Automatic Support for Traceability in a Generic Model Management Framework♦

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. In a MDA process, software artifacts are refined from the problem space (requirements) to the solution space (application). A model refinement involves the application of operators that perform tasks over models such as integrations and transformations, among others. We are working on a model management framework, called MOMENT (MOdel manageMENT), where model operators are defined independently of any metamodel in order to increase their reusability. This approach also increases the level of abstraction of solutions of this kind by working on models as first-class citizens, instead of working on the internal representation of a model at a programming level. In this context, traceability constitutes the mechanism to follow the transformations carried out over a model through several refinement steps. In this paper, we focus on the generic traceability support that the MOMENT framework provides. These capabilities allow the definition of generic complex operators that permit solving specific problems such as change propagation.

Keywords: Model-Driven Architecture, Model Management, traceability, software maintenance.

1. Introduction

Traceability is an important issue in environments where there is a process chain. In these cases, information about each step in the chain may be stored for further processing. For example, in the automotive industry, traceability makes recalls possible; in the food industry, it contributes to food safety; in the Software Engineering field, it provides support for requirements validation and improves the quality of the software development process.

In any scenario of the Software Engineering field, there is a manipulation of a software artifact. The capability of describing and querying the manipulation that has been performed on a specific artifact might be relevant to correlated tasks. However, traceability still remains in the background when software engineering problems are

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

solved. It is often misunderstood and burdensome due to the lack of tools that provide full automatic support for it [1, 2].

In the Model-Driven Architecture initiative [3] (MDA), a software artifact is viewed as a model. Typical tasks, such as code production, integration of applications, interoperability between applications, are performed on models directly. This allows the user to work at a conceptual level, and makes the identification of the elements needed to automate these tasks easier. These tasks are pervasive in many scenarios and are usually solved in an ad-hoc manner.

Following this model-driven approach, a new discipline, called Model Management, was proposed in [4]. This discipline considers models as first-class citizens and provides a set of generic operators to deal with them: *Merge*, *Diff*, *ModelGen*, etc. These operators provide a reusable solution to the tasks described above so that the user deals directly with models, rather than working on the internal representation of a model at a programming level. Several approaches to this discipline [5, 6] specify operators that are based on mappings to deal with models. A mapping is a relationship between an element of a domain model and an element of a range model that indicates that they represent the same element in different models. This means that mappings between two models must be explicitly defined in order to apply an operator to them.

Using our experience in applying the algebraic specification formalism to solve actual software engineering problems [7, 8], we are working on a model management framework called MOMENT (MOdel manageMENT). In our approach, we represent the relationships between two models in an implicit manner by means of an equivalence morphism that is defined between two metamodels. Our approach describes equivalence relationships between two models from a more abstract and reusable point of view. However, explicit mappings between two models are also beneficial when there is no definition of the equivalence morphism between two metamodels. We refer to mappings of this kind as traceability links.

In this paper, we focus on the automatic traceability support that is provided by our framework from a generic point of view. We define what a traceability model is in this setting and how they can be used to provide traceability support in many different scenarios: requirements, workflows, ontologies, etc. We also show how traceability support contributes to automate solutions such as the software maintenance.

The structure of the paper is as follows: Section 2 reviews the traceability management studies that have been performed in the Requirements Engineering field; Section 3 provides an example to illustrate the use of traceability; Section 4 provides an overview of our model management approach; Section 5 details the generic traceability support that is provided in our framework; Section 6 solves the problem of the case study with our model management operators; Section 7 presents some related work; finally, Section 8 summarizes the main features of our approach.

2 The Traceability Problem in Requirements Engineering

In Requirements Engineering, the IEEE Guide to Software Requirement Specifications [9] indicates that a software requirements specification is traceable if

the origin of each requirement is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

Based on this definition, Gotel and Finkelstein [10] described requirements traceability as the ability to describe and follow the life of a requirement, both forwards and backwards, through all the refinement steps in a software development process. Queries of this kind allow the user to know what refinements have been applied to a requirement (in a forward direction), and permit the identification of a requirement from a more specific software artifact such as code (in a backward direction). Therefore, traceability can be used for requirements validation and for providing support for software maintenance. Traceability also provides economic leverage as it details how the system has been developed and avoids the redundant development of certain parts.

To achieve traceability in a software development process, several tasks should be taken into account [2]:

1. Trace definition, to indicate the kinds of objects in our system that can be traced and what information is going to be defined in a trace.
2. Trace production, to indicate what activities, actions, decisions and events happening during software development generate traces for further use.
3. Trace extraction, to indicate how the traces produced in the previous task can be queried in order to achieve certain goals, such as requirements validation or software maintenance.
4. Trace verification, to maintain the integrity of the set of objects and traces.

There are many tools that provide requirements traceability management [11]. To provide efficient traceability management, these tools must resolve certain problems that are present in the industrial setting:

- The lack of a common guideline that describes how to define a traceability model through a well-defined metamodel and how to use it; for example, the way the UML standard provides support to object-oriented modeling. This is due to the variable nature of the traceability capture and use [2], which varies from one organization to another, from one project to another and even from one stakeholder to another.
- The *establish and end-user conflict* [10], where trace providers and users have different goals and priorities.
- The use of heterogeneous tools to define and manipulate the software artifacts involved in a software development process. Thus, the interoperability issue arises as an important feature to be taken into account in a traceability management tool.

These tools are also implemented in an ad-hoc way for the requirements traceability problem without taking into account that the same functionality can be used in other contexts.

3 A Software Maintenance Case Study: Change Propagation

In this case study, we use the change propagate scenario that was introduced in [5]. We illustrate it by means of a specific example shown in Fig. 1. We have defined the information structure of an application in a XML schema (XSD). To build a new

application that stores the information in a relational database, we reuse the metainformation that describes the XML schema. By applying a transformation[1] mechanism (step 1), we obtain the new relational database (RDB). The transformation mechanism also generates a set of links between the new generated RDB relational schema and the source XML schema in order to provide traceability support ($map_{XSD2RDB}$).

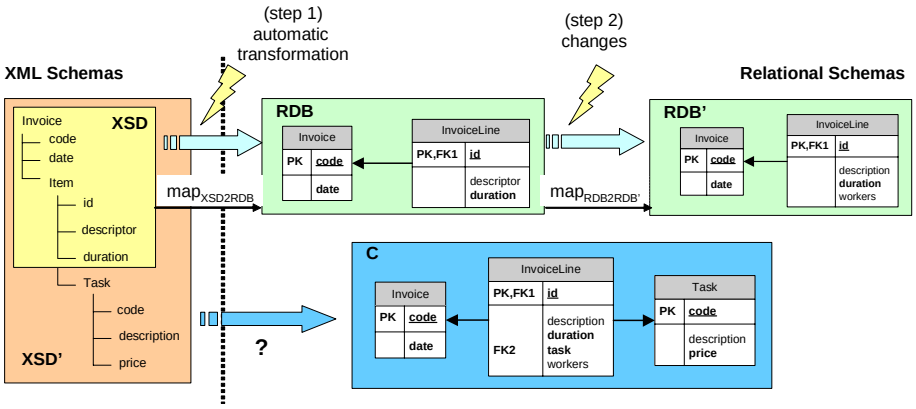

Fig. 1. An example of change propagation.

After obtaining a semantically equivalent relational database from the original XML schema, we continue with the development of the new system. This may involve changes in the application and in the database (step 2), obtaining the relational schema (RDB'). These changes are traced and stored by the tool that manages the model manipulation or by the user directly ($map_{RDB2RDB'}$).

Once the new system is developed, changes may occur in the requirements of the system, requiring modifications. It is easier to extend the XML schema than to modify the RDB database. At this point, the application of the transformation mechanism used in step 1 will discard the changes applied from RDB to RDB'.

A solution to this change propagation example can be performed by using model management operators, as shown in Section 6. In our approach, traceability links are used to automate the propagation of changes that were applied to the RDB relational schema, for the new system C.

4 The MOMENT Framework: a MOdel manageMENT Environment into Eclipse

The MDA initiative of the OMG consists of a family of standards that indicate how to define and how to use models to develop software applications in a MDE setting. Application integration and interoperability are two goals of this initiative, as indicated in the request for proposals for the new standard

[1] We use the term 'model transformation' for the mechanism that translates a model between two different metamodels.

Query/Views/Transformations [12]. Nevertheless, to achieve interoperability between applications, bridges built between them are still ad-hoc.

We are working on the application of the model management trend in the context of MDA. We have developed a framework, called MOMENT (MOdel manageMENT), which is embedded in the Eclipse platform. It provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [13]. EMF provides a close implementation to the MDA guidelines. This framework enables the automatic importation of software artifacts from heterogeneous datasources: UML models (by means of visual modeling environments), relational schemas of any relational database management system (through the Rational Rose tool) and XML schemas. Moreover, third-party researchers and developers are bringing new tools to work on ontologies through EMF [14, 15] and graphical Domain Specific Languages [16, 17]. Therefore, EMF has become an industrial framework for MDA.

4.1 Bridging the EMF and the Maude Technical Spaces

The concept of technical spaces (TS) was introduced by Kurtev et al. in the discussion on the problem of bridging different technologies [18]. A technical space is a working context with a set of concepts, a body of knowledge, tools, required skills, and possibilities [19]. For example, we use the EMF and Maude technical spaces for our framework. The EMF is characterized by its interoperability with industrial tools for solving actual Software Engineering problems. Maude constitutes the formal backbone for our model management approach.

The algebra of operators, which was proposed by Bernstein [20] to deal with models and mappings between models as first-class citizens, has been adapted and directly specified as a generic algebra by using the algebraic specification formalism Maude [21] in the MOMENT framework. This algebraic specification language belongs to the OBJ family, and its equational deduction mechanism animates the specification of an operator over a piece of data, providing the operational semantics for our model management operators. We have developed a plug-in that embeds the Maude environment into the Eclipse framework so that we can use it for our purposes.

In [7, 8], we envisioned the advantages of applying this formalism to solve actual problems in MDA such as model transformation. To fulfill this goal, we have defined two bridges between both technical spaces, at the M2-layer and at the M1-layer (using the Meta-Object Facility [22] terminology). Both of them permit the integration of MOMENT with EMF.

We have defined a projection mechanism at the M2-layer that obtains the algebraic specification[2] that corresponds to a specific metamodel automatically, by applying generative programming techniques. The inverse projection mechanism that obtains an EMF metamodel from an algebraic specification is not interesting in our tool,

[2] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel, they only permit the representation of information for model management issues.

because the algebraic specification must conform to several features in order to be used by our operators, and they should be automatically achieved. We also think that visual modeling environments are more suitable to define such metamodels.

At the model level, we have developed a bidirectional projection mechanism that permits us to project an EMF model as a term of an algebra and to project an EMF model from a term. In this case, the bidirectionality is needed to apply an operator to an input model, since the input model must be serialized as a term and the output term must be deserialized into an EMF model in order to be persisted.

4.2 Operators

In MOMENT, operators are defined in a parameterized module called MOMENT-OP. In this way, operators are defined generically. To apply these operators to specific models, this module must be instantiated by passing a metamodel as actual parameter. This task is automatically performed by the MOMENT tool.

To understand the solution that is given for the change propagation example in Section 6, we informally present some of the model management operators that we use in our approach by indicating their inputs, outputs and semantics:

1. *Cross* and *Merge*
   These operators correspond to well-defined set operations: intersection and disjoint union, respectively. Both operators receive two models (*A* and *B*) as input and produce a third model (*C*). The *Cross* operator returns a model *C* that contains elements that participate in both the *A* and *B* input models; while the *Merge* operator returns a model *C* that contains elements that belong to either the input model *A* or the input model *B*, deleting duplicated elements. Both operators also return two models of links (map_{AC} and map_{BC}) that relate the elements of each input model to the elements of the output model.
   Example: $<C, map_{AC}, map_{BC}> = Cross(A,B)$.
2. *Diff*
   This operator performs the difference between two input models (*A* and *B*). The difference between the two models (*C*) is the set of objects in model *A* that does not correspond to any element in model *B*.
3. *ModelGen*
   ModelGen performs the translation of a model *A*, which conforms to a source metamodel *MMA*, into a target metamodel *MMB*, obtaining model *B*. This transformation implies dealing with two metamodels. This is perfectly feasible in our approach due to the modularity and reusability that algebraic specifications provide. This operator also produces a model of links (map_{AB}) relating the elements of the input model to the elements of the generated model.
   Example: $<B, map_{AB}> = ModelGen_{MMA2MMB}(A)$.

5 Traceability Support in Model Management

All the definitions of Requirements Traceability stated in Section 2 have one feature in common: a trace provides information about a task that has been performed on a

source software artifact in a software development process, and relates it to the resulting software artifact. Traceability support must provide both the mechanism that is needed to define traceability links and the query functionality that permits link navigation. In this section, we define traceability in MDA through a model management lens, and we present a set of operators that provide traceability support in the MOMENT framework.

5.1 Generic Traceability Management

A MDA process consists of a sequence of operations performed over a set of models. These models conform to a metamodel and represent specific software artifacts. Operations such as model integration or model transformation can be directly supported by simple model management operators (Section 4.2). Other operations, such as the change propagation mechanism of the case study, can be specified as a complex operator made up of other operators.

Each simple operator carries out a manipulation over a set of input models. To fulfill this, the operator invokes a function that is defined at the metamodel level. The semantics of this function is defined axiomatically in equational logic, and each one of its axioms is called a manipulation rule. To register the task performed over a model, each operator automatically produces a set of links between the elements of a source model and the elements of the resulting model. Such links are stored as models and are used to provide support for traceability.

Following the model management approach, we define Generic Traceability Management as two main issues:

1. The definition of a traceability metamodel to indicate the information needed to link the elements of two different models that can belong to different metamodels in a specific context. The detail of the metamodel depends on the common understanding of the traceability management in a specific society. For example, a generic traceability metamodel may be described for the Requirements Engineering field, although it seems more feasible to define a traceability metamodel for each organization or even each project.
2. A mechanism to extract information from a traceability model independently of the metamodel used. This mechanism is made up of two kinds of operators:
   - Query operators that provide forward and backward navigation through a traceability model.
   - Traceability management operators to manipulate the traceability models in order to automate the reasoning over traceability links. For instance, the *Compose* operator permits chaining traceability links in order to make implicit traceability links explicit; and the *Match* operator permits the inference of traceability models between two models. Furthermore, a traceability model can also be manipulated by model management operators.

5.1.1 Definition of the Traceability Metamodel

To define the traceability metamodel, the user can use the UML notation. This work is done by the user for a specific working context. For the case study, we have specified a traceability metamodel which basically provides the constructs needed to

relate elements of a domain model to elements of a target model, independently of their metamodels. In Fig. 3, we show the part of the MOMENT framework metamodel that concerns the traceability support.

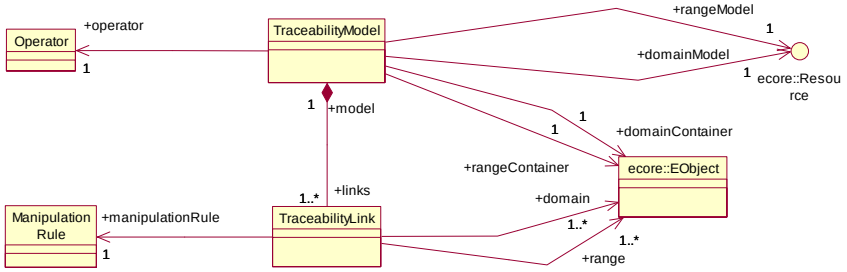

Fig. 2. The default traceability metamodel of the MOMENT framework.

In our specific metamodel, the *TraceabilityModel* class is the root element of the package. It allows us to define traceability models. An instance of the *TraceabilityModel* class contains: information about the storage of both the domain and the range model (represented by the interface *ecore::Resource*); which element of each model is the root (by means of the *domainContainer* and *rangeContainer* roles); the operator that has been applied to the domain model; and the links that constitute the traceability model.

The *TraceabilityLink* class indicates how to define a relationship between a set of elements of the domain model and a set of elements of the range model (by means of the *domain* and *range* roles). Each link is associated to the step of the model manipulation task that has produced it (through the *manipulationRule* role). In the metamodel of the figure, *ecore::Resource* and *ecore::EObject* refer to the interface *org.eclipse.emf.ecore.resource.Resource* and to the class *org.eclipse.emf.ecore.EObject*, respectively. The former permits access to a model stored physically. The latter permits access to any element of an EMF model so that any model defined by means of the EMF can be dealt with. The *Operator* class represents an operator that is defined algebraically in MOMENT. The *ManipulationRule* class defines the information needed to specify an axiom for the manipulation function used by the simple operators.

By applying the projection mechanism defined between the EMF TS and Maude TS, we obtain the algebraic specification of the traceability metamodel. This means that we can specify traceability models as sets of elements so that MOMENT operators can be used to manipulate them (*Merge*, *Cross*, *ModelGen*, …).

5.2 Traceability Operators

Once we have shown the definition of a specific traceability metamodel, we explain the generic traceability operators that are provided by MOMENT. The operators that provide support for traceability are defined generically in a parameterized algebraic specification, called *MOMENT-TRAC(Y :: BASICTMM)*.

Fig. 4 shows the elements involved in the parameter passing mechanism diagram. *BASICTMM* (BASIC Traceability MetaModel) is the algebraic specification of the

formal parameter, called theory in Maude. This theory declares some operators that guarantee the independence between the semantics of the generic traceability operators and the semantics of a specific metamodel. For example, an operator of this kind is *GetDomain*. It obtains the domain element of a traceability link independently of the syntactical representation of the link. Thus, the formal parameter behaves as an interface through which the generic operators can access the elements of a model that conforms to a specific traceability metamodel.

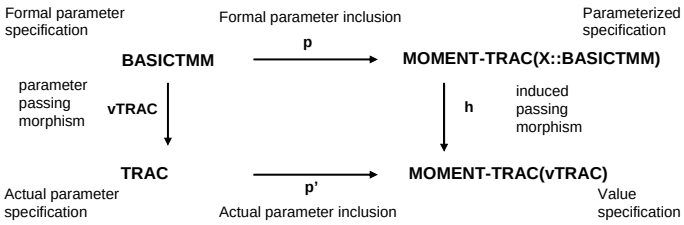

Fig. 3. The parameter passing diagram for the *MOMENT-TRAC(Y :: BASICTMM)* parameterized module.

TRAC is the algebraic specification obtained by the projection mechanism from a specific traceability metamodel. The *TRAC* specification constitutes the actual parameter for the *MOMENT-TRAC(Y :: BASICTMM)* module and defines the semantics of the operators that are only declared in the *BASICTMM* theory. The *vTRAC* view is the morphism that relates the elements of the *BASICTMM* formal parameter to the elements of the *TRAC* actual parameter.

The *MOMENT-TRAC(Y::BASICTMM)* parameterized algebraic specification contains the definition of the traceability operators that are independent of the specific TRAC traceability metamodel. The *MOMENT-TRAC(vTRAC)* value specification results from the instantiation[3] of the parameterized module with the specific *TRAC* traceability metamodel.

In this figure, p and p' are inclusion morphisms that indicate that the formal parameter specification is included in the parameterized specification, and that the actual parameter specification is included in the value specification, respectively. The h morphism is the induced passing morphism that relates the elements of the parameterized module to the elements of the *MOMENT(vTRAC)* value specification, by using the *vTRAC* parameter passing morphism.

The traceability operators defined in the *MOMENT-TRAC(X::BASICTMM)* parameterized module are classified in two groups: operators that provide support for navigability and operators that perform tasks on traceability models. In this paper, we focus on the first group of operators.

We define the operators that provide navigability through a traceability model with the following elements: two input models (A and B); a traceability model (map_{AB}) that relates the elements of the two input models and that has been automatically produced by an operator or manually produced by a user; a model (A') that is a sub-

[3] In the context of algebraic specifications, the instantiation of a parameterized module refers to the fact of passing an actual parameter to the parameterized module, obtaining the final value specification.

model of A (i.e. A' only contains elements that also belong to A); and a model (B') that is a sub-model of B. The traceability operators that are considered here are:

1. *Domain* and *Range*

   These operators provide the backward and forward navigation through a traceability model, respectively. Both operators obtain a model as output, which is not a traceability model.

   The operator *Domain* takes three models as input: a traceability model (map_{AB}), a domain model (*A*), and a range model (*B'*). The operator navigates the traceability links of the traceability model that have elements of *B'* as target elements, and returns a sub-model of *A* (*A'*), as shown in Fig. 5.a.

   The operator *Range* also receives three inputs: a traceability model (map_{AB}), a domain model (*A'*), and a range model (*B*). This operator performs the opposite task to the previous one: it navigates the traceability links that have elements of *A'* as domain elements and returns a sub-model of the range model *B* (*B'*), as shown in Fig. 5.b.

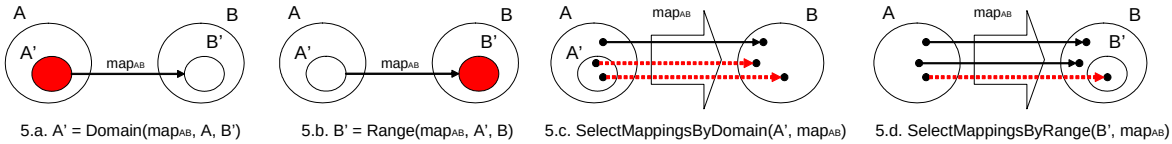

Fig. 4. Generic operators for traceability navigation.

2. *SelectMappingsByDomain* and *SelectMappingsByRange*

   These operators produce a traceability model as output and permit selection of parts of a traceability model.

   The operator *SelectMappingsByDomain* receives two input models: a domain model (*A'*) and a traceability model (map_{AB}). The operator extracts the traceability links of the map_{AB} traceability model that have elements of the *A'* model as domain elements and returns this sub-model. The traceability links that are added to the output traceability model are highlighted by a dotted line in Fig. 5.c.

   The operator *SelectMappingsByRange* receives two input models: a range model (*B'*) and a traceability model (map_{AB}). In this case, the operator extracts the traceability links of the map_{AB} traceability model that have elements of the *B'* model as range elements, and returns this sub-model, as shown in Fig. 5.d.

5.3 Process

Taking into account the process described in [2] to define a traceability model, we indicate how its tasks are performed in our tool:

1. Trace definition. Users can define their own metamodel to fit into a specific working context. Moreover the default traceability metamodel of the MOMENT framework can be used in its place. As seen above, the traceability metamodel can be defined using well-known graphical notations, such as UML, through EMF-compliant tools.
2. Trace production. By default, the traceability model is automatically generated by an operator when it performs a manipulation over a set of input models.

Nevertheless, traceability links can be defined manually by means of an editor generated from the traceability metamodel, following the EMF software development culture. Moreover, a traceability model can be inferred automatically between two models by using heuristics [23] or historical knowledge [24].

3. Trace extraction. The analysis of the knowledge provided by a set of traceability models can be useful to perform other tasks. The traceability operators are used to deal with this information in our framework. Such operators constitute an automatic reusable solution that provides support for traceability in many scenarios in the MDE field. Therefore, our framework provides automatic support for this step although the user has to reason about the extracted knowledge. In future works, heuristics may be applied in this step to achieve richer information.
4. Trace verification. The consistency of traceability models can be kept automatically when either their domain or their range model is modified, by means of the application of traceability operators. Consider that we have a domain model *A*, a range model *B* and a traceability model map_{AB} that has been defined between them. Three kinds of model modifications are available: addition of elements, modification of existing elements and deletion of elements. In the case of adding elements to a model, the traceability model remain consistent on the grounds that there is no connection between the new elements and the elements of the other model, unless this connection is defined explicitly afterwards. In the case of modifying and deleting elements of a model, links can be broken when the domain or the range elements are deleted. This problem is easily solved by using traceability operators. If we delete elements in model *A*, obtaining model *A'*, we can apply the *SelectMappingsByDomain* operator to obtain the new consistent traceability model map_{AB}': $map_{AB}' = SelectMappingsByDomain(A', map_{AB})$.

6 Application to the Case Study

The problem explained in the case study can be simplified as shown in Fig. 6, where the $map_{XSD2RDB'}$ traceability model can be easily obtained from the $map_{XSD2RDB}$ and $map_{RDB2RDB'}$ traceability models by means of the *Compose* operator. Therefore, the problem can be enunciated as follows:

We have the following models: an original XML schema (XSD); a XML schema (XSD'), which has been evolved from XSD; a relational database RDB', which has been generated from the XML schema XSD and modified afterwards; and a traceability model between XSD and RDB' ($map_{XSD2RDB'}$). The goal is to obtain a relational database from the XML schema XSD' that preserves the changes applied to RDB'.

This problem can be solved by the following complex operator:

operator PropagateChanges(XSD, XSD', RDB', $map_{XSD2RDB'}$) =
<Unmodified, $map_{XSD2Unmodified}$, $map_{XSD'2Unmodified}$> = Cross(XSD, XSD') (1)
RDB'' = Range($map_{XSD2RDB'}$, Unmodified, RDB') (2)
<newXSD> = Diff(XSD', Unmodified) (3)
<newRDB, $map_{newXSD2newRDB}$> = $ModelGen_{XSD2RDB}$(newXSD) (4)
<C, $map_{RDB''2C}$, $map_{newRDB2C}$> = Merge(RDB'', newRDB) (5)
return (C)

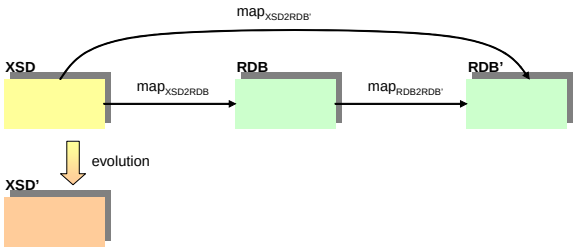

Fig. 5. Schematization of the case study problem.

This operator is made up of simple operators of the MOMENT algebra and the steps followed in the script are represented in Fig. 7. These steps are the following:

1. *Unmodified* is the part of the *XSD* model that remains unmodified in the *XSD'* model.
2. *RDB''* is the sub-model of *RDB'* that corresponds to the unmodified part of XSD'.
3. *newXSD* is the part of *XSD'* that has been added to the XSD model.
4. *newRDB* is the relational schema obtained from the translation of *newXSD* into the relational metamodel.
5. C is the final model obtained from the integration of the relational databases that we have obtained in steps 2 and 4.

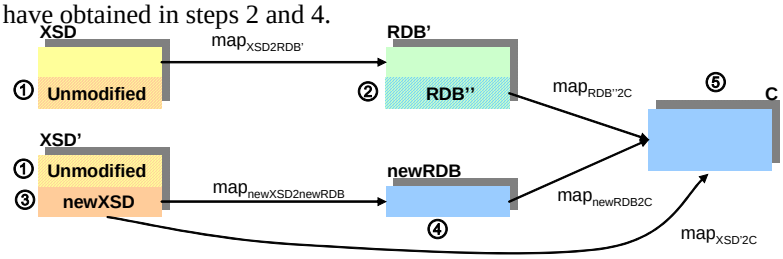

Fig. 6. Solution to the case study problem.

If we want to add traceability support to this operator to generate the traceability model that relates the XSD' model to the new model (C) as well, we only have to add the next step after step 5[4]:

<$map_{XSD'2C}$, $map_{mapUnmodified2C2mapXSD'2C}$, $map_{mapnewXSD2C2mapXSD'2C}$> = Merge(
Compose(Unmodified, SelectMappingsByDomain(Unmodified, $map_{XSD2RDB'}$),RDB'', $map_{RDB''2C}$, C),
Compose(newXSD, $map_{newXSD2newRDB}$, newRDB, $map_{newRDB2C}$, C))

[4] In this step, the operator Compose (*<map_{AC}> = Compose(A, map_{AB}, B, map_{BC}, C)*) has five input parameters: three models (*A*, *B*, *C*), and two traceability models, which are defined

This step merges two traceability models: one is defined between the unmodified part of XSD' and C, and the other is defined between the new part of XSD' and C. This step merges both traceability models by means of the *Merge* operator, in the same way that any two models that belong to the same metamodel are merged. The model $map_{XSD'2C}$ has to be added as a return value in the script.

The resulting complex operator solves the change propagation problem of the case study independently of the metamodels involved so that we can apply it to any combination of metamodels, instead of using the XSD and the relational metamodels.

7 Related work

In the Model Management field, tools do not deal with traceability directly. They usually work on mapping models, which define equivalence relationships between the elements of two models so that a model management operator can be defined generically. Rondo [5] and [6] are good examples of this approach. For instance, Rondo's Merge operator permits the integration of two models. It receives two models (*A* and *B*) and a mapping model (map_{AB}) between them as inputs, and it produces the merged model *C* and two new mapping models (map_{AC} and map_{BC}): $<C, map_{AC}, map_{BC}> = Merge\ (A, B, mapAB)$.

In MOMENT, mapping models are introduced as traceability models. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, the traceability relationships between the elements of two models, which are needed to apply an operator to them, are defined between the elements of their corresponding metamodels axiomatically within the corresponding operators. The collection of equivalence relationships between two metamodels constitutes a morphism that can be reused for all the operators of the MOMENT algebra. This permits a clearer specification of complex operators. In MOMENT, the Merge operator is as follows: $<C, map_{AC}, map_{BC}> = Merge(A, B)$. Mapping models are produced by the application of a simple operator to a set of models and keep information about the manipulation task that has been performed to a model. Therefore, we deal with these mapping models from a traceability standpoint.

The Generic Model Weaver AMW [25] is a tool that permits the definition of mapping models (called *weaving models*) between EMF models in the ATLAS Model Management Architecture. AMW provide a basic weaving metamodel that can be extended to permit the definition of complex mappings. These mappings are usually defined by the user, although they may be inferred by means of heuristics, as in [23]. In MOMENT, such mappings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators. We also support extension of the traceability metamodel. Although the simplicity of our initial traceability metamodel, it allows us to deal with complex operators satisfactorily.

---

between models *A* and *B* (map_{AB}) and between models *B* and *C* (map_{BC}). It concatenates both traceability models obtaining a new one that directly relates A to C.

8 Conclusions

In this paper, we have presented an overview of our model management approach, focusing on the automatic support that the MOMENT framework provides for traceability. To do this, we have based our approach on the traceability management studies that have been made in the Requirements Engineering field.

We have introduced the Generic Traceability Management concept in the MDA initiative through a Model Management lens. We have also discussed some operators and illustrated how they can be used to solve common software engineering scenarios, like the software maintenance case study presented here. The traceability support has been defined in the MOMENT framework generically so that it can be applied to any context (requirements, workflows, ontologies,…). It can also be customized with a specific traceability metamodel depending on the needs of each working context.

We provide a new vision of the traceability support with respect to previous model management approaches [5, 6], where all operators are based on mappings. In these approaches, the equivalence relationships between elements of two models are defined with specific explicit mappings at the model level. Such mappings are defined by the user or can be inferred by means of heuristics or historical knowledge. However the obtained mappings should be reviewed by a user, which can become burdensome when huge models are involved.

In MOMENT, such equivalence relationships are defined as morphisms between metamodels because algebraic specifications are used as the background formalism. The specification of an equivalence morphism at metamodel level contributes to a more abstract and reusable solution for model management. Users of our model management approach do not have to deal with algebraic specifications directly. The use of EMF to algebraically define models dramatically reduces the learning curve for dealing with models from a formal generic standpoint. It also increases the interoperability with many industrial software development tools.

As we are implementing the MOMENT tool as an Eclipse plugin, AMW constitutes a good environment to define our traceability models by the user. Nevertheless, we are developing our own editor for traceability models in order to add the chance of invoking traceability operators directly from the editor interface. In this way, we will be able to automatically compose traceability models or to navigate them from visual interfaces, using the underlying algebraic specification formalism.

The next step in the MOMENT framework development process is to provide support for the QVT Relations language in order to use it for the definition of equivalence relationships and transformations. The work presented in this paper constitutes the traceability support that MOMENT will provide for the QVT standard.

References

1. Palmer, J.D.: Traceability. Richard H. Thayer and Merlin Dorfman, Software Requirements Engineering, 2nd Edition, IEEE Computer Society, pages 412–422, 2000.
2. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. Software Engineering, 27(1):58–93, 2001.

3. OMG Model-Driven Architecture. http://www.omg.org/mda/
4. Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. Microsoft Research Technical Report MSR-TR-2000-53, June 2000, (short version in SIGMOD Record 29, 4 (Dec. '00)).
5. Melnik, S., E. Rahm, P. A. Bernstein, "Rondo: A Programming Platform for Generic Model Management," Proc. SIGMOD 2003, pp. 193-204 (PDF, 344KB). Extended version in Web Semantics, Volume 1, Number 1.
6. Song, G., Zhang, K., Kong, J.: Model Management Through Graph Transformation. IEEE VL/HCC'04. Rome, Italy. 2004.
7. Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.
8. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, UK. 2005.
9. M. Dorfman and R. Thayer. Guide to software requirements specification. IEEE Standards, Guidelines and Examples on System and Software Requirements Engineering, 1990.
10. Gotel, O. C. Z. and Finkelstein, A. C. W.: An Analysis of the Requirements Traceability Problem, Proceedings of the First International Conference on Requirements Engineering (ICRE '94), IEEE Computer Society Press, Colorado Springs, Colorado, U.S.A., April 18-22, pp. 94-101.
11. Requirements tools in the Volere web site: http://www.volere.co.uk/tools.htm
12. Object Management Group. Request for Proposal: MOF 2.0 Query / Views /Transformations RFP, 2002. ad/2002-04-10.
13. The EMF site: http://download.eclipse.org/tools/emf/scripts/home.php
14. Zhang, L., Yu, Y., Lu, J., Lin, C., Tu, K., Guo, M., Zhang, Z., Xie, G., Su, Z., Pan, Y.: ORIENT: Integrate Ontology Engineering into Industry Tooling Environment. In Proc. of the Third International Semantic Web Conference 2004, Hiroshima, Japan.
15. The Hyena web site: http://www.pst.ifi.lmu.de/~rauschma/hyena/
16. The Graphical Modeling Framework proposal: http://www.eclipse.org/proposals/eclipse-gmf/main.html
17. The JANE Model-specific editor generator web site: http://www.dstc.edu.au/Research/Projects/Pegamento/jane/
18. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
19. Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In Proceedings of INTEROP-ESA'05, Geneve, Switzerland. 2005.
20. Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.
21. 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.
22. OMG, "Meta Object Facility 1.4", http://www.omg.org/technology/documents/formal/mof.htm
23. Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. VLDB 2001.
24. Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P.: Corpus-based Schema Matching," Workshop on Information Integration on the Web, at IJCAI'2003, pp. 59-66.
25. Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, and Gueltas, G : AMW: a generic model weaver. Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). 2005.