Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the
Third Workshop on Software Evolution
through Transformations:
Embracing the Change
(SeTra 2006)

Exogenous Model Merging by means of Model Management Operators

Artur Boronat, José Á. Carsí and Isidro Ramos

19 pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Exogenous Model Merging by means of Model Management Operators

Artur Boronat[1], **José Á. Carsí** [2] **and Isidro Ramos**[3]

[1] aboronat@dsic.upv.es, [2] pcarsi@dsic.upv.es, [3] iramos@dsic.upv.es
http://issi.dsic.upv.es
Information Systems and Computation Department
Technical University of Valencia, Spain

Abstract: In Model-Driven Engineering, model merging plays a relevant role in the maintenance and evolution of model-based software. Depending on the amount of metamodels involved in a model merging process, we can classify model merging techniques in two categories: endogenous merging, when all the models to be merged conform to the same metamodel; and exogenous merging, when the models to be merged conform to different metamodels. MOMENT (MOdel manageMENT) is a framework that is integrated in the Eclipse platform, and provides a collection of generic set-oriented operators to manipulate MOF models, following the Model Management discipline. In this paper, we study how model transformations are useful in a model merging process and we provide a solution for both kinds of model merging by means of model management operators and the QVT Relations language.

Keywords: Model-Driven Architecture, Model Management, Exogenous Model Merging, QVT Relations

1 Introduction

Software merging is an essential aspect of the maintenance and evolution of large-scale information systems. Information systems can be specified by means of models in Model-Driven Engineering. Models collect the information that describes the information system at a high level of abstraction, which permits the development of the application in an automated way using generative programming techniques. The consolidation of the Meta-Object Facility standard [OMG04] as a four-layer architecture, where metamodels can be specified as a set of syntactical well-formedness rules to define models, permits the definition of modeling domains where merging processes can be performed. A model merging process can be defined over a metamodel. Then, any two well-formed models in this metamodel can be merged. Traditionally, the tasks that are involved in this process have usually been solved in an ad-hoc manner for a specific context or metamodel: relational databases [BLN86, BDK92], XML schemas [Beh00], OWL-DL ontologies [HM05] , aspect-oriented modeling [SGS$^+$04], UML models [OWK03], etc.

[Men02] presents a classification of merge approaches, where domain independence and customizability of a generic merge operator to a specific domain are desired features. However, the definition of metamodels by means of a common metamodeling language (like MOF, or any MOF-like implementation) is a desired feature that should be preserved on the grounds that it permits the development of generic infrastructures to manipulate models.

Following this direction, Model Management [BHP00] is a new emergent discipline that pursues an abstract reusable solution for problems of this kind, independently of the metamodel under study. The Model Management discipline deals with software artifacts by means of generic operators that do not depend on their internal implementation because they work on mappings between models [Ber03]. These operators treat models as first-class citizens and increase the level of abstraction of the solution avoiding programming tasks and improving the reusability of the solution.

As stated in [BLN86], a model merging process consists of three main phases: a model comparison phase, where elements of different models that are equivalent are found; a consistency checking phase, where conflicts that may appear if we merge equivalent elements are identified, defining a conflict resolution strategy to eliminate them; and a merging phase, where the equivalent elements that are found in the first step are merged taking into account the conflict strategy defined in the second step.

Generic model merging approaches provide support for these three phases in different ways. [AP03] uses MOF identifiers to compare elements in different versions of a same base model. [BP03, BCE$^+$06] provide a set of model management operators to define equivalence relationships between elements of different models by means mappings, which are used by a merge operator later on. [KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language permits the definition of equivalence relationships between elements of a metamodel that can be applied over elements of the corresponding models afterwards. The model merging language embeds the comparison language so that these equivalence relationships can be used in the merging process.

In our approach, we propose a set of model management operators that use the QVT Relations language [OMG05] to perform model comparison and model transformation. In a model merging process where two models are involved, the comparison phase is achieved by defining relations between elements of the same metamodel. The consistency phase is solved by defining a model transformation that takes the two models to be merged as input models. Finally, the merging phase is performed by a generic operator that uses the QVT Relations programs defined in the previous phases. Thus, we enhance the use of the QVT Relations language within the Model Management field, avoiding the definition of a new DSL for every model management operator. In this paper, we show how this approach can be used by providing an example of exogenous model merging, where the models to be merged conform[1] to different metamodels.

The structure of the paper is as follows: Section 2 presents the exogenous model merging problem; Section 3 introduces the *ModelGen* operator for model transformation; Section 4 introduces the *Merge* operator for model merging; Section 5 provides the solution for the example in Section 2; Section 6 provides some related works. Finally, Section 7 summarizes the main contributions of the paper.

[1] A model conforms to a metamodel if it is syntactically well-formed by using the constructs of the metamodel.

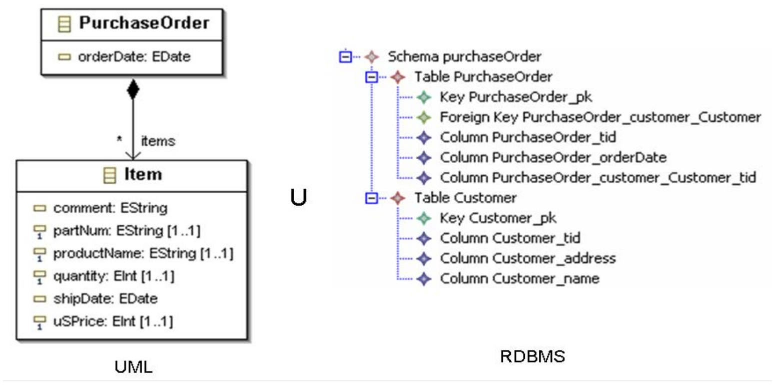


Figure 1: Exogenous model merging of a UML model and a relational schema.

2 Exogenous Model Merging Scenario

When two models are merged, an equivalence relation must be defined between their corresponding metamodels, associating their elements using a set of relationships. These relationships are used to identify equivalent elements in different models in order to avoid duplicated information in the merged model.

Generic approaches to merge models use this concept of equivalence relation, but they do not usually differentiate between an endogenous and an exogenous model merging. In Fig. 1, we provide an example of exogenous model merging: the integration of a UML model and a relational schema. We have used the Ecore metamodel [BBM03] as an implementation for the UML class diagram metamodel, and the relational metamodel that appears in the Query/View/Transformation (QVT) standard specification [OMG05]. In Fig. 1, the relational schema is shown in a tree-like form.

In this paper, we use this example to show that an exogenous model merging process is a generalization of an endogenous model merging process. Therefore, it can be broken down into simpler processes, which can be solved by means of model management operators. Our approach for solving the example consists of two steps: a model transformation that permits representing the UML model as a relational schema; and a model merging between relational schemas. We present how we deal with model transformation and endogenous model merging in the following sections.

3 The QVT Relations Language and the *ModelGen* Operator

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model elements of different candidate models. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If the target domain is defined in the QVT transformation as *enforce*, a transformation is performed by creating the corresponding elements in the target model. If the target domain is defined as *checkonly*, just a checking is performed without creating any new element in the target model. Both kinds of transformations are used in our approach.

A relation can be also constrained by two sets of predicates, a *when* clause and a *where* clause. The *when* clause specifies the conditions under which the relationship needs to hold. The *where* clause specifies the condition that must be satisfied by all model elements participating in the relation.

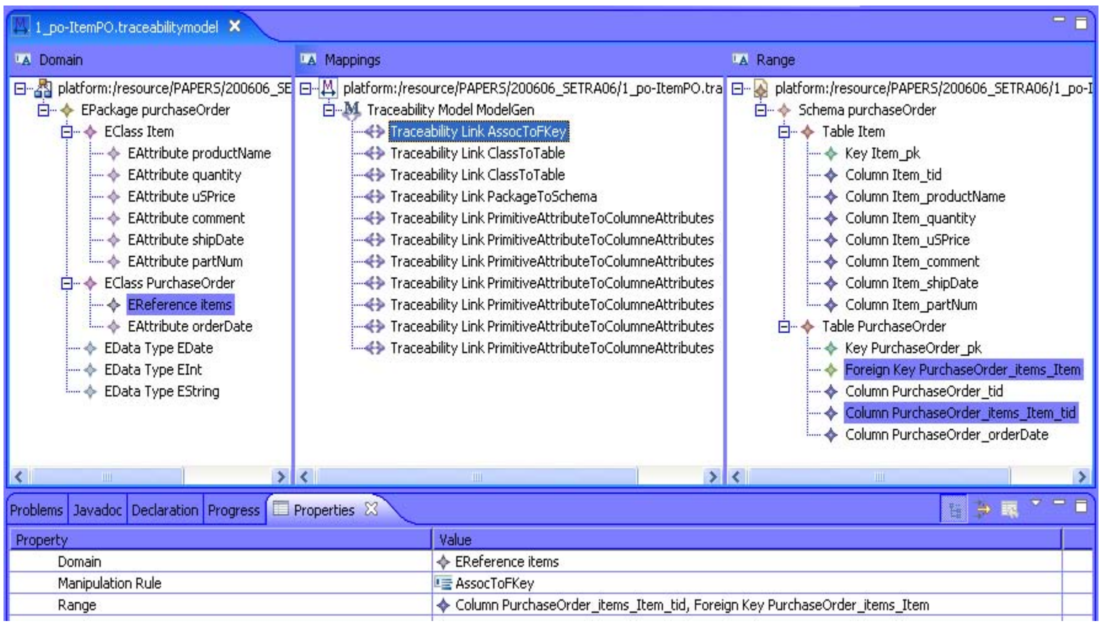
A transformation contains two kinds of relations: top-level (marked with the *top* keyword) and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

As example, we have taken the *UmlToRdbms* transformation that is presented in the MOF QVT final specification[2]. The top relation below specifies the transformation of a *Class* into a *Table*. By means of the *where* clause, the relation *ClassToTable* needs to hold only when the *PackageToSchema* relation holds between the package containing the class and the schema containing the table. By means of the *when* clause, the *ClassToTable* relation holds, the relation *AttributeToColumn* must also hold.

```
top relation ClassToTable {
    className: String;
    checkonly domain ecoreDomain c: EClass {
        ePackage = p:EPackage {},
        name=className
    };
    enforce domain rdbmsDomain t: Table {
        schema = s:Schema {},
        name = className,
        column = cl:Column {
            name = className + '_tid',
            type = 'NUMBER'
        },
        key = k:Key {
            name = className + '_pk',
            column=cl
        }
    };
    when {
        PackageToSchema(p, s);
    }
    where {
        AttributeToColumn(c, t, className);
    }
}
```

In MOMENT, a model transformation can be applied to several source models, which may or may not conform to the same metamodel. When the transformation is invoked, it generates one target model and a set of traceability models. A traceability model contains a set of traces

[2] In this paper, we are using a version of this transformation in which we consider Ecore as an implementation of the UML Class Diagram metamodel. The version of the transformation that is used is presented in Appendix B.



1_po-ItemPO.traceabilitymodel

Domain

```
platform:/resource/PAPERS/200606_SE
  EPackage purchaseOrder
    EClass Item
      EAttribute productName
      EAttribute quantity
      EAttribute uSPrice
      EAttribute comment
      EAttribute shipDate
      EAttribute partNum
    EClass PurchaseOrder
      EReference items
      EAttribute orderDate
    EData Type EDate
    EData Type EInt
    EData Type EString
```

Mappings

```
platform:/resource/PAPERS/200606_SETRA06/1_po-ItemPO.tra
  Traceability Model ModelGen
    Traceability Link AssocToFKey
    Traceability Link ClassToTable
    Traceability Link ClassToTable
    Traceability Link PackageToSchema
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
    Traceability Link PrimitiveAttributeToColumneAttributes
```

Range

```
platform:/resource/PAPERS/200606_SETRA06/1_po-I
  Schema purchaseOrder
    Table Item
      Key Item_pk
      Column Item_tid
      Column Item_productName
      Column Item_quantity
      Column Item_uSPrice
      Column Item_comment
      Column Item_shipDate
      Column Item_partNum
    Table PurchaseOrder
      Key PurchaseOrder_pk
      Foreign Key PurchaseOrder_items_Item
      Column PurchaseOrder_tid
      Column PurchaseOrder_items_Item_tid
      Column PurchaseOrder_orderDate
```

Problems | Javadoc | Declaration | Progress | Properties

Property	Value
Domain	EReference items
Manipulation Rule	AssocToFKey
Range	Column PurchaseOrder_items_Item_tid, Foreign Key PurchaseOrder_items_Item

Figure 2: Traceability Editor in the MOMENT Framework.

that relate the elements of the source model to the elements of the target model, indicating which transformation rule has been applied to each source element. A QVT Relations enforced transformation is executed by means of the *ModelGen* operator as follows:

$$< output\_model,\ trac_1, \ldots,\ trac_n >= ModelGen(transformation,\ input\_model_1, \ldots,\ input\_model_n)$$

where $transformation$ is the name of the QVT transformation; $input\_model_1, \ldots,\ input\_model_n$ are the input models, which may conform to different metamodels; $output\_model$ is the generated model; and $trac_1, \ldots,\ trac_n$ are the trace models that are generated for each one of the corresponding input models.

Fig. 2 presents the traceability editor of the MOMENT framework. This editor shows the trace model that is generated by the *UmlToRdbms* transformation, when it is applied to the UML model that is defined in Fig. 1. This transformation constitutes the first step of the exogenous model merging process. Trace models in our framework conform to our traceability metamodel, which was presented in [BCR05]. The traceability editor is constituted by three main frames, the left frame shows an input model of the transformation, the right frame shows the output generated model and the frame in the middle shows the traces that relate elements of the input model to elements of the target model. Traces also provide information about the transformation rule (or relation) that has been applied to source elements to generate the corresponding trace and the related target elements.

4 The *Merge* Operator

The *Merge* operator takes two models as input and produces a third one. If A and B are models that conform to the same metamodel, the application of the *Merge* operator on them produces a model C, which consists of the members of A together with the members of B, i.e. the union of A and B. Taking into account that duplicates are not allowed in a model, the union is disjoint.

To understand the semantics of the *Merge* operator in our example, we need to introduce two concepts: the equivalence relation, for finding duplicates by comparing models, and the conflict resolution strategy, for integrating them.

4.1 The equivalence relation

In an endogenous model merging, an equivalence relation is defined between elements that belong to different models that conform to the same metamodel. To define an equivalence relation among the elements of a model in our approach, the user can use the QVT Relation language in the checkonly mode. Only checkonly transformations with two domains are accepted in this context. Both domains have to refer to the same metamodel in our approach. For the example, we customize the *Merge* operator to merge relational schemas, i.e., models that conform to the *RDBMS* metamodel of Appendix A. To do so we use a checkonly QVT Transformation whose domains refer to the *RDBMS* metamodel. The user can add a QVT relation for each of the classes that appear in the metamodel when it is desired. Such QVT relations act as equivalence relationships that must hold over the elements of two *RDBMS* models. These QVT relations are used in the merging process to check when two elements are equivalent in order to eliminate duplicates.

For instance, the following relation can be defined to indicate that two tables are the same if they belong to the same schema and they have the same name by means of the *tableName* variable[3]:

```
top relation TableEquivalence {
        tableName: String;
        checkonly domain rdbmsDomain1 t1: Table {
                schema = s1:Schema {},
                name=tableName
        };
        checkonly domain rdbmsDomain2 t2: Table {
                schema = s2:Schema {},
                name=tableName
        };
        when {
                SchemaEquivalence(s1, s2);
        }
}
```

where the *SchemaEquivalence* is another QVT Relation defined within the same transformation, describing when two Schema instances are equivalent (for instance, by name). In our approach, this kind of equivalences may involve several instances of two models as in the above example,

[3] We have chosen these criteria for the example. Nevertheless, they can be customized to a specific metamodel by the user. Nothing impedes us to add semantic annotations to the elements of a model and use this information to determine which elements are equals or not.

where *Table* instances and *Schema* instances are used to check whether two tables are equivalent or not.

During the merging process, this checkonly transformation permits checking when groups of elements of different models represent duplicate elements so that they will be merged. In a checkonly QVT transformation, helper functions can be defined by using OCL expressions to manipulate and compare names, and to navigate the structure of the corresponding model. Thus, the user only has to be aware of the standard QVT Relations language and the domain-specific knowledge.

4.2 The conflict resolution strategy

During a model merging process, when two software artifacts (each of which belongs to a different model) are supposed to be equivalent, one of them must be erased. Their syntactical differences may cast doubt on which should be the syntactical structure for the merged element. Here, the conflict resolution strategy comes into play. The conflict resolution strategy is a model transformation that has two input models and one output model, the merged one. The generic semantics of this strategy in our framework consists of the preferred model strategy. When the *Merge* operator is applied to two models, one has to be chosen as preferred (the first argument of the *Merge* operator). In this way, when two groups of elements (that belong to different models) are equivalent due to an equivalence relation, the elements of the preferred model prevail although they may differ syntactically.

To refine the *Merge* operator, the conflict resolution strategy can also be customized. During the merging process, when the *Merge* operator finds two duplicates, they should be integrated. This integration involves a transformation process where information of both duplicates may be taken into account to define the merged model. Thus, an enforced QVT transformation can be used to customize the conflict resolution strategy in the same way a checkonly QVT transformation is used to customize the generic equivalence relation.

A QVT transformation that is used to define a specific conflict resolution strategy has three domains. All of them refer to the metamodel under study (*RDBMS* in our example). The first two domains are defined as checkonly and they only query the two input models of the *Merge* operator. The third domain is defined as enforce and is the one that produces merged elements. In the case study, when we integrate two tables that are equivalent (because they have the same name), we have to integrate their respective columns, primary keys and foreign keys. The following QVT Relation is intended to perform this task:

```
top relation TableMerging {
    tableName: String;
    checkonly domain rdbmsDomain1 t1: Table {
        schema = s1:Schema {},
        name = tableName
    };
    checkonly domain rdbmsDomain2 t2: Table {
        schema = s2:Schema {},
        name = tableName
    };
    enforce domain rdbmsDomain3 t3: Table {
        schema = s3:Schema {},
        name = tableName
    };
    when {
        SchemaMerging(s1, s2, s3);
    }
    where {
        ColumnMerging(t1, t2, t3);
        PKMerging(t1, t2, t3);
        FKMerging(t1, t2, t3);
    }
}
```

where the *SchemaMerging* QVT relation, which is invoked in the *when* clause, ensures that the container schemas of both *Table* instances must be equivalent in order to apply the current relation to the involved tables. The QVT Relations that are invoked in the *where* clause ensure that the merging process will go on by merging columns, primary keys and foreign keys of the involved tables.

The enforce QVT transformation that the user defines to customize the conflict resolution strategy is automatically compiled into a *ModelGen* equation as briefly introduced in the previous section [4].

4.3 The *Merge* operator

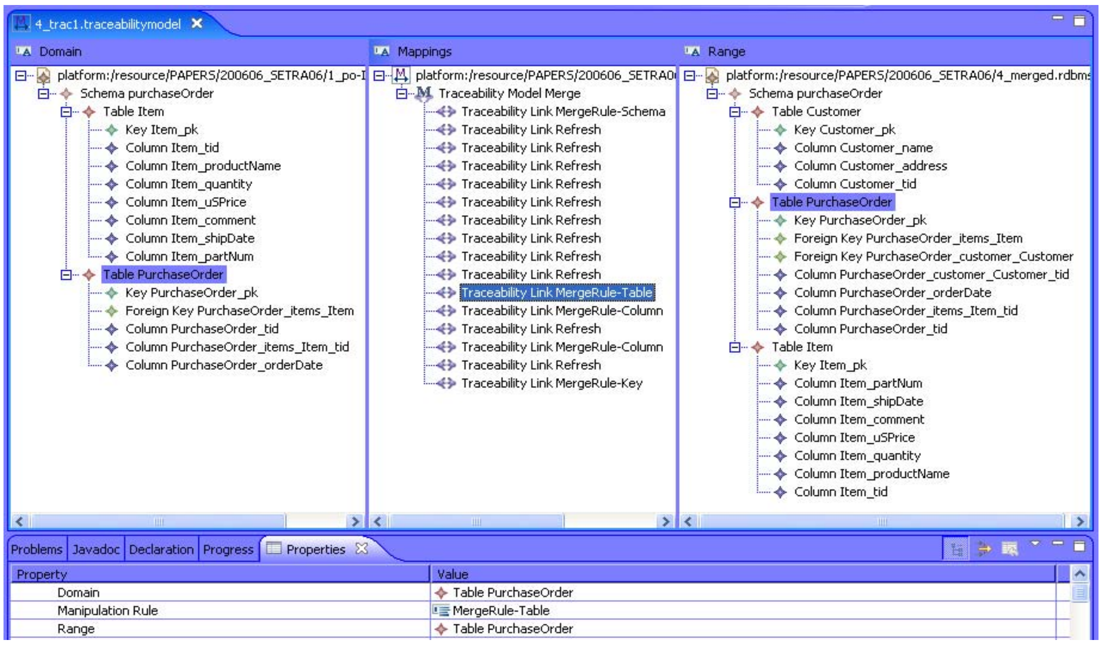
The *Merge* operator takes two models that conform to the same metamodel as inputs. The outputs of the *Merge* operator are a merged model (*merged_model*) and two models of traces ($trac_1$ and $trac_2$) that relate the elements of each input model (*model*1 and *model*2) to the elements of the output merged model. The operator is used as follows:

$$< merged\_model, trac_1, trac_2 >= Merge(model1, model2)$$

The *Merge* operator uses the equivalence relation that is defined for a metamodel to detect duplicated elements between the two input models. When two duplicated elements are found, the conflict resolution strategy is applied to them in order to obtain merged elements, which are then added to the output model. The elements that belong to only one model, without being duplicated in the other one, are copied into the merged model.

The two output trace models are automatically generated by the *Merge* operator on the grounds that it reuses the model transformation mechanism that is described in Section 3, through the conflict resolution strategy. These trace models provide full support for keeping traceability between the input models and the new merged one. The second step of the exogenous model merging in the example constitutes a merging process that involves the model *RDBMS'* and the model *RDBMS*. The model *RDBMS'* is the result of applying the *UmlToRdbms* transformation (defined in Appendix B) to the model *UML* that is defined in Fig. 1, as explained in Section 3. The model *RDBMS* is provided in Fig. 1. In Fig. 3, we show the trace model that is generated during this merging process for the *RDMBS'* model (shown in the left frame of the editor). The model that appears in the right frame of the editor is the final merged relational schema.

[4] More information about the semantics of the *Merge* operator can be found in [BCRL06]


4_trac1.traceabilitymodel

Domain

- platform:/resource/PAPERS/200606_SETRA06/1_po-I
  - Schema purchaseOrder
    - Table Item
      - Key Item_pk
      - Column Item_tid
      - Column Item_productName
      - Column Item_quantity
      - Column Item_uSPrice
      - Column Item_comment
      - Column Item_shipDate
      - Column Item_partNum
    - Table PurchaseOrder
      - Key PurchaseOrder_pk
      - Foreign Key PurchaseOrder_items_Item
      - Column PurchaseOrder_tid
      - Column PurchaseOrder_items_Item_tid
      - Column PurchaseOrder_orderDate

Mappings

- platform:/resource/PAPERS/200606_SETRA0
  - Traceability Model Merge
    - Traceability Link MergeRule-Schema
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link Refresh
    - Traceability Link MergeRule-Table
    - Traceability Link MergeRule-Column
    - Traceability Link Refresh
    - Traceability Link MergeRule-Column
    - Traceability Link Refresh
    - Traceability Link MergeRule-Key

Range

- platform:/resource/PAPERS/200606_SETRA06/4_merged.rdbms
  - Schema purchaseOrder
    - Table Customer
      - Key Customer_pk
      - Column Customer_name
      - Column Customer_address
      - Column Customer_tid
    - Table PurchaseOrder
      - Key PurchaseOrder_pk
      - Foreign Key PurchaseOrder_items_Item
      - Foreign Key PurchaseOrder_customer_Customer
      - Column PurchaseOrder_customer_Customer_tid
      - Column PurchaseOrder_orderDate
      - Column PurchaseOrder_items_Item_tid
      - Column PurchaseOrder_tid
    - Table Item
      - Key Item_pk
      - Column Item_partNum
      - Column Item_shipDate
      - Column Item_comment
      - Column Item_uSPrice
      - Column Item_quantity
      - Column Item_productName
      - Column Item_tid

Problems | Javadoc | Declaration | Progress | Properties

Property	Value
Domain	Table PurchaseOrder
Manipulation Rule	MergeRule-Table
Range	Table PurchaseOrder

Figure 3: Trace model that is produced during the merging of the models *RDBMS’* and *RDBMS*.

5 Exogenous Model Merging in MOMENT

The exogenous model merging problem consists in the merging of two models that conform to different metamodels, as in the example in Section 2. This problem can be divided into simpler ones that can be solved by two simple model management operators. A composite operator, called *ExogenousMerge*, can be defined for this purpose by composing the *Merge* operator and the *ModelGen* operator. This operator has three arguments: the model A, which conforms to the metamodel *MMA* (the *Ecore* metamodel in our example); the model B, which conforms to the metamodel *MMB* (the *RDBMS* metamodel in our example); and the name of the QVT transformation that must be defined between between the metamodels *MMA* and *MMB* (*umlToRdbms* in our example). In the first step, model A is transformed into a model B’, which conforms to the metamodel *MMB* by means of the operator *ModelGen*. This step has been performed in Section 3. In the second step, models B and B’ are merged within the metamodel *MMB*. This step has been performed in Section 4. Finally, the merged model *result* is the output of the composite operator. The definition of the *ExogenousMerge* composite operator is as follows:

```
operator ExogenousMerge (A : MMA, B : MMB, T : Transformation) =
        <B’, $map_{A->B'}$ > = ModelGen(T, A)                          (1)
        <result, $map_{A->B'}$, $map_{A->B}$> = Merge (B’, B)          (2)
return (result)
```

The *ExogenousMerge* operator is defined independently of any metamodel so that it can be reused to merge two models that conform to any metamodel. In this example, we have not taken into account the trace models that are generated by the *ModelGen* and *Merge* operators. Nevertheless, another version of the operator could generate traceability models as result of the *ExogenousMerge* operator.

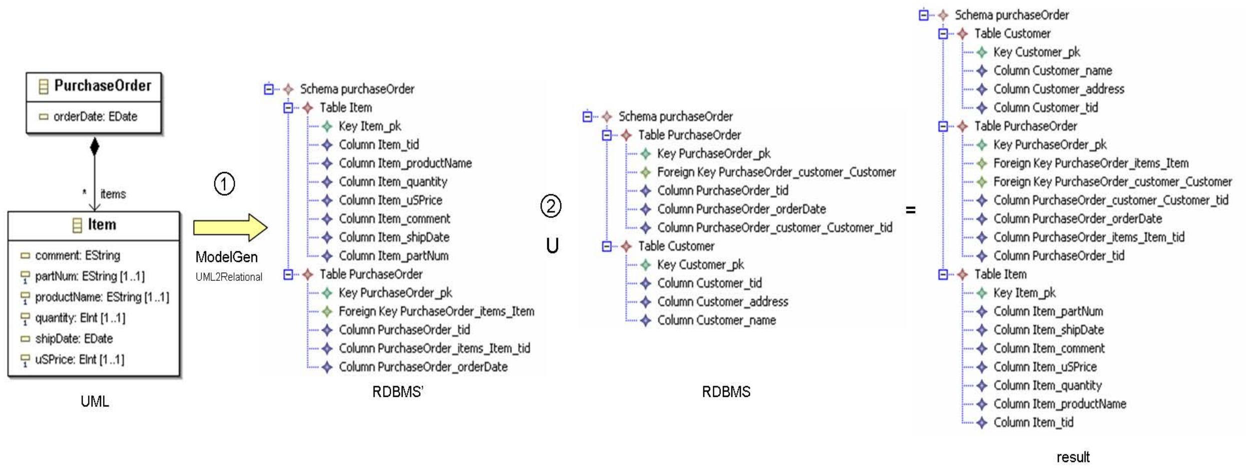


Figure 4: Application of the *ExogenousMerge* operator to the example in Section 2.

Fig. 4 graphically represents the merging process that is performed by the operator *ExogenousMerge* for solving the example that is shown in Section 2. In the example, parameter A corresponds to the UML model and parameter B corresponds to the RDBMS model in Fig. 4. To be able to apply the operator, the equivalence relation for the RDBMS metamodel and the transformation function between the UML and the Relational metamodels must be previously defined by the user.

6 Related Work

Generic model merging approaches take into account the phases that were discussed in [BLN86] to merge database schemas. These approaches can be differentiated by the mechanism that is used to perform model comparison.

[AP03] uses MOF identifiers to compare elements in different versions of a same base model. Although this approach is effective, only versions of a same base model can be compared and merged.

[BP03, BCE$^+$06] provide a set of model management operators to define equivalence relationships between elements of different models by means of mappings, which are used by a merge operator later on. In this approach, the *Merge* operator receives two models (A and B) and a mapping model (mapAB) between them as inputs, and it produces the merged model C and two new mapping models *(mapAC and mapBC): <C, mapAC, mapBC> = Merge (A, B, mapAB)*.

In the AMMA platform [FJ05], the Generic Model Weaver AMW is a tool that permits the definition of mapping models (called weaving models) between MOF models in the ATLAS

Model Management Architecture. AMW provides 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 [MBR01]. These mapping models are used, together with the mapped models in a model transformation to perform a model composition.

In MOMENT, mapping models are introduced as trace models that are generated by model management operators. This is because operators do not have to rely on them to be applied to a set of models. In MOMENT, mappings between the elements of two models are defined between the elements of their corresponding metamodels by means of checkonly QVT Relations. This permits a clearer specification of composite operators. Trace 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.

[KPP06] proposes several domain-specific languages to define model comparison and model merging over metamodels. The model comparison language enhances the definition of equivalence relationships between elements of a metamodel that can be applied over the elements of the corresponding models afterwards. In this language, a differentiation between matching and conformance is provided. While a matching mapping indicates when two elements are equivalent, a conformance mapping indicates when two elements are equivalent and consistent to be merged. In this approach, when an equivalence relationship based on names is used, two elements do not conform to each other if they have different types, for instance. In our approach, we use the QVT Relations language to perform model comparison and model transformation. This feature aims at decreasing the learning curve of our framework since there is only one language, which has been specified as an standard. The QVT Relation language does not provide such a differentiation between conformance and matching. Since two elements that do not conform to each other are usually interpreted as an error, we collapse the conformance and matching conditions in a relation. However, a transformation with conformance relations could be defined for a specific metamodel. Then, this transformation could be specialized with user-defined checkonly relations for defining equivalence relationships.

7 Conclusions

Model merging plays a relevant role in the maintenance and evolution of model-based software. Systems of this kind are usually represented by models that conform to different metamodels. Thus, two kinds of merging processes arise by considering the amount of metamodels that are involved: endogenous merging and exogenous merging. In an endogenous merging process, the models that are merged conform to the same metamodel. In an exogenous merging process, the models that are merged conform to different metamodels.

The MOMENT framework is a model management framework that provides operators to manipulate models on top of a MOF architecture, such as *Merge* for model merging and *ModelGen* for model transformations. In our approach, model management operators are defined independently of any metamodel, keeping a generic infrastructure, but they might be customized by an expert user with domain-specific knowledge by means of standard languages, such as OCL and QVT.

In this paper, we have presented how model transformations are supported in MOMENT through the QVT Relations language and how model transformations play an important role in a model merging process. We have used the standard QVT Relations for this purpose instead of providing new languages for model comparison and model merging. To study the aforementioned kinds of model merging, we have described a solution for an endogenous model merging process by using model transformations through the *Merge* operator. Finally, we have provided a generic solution for exogenous model merging by reusing model transformations and endogenous model merging.

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.

We are grateful to Abel Gómez, Pascual Queralt, Joaquín Oriente and Luis Hoyos for their effort in the development of the MOMENT Framework.

Bibliography

[AP03] M. Alanen, I. Porres. Difference and Union of Models. In Stevens et al. (eds.), *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*. LNCS 2863, pp. 2–17. Springer, 2003.

[BBM03] F. Budinsky, S. A. Brodsky, E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BCE$^+$06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, M. Sabetzadeh. A manifesto for model merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. Pp. 5–12. ACM Press, New York, NY, USA, 2006.

[BCR05] A. Boronat, J. A. Carsí, I. Ramos. Automatic Support for Traceability in a Generic Model Management Framework. In Hartman and Kreische (eds.), *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*. Lecture Notes in Computer Science 3748, pp. 316–330. Springer, 2005.

[BCRL06] A. Boronat, J. A. Carsí, I. Ramos, P. Letelier. Formal Model Merging Applied to Class Diagram Integration. *Electr. Notes Theor. Comput. Sci.*, 2006.

[BDK92] P. Buneman, S. B. Davidson, A. Kosky. Theoretical Aspects of Schema Merging. In *Extending Database Technology*. Pp. 152–167. 1992.

[Beh00] R. Behrens. A Grammar Based Model for XML Schema Integration. *Lecture Notes in Computer Science* 1832:172, 2000.

[Ber03] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*. 2003.

[BHP00] P. A. Bernstein, A. Y. Halevy, R. A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 29(4):55–63, 2000.

[BLN86] C. Batini, M. Lenzerini, S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.* 18(4):323–364, 1986.

[BP03] P. A. Bernstein, R. A. Pottinger. Merging Models Based on Given Correspondences. In *Proceedings of the 29th VLDB Conference*. Berlin, 2003.
http://www.cs.washington.edu/homes/rap/publications/pottinger-bernstein-vldb03.pdf

[FJ05] M. D. D. Fabro, F. Jouault. Model Transformation and Weaving in the AMMA Platform. In *Pre-proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE'05), Workshop*. Pp. 71–77. Centro de Ciências e Tecnologias de Computaao, Departemento de Informatica, Universidade do Minho, Braga, Portugal, 2005.

[HM05] P. Haase, B. Motik. A mapping system for the integration of OWL-DL ontologies. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*. Pp. 9–16. ACM Press, New York, NY, USA, 2005.

[KPP06] D. S. Kolovos, R. F. Paige, F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*. Pp. 13–20. ACM Press, New York, NY, USA, 2006.

[MBR01] J. Madhavan, P. A. Bernstein, E. Rahm. Generic Schema Matching Using Cupid. In *Proc. VLDB 2001*. Pp. 49–58. 2001.
http://www.research.microsoft.com/research/db/ModelMgt/CupidVLDB01.pdf

[Men02] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* 28(5):449–462, 2002.

[OMG04] OMG, Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/04-10-15). 2004.
http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[OMG05] OMG, Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005.
http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[OWK03] D. Ohst, M. Welle, U. Kelter. Differences between versions of UML diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. Pp. 227–236. ACM Press, New York, NY, USA, 2003.

[SGS+04] G. Straw, G. Georg, E. Song, S. Ghosh, R. B. France, J. M. Bieman. Model Composition Directives. In *UML*. Pp. 84–97. 2004.

A RDBMS Metamodel

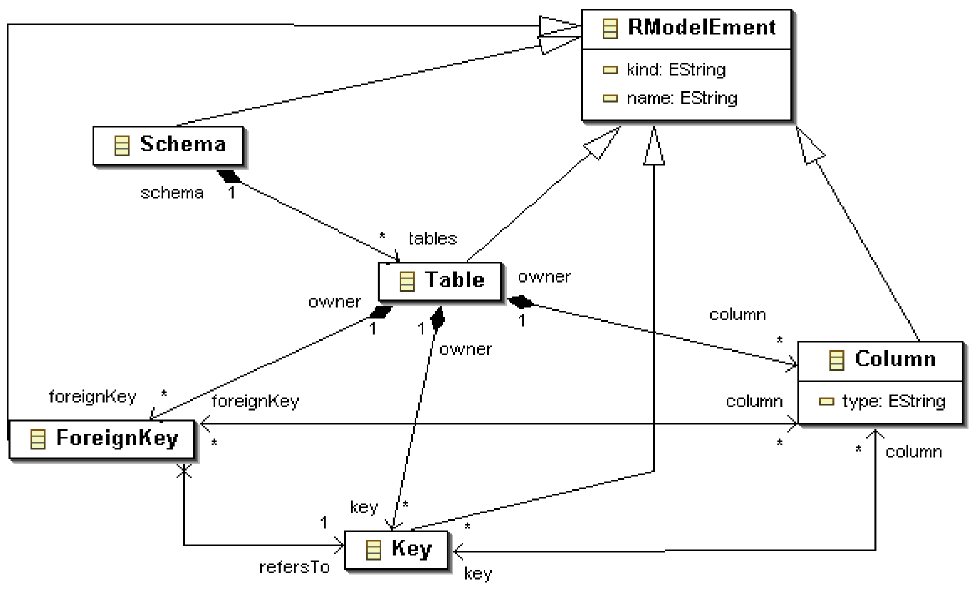


Figure 5: RDBMS metamodel.

B An Ecore to RDBMS Transformation by means of the QVT Relations Language

```
transformation umlToRdbms(ecoreDomain:ecore, rdbmsDomain:rdbms) {
        key Schema {name};
        key Table {schema,name};
        key Column {owner,name};
        key ForeignKey {owner,name};

        top relation PackageToSchema {
            packageName: String;
            checkonly domain ecoreDomain p:EPackage {
                name=packageName
            };
            enforce domain rdbmsDomain s:Schema {
                name=packageName
            };
        }//end PackageToSchema

        top relation ClassToTable {
            className: String;
```

```
    checkonly domain ecoreDomain c: EClass {
        ePackage = p:EPackage {},
        name=className
    };
    enforce domain rdbmsDomain t: Table {
        schema = s:Schema {},
        name = className,
        column = cl:Column {
            name = className + '_tid',
            type = 'NUMBER'
        },
        key = k:Key {
            name = className + '_pk',
            column=cl
        }
    };
    when {
        PackageToSchema(p, s);
    }
    where {
        AttributeToColumn(c, t, className);
    }
}//end ClassToTable

relation AttributeToColumn
{
        checkonly domain ecoreDomain c:EClass {};
        checkonly domain rdbmsDomain t:Table {};
    primitive domain prefix:String;
    where {
        PrimitiveAttributeToColumn(c, t, prefix);
        SuperAttributeToColumn(c, t, prefix);
    }
}//end AttributeToColumn

relation PrimitiveAttributeToColumn
{
    attributeName, columnName, sqltype: String;
    checkonly domain ecoreDomain c:EClass {
        eAttributes = a:EAttribute {}
    };
    checkonly domain rdbmsDomain t:Table {};
    primitive domain prefix:String;
    where {
        PrimitiveAttributeToColumneAttributes(a,t,prefix);
    }
}//end PrimitiveAttributeToColumn

relation PrimitiveAttributeToColumneAttributes
{
    attributeName, columnName, ecoreTypeName, sqltype: String;
    checkonly domain ecoreDomain a:EAttribute {
        name = attributeName,
        eType = ecoretype: EDataType {
            name = ecoreTypeName
        }
    };
    checkonly domain rdbmsDomain t:Table{};
    enforce domain rdbmsDomain cl:Column {
        name = (
            if (prefix = ”) then
                attributeName
            else
                prefix + ’_’ + attributeName
            endif
        ),
        type = PrimitiveTypeToSqlType(ecoreTypeName),
        owner = t
    };
    primitive domain prefix:String;
    when {
        IsPrimitiveDatatype(ecoreTypeName);
    }
}//end relation

relation SuperAttributeToColumn
    {
    checkonly domain ecoreDomain c: EClass {
        eSuperTypes = sc:EClass {}
    };
    checkonly domain rdbmsDomain t:Table {};
        primitive domain prefix: String;
    where {
        AttributeToColumn(sc, t, prefix);
    }
}

top relation AssocToFKey
{
        srcTbl, destTbl: Table;
        pKey: Key;
        referenceName, sourceClassName, targetClassName: String;
            checkonly domain ecoreDomain ref: EReference {
            name = referenceName,
            eContainingClass = sc:EClass {
                name = sourceClassName
            },
            eType = tc:EClass {
                name = targetClassName
            }
        };
        enforce domain rdbmsDomain fk:ForeignKey {
            name = sourceClassName + '_' + referenceName + '_' + targetClassName,
            owner = srcTbl,
            column = fkc:Column {
                name = sourceClassName + '_' + referenceName + '_' + targetClassName +
'_tid',
                type = 'NUMBER',
                owner = srcTbl
            },
            refersTo = ObtainReferredPrimaryKey(destTbl)
        };
        when {
            ClassToTable(sc, srcTbl);
            ClassToTable(tc, destTbl);
        }
    }

    function ObtainReferredPrimaryKey(table: Table):Key
    {
        table.key
    }

    function IsPrimitiveDatatype(datatype: String):Bool
    {
        ((datatype = 'EInt') or (datatype = 'EBoolean') or (datatype = 'EString') or (datatype
= 'EDate'))
    }

    function PrimitiveTypeToSqlType(primitiveType:String):String
    {
        if (primitiveType='EInt')
            then 'NUMBER'
        else if (primitiveType='EBoolean')
```

```
                then ’BOOLEAN’
                else
                    ’VARCHAR’
                endif
            endif
        }

        function IsDirectedReference(ref:EReference):Bool
        {
            (ref.eOpposite -> size() = 0)
        }
    }
```