Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine

Artur Boronat, José Á. Carsí, Isidro Ramos
Department of Information Systems and Computation
Polytechnic University of Valencia
Camí de Vera s/n
46022 Valencia-Spain
{aboronat, pcarsi, iramos}@dsic.upv.es

Abstract

Legacy systems are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. As time goes on, the maintenance of this software becomes more complex, expensive and painful. The OMG's MDA initiative provides some guides to develop software that raise models and transformations between them as first class citizens. This point of view provides a precise way to develop a new application from the legacy one. The MOMENT Framework supports automatic formal model transformations in MDA. This model transformation approach is based on the algebraic specification of models and benefits from mature term rewriting system technology to perform model transformation using rewriting logic. In this paper, we present how we apply this formal transformation mechanism to recover a legacy relational database, obtaining a UML-based application. This approach keeps the legacy knowledge by pumping the legacy data to the new database generated from the UML model. Our approach enhances the integration between formal environments and industrial technologies such as .NET technology, and exploits the best features of both.

1. Introduction

Legacy systems can be defined informally as "software we do not know what to do with, but it is still performing a useful job" [1]. They are information systems that have been developed by means of methods, tools and database management systems that have become obsolete, but they are still being used due to their reliability. They are characterized by the following features:

- Software architecture based on obsolete technology that has probably been patched in order to adapt to new changes in requirements. This fact complicates the maintenance of the application.
- Poor, complex documentation that prevents effective maintenance or software updating, making it necessary to check the source code to understand the functionality of the system.
- Cumulative experience working with the system that has filled its database with information that is significant for the company.

As in all complex systems with a medium life cycle, the requirements for this kind of applications go on changing at the same rate as technology does. There are two main approaches to performing changes in these systems. On the one hand, there is the patching of the legacy system that has obsolete technology code. The disadvantages to this approach are that the technology does not consider new features to improve either code reuse, quality or documentation generation, and that the staff that will develop the new part of the system needs to be trained. On the other hand, the whole system can be developed with a new technology taking advantage of all its features. Both approaches imply a high cost, but we prefer the second option because the first one only temporarily delays the translation into a new technology, making maintenance harder and harder each time the system is changed.

The OMG's Model Driven Architecture (MDA) initiative [2] is set in this context and provides several proposals to define models, through the standard Meta-Object Facility (MOF) [3]. It also offers proposals to perform model transformations by means of the Query/View/Transformations (QVT) language, which is still in its early stages [4]. While a lot of attention has been given to the transformation of platform-

independent models into platform-specific models, the scope of MDA goes beyond this in an attempt to model all the features of a software product throughout its life cycle. Nonetheless, a precise technique to provide formal support for the entire process of model transformation has not yet been developed.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. MOMENT is based on an algebra that is made up of *sorts* and operators that permit the representation of any model as a term that can be automatically manipulated by means of operators. The MOMENT Framework benefits from the best features of current visual CASE tools and the main advantages of formal environments such as term rewriting systems, combining the best of both industry and research.

This paper presents the application of the MOMENT framework to a reengineering problem where a UML model is obtained from a legacy relational database by means of the MOMENT model transformation, which relies on a term rewriting system.

The paper is structured as follows: Section 2 indicates an example to illustrate the transformation process through the paper; Section 3 provides an overview of the MOMENT Framework; Section 4 presents the model transformation mechanism of the MOMENT Framework, focusing on the use of a Term Rewriting System (TRS) as a formal environment to perform automatic model transformations; Section 5 presents other approaches to provide automatic support for model transformations. Finally, Section 6 summarizes the work and indicates the future directions of our research tasks.

2. Example of a Common Legacy System Recovering Process

Consider a car maintenance company that has worked for a long time for a large car dealership. The maintenance company works 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 its 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 although the database layer remains a relational database. This time, new integrity constraints are provided in the new relational database in order to improve maintainability.

Consider the part of the legacy system that stores information about invoices in the example. Each invoice contains data about the task performed in a specific period of time and at a specific price.

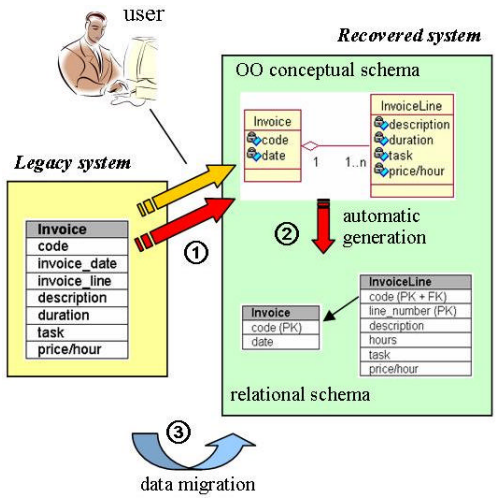

Figure 1. Legacy Database Recovering Example

To recover the legacy system, a designer has to build a semantically equivalent OO conceptual schema that captures the semantics that is disorganized in the legacy system. This task is usually done manually and involves high development costs. What is worse is that the human factor does not guarantee an error-free process to obtain a correct OO conceptual schema. Step (1) constitutes a manual, reverse-engineering process where the designer detects that the legacy table can be broken down into two classes: one containing the information about a performed task during a period of time and another one representing the collection of performed tasks for a specific customer, i.e., the *InvoiceLine* class and the *Invoice* class. In this step, works like those by [5], [6] and [7] can be applied to obtain OO models from relational schemas.

Once the OO model is complete, the relational database has to be generated (2). Here the designer can use many CASE tools such as Rational Rose, Together or System Architect in order to generate the new relational schema automatically. Despite obtaining a relational schema, these tools do not take into account legacy data.

The cumulative experience of the maintenance company is collected in its database and, it is expected to be preserved in the new database (3). Several DBMS allow for data migration using their ETL (Extract, Transform & Load) tools. This migration can be done

by means of SQL statements or user defined scripts which can be executed on the database. Although ETL tools provide friendly interfaces to migrate data between databases, DB administrators must write migration code manually, and this is very costly in terms of people and time.

In next sections, we explain the MOMENT Framework and the way in which its application provides a more efficient solution to this reengineering problem.

3. The MOMENT Framework

The MOMENT (MOment manageMENT) Framework is a modular architecture divided into the three traditional layers: interface, functionality and persistence. In each one of them, the environment benefits from mature tools, such as graphical CASE tools at the interface layer, term rewriting systems at the functionality layer, and RDF repositories at the persistence layer. Hence, the MOMENT Framework aims at using the best features of each environment, bringing industrial modeling tools closer to more formal systems. Figure. 2 shows an overview of the MOMENT Framework.

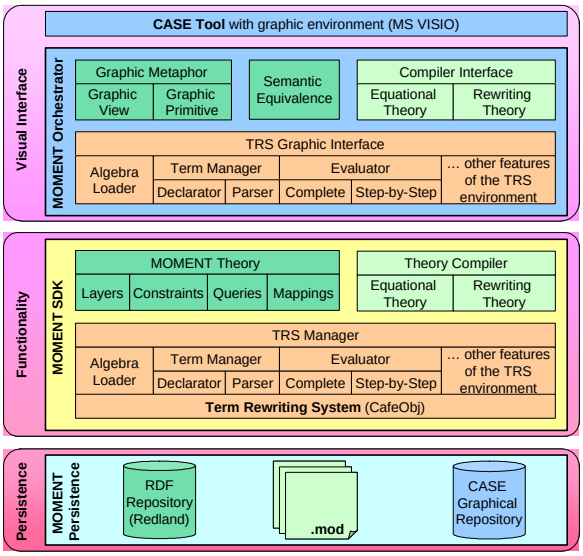

Figure 2. The MOMENT Framework

The functionality layer permits the representation of models and the performance of transformations over them. The core of the functionality layer is a module called *MOMENT Theory*, which allows model representation and manipulation by means of an algebraic approach. We use the expressiveness of the algebra that the platform is based on to define and represent a model as an algebraic term. This algebra represents models by means of terms of a *sort* called *Schema*. These terms are made up of by concepts and properties. The concepts are the main entities of a model, and the properties either describe them with values or establish relationships between them. The properties contain information about cardinality, indicating how many concepts can be related to the owner of the property.

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 [8] and CDIF [9]) and on the more modern four-layer framework proposed in the MOF specification [3]. In our work, we divide the platform into 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.

The *MOMENT Theory* module also provides a mechanism to define transformations between metamodels. The *TRS Manager* module wraps a TRS, which carries out the model transformation by applying a set of rewriting rules automatically. We have used the CafeOBJ environment as TRS [10]. The *Theory Compiler* module permits the compilation of the algebraic specification of a metamodel into a theory based on equational logic. It also compiles the defined mappings between the elements of the metamodels into a theory based on rewriting logic in order to perform the model transformation on the wrapped TRS. Some of these modules have been developed using the functional language F# [11], which provides convenient features to work with algebraic

specifications and with imperative programming environments such as .NET technology. A combination of functional languages and algebraic specification languages has permitted us to reach our goals. On the one hand, the MOMENT algebra is implemented in F#, which provides efficient structures for navigation and specification manipulation. On the other hand, a TRS provides a suitable environment to support automatic model transformation.

4. Legacy Systems Recovery Process in MOMENT

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. Models are defined using modeling languages, but when those languages are intended to be used for anything more sophisticated than drawing pictures, both their syntax and their semantics must be specified. In this case, the use of formal languages usually involves dealing with their complex syntax, making them unpopular in industry. In this sense, the MOMENT Framework is user-friendly and permits the use of formal techniques from well-known CASE tools to both define models by means of algebraic specifications and to perform model transformations using rewriting logic [12].

Rebuilding a legacy system into a semantically equivalent one with a new technology can be treated as a model transformation problem in the MDA context. Our tool provides a generic model transformation mechanism that has been applied to recover legacy databases. The reengineering process is composed of two steps:

a) A data reverse engineering process that extracts an abstract description from the legacy system database in order to know its structure and its behavior. Changes can be applied to it in order to adapt the systems to new requirements or to new technologies, such as the change between the structured paradigm and the OO paradigm. Our tool recovers a legacy database obtaining the static component of an equivalent OO conceptual schema using formal methods.
b) A forward engineering process that generates the software application (its structure in our case) based on a specific technology from the abstract description extracted from the legacy system. We use the Rose Data Modeler add-in [13] of the Rational Rose tool case to generate a new relational schema from the OO conceptual schema.

Our tool also allows for data migration from the legacy database to the recently generated one, keeping the knowledge stored in the old database. The data reverse engineering process and the data migration process followed by our approach will reduce the time invested and the number of people involved in the data evolution process. This optimization is reached by the automatic tasks that are performed by the tool in three phases. Despite the fact that these tasks are performed automatically, the results can be freely modified by the analyst. In this case, the process is semi-automatic. The tool performs the following three phases in order to reach this goal:

a) A UML conceptual schema is obtained by applying a data reverse engineering process in order to recover a relational legacy database. Both relational and UML conceptual schemas are represented as terms of an algebra and the correspondences between terms are specified using term rewriting rules.
b) The rewriting rules applied in the first phase and the patterns used by the Rose Data Modeler add-in to generate the new relational schema are used to describe a data migration plan which is specified using a declarative language.
c) The data migration plan is compiled into DTS[1] packages whose execution automatically migrates data from the legacy database to the new one.

In the following sections, we present the three phases in more detail, illustrating their application by means of the example of the motivating scenario.

4.1. Reengineering Process

The input of this phase is a relational schema of a legacy database and is the input to the main tool. This phase generates two XML documents as output: one representing the generated UML conceptual schema in XMI format ([19]) and another one that contains the rewriting rules applied to obtain the final UML conceptual schema. In this phase the process that produces these outputs follows three steps:

a) The reading of the relational schema of the legacy database. The access to the relational schema is performed by means of an API, named RSAO in Figure. 3, in order to access heterogeneous databases. The information that describes the legacy relational schema is used to instantiate elements of the relational schema at the M1-layer as a model. This phase also considers features of the old DBMS or other repository forms which do not allow for the

[1] Data Transformation Services (DTS) is a technology of SQL Server that allows the transfer of data among heterogeneous databases.

definition of constraints (either integrity or reference constraints). These constraints, implemented by code in the best case scenario, are not explicitly defined in the legacy database structure. Thus, user interaction may be necessary to provide additional information to obtain a complete relational conceptual schema. This extra information is added to the relational model obtained from the relational database by means of a graphical interface that hides the algebraic formalisms to the user.

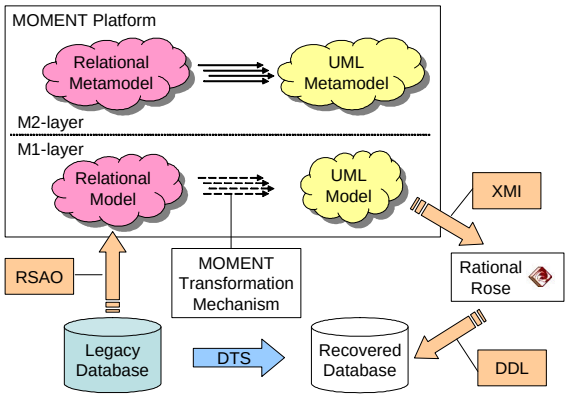

Figure 3. MOMENT solution to the legacy system recovering problem

b) Translation of the relational model into an UML term by means of the model transformation mechanism of the MOMENT Framework. Here the user may decide to apply other rewriting rule than the default rules chosen by the TRS in order to generate a more semantically accurate UML term. The applied rewriting rules in the translation process are written down to a XML document that will be used in the second phase.

c) The UML model obtained from the MOMENT transformation is stored at the M1-layer of the platform. It can be exported in an XMI document, which can be imported in turn by most of the CASE tools that manage UML diagrams, such as Rational Rose. These CASE tools provide support to manipulate a UML model and many other services such as generate its code or its database. Although the generation of the relational schema can be treated as a model transformation issue in the MOMENT platform, we also allow the communication with other CASE tools by means of XMI documents to benefit from their modeling features.

In this section, we present the way in which the MOMENT model transformation mechanism is applied to generate a UML model from a relational schema, providing formal support for generic model transformation in the MDA context. First, we explain a general overview of the transformation mechanism, and later, we focus on the most formal phases of the process.

4.1.1. Overview of the MOMENT model transformation process

Transforming any model using the MOMENT Framework constitutes a process that is detailed in Figure. 4. To obtain the corresponding UML model from the relational schema of the motivating example, we perform the following steps:

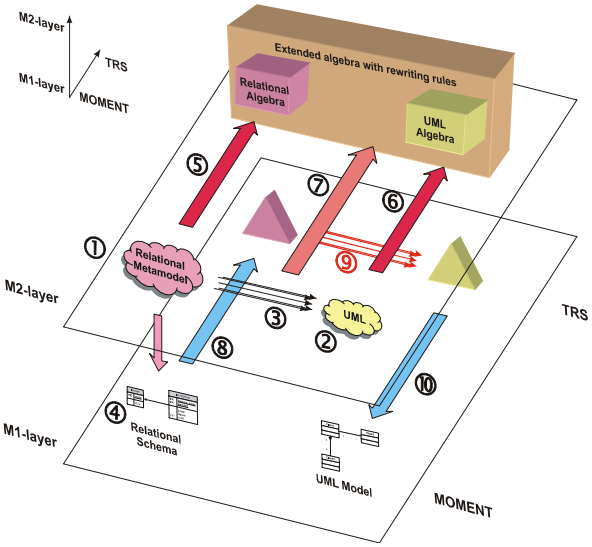

Figure 4. Overview of the MOMENT model transformation mechanism

– (1) and (2): We specify both relational and UML metamodels, respectively, at the M2-layer of the MOMENT platform using the operations of the MOMENT algebra. Each one of the metamodels is a schema made up of concepts, which describe the main entities of the ontology, and by properties, which describine the concepts by specifying values and establishing relationships between them. These algebraic specifications are performed through visual wizards that are embedded in a specific CASE tool to disguise the equational logic formalism.

– (3): Mappings are specified between the concepts of both metamodels at the M2-layer by means of a script language, indicating semantic relationships. There are two kinds of equivalence mappings that can be expressed in this language:

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 association of the UML metamodel, we have to relate the foreign key, the unique constraint and the not null value constraint concepts to the association concept. This is because all three of these concepts of the relational metamodel provide the necessary knowledge to define an association between two classes in the UML metamodel, such as the cardinalities of the association.

- (4): The original relational schema is specified by means of concepts and properties in a schema of the M1-layer of the MOMENT platform. Both concepts and properties are instances of the elements of the relational metamodel defined in step 1.
- (5) and (6): Both relational and UML metamodels, respectively, are compiled into algebraic theories by means of the *Theory Compiler* module of the Framework. The compilation uses the concepts to define the sorts of the new theory and the properties to define constructors and query operators. The generated theories are interpreted by the CafeOBJ TRS, providing the respective algebras to define models in the TRS as algebraic terms.
- (7): The semantic mapping that is specified between the concepts of both metamodels at the M2-layer is also compiled into another theory that extends the theories described above with a set of rewriting rules. This theory indicates how to transform a model of the source metamodel (relational metamodel) into a new model of the target metamodel (UML) in an automatic way.
- (8): The original relational schema, which is defined in step (4) at the M1-layer of the MOMENT platform, is compiled into a term of the relational algebra in the CafeOBJ TRS by means of the *Term Manager* module of the Framework.
- (9): The TRS evaluates the term that represents the initial relational schema in the algebra obtained in step (7). The user can manage this process through the *Evaluator* module of the Framework. The evaluation process can be carried in a step-by-step mode or in only one step with the full-evaluation mode, benefiting from the evaluation features of the TRS. The TRS reduces the initial term by applying the rewriting rules obtained in step (7), generating a term of the target algebra.
- (10): This is the last step of the model transformation process. It parses the obtained term in step (9), defining a model in the M1-layer as an instance of the target metamodel defined at the M2-layer. There, it is disguised with the visual metaphor associated to the target metamodel in the graphical CASE tool.

Previously to the transformation process, the source and target metamodels (step (1) and (2)), and the semantic mappings between the elements of both metamodels (step (3)) must be defined on the platform. In the model transformation process, the user only interacts with the MOMENT platform when defining the initial model (step (4)). The other steps are automatically carried out by the Framework, although the user can participate in the evaluation process by specifying the rewriting rules to be applied by the TRS at each step of the term reduction.

In the following sections, we explain phases (5), (6), (7), (8) and (9) in more detail, indicating how the TRS is able to perform model transformations providing formal support to the objectives of MDA.

4.1.2. Compilation of equational logic based theories

The relational and UML metamodels defined at the M2-layer of the MOMENT platform are compiled into theories based on equational logic in steps (5) and (6), respectively. The compilation of MOMENT metamodels into equational theories uses the concepts of the metamodel to obtain the sorts of the theory; for instance, the sorts Table, Field, ForeignKey for the relational metamodel, as well as the identifiers for these sorts, i.e., the sorts TableId, FieldId and ForeignKeyId. The properties of a MOMENT metamodel provide information about the structure of the term of a sort by means of the cardinalities. Thus, when a concept *A* is related to a concept *B* by means of a property that has cardinality 1..1, the constructor of the sort *A* looks like this: *op a _ : B → A*. Nevertheless, if the minimum cardinality is zero or the maximum cardinality is *many*, then the constructor for a term of the sort *A* looks like this: *op a _ : ListB → A*, where *ListB* is a *sort* that permits the definition of lists, whose items are terms of *sort B*. As CafeOBJ belongs to the OBJ language family, it permits equational specification through several equational theories, such as associativity, commutativity, identity, idempotence and combinations

between all these. This feature is reflected at the execution level by term rewriting by means of such equational theories.

Figure. 5 shows the constructors of the compiled theory for the relational metamodel; and Figure. 6 shows the constructors for the UML metamodel. We obviate the definition of sorts and other constructors in the theory, as well as the definition of query operators, focusing on the elements of the metamodels that permit us to illustrate the example. We must point out that the constructors obtained for the UML theory permit us to define terms that represent UML-compliant models.

```
-- FIELD:  id, type, is nnv, is pk, tableid, dbid
op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId
DatabaseId ->
Field {constr}
-- FOREIGNKEY:  id, field list, related table, is unique, table that
-- contains the fk, database
op foreignKey _ _ _ _ _ _ : ForeignKeyId ListField TableId Bool
TableId DatabaseId -> ForeignKey {constr}
-- TABLE
op table _ _ _ _ : TableId ListField ListForeignKey DatabaseId ->
Table {constr}
-- DATABASE
op database _ _ : DatabaseId ListTable -> Database {constr}
```

Figure 5. Part of the relational theory

```
-- ATTRIBUTE:  id, type, required, constant, identifier, ClassId,
-- schemaid
op attribute _ _ _ _ _ _ _ : AttributeId Datatype Bool Bool Bool
ClassId
SchemaId -> Attribute {constr}
-- ASSOCIATION
op association _ _ _ _ : AssociationId AssociationEndId
AssociationEndId SchemaId -> Association {constr}
-- ASSOCIATIONEND: id, id of the class, id of the association,
-- isNavigable, ordering, aggregation, min card., max card,
-- changeability, visibility, id of the schema
op associationEnd _ _ _ _ _ _ _ _ _ _ _ : AssociationEndId
ClassId AssociationId Bool OrderingKind AggregationKind
Cardinality Cardinality ChangeableKind VisibleKind SchemaId ->
AssociationEnd {constr}
-- CLASS
op class _ _ _ : ClassId ListAttribute SchemaId -> Class {constr}
-- OOSCHEMA
op ooSchema _ _ : SchemaId ListClass -> OOSchema {constr}
```

Figure 6. Part of the UML theory

4.1.3. Compilation of rewriting logic-based theories

To transform the relational schema of the example, semantic mappings have been defined between the concepts of both source and target metamodels in step (3). These mappings are compiled into an algebra that extends both relational and UML algebras (steps (5) and (6)) with a set of rewriting rules describing the guidelines for the model transformation. These rules are automatically applied by the TRS rewriting the initial term into a term of the target algebra. The new algebra constitutes the context where semantical relationships between the source and target ontologies are defined. To allow the transformation process, the new algebra must relate the *sorts* of the initial algebra (relational metamodel) to the *sorts* of the target algebra (UML metamodel). Relationships between the sorts of both algebras result in a subsort order that involves all the sorts. For instance, in the example, the sort Table is a subsort of the sort Class, indicating that a class can take the place of a table that was there before. Subsort relationships affect all the sorts of both algebras, even identifiers and lists, because they are the related concepts in the MOMENT algebra.

The properties that relate concepts in the MOMENT algebra define a canonical order among the sorts of the compiled algebras. This order is taken into account to generate the rewriting rules. We present the rewriting rules that are applied to the relational schema of the example to obtain a semantically equivalent UML model in CafeOBJ syntax:

a) Field

A field of a table becomes an attribute of a class in the term that represents a UML model. The rule reuses the features of the field (datatype, whether it is null or not and whether it is primary key) to generate an attribute. Field features also indicate the attribute datatype, whether it is required or not and whether it is the identifier of the class to which it belongs:

op field _ _ _ _ _ _ : FieldId Datatype Bool Bool TableId DatabaseId -> Attribute

eq field FI D NNV PK TI DBI = attribute FI D NNV false PK TI DBI .

b) Foreign Key

A foreign key can define an association between two classes in the UML context. The following rule is applied when the foreign key is unique and not null, obtaining an association 1..1 - 0..* between the classes generated from the related tables.

op foreignKey _ _ _ _ _ _ : ForeignKeyId ListAttribute TableId Bool TableId DatabaseId -> ListClass

ceq foreignKey FkI LA RTI U TI DBI = (association FkI TI RTI DBI) (associationEnd TI TI FkI true unordered aggregate card 0 many frozen public DBI) (associationEnd RTI RTI FkI true unordered none card 1 card 1 frozen public DBI) if U and isRequired (LA) .

c) Table

A table becomes a class. The rewriting rules must take into account the fact that a table is made up of fields and foreign keys, so that a field will become an attribute of the new class and a foreign key will

become a set of elements of the UML model, i.e., an association and two association ends, according to the UML metamodel.

op table _ _ _ _ : TableId ListAttribute ListClass DatabaseId -> ListClass

eq table TI LA nilForeignKey DBI = (class TI LA DBI) .

eq table TI LA LC DBI = (class TI LA DBI) LC .

d) Database

Finally, a database is rewritten into a term of the sort OOSchema, representing the target UML model, by means of the following rule:

op database _ _ : DatabaseId ListClass -> OOSchema

eq database DBI LC = ooSchema DBI LC .

Relational Schema (a)

```
database 'InvoiceRS'
((table 'Invoice'
  ((field 'code' integer true true 'Invoice' 'InvoiceRS')
   (field 'date' datetime false false 'Invoice' 'InvoiceRS'))
  nilForeignKey
  'InvoiceRS')
(table 'InvoiceLine'
  ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS')
   (field 'number' integer true true 'InvoiceLine' 'InvoiceRS')
   (field 'description' string false false 'InvoiceLine' 'InvoiceRS')
   (field 'date' datetime true false 'InvoiceLine' 'InvoiceRS')
   (field 'hours' decimal true false 'InvoiceLine' 'InvoiceRS'))
  ((foreignKey 'FK-InvoiceLine-Invoice'
     ((field 'code' integer true true 'InvoiceLine' 'InvoiceRS'))
     'Invoice‘ true 'InvoiceLine‘ 'InvoiceRS'))
  'InvoiceRS'))
```

```
OOSchema 'InvoiceRS'
((class 'Invoice'
  ((attribute 'code' integer true false true 'Invoice' 'InvoiceRS')
   (attribute 'date' datetime false false false 'Invoice‘ 'InvoiceRS'))
  'InvoiceRS‘)
(class 'InvoiceLine'
  ((attribute 'code' integer true false true 'InvoiceLine' 'InvoiceRS')
   (attribute 'number' integer true false true 'InvoiceLine' 'InvoiceRS')
   (attribute 'description' string false false false 'InvoiceLine' 'InvoiceRS')
   (attribute 'date' datetime true false false 'InvoiceLine' 'InvoiceRS')
   (attribute 'hours' decimal true false false 'InvoiceLine' 'InvoiceRS'))
  'InvoiceRS‘)
(association 'FK-InvoiceLine-Invoice' 'a' 'b' 'InvoiceRS')
(associationEnd 'a' 'InvoiceLine' 'FK-InvoiceLine-Invoice' 'InvoiceRS')
(associationEnd 'a' 'Invoice' 'FK-InvoiceLine-Invoice' 'InvoiceRS'))
```

UML Model (b)

Figure 7. Original term representing the source relational schema (7.a), and the generated term representing the target UML model (7.b).

4.1.4. Term rewriting process

Step (8) compiles the relational schema defined in the M1-layer of the MOMENT platform, obtaining the algebraic term in Figure. 7.a. The TRS applies the rewriting rules specified above to this initial term, obtaining a term of the target algebra (UML), shown in Figure. 7.b. This term is parsed and is defined as a UML model in the M1-layer of the MOMENT platform. There, it is automatically related to graphical pictures in a specific CASE tool.

During the term rewriting process, some additional information could be required in order to perform a correct transformation, as the metamodels do not have the same expressive power. For instance when a transformation case has not been taken into account or when several rewriting rules can be applied to the source model. In these cases a visual wizard helps the user to chose one option or even to add a new transformation rule, providing a visual interface for the CafeOBJ interpreter.

To benefit from the MOMENT features, we have integrated the functionality of the MOMENT Framework into a visual modeling environment [14]. In this way, we can relate algebraic specifications to visual notations so that the user can use these graphics to build a model. The CASE tool we have chosen is MS Visio [15]. We have developed an add-in that permits the definition of metamodels with concepts and properties. Figure. 8 shows the interface that permits the definition of the graphical symbol of a class in the MOMENT platform.

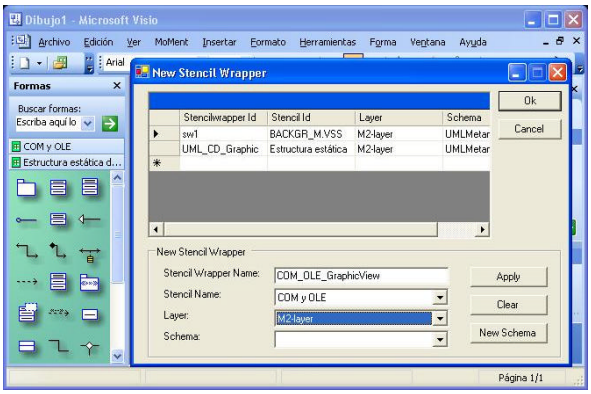

Figure 8. Visual interface to define a graphical primitive algebraically

In visual CASE tools, models are usually defined by dropping graphical primitives on a sheet where the model is defined. By means of the developed add-in, dropping a primitive on the sheet does not only add a figure to the model but it also defines it algebraically, specifying the model so that it can be manipulated afterwards.

5. Relational Migration Plan Generator

This phase generates a migration plan that specifies what information must be copied from the legacy database to the database of the new OO application. Its

inputs are two XML documents that contain the mappings:

- between elements of the legacy relational schema and elements of the recently generated OO conceptual schema.
- between elements of this OO conceptual schema and elements of the new relational schema generated by the Rose Data Modeler add-in.

The migration plan generator applies a set of patterns to the input correspondences and produces a migration plan that is specified using a relational declarative language indicating what information has to be copied from the legacy database to the new one. The use of a declarative language provides independence from the specific DBMSs used for supporting the databases. Additionally, this phase checks the constraints of the target database in order to avoid constraint violation.

5.1. Relational Migration Plan

A relational migration plan specifies the actions that must be performed in order to copy data from the legacy database to the new database, generated from the recovered OO conceptual schema. The migration plan consists of a set of migration modules. There exists one migration module for each specific table of the target database. Therefore, a migration module assigns a view over the legacy database to a target table indicating where to find the source data.

To specify the data copy process, a migration module contains a set of mappings between columns of the source view and the target table. Those mappings constitute the migration expressions that can be used in a migration plan and they are specified by means of the relational declarative language.

The automatic generation of the migration plan considers its structure and its contents. Thus, two kinds of patterns are used: migration patterns and migration expression patterns. Migration patterns generate the structure of the migration plan following the ordering of the tables of the new database. Migration expression patterns generate their content by means of migration expressions that represent mappings between columns of the relational tables.

The migration plan generator gets the applied rewriting rules of the reengineering process from the two input XML documents, one from the data reverse engineering phase and another one from the Rose Data Modeler. These rules provide enough information to determine how many migration modules are needed and which tables of the legacy database form the source view for each module. Thus, the generator constructs the migration modules by applying the migration patterns. Then, it applies the migration expression patterns in order to link attributes of the source view with attributes of the target table in each migration module, reflecting the generation process followed to obtain the target database. Once the migration plan is finished, the generator writes it in an XML document, ready to be compiled into a specific technology in order to perform the data migration.

6. Migration Plan compiler

This phase performs the compilation of the migration plan to a specific technology and its execution in order to carry out the physical data migration. We use the Data Transformation Services (DTS) of Microsoft SQL Server. This tool allows data migration between heterogeneous relational DBMS by applying data transformation services in order to fulfill target database requirements.

The compiler receives an XML document that describes the migration plan from the second phase of RELS and obtains a set of DTS packages that are able to perform the specified migration plan between the legacy database and the new one.

7. Related works

The MétaGen project [16] has dealt with model engineering since 1991, aiming at a fully automatic generation of a conventional application from a description given by its intended user. Such a description is performed by means of PIR3, a variant of what is known in the Database community as Entity-Relationship Model. In [17], Revault et al. compared three metamodeling formalisms to share the experience they acquired during the MétaGen project. In that paper, they presented a way to transform MOF-based metamodels into PIR3-based metamodels so that the metamodels could benefit from the MétaGen tools. This proposal constrains the expressivity of the source metamodels because the Object-Oriented paradigm is richer than the Entity-Relationship paradigm. Our model management approach supports the definition of several metamodeling languages such as MOF or PIR3, considering them as models at the same abstraction level and using generic operators. This makes automated transformations between models of both metamodels possible, without loss of expressivity.

XML [18], the standard for data communication between applications is also used to represent models and metamodels by means of the XMI specification [19]. MOF defines a meta-metamodel, while XMI

indicates the physical representation for the metamodels and the models that can be defined with it. In the model transformation field, there is a XML specification that allows the transformation of a XML document into another one, called XSLT [20]. It could be used to transform models that are represented in the XML format. Comparing XSLT to term rewriting systems, there are some differences that should be pointed out:

- Although XSLT is said to be a declarative language, control instructions, such as jumps and loops, can be used to guide the transformation process. In contrast, a term rewriting system takes over the transformation rule evaluation process.
- Writing an XSLT program is a long and painful process which implies poor readability and high maintenance cost for associated programs. Also, writing an XSLT program requires good skills in the MOF and XMI specification, because when using XPATH and XSLT, the developer must take into account the structure of models which depends on metamodels. These metamodels are in turn widely influenced by MOF and XMI. By expressing metamodels as algebras, we can deal with a more specific syntax that reflects their semantics using the algebras as domain specific languages. Therefore, writing models (and consequently transformation rules) becomes easier and more comprehensible.
- Transformation rules in XSLT are applied without taking into account the target XML schema (metamodel when transforming models), implying a posterior checking to determine whether the obtained document is a valid XML document that conforms to the target schema. Rewriting rules in an algebra take into account the source and the target algebras so that a posterior checking is no longer needed.
- Executing an XSLT program is not user-friendly for model transformation because there are no error messages to advise the user about an incorrect transformation.

The MTRANS Framework [21] provides an abstract language to define transformation rules that are compiled to XSLT. Even though this language is more compact and easier to understand than XSLT, it still keeps instructions to manage the transformation rules evaluation. Nevertheless, transformations using XML technology imply the use of standard specifications that are industrially supported while term rewriting systems usually remain within the field of research.

8. Conclusions

Nowadays, software applications have become complex combinations of technology, which have to be well understood in order to manage them. The development of software artifacts involves models that can be mixed with others to obtain an entire system from partial views or that can be interconnected with others in order to guarantee both interoperability in a distributed environment and their implementations.

MDA raises the level of abstraction in the software development process by treating models as primary artifacts. MDA potentially covers the modeling of all aspects of a system throughout its life cycle, making software development processes easier and more automated.

The MOMENT (MOdel manageMENT) platform follows this trend by providing a framework where models can be represented using an algebraic approach. The MOMENT Framework benefits from the best features of current visual CASE tools and from the main advantages of formal environments such as term rewriting systems, combining both industrial and research features.

In this paper, we have presented the generic model transformation mechanism provided by the MOMENT Framework, focusing on the use of the CafeOBJ Term Rewriting System to perform automatic translations of models. This mechanism has been applied to legacy relational database recovering in the MDA context, obtaining a UML model from a relational schema. The functionality of TRSs permits us to deal with model transformation from a more abstract point of view, since the application of rewriting rules can be performed in a transparent way. This fact allows us to focus all the efforts on the specification of models without having to take the evaluation logic into account. Our approach constitutes an algebraic baseline to cope with the future model transformation language QVT, providing a user-friendly environment to manipulate models from a visual CASE tool [14]. In [22], we present the fundamental mainstay on which we have built our MOMENT platform taking into account our previous experience in the industrial project RELS, a tool for the recovery of legacy systems.

Currently, we are working with transformations between relational schemas and UML models. In the near future, we will also take into account software architecture specifications by means of PRISMA ADL [23], studying semantic interoperability between software architectures and other types of software artifacts represented through UML models.

9. Acknowledgments

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

10. References

[1] Ward, M.P., and Bennett, K.H., "Formal Methods to Aid the Evolution of Software"; *Journal of Software Maintenance: Research and Practice*, Vol 7, no 3, May-June 1995, pp 203-219.

[2] OMG, "The Model-Driven Architecture, Guide Version 1.0.1", OMG Document: omg/2003-06-01. Available from www.omg.org

[3] OMG, "Meta Object Facility 1.4", OMG Document: formal/02-04-03. Available from www.omg.org

[4] OMG, "MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2002-04-10. Available from www.omg.org

[5] Hainaut, J. L., Henrard, J., Roland, D., Englebert, V., and Hick, J.M., "Structure Elicitation in Database Reverse Engineering", WCRE'96 3 (1996), 131-140.

[6] Premerlani, W.J., and Blaha, M.: "An approach for reverse engineering of relational databases"; *Communications of the ACM* 37,5 (1994), 42-49.

[7] Ramanathan, S., and Hodges, J.: "Reverse Engineering Relational Schemas to Object-Oriented Schemas"; Technical Report MSU-960701, Mississippi State University, Mississippi, 1996.

[8] 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.

[9] CDIF Technical Committee, "CDIF Framework for Modeling and Extensibility", Electronic Industries Association, EIA/IS-107, January 1994. See http://www.cdif.org/

[10] Razvan Diaconescu, and Kokichi Futatsugi, "An overview of CafeOBJ", *Electronic Notes in Theoretical Computer Science* vol. 15: (2000)

[11] Microsoft Research (Don Syme), "The F# official web site": http://research.microsoft.com/projects/ilx/fsharp.aspx

[12] N. Martí-Oliet and J. Meseguer, Rewriting logic: roadmap and bibliography, *Theoretical Computer Science*, vol. 285, issue 2 (August 2002), pp. 121-154.

[13] Boggs, W., Boggs, M.: "Mastering UML with Rational Rose 2002"; Sybex. January 2002.

[14] A. Boronat, J. Á. Carsí, I. Ramos, "An Architecture for the Definition of Graphic Metaphors for Metamodels", Software Engineering and Databases Workshop (Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2004). Málaga, 10-12 November 2004 (in Spanish).

[15] Graham Wideman, "Microsoft Visio 2003 Developer's Survival Pack", Trafford, 2004.

[16] N. Revault, H.A. Sahraoui, G. Blain and J.F. Perrot, A Metamodeling technique: "The MÉTAGEN system", TOOLS 16: TOOLS Europe'95, Prentice Hall, pp. 127-139. Versailles, France. Mar. 1995.

[17] N. Revault, X. Blanc and J. F. Perrot, "On Meta-Modeling Formalisms and Rule-Based Model Transforms", Comm. at workshop, In Iwme'00 workshop at Ecoop'00, Jean Bézivin and Johannes Ernst (ed), Sophia Antipolis and Cannes, France, June, 2000.

[18] Elliotte Rusty Harold, "XML Bible", IDG Books Worldwide, 1999.

[19] OMG, XML Metadata Interchange (XMI) Specification, OMG Document formal/02-01-01.

[20] W3C, XSL Transformations (XSLT) v1.0. W3C Recommendation, http://www.w3.org/TR/xslt, Nov. 1999.

[21] M. Peltier, J. Bézivin, and G. Guillaume, "MTRANS: A general framework, based on XSLT, for model transformations". In WTUML'01, Proceedsings of the Workshop on Transformations in UML, Genova, Italy, Apr. 2001.

[22] A. Boronat, J. Pérez, J. Á. Carsí, and I. Ramos, "Two experiencies in software dynamics". *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. April 2004.

[23] J. Pérez, I. Ramos , J. Jaén, P. Letelier, E. Navarro, "*PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*", 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 © IEEE Computer Society Press pp. 59-66.