operator, possibly followed by a customization step; viceversa, we make no specific assumption on how I_2 has been obtained: it could be some transformation (specified by means of a Datalog program or in some other way), or a manual modification or evolution of I_1, or it could even come from an external input.

Then the procedure is as follows.

1. $G_2'^{-} = \text{DIFF}(I_1, I_2)$
   Here we use the DIFF operator to detect which elements of the implementation schema I_1 do not appear in the revised version I_2: these are the elements belonging to I_1 but not to I_2 (i.e. the removed elements).
2. $G_2'^{+} = \text{DIFF}(I_2, I_1)$
   This difference (with parameters swapped with respect to the previous one) allows to compute which elements have been added in the revision which led from I_1 to I_2. In fact, these elements are all the ones present in I_2 but not in I_1.
3. $S_3'^{-}$ is obtained by applying to $G_2'^{-}$ the reverse of the mapping map_1. The details then depend on the way map_1 is defined. In the common case where it is an automatic translation from the specification model to the implementation one (an application of MODELGEN), possibly followed by a customization, we

have that reverse can be done with MODELGEN as well, with a translation from the implementation model to the specification one. This ignores the possible customizations, under the assumption that changes in I_1 (yielding I_2) do not involve customized elements. In fact, if this is the case, $G_2'^{-}$ will not include the customized elements, since they are removed by the difference step. It should be noted that in general the existence of the inverse of a given translation is not guaranteed. We will discuss this issue later in this section.

4. Similarly for the other difference: $S_3'^{+}$ is obtained by applying to $G_2'^{+}$ the reverse of the mapping map_1.
5. $H = \text{MERGE}(S_1, S_3'^{+})$
   H is the union of the original specification S_1 with the reversed difference $S_3'^{+}$ containing the added elements. Therefore, H contains all the original elements plus the added ones.
6. $S_2 = \text{DIFF}(H, S_3'^{-})$
   The last operation of the procedure subtracts $S_3'^{-}$ from the temporary result H, because the elements in $S_3'^{-}$ are those that correspond to the elements removed in the implementation.

It is clear that this procedure does not require information about the models of the source schemas, since the operators act at MISM metalevel, dealing with constructs directly, however the model awareness of MISM guarantees the model closure. In fact, in the same way as we do for translations in our previous tool MIDST (see Section 2), we apply our operators in the supermodel framework, and the procedure is preceded and followed by copy steps, the first from the specific source model to the supermodel and the second from the supermodel to the specific model, which essentially rename constructs. An example should get the meaning across: suppose the specification data model is ER, while the implementation belongs to the relational model. Before applying the DIFF between I_1 and I_2, we rename all the elements in terms of constructs of the supermodel. After this step, there is no need to take into account the model-specific constructs anymore and the procedure can continue with respect to model-generic constructs only. Then, since the operators are defined in such a way that the difference between two schemas of a model belongs to that model, then we are guaranteed that the two differences in the procedure belong to the relational model as well. Finally, we apply MERGE and DIFF on ER schemas. These operators work independently of the model. However, we are sure that the results will also belong to the ER model because, as we have illustrated, the operators do not add any new element.

Moreover, it is important to observe that, if S_1, I_1 and I_2 are proper (and coherent,[5] as we always assume) schemas, then the result S_2 of the script is a proper schema as well. Consider the last two steps of the procedure ((i) $H = \text{MERGE}(S_1, S_3'^{+})$, (ii) $S_2 = \text{DIFF}(H, S_3'^{-})$): S_1 is assumed to be proper (the script starts from a specification without stubs). $S_3'^{+}$ contains added elements which

[5] As we said in Section 3.2, a schema is coherent if all its constructs have no dangling references to other constructs.

may refer to stub parents. However, as I_1 and I_2 are coherent, we have that non-stub equivalents for these stub parents are already present in S_1. Therefore H is proper. $S_3'^{-}$ contains the removed constructs. Then, as I_2 is coherent, in $S_3'^{-}$ we cannot come across the removal of parent elements when their descendants are preserved. Therefore S_2 is proper.

In the above procedure, we have referred to applications of MODELGEN from the specification model to the implementation one and viceversa, as if they were one the inverse of the other. This need not be always the case, because models have different expressive power. However, from the practical point of view, we have reasonable solutions, as follows. A preliminary observation is that our translations can be seen as schema mappings where the correspondences are represented by Skolem functions. In general, schema mappings are not always invertible according to the strict definition, but in the literature there are proposals for relaxed constraints guaranteeing the existence of a kind of inverse mapping. According to Fagin et al. [15] a Local As View (LAV) schema mapping, having a set of Tuple Generating Dependencies (TGDs) where their left-hand sides are singleton, always admits a *quasi-inverse* corresponding mapping. Let us consider a mapping m and a source schema S; applying m to S we obtain another schema T. A quasi-inverse mapping does not permit to reobtain S (with its original data) from T, however, it allows to obtain a schema S^* such that applying m to it we have T again (with all its data). In our approach the only translation rules dealing with the actual data are the ones involving Lexicals. All these rules are LAV TGDs and therefore the whole translation is a LAV schema mapping and so each translation admits at least a quasi-inverse one that is part of the MISM repository. In general, a translation can lead to loss of information (i.e. when we translate a model into a less expressive one); in such cases it is not possible to define an inverse translation, but only a quasi-inverse one. It is worth noting that this loss of information has already been accepted by the user of the system when performing the first translation (from the specification to the implementation). Moreover, this is the only loss of information of the whole process. In fact after the first translation, it is possible to apply the quasi-inverse translation and the direct one repeatedly always obtaining the same schemas (with the same data). The inverse (quasi-inverse) translation does not cause loss of information even if it turns a model into a more expressive one. In fact, the input schema of the inverse translation has been obtained from a schema of a less expressive model; therefore it contains only structures that can be represented in such a model.

5.2 Application of the Round-Trip Solving Procedure

Now we present the details of the application of the round-trip solving procedure described in Subsection 5.1 to the case already shown in Figure 2. The specification domain is the ER model, while the implementations are relational schemas. It is a common scenario in which high level specifications are conceptually designed with an ER schema. The implementation, which in this situation belongs

to the relational model, is then derived from the ER through the application of a translation rule.

The various steps are shown in Figure 8. Schema S_1 is composed of two entities, *Project* and *Manager*, and has a relationship R between them. *PCode* and *Title* are *Project* attributes (*PCode* is key), while *SSN*, *Name* and *EID* are *Manager* attributes (*SSN* is key).

Map_1 is implemented in two parts: a first part of the transformation is represented by ER-to-relational translation rule. A second part of it consists of the customization step which splits *Name* into *FName* and *LName*.

The transformation from the old to the new implementation modifies the table *Project* by changing the name of its column *MgrSSN* (to *MgrID*); it also modifies the *Manager* by adding the column *Title* and changing its key (from *SSN* to *EID*). The foreign key that in I_1 connects the column *SSN* with the table *Manager*, does not exist anymore, it is replaced by a new foreign key from the column *MgrID* of *Project* to the table *Manager*.

The first step of the solving procedure is the double application of the DIFF rules to I_1 and I_2 which yields $G_2'^-$ (negative difference) and $G_2'^+$ (positive difference), as we have already seen with examples for the operator in Section 3.2.

Then each semi-difference is reversed with the application of the MODELGEN operator, with the ER model as a target. In the case under examination, the reverse translation is simple, while in general it might be much more complex. Notice that in the application of the reverse rule, the stubness property of elements is preserved, then for example the entity *Project* in $S_3'^+$ is stub as well as in $G_2'^+$. Notice that the foreign key of $G_2'^-$ is reversed into the relationship R (that is the same as in S_1,[6] while the foreign key of $G_2'^+$ is reversed into the relationship R_1 (that is different from the one in S_1).

Now we have three different versions of the specification: the original one, S_1, together with $S_3'^-$, including all the elements that have to be removed, and $S_3'^+$, containing all the added elements.

The set-oriented merge of schemas S_1 and $S_3'^+$ leads to an updated specification, H, containing all the initial elements plus the added ones. Then in H we have *Project* with *PCode* (coming from S_1) and *Title* (from S_1) (the table *Project* is not stub anymore since it comes from S_1); moreover, there is the table *Manager* (non-stub for the same reason as *Project*) with the attributes *Name* (coming from S_1), *SSN* (from S_3^+), *SSN* (key) (from S_1), *EID* (from S_1), *EID* (key) (from S_3^+) and *Title* (from S_3^+). H also contains two relationships, R (coming from S_1) and R_1 (from $S_3'^+$).

Finally, we need to subtract from H all the non-stub elements in $S_3'^-$. Therefore, *SSN* (key) and *EID* are not present in the obtained result S_2. The relationship R of H is also present in $S_3'^-$, so the only relationship between *Project* and *Manager* in S_2 will be R_1.

[6] We can get back the "original" name because each construct has a name property; hence also the foreign key has a name property (not shown in figure) in our construct-based representation; in detail, we instantiated the name of the foreign key during the translation from S_1 to I_1 and we did the same during this step.

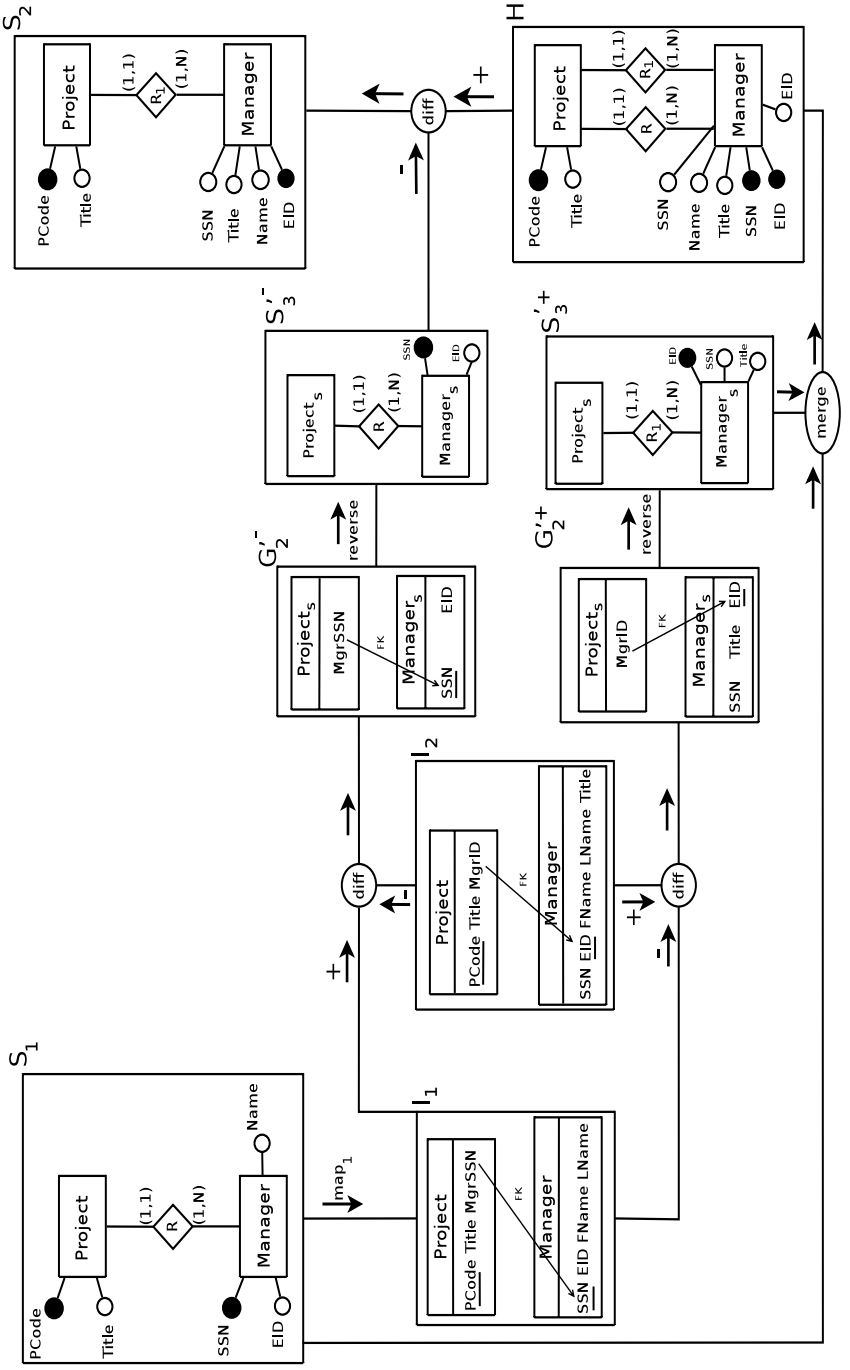

Fig. 8. An example of application of the round-trip solving procedure

6 Related Work

This paper illustrates a general approach to model management and relies on our previous work on model-generic schema and data translation [1,3,4,5] describing our conception and implementation of the MODELGEN operator. There are many proposals addressing model management problems which have been put forward since the original formulation of the problem.

In [7] Bernstein et al. recognize the possibility of a generic metadata approach to model management: their theoretical formalizations [8] and later studies converged into Rondo, a programming platform for model management [23]. However their approach is not supported by a description of models and so they pursue model independence without a concrete characterization of models and they cannot associate schemas with models. Conversely, MIDST (and now MISM) uses a dictionary of models and schemas to actually represent models and allows transparent transformations on them.

A parallel but orthogonal approach to model management problems, is that of Clio [16,17,19,24,28] whose aim is the development of a user aiding environment that allows the specification of a mapping between two instances and, consequently, generates the rules to implement the high level specified correspondences. Clio mainly offers a solution to data exchange problems by generating directly executable, though approximate, mappings between schemas. Similarly to Rondo, it lacks a model-independent representation of schemas and a representation of models.

A recent approach to schema evolution is PRISM [14]. Citing the authors, PRISM provides an intuitive, operational interface, used by the database administrator to evaluate the effect of possible evolution steps with respect to redundancy, information preservation, and impact on queries. In detail, the administrator can use a Schema Modification Operators (SMO) [9] language in order to specify schema changes and check whether such a modification could cause information loss, introduce redundancy, or grant invertibility. Moreover, the system allows for an automatic migration of the data, grants compatibility with old queries (i.e. against an old schema), and maintains the schema history. We propose something wider in which this approach can fit well: with reference to our running example, for instance, we could use similar techniques in order to constrain the evolutionary step between implementation schemas, thus granting the aforementioned desirable properties.

Our approach, together with Bernstein's, is more general and proposes a global platform for model management where the generation of executable mappings, like Clio's or PRISM's, is a complementary feature.

7 Conclusions

In this paper, we have discussed a paradigm and a concrete platform allowing model-independent solutions to a wide range of model management problems. We have provided effective definitions and implementations of model management operators which can be directly executed by the MISM platform. The

operators defined in this way have been used to assemble a solution to major model management problems.

A major target of the model management research is the development of an advanced software system managing all the involved problems (model management system). Such a system aims at providing applications with an abstraction layer towards data programmability issues, that is, the whole spectrum of application problems concerning data manipulation. The approach presented in this paper lies in this direction. MIDST represents a framework for model management problems; MISM is an enhanced version, where operators and solving procedures are specifically designed to maximize the abstraction level together with an effective and sound representation of schemas and models. In parallel, we are working on the development of runtime strategies and algorithms in order to make our solutions in step with large operational databases as well as compliant with the most expressive data models.

Acknowledgement

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

References

1. Atzeni, P., Cappellari, P., Bernstein, P.A.: Modelgen: Model independent schema translation. In: ICDE Conference, pp. 1111–1112. IEEE Computer Society, Los Alamitos (2005)
2. Atzeni, P., Cappellari, P., Bernstein, P.A.: A multilevel dictionary for model management. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 160–175. Springer, Heidelberg (2005)
3. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model-independent schema and data translation. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 368–385. Springer, Heidelberg (2006)
4. Atzeni, P., Cappellari, P., Gianforme, G.: MIDST: model independent schema and data translation. In: SIGMOD Conference, pp. 1134–1136. ACM, New York (2007)
5. Atzeni, P., Cappellari, P., Torlone, R., Bernstein, P.A., Gianforme, G.: Model-independent schema translation. VLDB J. 17(6), 1347–1370 (2008)
6. Atzeni, P., Torlone, R.: Management of multiple models in an extensible database design tool. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 79–95. Springer, Heidelberg (1996)
7. Bernstein, P., Haas, L., Jarke, M., Rahm, E., Wiederhold, G.: Panel: Is generic metadata management feasible? In: VLDB Conference, pp. 660–662 (2000)
8. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR Conference, pp. 209–220 (2003)
9. Bernstein, P.A., Green, T.J., Melnik, S., Nash, A.: Implementing mapping composition. VLDB J. 17(2), 333–353 (2008)
10. Bernstein, P.A., Halevy, A.Y., Pottinger, R.: A vision of management of complex models. SIGMOD Record 29(4), 55–63 (2000)

11. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: SIGMOD Conference, pp. 1–12. ACM, New York (2007)
12. Codd, E.: A relational model for large shared data banks. CACM 13(6), 377–387 (1970)
13. Codd, E.: Relational database: A practical foundation for productivity. CACM 25(2), 109–117 (1982)
14. Curino, C., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the PRISM workbench. PVLDB 1(1), 761–772 (2008)
15. Fagin, R., Kolaitis, P., Popa, L., Tan, W.: Quasi-inverses of schema mappings. ACM Trans. Database Syst. 33(2), 1–52 (2008)
16. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theor. Comput. Sci. 336(1), 89–124 (2005)
17. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. 30(1), 174–210 (2005)
18. Haas, L.M.: Beauty and the beast: The theory and practice of information integration. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 28–43. Springer, Heidelberg (2006)
19. Haas, L.M., Hernández, M.A., Ho, H., Popa, L., Roth, M.: Clio grows up: from research prototype to industrial tool. In: SIGMOD Conference, pp. 805–810. ACM, New York (2005)
20. Halevy, A.Y., Ashish, N., Bitton, D., Carey, M.J., Draper, D., Pollock, J., Rosenthal, A., Sikka, V.: Enterprise information integration: successes, challenges and controversies. In: SIGMOD Conference, pp. 778–787. ACM, New York (2005)
21. Hull, R., King, R.: Semantic database modelling: Survey, applications and research issues. ACM Computing Surveys 19(3), 201–260 (1987)
22. McGee, W.C.: Generalization: Key to successful electronic data processing. J. ACM 6(1), 1–23 (1959)
23. Melnik, S.: Generic Model Management: Concepts and Algorithms. Springer, Heidelberg (2004)
24. Miller, R.J., Haas, L.M., Hernández, M.A.: Schema mapping as query discovery. In: VLDB Conference, pp. 77–88 (2000)
25. Mork, P., Bernstein, P.A., Melnik, S.: Teaching a schema translator to produce O/R views. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 102–119. Springer, Heidelberg (2007)
26. Papotti, P., Torlone, R.: Heterogeneous data translation through XML conversion. J. Web Eng. 4(3), 189–204 (2005)
27. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. 10(4), 334–350 (2001)
28. Velegrakis, Y., Miller, R.J., Popa, L.: Mapping adaptation under evolving schemas. In: VLDB Conference, pp. 584–595 (2003)