Automatic model generation in model management[♦]

Artur Boronat, Isidro Ramos, José Á. Carsí

Department of Information Systems and Computation
Polytechnic University of Valencia
C/Camí de Vera s/n
46022 Valencia-Spain
{aboronat | iramos | pcarsi}@dsic.upv.es

Abstract. Model management aims at solving problems that stem from model representation and its manipulation by considering models as first-class citizens that are manipulated by means of generic operators. MOMENT is a prototype that supports generic model management using an algebraic approach within the four-layered metamodeling culture of MOF [1]. In this paper, we focus on the automatic generation of schemas that belong to different metamodels, using a term-rewriting system approach. We present the type system of the algebra that we use to represent models in the MOMENT prototype, and we describe our generic operator that automatically translates schemas between different metamodels: the operator *generate*. This algebra has been implemented using the functional language F#, which allows us to validate the correctness of our approach.

Keywords: information modeling, reverse engineering, model management, algebraic morphisms, schema generation.

1. Introduction

Nowadays, many companies are working with software products that were developed several years ago. Such applications have undergone changes to adapt to new requirements, but the lack of a specification makes this task more and more difficult. Software reverse engineering is the process that analyzes an application in an attempt to create a representation of it at a higher level of abstraction than the source code [2]. Therefore, reverse engineering is a process of design recovery. There are several CASE tools that support reverse engineering, such as Rational Rose [3], System Architect [4] or DB-Main [5]. These tools provide wizards that build a design specification of the data structure from a relational database or even wizards that detect and to organize functional services in class diagrams from the source code of the application.

Although these tools support automatic reverse engineering onto specific design methods, such as relational or object-oriented (OO for short), they do not take into account a change in the use of design methods. For instance, an application could

♦ This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

have been developed in the early 90s following the structured paradigm and the relational model for the database. Now, the application of a reverse engineering mechanism to obtain the relational schema of the database might not be the best solution to obtain an abstract description of the application. The development company will likely use a newer paradigm to design the software. In such a case, the research field of model management provides advantages to improve the reverse engineering process by manipulating data models.

A model is an abstract representation of the reality that enables communication among heterogeneous stakeholders so that they can understand each other. In our approach, a model is a structure that abstracts a design artifact such as a database schema, an OO conceptual schema, an interface definition, an XML DTD, or a semantic network [6]. Model management aims at solving problems with model representation and its manipulation by considering models as first-class citizens that are manipulated by means of abstract operators. The MOMENT (MOdel manageMENT) platform is a tool that allows model representation and manipulation by means of an algebraic approach. This approach allows the automation of model manipulation tasks, which can improve a reverse engineering process such as schema or model generation.

Focusing on this point, we introduce an algebraic approach to automatically generate schemas among different metamodels within the MOMENT prototype. The paper is structured as follows: Section 2 presents related work to contextualize our schema generation approach; Section 3 offers an application example; Section 4 presents an overview of the MOMENT prototype; Section 5 explains the algebraic operator *generate* and how it works; Section 6 presents some conclusions and future work.

2. Related work

Data reverse engineering can be treated generically from the perspective of model management. In this sense, the essentials of a model theory for generic schema management are presented in [7]. This model theory is applicable to a variety of data models such as the relational, object-oriented, and XML models, allowing model transformations by means of categorical morphisms. RONDO [8] is a tool based on this approach. It represents models by means of a graph theory and a set of high level operators that manipulate such models and mappings between them through a category theory. Models are translated into graphs by means of specific converters for each metamodel. These algebraic operators are based on imperative algorithms, such as CUPID [11]. CUPID is an algorithm for matching schemas in the RONDO tool.

In the MOMENT platform, we use the framework that is proposed in the Meta-Object Facility specification (MOF). MOF is one of the OMG family of standards for modeling distributed software architectures and systems. It defines an abstract language and a four-layered framework for specifying, constructing and managing neutral technology metamodels. A metamodel is an abstract language for some kinds of metadata. MOF defines a framework for implementing repositories that hold the metadata described by the metamodels. This framework has inspired our platform for model management although we do not use the same vocabulary to describe metamodels. In the case of MOF, the two most abstract layers offer a higher view of a specific model involving metamodel management.

3. Motivating example

Consider a car maintenance company that has worked a long time for a large car dealership. The maintenance company has always worked with an old C application where the information is stored in a simple relational database that does not even consider integrity constraints. The car dealership has recently acquired the car maintenance company and the president has decided to migrate the old application to a new OO technology in order to improve maintenance and efficiency. Therefore, the target application will be developed by means of an OO programming language.

Suppose that a part of the original database is a table *Invoice*, as shown in Fig. 1. To obtain a class that is semantically equivalent to this table, a designer usually builds it manually, which involves high development costs since the entire initial database is taken into account. What is worse is that this process is error-prone due to the human factor.

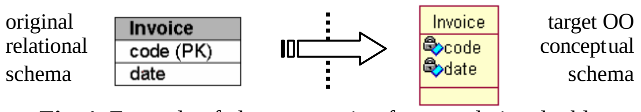

Fig. 1. Example of class generation from a relational table

4. MOMENT overview

The MOMENT (MOdel manageMENT) platform is a tool that allows model representation and manipulation by means of an algebraic approach. We use the expressiveness of the algebra, which the platform is based on, to define and represent a model as a formal term. Operators of the algebra, also called morphisms, perform transformations over the terms of the algebra.

The MOMENT platform uses several metadata layers to describe any kind of information including new metadata types. This architecture is based on both the classical four-layer metamodeling architecture, following standards such as ISO [9] and CDIF [10], and on the more modern four-layer framework proposed in the MOF specification [1]. In our work, we structure the platform in four abstract layers:

– The M0-layer collects the examples of all the models, i.e., it holds the information that is described by a data model of the M1-layer.
– The M1-layer contains the metadata that describes data in the M0-layer and aggregates it by means of models. This layer provides services to collect examples of a reality in the lowest layer.
– The M2-layer contains the descriptions (meta-metadata) that define the structure and semantics of the metadata located at the M1-layer. This layer groups meta-

metadata as metamodels. A metamodel is an “abstract language” that describes different kinds of data. The M2-layer provides services to manage models in the next lower layer.

– The M3-layer is the platform core, containing services to specify any metamodel with the same common representation mechanism. It is the most abstract layer in the platform. It contains the description of the structure and the semantics of the meta-metadata, which is located at the M2-layer. This layer provides the “abstract language” to define different kinds of metadata.

We have developed a prototype of MOMENT that runs on the .NET platform. The core of the prototype is an algebra, which provides a set of sorts and constructors to define models and a set of operators to manipulate them. To implement this algebra, we have used the F# programming [13] language for two main reasons: to bring a formal model management approach closer to an industrial programming environment, such as .NET; and to benefit from the functional programming advantages, such as independence from the control logics and a strong inference type. F# is a version of the Objective Caml programming language [12] on the .NET platform.

4.1. The MOMENT algebra

The MOMENT algebra aims to represent models of any kind as algebraic terms in order to automate model transformation tasks in a precise, formal way. Reaching this objective implies choosing a basic specification language that permits us to describe any piece of data. We have chosen the Resource Description Language (RDF) [14] for this purpose. RDF is an emergent standard proposed by the World Wide Web Consortium (W3C) that is becoming the “de facto” standard for the Semantic Web [15]. This language provides the foundation for metadata interoperability across different web resource description communities.

This algebra offers the core services of the MOMENT platform and is used to define its four meta-layers. The algebra consists of two main elements: sorts and their operations. Such operations consist of both constructors, which allow for the definition of the structure of the platform and the representation of the models, and operators, which perform management tasks over the models defined in the platform. Four main sorts permit the definition of a model as a term:

1. *Concept*

   A concept represents an RDF resource and is identified by a URI. It defines an entity that can be described by means of properties. The constructor of this sort is expressed in F# notation as follows:

```
Concept = NilConcept | Concept of (Concept * string)
```

where *NilConcept* represents a null concept term; the first argument of the constructor *Concept* is a term of the sort *Concept* that represents its metaconcept in the next upper abstraction layer, and the second argument is its identifier.

2. *Property*

   To define the relationships that relate a subject element to an object element, we focus on the RDF statement structure. Such relationships are specified by means of the Property sort.

We express the constructor of this sort in F# notation, as follows:

```
Property = NilProperty
   | Property of (Property * string * Cardinality * Cardinality * Node * Concept)
```

- Parent property indicating its type.
- URI that identifies the predicate of the property.
- Minimum cardinality of the property that indicates the minimum amount of instances of the range concept, which must be related to the subject node.
- Maximum cardinality of the property that indicates the maximum amount of instances of the range concept that can be related to the subject node.
- Subject element that receives the property. This can be a concept or another property, because a property may involve other properties.
- Object element that constitutes the value of the property. A property cannot be the object of another property for two reasons: it would make the RDF specification more difficult to understand, and it does not provide additional information.

3. *Schema*

In our context, a schema term represents a collection of concepts and properties that describe such concepts.

4. *Level*

A level term represents an abstraction layer in the platform. Four terms of this sort constitute the four-layer structure of the platform. The term M3-layer represents the most abstract layer in the platform and contains the MOMENT schema. This schema contains the term Concept and the term Property; the latter relates two concept terms, constituting the minimalist structure that we use to represent a model at a lower layer.

To apply our schema generation approach, we show how we can represent the relational and OO metamodels at the M2-layer, and how we can represent their schemas (also called models) at the M1-layer. We only describe the essentials of the metamodels that will be useful to present the operator *generate* in Section 5. The *Relational Metamodel* is a schema term that contains the concepts and properties that constitute the terminology to define a relational schema. *Table* and *Column* are represented by means of concept terms, which are related to each other through a property *table_column* in the relational metamodel at the M2-layer. This metamodel allows the definition of the concept *Invoice* as a table by means of the operator *new_concept*. In an identical way, the concepts *Code* and *Date* are defined as column terms, which are related to a table by means of instances of the property *table_column* defined at the M2-layer.

In a similar manner, the *OO Metamodel* is defined at the M2-layer, allowing the definition of the OO schema term that represents the OO model of the motivating example at the M1-layer.

5. Algebraic schema generation: the operator *generate*

The operator *generate* is a morphism that permits the translation of a model of a specific metamodel into a model of a different metamodel. In our case study, we generate an OO conceptual schema from a relational schema, both of which are specified by means of terms of our algebra sorts.

This morphism is defined at the M2-layer using mappings between elements of two different metamodels. These mappings are instances of the *MOMENTEquivalence* property defined at the M3-layer and indicate equivalence relationships between concepts of the two metamodels. These mappings must be defined by the user.

In the following sections, we explain the definition of equivalence mappings between elements of the two metamodels used in the motivating example. Then, we present the set of functions that define the generic morphism *generate*. Finally, we indicate the specific axioms that we have implemented in F# to support the schema generation between the relational and the OO metamodels.

5.1. Metamodel equivalence mappings

The metamodel equivalence mappings are instances of the *MOMENTEquivalence* property of the M3-layer. They permit the establishment of correspondences between concepts of two different metamodels indicating that they represent either a similar semantic meaning in different metamodels or model definition vocabularies.

There are two kinds of equivalence mappings:

a) Simple mappings, which define a simple correspondence between two concepts that belong to different metamodels; for instance, between a table and a class, or between a column of a table and an attribute of a class.
b) Complex mappings, which define correspondences between elements of a source metamodel and a target metamodel. These mappings relate two structures of concepts that represent a similar semantic meaning. For instance, to define an equivalence relationship between a foreign key of the relational metamodel and an aggregation of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the aggregation concept. This is because all three of these concepts of the relational metamodel provide the necessary knowledge to define an aggregation between two classes in the UML metamodel, such as the cardinalities of the aggregation.

In this paper, we focus on simple mappings, presenting a generic morphism to use at the M1-layer in the next section.

5.2. The morphism *generate*

To translate a schema of a source metamodel into another schema of a target metamodel, we use the morphism *generate*. This morphism is applied to a schema term of the source metamodel and defines a new schema of the target metamodel. Then, it checks all the concepts of the source schema one by one, generating the corresponding concept in the target schema in each case.

To process a concept, the operator *generate* makes use of axioms or rewriting rules. Each one of them is formed by two kinds of functions: a condition and a generation function. On the one hand, the condition function checks the properties of a concept in order to select which generation function should be applied. These conditions take into account the precedence order that exists between the concepts of a specific metamodel when this order is used to define a model. For instance, when we define a relational schema, we cannot define a column if the table that it belongs to is not defined previously. On the other hand, the generation function implies the definition of concepts and properties in the target schema by following four steps, as shown in Fig. 2:

1. The concept is reified in its metaconcept; that is, if the concept to be processed is the table *Invoice*, we obtain the metaconcept *Table* of the relational metamodel.
2. Once we know the corresponding metaconcept of the source metamodel, we query the equivalence that relates it to a concept of the target metamodel. In the case of a table of the *Relational Metamodel*, we obtain the concept *Class* of the *OO Metamodel*.
3. The operator *generate* instantiates the concept of the target metamodel, which becomes a metaconcept for its instance, i.e., the concept *Class* of the *OO metamodel* becomes the metaconcept for its instance *OO-Invoice*. The new concept, which has been generated in the new target schema at the M1-layer, is equivalent to the original concept in step 1, through the equivalence relationship that we have defined before.
4. Finally, the operator instantiates the equivalence defined at the M2-layer between the *Metaconcept* of the source *Concept* and the *Metaconcept'* of the new generated *Concept'*. The instantiation defines a new property in the source schema at the M1-layer that has the source *Concept* as domain and the target *Concept'* as range.

The morphism *generate* is one of the MOMENT algebra operators and has been implemented in F# as part of our prototype. To automatically generate models among different metamodels, we only have to add specific rewriting rules to the operator. These axioms are applied by means of the pattern matching of the F# language to translate the source model into the target one. Focusing on simple equivalence mappings, we take into account three possible cases of rewriting rules in order to generate OO models from relational schemas: the generation of a class, the generation of an attribute when the class that contains it has been generated before, and the generation of an attribute when the class that contains it has not yet been generated.

5.2.1. Table-Class equivalence

To take into account the translation of a table, the following condition and generation functions appear in the code of the operator:

```
| Concept(_,"Table", _) ->
        generate_r_table_2_oo_class r_schema r_concept oo_schema
```

where *generate_r_table_2_oo_class* performs the translation of the relational table into a class in the target OO schema.

When the operator processes a concept of the relational schema, it reifies the concept obtaining the metaconcept that describes the type of this concept. If the current concept is the term *Invoice* (shown in Fig. 2), its metaconcept is the term *Table* in the relational metamodel (1). The operator searches for the equivalence mapping that relates it to a concept of the OO Metamodel, obtaining the concept *Class* (2). This concept is instantiated in the new OO schema at the M1-layer (3). The identifier of the new concept is obtained from the URI that identifies the original relational table. Currently, we add the prefix *OO-* to the URI: *OO-Invoice*. Finally, the equivalence mapping *equivalence_table_class* defined at the M2-layer is instantiated in the property *equivalence_table_class_1* at the M1-layer, relating the original concept *Invoice* of the relational schema to the new generated concept *OO-Invoice* of the new OO schema (4).

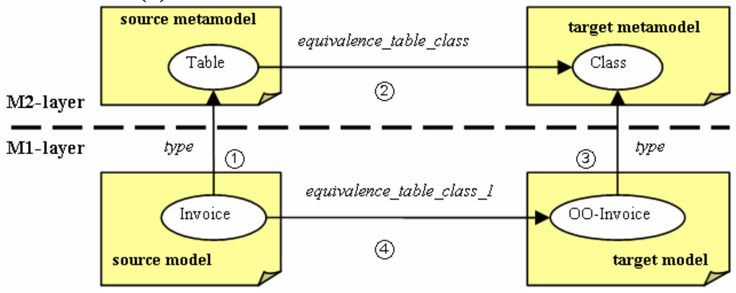

Fig. 2. Description of the rewriting process applied to a table

5.2.2. Column -Attribute equivalence

To translate a column of a table of a relational schema into an attribute of a class in a target OO schema, we have to take into account two situations: when the table that contains the column has already been translated and when it has not.

To deal with the first case, the following conditions and generation functions are specified in the code of the operator *generate*:

```
| Concept(_,"Column", _)
    when (validate_column_with_table r_schema r_concept oo_schema) ->
        generate_r_column_2_oo_attribute r_schema r_concept oo_schema
```

where the condition *validate_column_with_table* checks whether the table is translated and the generation function *generate_r_column_2_oo_attribute* translates the original column into an attribute in the target OO schema.

Fig. 3 illustrates the generation process followed in this case. Assume that we are processing the column *Code* of the table Invoice in Fig. 3. To determine whether the table *Invoice* has already been translated, the axiom finds an instance of the *table_column* property that relates the column *Invoice* to a table (1). Once we have the concept *Invoice* representing the relational table, the condition checks the existence of an instance of the property *equivalence_table_class*, which relates the table Invoice to a class in the target schema at the M1-layer (2). In this case, the generation function is applied, reifying the concept *Code* to the concept *Column* (3). Querying the equivalence mapping *equivalence_column_attribute* (4), the operator obtains the concept of the *OO Metamodel* that is equivalent to the concept *Column*, i.e., the concept *Attribute*. Then, the operator instantiates it providing an identifier, which is obtained from the original concept *Code* (5). By means of the property *equivalence_table_class_0* defined at the M1-layer, the operator obtains the class that is going to contain the recently generated attribute. By doing so, the property *class_attribute* of the OO metamodel is instantiated by relating the class *Invoice* and the attribute *Code* (6). Finally, the property *equivalence_column_attribute* is instantiated relating the original concept *Code* and the generated concept *OO-Code* (7).

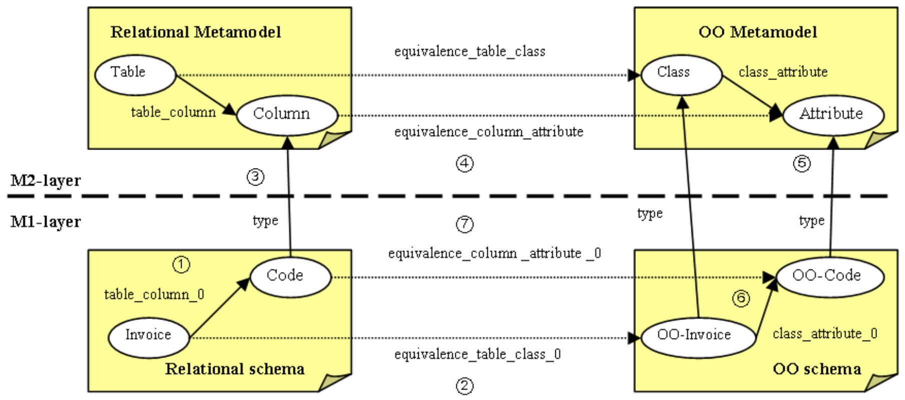

Fig. 3. Description of the rewriting process applied to column

In the second case, the axiom just checks the opposite condition of the first case, i.e., there is no class that is related to the concept *Invoice* of the relational schema by means of an equivalence mapping. Here, the generation function is a composition of the generation rule for a table and the generation rule for a column whose table has already been generated.

Equivalence mappings generated at the M1-layer are used by another operator of the MOMENT algebra, the operator *migrate*, which automatically produces a migration plan that indicates how the information of the source model at the M0-layer can be migrated to the information container of the target model. Therefore, these mappings allow us to perform data migration among models at the M0-layer.

6. Conclusions and future work

Reverse engineering constitutes a process that is currently present in software development companies. However, model management [6] is an emerging research field that aims at resolving data model integration and interoperability by means of generic operators. We have developed a prototype that permits the representation of models by means of the essential concepts of RDF, following the MOF metamodeling culture. For the moment, we have focused on the specification of relational schemas and object-oriented models. Model management tasks can help to improve a reverse engineering process as well as help to directly deal with the data models that are held by the applications. In this paper, we have focused on one of these tasks: schema (or model) generation.

We have presented two fundamental mainstays, which we have built our MOMENT platform on, taking into account our previous experience in the industrial project RELS [16]. A tool for the recovery of legacy systems has been built, using a term rewriting system to translate relational schemas into OO conceptual schemas and performing data migration from the legacy database to the database of the new application.

In this paper, we have considered an example of schema generation and we have presented an overview of our platform for model management taking into account the algebra used to represent and manipulate models. We have gone beyond on schema generation explaining the algebraic operator *generate* that we use to automatically generate the basic parts of a model by means of simple mappings between elements of different metamodels. Both the operator and the entire algebra have been implemented in F# [15] to be able to deal with models without the complexities that have to be taken into account in an algorithmic approach, such as in RONDO [8]. Thus, our approach is more generic and improves the scalability of the algebraic operators by simply adding axioms to the presentation of the algebra, as we have shown above.

Future work will take into account complex mappings that provide full support for generating schemas. We will also consider metamodels that are different from the relational and the OO metamodels in order to validate the generic applicability of our approach.

References

1. OMG: *Meta-Object Facility Specification version 1.4*. April 2002. http://www.omg.org/technology/documents/formal/mof.htm
2. Pressman, Roger s.: Software Engineering: A Practitioner's Approach. European Edition. McGraw-Hill, 2000.
3. Rational Software, http://www.rational.com/products/rose/
4. System Architect, http://www.popkin.com/products/sa2001/systemarchitect.htm
5. DB-Main, http://www.fundp.ac.be/recherche/unites/publications/en/2987.html
6. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).
7. Alagic, S. and Bernstein, P.A.: A Model Theory for Generic Schema Management. In Proceedings of DBPL'01, G. Ghelli and G. Grahne (eds), Springer-Verlag, (2001).
8. S. Melnik, E. Rahm, P. A. Bernstein: Rondo: A Programming Platform for Generic Model Management (Extended Version). Technical Report, Leipzig University, 2003. Available at http://dol.uni-leipzig.de/pub/2003-3
9. ISO/IEC 10746-1, 2, 3, 4 | ITU-T Recommendation X.901, X.902, X.903, X.904 « Open Distributed Processing - Reference Model". OMG, 1995-96.
10. CDIF Technical Committee: CDIF Framework for Modeling and Extensibility. Electronic Industries Assocaiation, EIA/IS-107, January 1994. See http://www.cdif.org/.
11. Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. MSR Tech. Report MSR-TR-2001-58, 2001, http://www.research.microsoft.com/pubs (short version in VLDB 2001).
12. E. Cahilloux, P. Manoury, B. Pagano: Developing Applications With Objective Caml, Éditions O'Reilly, 2000.
13. Microsoft Research F# Project. http://research.microsoft.com/projects/ilx/fsharp.aspx
14. World Wide Web Consortium, Resource Description Framework (RDF), http://www.w3.org/RDF/
15. World Wide Web Consortium, Semantic Web, http://www.w3.org/2001/sw/
16. Perez J., Anaya V., Cubel J.M., Domiguez F., Boronat A., Ramos I., Carsí J.A.: Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas. SET 2002, Software Evolution Through Transformations: Towards uniform support throughout the software life-cycle, Barcelona - Spain, October 2002.