An Approach for Cross-Model Semantic Transformation on the .NET Framework

Artur Boronat, José Á. Carsí, Isidro Ramos, Julián Pedrós
Department of Information Systems and Computation
Technical University of Valencia
Camí de Vera s/n
46022 Valencia (Spain)
{aboronat | pcarsi | iramos | jpedros}@dsic.upv.es

ABSTRACT

Model-Driven Development is a suitable approach for improving productivity and quality in the software development process by raising the level of abstraction of software artifacts from code to models. In this context, code generation has traditionally been the star feature. Working on models also provides more reusable solutions to problems that have to be solved in an ad-hoc manner using the .NET technology: interoperability between applications, integration of applications, legacy system recovery, software evolution, maintainability, etc. One mechanism for dealing with models is model transformation. Although several tools follow this approach to generate code that targets the .NET platform, there are no tools based on .NET technology that provide model manipulation such as transformations. In this paper, we present a platform that permits the formal representation of models and an operator to transform models in a declarative way. This platform has been implemented using the F# functional programming language, presenting its advantages over an implementation using an imperative programming language such as C#. The platform has been integrated into the Visio modeling environment by means of an add-in to deal with formal models through visual metaphors (visual notation). To our knowledge, this solution is the first approach for dealing with cross-model semantic interoperability on the .NET technology.

Keywords

Model-driven development, model transformation, graphical notation, MS Visio 2003, F#, Office managed COM add-in, cross-language interoperability.

1. INTRODUCTION

Model-Driven Development (MDD for short) [Sel03] is a suitable approach to combat the complexity of software development by means of principles such as abstraction and modularity, which improve the quality, reuse, and scalability of software artifacts. This discipline also improves the productivity and quality in the software development process to obtain automatically error-free code that is easy to maintain. Following this approach, a software artifact is modeled at a high level of abstraction where technical details are not as important as semantics. The structure and semantics of a software artifact are modeled by using an ontology or metamodel (a vocabulary that provides constructs to specify a model in a determined manner). A metamodel can be domain-independent such as UML, or domain-specific, taking into account specific types of software systems, such as banks, electronical circuits, business modeling, etc.

In accordance with [Cza00], the MDD approach based on UML-like metamodels is called Object-Oriented Analysis and Design, while the MDD approach based on domain-specific metamodels is called Domain Engineering. Microsoft has shown a growing interest in the MDD discipline by adding designers to the Visual Studio environment in order to build software artifacts by means of models. This technology should be expanded to cover domains of interest to their customers, such as code visualization, business modeling, etc., thereby

.NET Technologies'2005 *conference proceedings,*
ISBN 80-903100-4-4

applying Domain Engineering from a commercial standpoint [Coo04].

In MDD, models defined by means of metamodels are usually transformed into code, providing the final application that can be directly compiled and executed on a specific platform, such as .NET. There are lots of tools that provide code generation based on models in the .NET world, called model compilers: from visual modeling environments (such as Visio, Rational XDE [Rat] among others) to development environments (such as the Visual Studio .NET).

However, generating code from models forces the programmer to work on code in order to face well-known problems in the software engineering field: round-trip, application integration, legacy system recovery, refactorization, software evolution, maintainability, etc. These problems can be solved at a more abstract level by dealing with models directly, obtaining the same advantages that the MDD discipline obtains for the software development process. This is the point where model transformations come into play [Sen03]. To provide support for model transformations, two main issues have to be taken into account: model representation to structure the information in some accessible manner and a transformation mechanism to manipulate such models. Although this issue is becoming well-known in the research field [Cza03], to our knowledge, there are no tools based on the .NET technology to achieve transformations of this kind. A solution of this nature would improve the productivity and the quality in the integration of .NET-based applications at a high level of abstraction, rather than just benefiting from the cross-language interoperability that the .NET Framework provides at code level. Therefore, a solution of this nature would achieve cross-model interoperability.

In this paper, we provide a solution along these lines. We present a mechanism that takes advantage of the MS Visio modeling tool in order to describe the structure of visual models in a formal manner. This mechanism uses a platform to represent and store software artifacts in four layers, where metamodels and models are taken into account. Models defined on the platform can be transformed in a declarative fashion by using the platform operator *generate*, which permits the translation of a model between different metamodels.

This platform has been developed using the functional language F# [Fsh]. Taking into account its advantages over conventional OO languages such as C#, models are formally described in an algebraic fashion.

Our solution takes advantage of the .NET cross-language interoperability and the Office extension mechanism by means of managed COM add-ins. It extends the Visio tool by using the most suitable language in each context: F# to implement the definition of formal models and their manipulation, and C# to integrate this functionality into the Visio tool.

The structure of the paper is as follows: in Section 2, we discuss and compare the C# and F# programming languages, evaluating their suitability in our solution; Section 3 presents a platform that enables the definition of models in an algebraic fashion; Section 4 presents the add-in that integrates this platform into the Visio modeling environment, enabling the manipulation of formal models by means of graphical metaphors (graphical notation); Section 5 describes the F# definition of the model transformation mechanism that is provided by the platform; finally, Section 6 summarizes our contributions.

2. F# versus C#

F# is a functional programming language that targets the .NET platform. F# has been developed at MS Research Cambridge and is a version of the Caml programming language [Cah00], which belongs to the ML languages family. F# is well integrated in the Visual Studio environment[1] and provides certain features that are inherited from Caml, which make it interesting for our purposes.

F# is based on the lambda-calculus model [Rea93] by means of a strict (eager) evaluation strategy. Therefore, it permits the definition of a program independently from the evaluation strategy used, that is without mixing functionality and control logic as is necessary in C#.

F# provides richer constructs to declare types like sum types, among others. A sum type permits the definition of a type by means of constructor patterns, each of which may have arguments. Sum types allow us to describe the signature of an algebraic specification [Ehr85], where the name of the type is the sort, and the constructor patterns are the constructors of a sort. This comparison allows us to deal with models from an algebraic point of view, where semantics of models can be described formally by means of Abtract Data Types. This feature is not feasible in C# intuitively, although it can be simulated in the same way that such constructors are invoked from C# code by means of static methods.

[1] Although we used F# version 0.6.4.1 for this solution.

F# provides a conditional pattern matching mechanism that enables the definition of functions over sum types in an intuitive way by applying a pattern to each constructor in order to perform a task. This mechanism can be simulated in a more complex way in C# by means of the *switch* statement and the addition of *if* statements inside each *case* of the *switch* statement.

The F# compiler infers the types of the declaration of a function statically (the types of its arguments and the type of its closure, i.e. the type of the returned value), so that these types do not have to be indicated in the definition of the function. This feature makes the definition of F# programs easier.

As all values are functions in F#, we can use lists of functions whenever we need them, rather than using delegates, as it happens in C#. This also provides parametric polymorphism that is used to provide some parametric functions that deal with lists without knowing the types of their elements: *map* to apply a function to the elements of a list, *find* to search the elements of a list that validate a condition, *exists* to know if some elements of a list validate a condition, etc. This feature, called generics, has not been added to the current release of the .NET Framework, although it will be added to the next release [Yu04].

Although F# is a functional language, it also provides imperative features such as references (pointer to a value), which allow us to manipulate the memory state whenever necessary for the sake of efficiency. Furthermore, the last and the most important feature of F# is its full interoperability with languages that target the .NET platform, such as C#, by means of the ILX extensions [Sym01] to the IL language.

All these considerations have encouraged us to use F# for the implementation of our solution to deal with models from a formal standpoint, on the grounds that we can use C# to integrate our solution to tools based on the .NET technology, such as the Visio modeling environment.

3. ALGEBRAIC REPRESENTATION OF MODELS BY MEANS OF F#

Our approach constitutes a platform that uses several metadata layers to describe any kind of information. In our work, we consider software artifacts in four abstract layers (as shown in Figure 1):

- 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 (metamodels) that define the structure and semantics of the models located at the M1-layer. A metamodel is an "abstract language" that describes different kinds of data.
- 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 for metamodels. This layer provides the "abstract language" to define different kinds of metadata.

The core of the prototype is an algebra that 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 language for two main reasons: to bring a formal model transformation approach closer to an industrial programming environment, such as .NET, and to benefit from the functional programming advantages presented above.

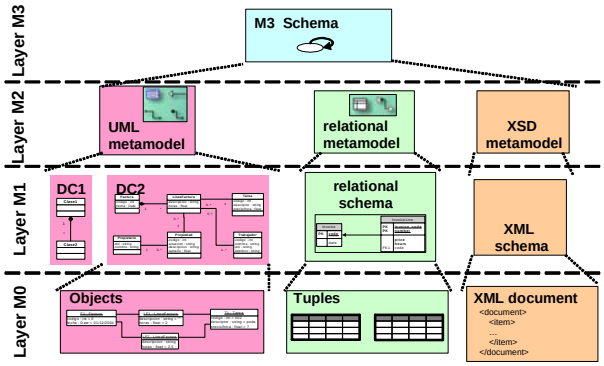

Figure 1. Graphical representation of the four-layered platform.

An Algebra for Representing Models

The algebra aims to represent models of any kind as algebraic terms in order to automate model transformation tasks in a precise, formal way. Achieving this objective implies choosing a basic specification language that permits us to describe any piece of data.

We have developed a platform based on this algebra that permits the representation of software artifacts in the four meta-layers explained above. Four main sorts permit the definition of a model as a term in the algebra:

1. *Concept*

   A concept represents an entity that can be described by means of properties. The constructor of this sort is defined 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*

   A property is a relationship that relates either a concept or a property (subject of the property) to a concept (the object of the property), following the RDF philosophy to describe metadata [W3C]. 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)
```

where *NilProperty* represents the null property term and the arguments of the constructor *Property* are the following elements in order of appearance:

- Parent property indicating its type.
- Identifier 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 on the grounds that 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 a 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 a basic vocabulary to define metamodels at the M2-layer, i.e. a simplified meta-metamodel. This schema contains the term Concept and the term Property; the latter relates two concept terms, constituting the minimal structure that we use to represent a model at a lower layer. The four layers of the platform are defined as values that can be accessed by means of references (pointers to a value). This simplifies the definition of transformation rules and enhances efficiency.

For instance, the *Relational Metamodel* is a schema term that contains the concepts and properties that constitute the terminology to define a relational schema, as shown in Figure 2. For instance, *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. In an identical way, the concept *Code* is defined as a column, which is related to the table *Invoice* by means of an instance of the property *table/Column*, i.e. by means of the invoice/Code property.

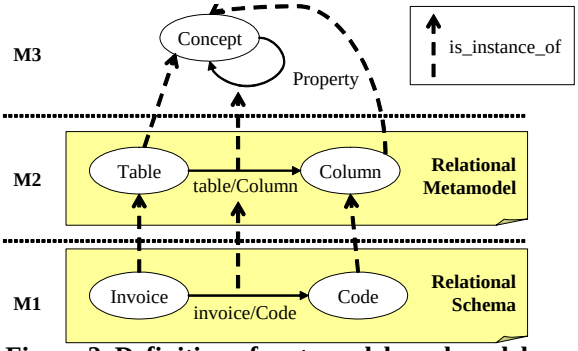

Figure 2. Definition of metamodels and models on the platform.

4. VISIO AS A VISUAL ENVIRONMENT FOR DEALING WITH ALGEBRAIC MODELS

Taking into account the four-layered platform based on the functional implementation of the presented algebra, we have developed an add-in for MS Visio 2003, called *Platform Integrator*. This add-in permits the association of a graphic metaphor with a formal metamodel in the Visio modelling environment and the automatic definition of its models as algebraic terms.

The customization of the Visio modeling visual environment is performed by means of add-ons, i.e. sets of stencils that provide the graphical information needed to define the graphical notation for a metamodel. To extend the tool, a type of module, called an add-in, is used to add functionality. Given the easy extension that such add-ins provide by means of managed COM (Component Object Model), Visio is the selected tool to embed our model repository. The formal definition of models, which our add-in provides, allows us to transform models as we present in the following section, rather than merely defining the models graphically.

Outside the Add-in

The add-in architecture is divided into three layers: the interface that graphically represents metamodels and models; the middle layer that permits the association of such graphics to algebraic representation of models; and the persistence layer that stores all the information.

In the middle layer, the module *Platform Integrator* enables the definition of associations between the graphical elements of the interface of Visio and the algebraic terms that define software artifacts in the four-layered platform. Such associations are stored in the same platform as instances of UML classes at the M0-layer by means of the *UMLSupport* library.

The persistence layer consists of two types of storage units: the one provided by Visio and the one provided by model repository of the platform.

In Visio, graphical models are stored by means of two types of files: .vss files that store the model defined in the shapesheet, and .vst files that provide the templates with masters (stencils), which enable the definition of shapes in the shapesheet. Visio provides several templates with several kinds of masters to define a large variety of models by default. Nevertheless, a user can define new templates to define other types of models.

The four-layered platform stores the information in a RDF repository on the grounds that the concepts and properties used in the platform are equivalent to RDF resources and properties, respectively. The repository used is Redland [Bec01], which we have embedded in a visual studio project and compiled on the .NET platform by means of the managed C++ programming language. In this repository, we store schemas that belong to any layer of the platform, and we store associations between graphical elements of the modeling environment and algebra terms.

Inside the Add-in

The graphical elements of the Visio interface are related to platform elements by means of the module *Platform Integrator*. To present both the definition of a graphical metaphor related to a metamodel and the definition of formal models by means of this association, we focus on the M2-layer and the M1-layer of the platform. These layers store information of metamodels and models, respectively. In this way, a schema of the M2-layer is related to a Visio stencil, while a schema of the M1-layer is related to a Visio shapesheet. To graphically represent the concepts and the properties that constitute a metamodel at the M2-layer, we use the masters that define the chosen stencil. In the case of the graphical representation of elements that constitute a model at the M1-layer, we use shapes that are defined by means of masters of a stencil.

The association mechanism that relates a formal model to a graphical representation has been modelled in Figure 3 using UML notation. In this model, the *SchemaWrapper* class contains the information needed to relate a schema to its visual representation, while the class *NodeWrapper* contains the information that relates a concept or a property to a specific image. Specializations of both classes identify whether a schema is a metamodel (*GraphicViewWrapper*) or a model (*GraphicModelWrapper*), and whether either a concept or a property is a metamodel element (*GraphicPrimitiveWrapper*) or a model element (*PictureWrapper*). In the case of a metamodel, an instance of the *GraphicViewWrapper* class relates a schema of the M2-layer to a stencil, and an instance of *GraphicPrimitiveWrapper* class relates a node of the schema to a master. In the case of a model, an instance of the *GraphicModelWrapper* class relates a schema of the M1-layer to a shapesheet, and an instance of the *PictureWrapper* class relates a node of the schema to a shape.

Storage of UML Software Artifacts

The association between Visio graphical elements and platform elements (defined in the UML class diagram in Figure 3) is stored in the same four-layered platform. In this way, the platform is used as an object-oriented repository on the grounds that it enables both the definition of UML models at the M1-layer and the definition of their instances at the M0-layer.

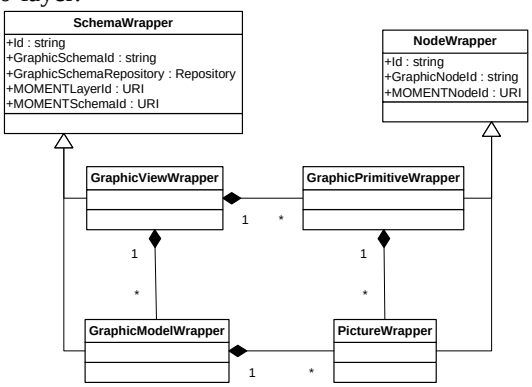

Figure 3. UML Model of the association mechanism between graphical elements and algebraic terms.

To achieve this, we have specified part of the UML metamodel as a schema at the M2-layer of the platform, taking into account classes and associations. The class diagram in Figure 3 has been specified in a schema at the M1-layer of the platform as an instance of this UML metamodel. Therefore, to

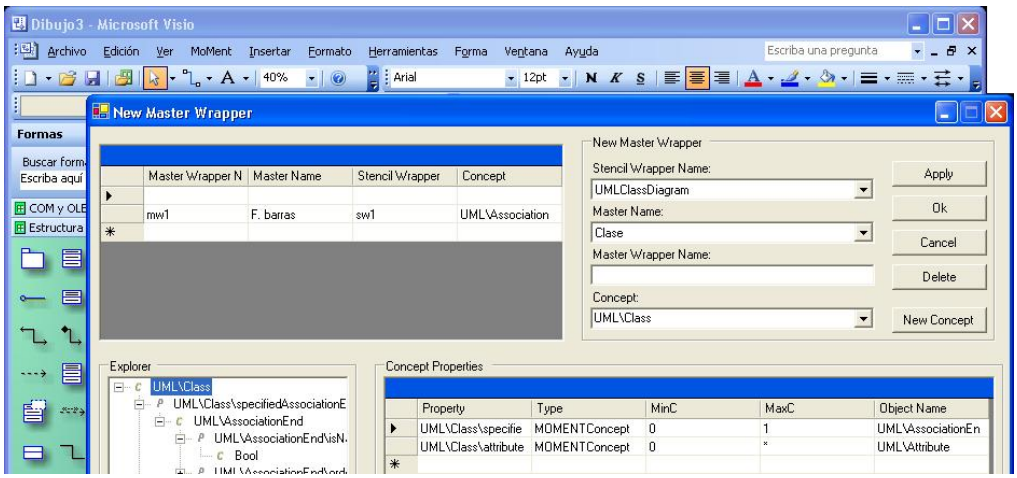

Figure 4. Interface to graphically define concepts and properties of a metamodel.

define associations between elements of the platform and Visio graphical elements, a schema can be defined at the M0-layer of the platform by instantiating the classes that constitute the model at the M1-layer.

Definition of Graphic Metaphors for Metamodels

To define a metamodel by means of Visio, we associate a schema of the M2-layer of the platform to a stencil. Then, each of its masters is related to a node of the schema by means of the interface in Figure 4, completing the graphical metaphor related to the metamodel. The formal metamodel can be directly defined on the platform by means of the Visio interface; it can also be loaded from the platform.

To define a master in the interface shown in the Figure, an association between a metamodel of the platform and a stencil must be selected. Then, a master of the stencil and a node of the schema are selected and related by means of a new association. After this, the hidden properties of a node (i.e. the properties that are not related to a master directly) can be accessed. They are shown in the list that appears at the bottom of the interface, and they can also be navigated recursively by means of the tree, placed on the left part of the interface.

Once a metamodel is graphically defined in Visio, it can be used to define visual models by means of the *drag-and-drop* mechanism, by dropping masters of the stencil onto the *shapesheet*. We enrich this mechanism so that the shape is not only graphically defined in the *shapesheet* but also its contents are defined in the platform. This functionality is embedded in the *Platform Integrator* module, providing this functionality automatically in a transparent manner to the user. Thereby, we not only define a graphical model but also provide the semantic information related to a metamodel.

Therefore, we can define formal models in a visual manner so that they can be manipulated by means of transformations as we explain in Section 5.

5. TRANSFORMATIONS

The operator *generate* permits the translation of a model of a specific metamodel into a model of a different metamodel. The semantics of the operator is defined denotationally by means of the pattern matching mechanism of F#.

The operation *generate* defines an evaluation strategy operationally (by means of the pattern matching mechanism) in order to enable the definition of transformation rules in a declarative manner. In this way, transformation rules are defined like axioms that do not take into account the rule evaluation strategy embedded in the operator *generate*. In this section, we introduce the *likeness* relation[2], which enables the definition of transformation rules based on metamodels. Then, we describe the structure of a transformation rule. Finally, we define the operational semantics of the operator *generate*.

Semantical Relationships between Metamodels

In our approach, transformations are based on metamodels. Applying a transformation to a source model involves two metamodels: a source metamodel that describes the structure and semantics of the source model, and a target metamodel that provides the structure and semantics for the new model to be generated. Two types of transformations can be distinguished taking into account the target metamodel:

[2] We have chosen the name *likeness* instead of *equivalence,* on the grounds that the equivalence relation is defined between elements of different metamodels, which cannot be equal.

– Intra-metamodel: when both the target and the source metamodels are the same. In this paper, we do not discuss this type of transformation.

– Inter-metamodel: when both the target and the source metamodels are different.

By basing our transformations on metamodels, we can specify them from an abstract point of view taking into account the metainformation that constitutes both the source and the target metamodels. This way, transformation rules can be viewed as patterns that can apply to any model of the source metamodel. To define such rules, we introduce the *likeness* relation between elements of two metamodels.

The *likeness* relation is based on mappings between the elements of both the source and the target metamodels. A mapping is a property that is an instance of a property, called *Likeness*, defined at the M3-layer. A mapping relates a concept of the source metamodel to a concept of the target metamodel. For example, indicating that a Table in the relational metamodel is like a Class in the UML metamodel. A set of mappings conforms a *likeness* relationship indicating that the concepts, which participate in these mappings, represent a similar semantic meaning in their respective metamodels.

A *likeness* relationship between elements of two metamodels may involve more than one element of either the source or the target metamodels. Thus, we distinguish between:

– Simple *likeness* relationships: specified by means of only one mapping.

– Complex *likeness* relationships: specified by a set of mappings involving several elements from either the source or the target metamodels. For example, 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 these three 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.

A *likeness* relation between two metamodels is defined as the union of all *likeness* relationships established between the elements of both metamodels. As a first approach to provide cross-model semantic interoperability on the .NET platform, we only focus on simple *likeness* relationships.

Transformation Rules

The operator *generate* is applied to a schema of the source metamodel and defines a new schema of the target metamodel. To achieve the transformation, this operator is based on a *likeness* relation defined between both the source and the target metamodels. By means of this relation, the operator knows what should be generated from a set of concepts of a source model.

To process the elements of the source schema, the operator *generate* makes use of transformation rules defined declaratively. Each one of them is divided into two functions: a condition and a body. When a group of elements of the source model is processed, the condition checks their properties in order to select which generation function should be applied. These conditions take into account the order of precedence 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 body of the transformation rule involves the definition of concepts and properties in the target schema.

The operator *generate* automatically generates models among different metamodels taking into account a set of transformation rules, which are applied following a specific evaluation strategy. The set of transformation rules is defined independently of the evaluation strategy chosen. To achieve this issue, all the transformation rules must have the same declaration so that the operator *generate* knows how to apply them. By declaration, we mean the declaration expression of a function in a F# interface (.mli), which involves the symbol that identifies the function value, the types of the arguments and the type of the function value (i.e. the type of the closure). Therefore, we denote the declaration expression of a function as follows:

```
val function_name :
    arg1_type -> … -> argn_type
-> closure_type
```

This is the value declaration inferred statically by the compiler where *val* is a reserved construct that indicates the declaration of a value; *function_name* is the symbol that identifies the function; *arg1_type -> … -> argn_type* are the types of the argument list of the function; and *closure_type* is the type of the closure (which is viewed as the type of the returned value in imperative programming).

As a transformation rule is divided into a condition function and a body function, we present their declarations in the following subsections.

5.1.1 Condition function

A condition function is the mechanism that indicates when a transformation rule can be applied. There is one, and only one, condition function for each function body. This means that a condition can indicate the suitability of only one transformation rule in a specific context, although many transformation rules can be applied to the same group of elements of a source model. The declaration expression for a condition function is as follows:

> ***val*** *condition_name : Schema -> Node -> bool*

where *condition_name* is a symbol that identifies the condition, the first argument is the source schema to be translated, and the second argument is the current node of the source schema to be translated.

The condition function checks wheter or not the specified node validates a set of requirements in order to determine if it can be translated into the target schema by means of the body function of the transformation rule. Finally, the closure of the condition function is a Boolean value indicating the suitability of the transformation rule that contains the condition.

5.1.2 Body function

The body function of a transformation rule materializes a *likeness* relationship, defined between elements of both the source and the target metamodels. This materialization involves both the definition of new elements into the target schema and specific mappings between the elements of the source schema, which are involved in the transformation rule. Such mappings provide support for traceability. Given a transformation process between two models, traceability [Got94] enables the identification of elements that are related by means of the application of a transformation rule and that belong to different models. Traceability support enhances mechanisms such as change propagation and round-trip between models.

To explain the semantics of the operator *generate*, we use the generation of a UML model from a relational schema as an example. The materialization of a *likeness* relationship at the M1-layer (between models) is obtained by four steps, shown in Figure 5:

1. The concept is reified in its metaconcept; that is, if the concept to be processed is the table *Invoice*, we obtain the metaconcepts *Table* of the relational metamodel.
2. Once we know the corresponding metaconcept of the source metamodel, we navigate the *likeness* relationship 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 similar to the original concept in step 1, through the *likeness* relationship that we have defined before.
4. Finally, the operator instantiates the *likeness* relationship 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 mapping in the traceability schema at the M1-layer, which is a property that has the source *Concept* as domain and the target *Concept'* as range.

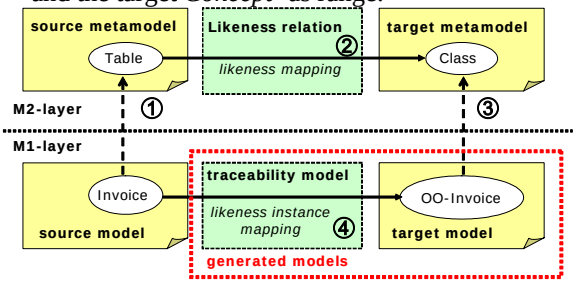

Figure 5. Description of the transformation process

The declaration of a body function is as follows:

> ***val*** *body_name : Schema -> Node -> unit*

where *body_name* is the symbol that identifies the body function, the first argument is a schema that contains the specific mappings between elements of both the source and the target models, and a node is the element of the source model to be translated. A body function knows the source and the target models by means of the mapping schema, which contains this information.

The type of the closure of a body function applied to a mapping schema and a node is the unit type[3]. A body function carries out side effects by accessing the layers of the platform (term of type Level) by means of references to them. Although these side effects decrease the level of abstraction of our functional approach, they avoid having to pass a whole layer as an argument for each transformation rule in order to improve efficiency. Side effects

[3] This type describes a set which possesses only a single element, which is denoted by (). This means that this function simulates the notion of procedure, just as the type *void* does in the C language.

produced by a body function involve the insertion of new elements into the target schema and new mappings into the mapping schema, both of which are located at the M1-layer. To understand the application of a transformation rule in more detail see [Bor04].

The Operator *generate*

The operator *generate* carries out the evaluation of a set of transformation rules on a source model, which is defined at the M1-layer of the platform. This obtains a new target model and a traceability model between the elements of the source models and the elements of the new generated model. The generated models are also defined at the M1-layer, as shown in Figure 5. The operator *generate* is a function whose declaration is as follows:

```
val generate :
    (bool * unit) list ->
    string -> string -> string
-> bool
```

where the first argument is a list of pairs of functions, in which the first element is a condition function and the second is a body function (i.e. each pair is a transformation rule); the second argument is the name of the source model placed at the M1-layer; the third argument is the name of the schema that contains the *likeness* relationship between the source and the target metamodels (i.e. the schema that provides the *likeness* relation); and the fourth is the name of the new target schema to be generated. This function returns a Boolean value indicating whether or not the model transformation has been performed correctly.

The evaluation process carried out by the operator *generate* is split into three steps: initialization of new schemas at the M1-layer, solution search, and transformation.

First, the operator defines two empty schemas at the M1-layer of the platform:

- Definition of the traceability model.
- Definition of an empty target schema as instance of the target metamodel with the name specified as the fourth argument. The target metamodel is known by means of the model of *likeness* mappings of the M2-layer, which is specified as the third argument.

Second, the operator searches for a solution for the source model transformation. This solution consists of a list of ordered nodes of the source model. The application of transformation rules to the ordered nodes produces the target schema. This step is needed because F# does not provide any mechanism to support the evaluation of the nodes of a schema in an automated and intuitive way. This inconvenience is due to the definition of a schema as a list of nodes, or even as a set. In other languages, this problem is avoided by means of a backtracking mechanism, such as in CLIPS [Cli], or by means of the commutativity property, such as in the algebraic language Maude [Cla02]. A solution is reached when all the nodes of the source model, whose parent participates in a *likeness* relationship in the specified *likeness* relation, have been added to the solution list. In the case that no solution is found, the transformation process is stopped and the operator returns a false value.

```
let rec apply_solution list_solution_concepts
list_transformation_rules sch_m1_source
sch_m1_mappings =
    match list_solution_concepts with
    | [] -> true
    | h::t ->
        let _ = apply_axiom
            list_transformation_rules
                sch_m1_source sch_m1_mappings h
        in
            apply_solution t list_transforamtion_rules
                sch_m1_source sch_m1_mappings
```

Figure 6. F# definition of the apply_axiom function.

Last, the list of transformation rules given as first argument is applied to the nodes of the solution list provided by the second step. The application of transformation rules is reached by means of the *apply_solution* function, which uses the pattern matching mechanism of the F# programming language, as shown in the code in Figure 6. The *apply_solution* function is recursive (indicated by the construct *rec*) and applies the *apply_rule* function to the first node h (head) of the list. This function searches for a suitable transformation rule in the list by means of its respective condition function, and applies the body of the rule to the node h. It inserts a set of nodes into the target schema and inserts the corresponding mappings into the traceability schema. To transform the entire list of nodes of the source model, the *apply_solution* function is applied to the rest of nodes of the solution list t (tail) recursively. When no node is left, the transformation is concluded.

The function *generate* produces side effects due to the application of transformation rules to the elements of the source model. These side effects are changes to the state of the M1-layer, which involve the addition of the generated target model and the traceability model to the M1-layer.

6. CONCLUSION

In this paper, we have presented a solution for transforming models by means of the Visio modeling

environment following a MDD approach. To achieve this, we built a platform that permits the definition of software artifacts following a four-layered approach, which involves metamodels and models, from an algebraic point of view. The platform provides a mechanism to transform models in a declarative way between two metamodels. This mechanism is embodied by the operator generate that receives a list of transformation rules that are applied to a source model in order to translate it into a model of a target metamodel. The application of a transformation provides support for traceability between the source model and the generated one.

The platform has been implemented with the F# programming language. We have also discussed its advantages over other languages that target the .NET platform, such as C#.

The platform has been integrated into the Visio modeling environment by means of an Office managed COM add-in. This allows us to deal with formal models in a visual manner through graphical metaphors. The platform also acts as a repository of formal models. This feature has been used to store the associations between the graphical elements of the Visio interface with the formal definitions stored in the platform, in a UML-based manner.

To our knowledge, this is the first approach to support cross-model semantic interoperability from a modeling environment based on .NET technology.

7. ACKNOWLEDGEMENTS

This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01, and the National Project PBC-03-00 “Methodologies of dynamic user interfaces development”.

8. REFERENCES

[Bec01] Beckett, D. The Design and Implementation of the Redland RDF Application Framework. 10th WWW Conf. May 2-5, 2001, Hong Kong.

[Bor04] Boronat, A., Ramos, I., Carsí, J. Á. Automatic Model Generation in Model Management. Springer-Verlag GMBH. Proceedings of CIT 2004. India, 2004.

[Cah00] Cahilloux, E., Manoury, P., Pagano, B.: Developing Applications With Objective Caml, Éditions O’Reilly, 2000.

[Cla02] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F. Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.

[Cli] Clips documentation. http://www.cis.ksu.edu/ VirtualHelp/Info/clips.html

[Coo04] Cook, S. Domain-Specific Modeling and Model Driven Architecture, MDA Journal, January 2004

[Cza00] Czarnecki, K. and Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications. Addison Wesley, Boston, 2000.

[Cza03] Czarnecki, K., Helsen, S. Classification of Model Transformation Approaches. In Proceedings OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Ehr85] Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Springer-Verlag Berlin Heidelberg New York Tokio (1985). ISBN: 3-540-13718-1.

[Fsh] Fsharp home page. http://research.microsoft. com/ projects/ilx/fsharp.aspx

[Got94] Gotel, O. C. Z., Finkelstein, A. C. W. An Analysis of the Requirements Traceability Problem. In Proceedings of First International Conference on Requirements Engineering (ICRE). Colorado, USA. 1994.

[Rat] Rational Rose XDE Developer. http://www-306.ibm.com/software/awdtools/developer/rosexde/

[Rea93] Reade, C. Elements of Functional Programming. Addison-Wesley, 1993.

[Sel03] Selic, B.: The Pragmatics of Model-Driven Development. *IEEE Software*, ISSN 0740-7459. September 2003, pp. 19-25.

[Sen03] Sendall, S., Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software. September/October 2003 (Vol. 20, No. 5), pp. 42-45.

[Sym01] Syme, D. ILX: Extending the .NET Common IL for Functional Language Interoperability. In Proceedings of Workshop Babel 01, Florence, Italy. September 2001.

[Wid04] Wideman G., *Microsoft Visio 2003 Developer’s Survival Pack*, 2004.

[W3C] W3C, Resource Description Framework (RDF), http://www.w3.org/RDF/

[Yu04] Yu, D., Kennedy, A., Syme, D.. Formalization of Generics for the .NET Common Language Runtime. In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy, January 2004.