Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 166 (2007) 5–26
www.elsevier.com/locate/entcs

Formal Model Merging Applied to Class Diagram Integration

Artur Boronat[1] José Á. Carsí[2] Isidro Ramos[3]
Patricio Letelier[4]

Information Systems and Computation Department
Polytechnic University of Valencia
Valencia, Spain

Abstract

The integration of software artifacts is present in many scenarios of the Software Engineering field: object-oriented modeling, relational databases, XML schemas, ontologies, aspect-oriented programming, etc. In Model Management, software artifacts are viewed as models that can be manipulated by means of generic operators, which are specified independently of the context in which they are used. One of these operators is *Merge*, which enables the automated integration of models. Solutions for merging models that are achieved by applying this operator are more abstract and reusable than the ad-hoc solutions that are pervasive in many contexts of the Software Engineering field. In this paper, we present our automated approach for generic model merging from a practical standpoint, providing support for conflict resolution and traceability between software artifacts by using the QVT Relations language. We focus on the definition of our operator *Merge*, applying it to Class Diagrams integration.

Keywords: Model-Driven Engineering, Model Management, model merging, conflict resolution, QVT Relations.

1 Introduction

The Model-Driven Development philosophy [15] considers models as the main assets in the software development process. Models collect the information that describes the information system at a high abstraction level, which permits the development of the application in an automated way following generative programming techniques [12]. In this process, models constitute software artifacts that experience refinements from the problem space (where they capture the requirements of the application)

[1] Email: aboronat@dsic.upv.es
[2] Email: pcarsi@dsic.upv.es
[3] Email: iramos@dsic.upv.es
[4] Email: letelier@dsic.upv.es

doi:10.1016/j.entcs.2006.06.013

to the solution space (where they specify the design, development and deployment of the final software product).

During this refinement process, several tasks are applied to models such as transformation and integration tasks. These tasks can be performed from a model management point of view. Model Management was presented in [2] as an approach to deal with software artifacts by means of generic operators that do not depend on metamodels by working on mappings between models. Operators of this kind deal with models as first-class citizens, increasing the level of abstraction by avoiding working at a programming level and improving the reusability of the solution.

Based on our experience in formal model transformation and data migration [8], we are working on the application of the model management trend in the context of the Model-Driven Development. We have developed a framework, called MOMENT (MOdel manageMENT) [25], which is embedded into the Eclipse platform and that provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [14]. Some of the simple operators defined are: the union, intersection and difference between two models, the transformation of a set of models to other model applying a QVT transformation, the navigation through mappings, and so on. Complex operators can be defined by composition of other operators. In this paper, we present the operator *Merge* of the MOMENT framework from a practical point of view. The underlying formalism of our model management approach is Maude [11]. We apply it as a novel solution for the integration of UML Class Diagrams in a Use Case Driven software development process.

The structure of the paper is as follows: Section 2 presents a case study used as an example in the rest of the paper; Section 3 describes our approach for dealing with models by means of an industrial modeling tool, and also presents the generic semantics of the operator *Merge*; Section 4 presents the customization of the operator *Merge* to the UML metamodel; Section 5 explains the application of the operator *Merge* to the case study; Section 6 discusses some related work; and Section 7 summarizes the advantages of our approach.

2 Case Study: Use Case Analysis using Partial Class Diagrams

Software development methodologies based on UML propose an approach where the process is Use Case Driven [17,18]. This means that all artifacts (including the Analysis and Design Model, its implementation and the associated test specifications) have traceability links from Use Cases. These artifacts are refined through several transformation steps. Obtaining the Analysis Model from the Use Case Model is possibly the transformation that has the least chance of achieving total automation. The Use Case Model must sacrifice precision in order to facilitate readability and validation so that the analysis of use cases is mainly a manual activity.

When the Use Case Model has many use cases, managing traceability between each use case and the corresponding elements in the resulting class diagram can be a difficult task. In this scenario, it seems reasonable to work with each use case

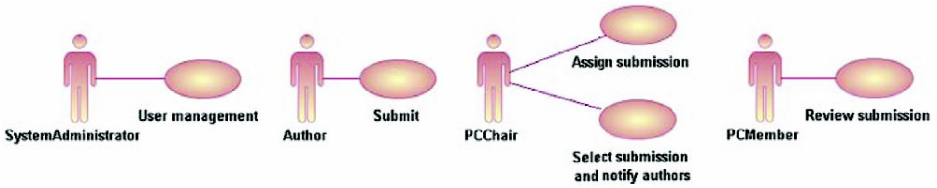

Fig. 1. Use Case Model.

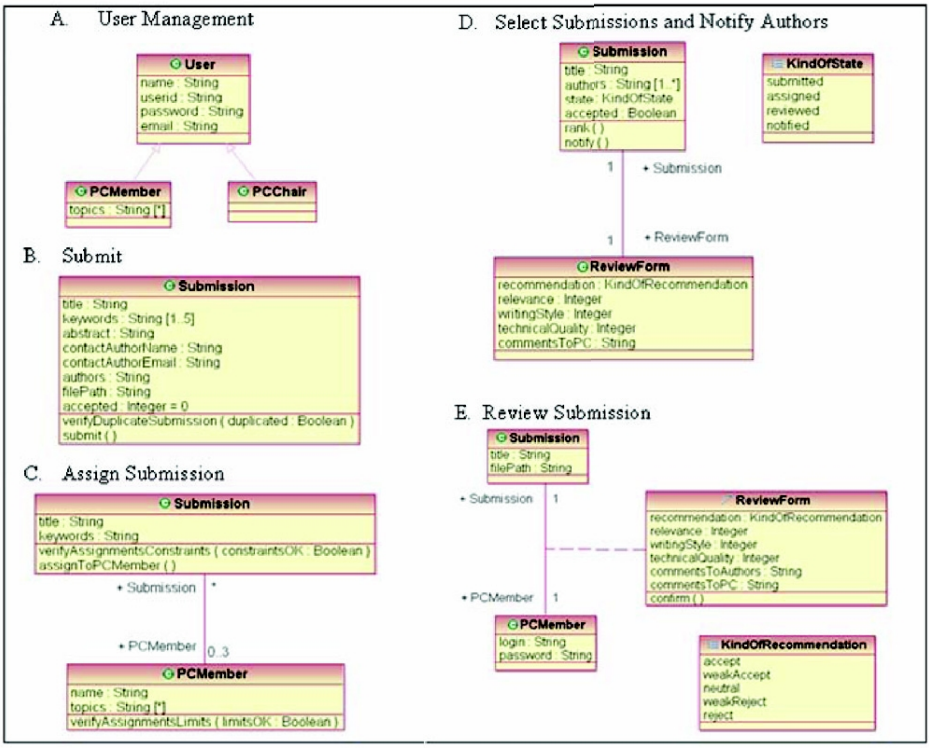

Fig. 2. Partial models associated to the corresponding Use Cases.

separately and to register its partial class diagram (which is a piece of the resulting class diagram that represents the Analysis Model). Regarding traceability, this strategy is a pragmatic solution, but when several team members work in parallel with different use cases, inconsistencies or conflicts among partial models often arise, which must be solved when obtaining the integrated model.

We present a case study that illustrates how our operator *Merge* can be used effectively to deal with the required needs established above. We present part of a system for managing submissions that are received in a conference. In our example, we will focus on the fragment of the Use Case Model shown in Fig. 1. The actor System Administrator manages user accounts. Authors submit papers to the conference. The PCChair assigns submissions to PCMembers. Each submission is assessed by several PCMembers using review forms. When all the reviews are completed, the PCChair ranks the submissions according to the assessment contained in the review forms. Since there is a limit to the numbers of papers that can be

presented, and taking into account the established ranking, some submissions are selected, and the rest are rejected. Then, all authors are notified by email attaching the review forms of their submission. Fig. 2 shows the Class Diagrams that support the functionality required for the corresponding Use Case.

3 The Generic Semantics of the *Merge* Operator

In a Model-Driven Development context [21], models consist of sets of elements that describe some physical, abstract, or hypothetical reality. In the process of defining a model, abstraction and classification are guidelines to be taken into account. A metamodel is simply a model of a modeling language. It defines the structure, and constraints for a family of models.

In our framework, a metamodel is viewed as an algebraic specification where the model management operators are defined so that they can be applied to all the models that conform to the metamodel. In the following sections, we describe the generic infrastructure for applying the *Merge* model to models that conform to a specific metamodel, and we focus on the generic semantics of the *Merge* operator in further details.

3.1 Maude

Maude [11] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming. The *Merge* operator has been specified in Maude functional modules[5]. Functional modules describe data types and operations on them by means of membership equational theories. Mathematically, such a theory can be described as a pair $(\Sigma, E \bigcup A)$, where: Σ is the signature that specifies the type structure (sorts, subsorts, kinds, and overloaded operators); E is the collection of equations and memberships declared in the functional module; and A is the collection of equational attributes (associativity, commutativity, and so on) that are declared for the different operators. Computation is the form of equational deduction in which equations are used from left to right as simplification rules, with the rules being Church-Rosser and terminating.

3.2 The QVT Relations language

A model merging process involves a composition task that can be viewed as a model transformation process, where the two input models to be merged are also the inputs for the model transformation. The *Merge* operator has been defined in a generic way in the MOMENT Framework but it can also be specialized by a domain-specific expert user. To provide support for extensibility in our approach, we have chosen the standard QVT Relations language [23].

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transfor-

[5] In Appendix A, we have enclosed the notation that has been used to define operators and equations along the paper. We refer to [10] for further details.

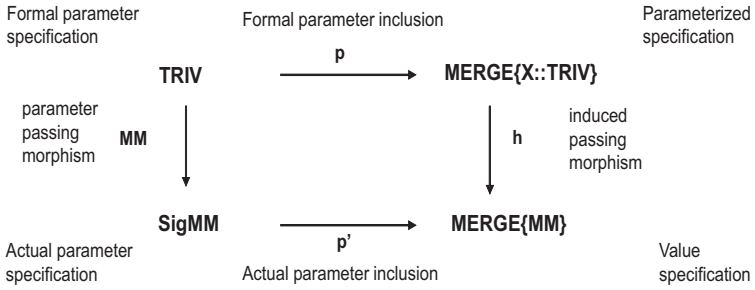

Fig. 3. The parameter passing diagram for the *MERGE*{*X :: TRIV*} parameterized module.

mation 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 enforceable, a transformation is performed. If the target domain is defined as checkonly, just a checking is performed. Both kinds of transformations are used in our approach, as we explain in the following sections.

3.3 Generic Infrastructure for the Merge *Operator*

The *Merge* operator is specified in the parameterized theory MERGE{X :: TRIV}. This theory also extends other theories that provide support for OCL [7] and for the QVT Relations language [6]. OCL and QVT are standard languages for performing queries [6] and model transformations, respectively, when the semantics of the *Merge* operator is specialized.

Fig. 3 represents the parameterization mechanism that shows how a *Merge* operator can be applied to different metamodels in the MOMENT Framework. In the following sections, the elements involved in the diagram are explained.

3.3.1 The formal parameter

TRIV is the algebraic specification of the formal parameter, which is called theory in Maude. The TRIV theory is constituted by only one sort: *fth TRIV is sort Elt . endfth* .

3.3.2 The actual parameter

A signature called *SigMM*, which provides the constructors of a specific metamodel, is specified in order to represent a model by means of algebraic terms. The *SigMM* specification constitutes the actual parameter for the *MERGE*{*X :: TRIV*} module and provides a constructor for each type that is defined in the metamodel and an inheritance hierarchy among the types that appear in the metamodel.

As example, we consider the Ecore metamodel [9] as an implementation of the UML Class Diagram metamodel. In the Ecore metamodel, we have the constructs

[6] In this context, we use OCL as a query language, as recommended in the QVT standard.

that define a class, an attribute, an operation, among others. This signature is automatically generated from a metamodel taking into account the following elements that can appear in a metamodel: non-abstract classes, which are represented by means of algebraic constructors whose arguments are the attributes (simple data types) and references (collections of identifiers); and class hierarchies, which are represented by means of subsort relationships. In the signature that has been generated for the Ecore metamodel [7], the class *EClass* is represented by means of the ecore-EClass sort, whose constructor is:

```
op '(ecore-EClass _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _') :
    Qid                *** identifier
    String             *** name
    String             *** instanceClassName
    String             *** instanceClass
    String             *** defaultValue
    Bool               *** abstract
    Bool               *** interface
    OrderedSet {QID}            *** eAnnotations
    OrderedSet {QID}            *** ePackage
. . . -> ecore-EClass [ctor] .
```

All the sorts that are generated are subsort of an artificial sort, called ecoreNode for the example. In the membership equational logic underlying Maude, sorts are grouped into equivalence classes, called kinds, through the subsort relation. By adding a supersort to all the other sorts, we are collapsing all the sorts into the same kind so that all the terms (representing class instances) that belong to the sorts (representing classes) of the signature can be used as an element in a collection. The signature that is generated from an EMF metamodel constitutes the actual parameter for the module *MERGE*{*X* :: *TRIV*} . This task is automatically performed by MOMENT from Ecore models.

3.3.3 The parameterized module MERGE{X :: TRIV}

A parameterized module called *MERGE*{*X* :: *TRIV*}, which provides the generic specification for the *Merge* operator. In MOMENT, the *Merge* operator is defined axiomatically using the Maude algebraic language. Maude allows us to specify the operator in an abstract, modular and scalable manner so that we can define its semantics from a generic and reusable point of view. The specification of the *Merge* operator is explained in the Section 3.4.4 and Appendix B.

The *MERGE*{*X* :: *TRIV*} module also provides the constructors that are needed to specify a model as a set of elements, by extending our algebraic specification of OCL [7] . To understand how we represent models as sets, we introduce the underlying OCL support for Set collections.

Among the four collection types that are provided by OCL, a Set is a collection that contains instances of a valid OCL type, where order is not relevant and duplicate elements are not allowed. To represent a group of elements that are not ordered, we introduce the sort *Magma*{*X*}. The constructor for this sort has the symbol ",", which is associative and commutative. Thus, working with integers, "1, 2, 3" is a term that represents a valid *Magma*{*Int*}. In addition, we can state

[7] The whole signature that has been generated for the Ecore metamodel is attached in Appendix B.

that "1,2,3" and "3,2,1" represent the same group of elements modulo the commutative and associative attributes. The Maude code that specifies the Set type in our specification of OCL is as follows:

(i) ***sort*** *Magma{X}* .

(ii) ***subsort*** *X$Elt < Magma{X}* .

(iii) ***sorts*** *Collection{X} Set{X} OrderedSet{X}* .

(iv) ***subsort*** *Collection{X} < X$Elt* .

(v) ***subsorts*** *Set{X} < Collection{X}* .

(vi) ***op*** *empty-magma : -> Magma{X} [ctor]* .

(vii) ***op*** *_,_ : Magma{X} Magma{X} -> Magma{X} [ctor assoc comm id: empty-magma]* .

(viii) ***op*** *Set{_} : Magma{X} -> Set{X} [ctor]* .

(ix) ***op*** *empty-set : -> Set{X} [ctor]* .

(x) ***eq*** *Set{ empty-magma } = empty-set* .

Terms of the sort *Magma{X}* are used to define sets (line (viii)). The sort Collection{X} can be considered as an abstract concept on the grounds that there is no specific constructor for it. The equation (x) guarantees the consistency of an empty set on the grounds that a set that contains an empty group of elements (*empty-magma*) is also an *empty-set*.

3.3.4 The MM view

A view has been defined for each Maude simple data type in order to deal with collections of simple data types. For instance, to deal with collections of integers, the following view is defined: *view Int from TRIV to INT is sort Elt to Int . endv*

This view is used to instantiate the *MERGE{X :: TRIV}* module as *MERGE{Int}*. This way, the following example is a valid collection of integers: *Set{1, 2, 3}* .

The *MM* view is the morphism that relates the elements of the TRIV formal parameter to the elements of the *SigMM* actual parameter. To represent models as Sets, we map the supersort that is generated for a metamodel (ecoreNode for the example) to the sort Elt of the TRIV theory, by means of an *MM* view. For the example: *from TRIV to sigecore is sort Elt to ecoreNode . endv* .

Thus, we can use the constructors that constitute the *SigMM* theory to define terms that represent the corresponding class instances. For example, *Set{(ecore-EClass 'a "EClass" "" "" "" false false empty-orderedset . . .)}* is the representation of a model that only contains a class called "EClass".

3.3.5 The instantiated module SpMM

A module called *SpMM* instantiates the parameterized module *MERGE{X :: TRIV}* by passing *SigMM* as actual parameter by means of the *MM* view. In the instantiation process, the *Merge* operator is customized to the constructs of the metamodel, represented by *SigMM*.

This provides the constructors that are needed to specify a model that conform to this metamodel as a set of elements, as explained above. This fact also provides the *Merge* operator that can be automatically applied to models that conform to this metamodel. To enable the manipulation of UML models, they have to be represented as terms of the *SpMM* algebraic specification, i.e. as a sets of terms that represent class instances (by means of the constructor operators that are obtained from the classes of a metamodel).

In the *SpMM* module, the specification of the *Merge* operator can also be specialized to a metamodel by simply adding new axioms to the *Merge* operator in an ad-hoc and more accurate way, taking advantage of both complementary standpoints: generic infrastructure and domain-specific knowledge. These features are further developed in Sections 4 and 5.

3.3.6 Inclusion morphisms p and p'

In Fig. 3, 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 Merge{*MM*} value specification by using the *MM* parameter passing morphism.

3.4 The Generic Semantics of the Operator Merge

The *Merge* operator takes two models as input and produces a third one. If A and B are models (represented as terms) in a specific metamodel algebraic specification, 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 define the semantics of the *Merge* operator, we need to introduce three concepts: the equivalence relation, the conflict resolution strategy and the refreshment of a construct.

3.4.1 Equivalence relation

A semantic equivalence relation is defined between elements that belong to different models but to the same metamodel. This relation is embodied by the operator *Equals*. The declaration operator *Equals* is as follows:

op *Equals : Set{X} Set{X} Set{X} Set{X} -> Bool .*

where the first argument is the first model to be compared that may contain only one element; the second argument is the entire model that contains the first argument as a subset; the third argument is the second model to be compared and it also may contain only one element; the fourth model represents the second input model and contains the third argument as a subset. The second and the fourth arguments are needed because we are defining all the theories as functional modules in membership equational logic. Thus, the notion of memory state does not exist in this context

and those arguments permit the navigation through the structure of the entire input models in order to check if the equivalence relation holds.

The generic semantics of the *Equals* operation coincides with the syntactical equivalence. The equational specification of the *Equals* operation is as follows:

```
vars N : X$Elt .
vars Model1 Model1b Model2 Model2b : Set{X} .
vars M1 M2 : Magma{X} .
eq Equals( Set{N, M1}, Model1, Set{N,M2}, Model2 ) =
     Equals( Set{M1}, Model1, Set{M2}, Model2 ) .
eq Equals( empty-set, Model1, empty-set, Model2 ) = true .
eq Equals( Model1, Model1b, Model2, Model2b ) = false [owise] .
```

where the sort *X$Elt* represents an element of a model (*Set*{*X*}), *Set*{*X*} is the generic sort for a set collection and *MagmaX* is the sort that represents groups of elements of a set. The first equation states that if the element N (first argument) also exists in the second model (third argument) they are equivalent (this equation is applied to all the elements of the models 1 and model 2 recursively); the second equation constitutes the base case for the recursion; the third equation permits indicating when there is an element in Model1 or Model2 is not equivalent. This generic semantics can be enriched by means of QVT checkonly relations that take into account the structure of a specific metamodel. This is explained in more detail in Section 4.

3.4.2 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 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 provided by the operator *Resolve*, whose generic semantics consists of the preferred model strategy. When the *Merge* operator is applied to two models, one has to be chosen as preferred. In this way, when two groups of elements (that belong to different models) are equivalent due to the *Equals* operation, although they differ syntactically, the elements of the preferred model prevail. The signature of the *Resolve* operator is as follows:

```
op Resolve : Set{X} Set{X} Set{X} Set{X} -> Tuple{ X, TraceabilityMetamodel, TraceabilityMetamodel
                                                 } .
```

where the first argument is the first model to be merged that contains only one element; the second argument is the entire model that contains the first argument as a subset; the third argument is the second model to be merged and it also contains only one element; the fourth model represents the second whole model and contains the third argument as a subset. The second argument of the operator (first input model of the Merge operator) is taken as the preferred model.

The generic semantics of the *Resolve* operator is as follows:

```
eq Resolve ( Set{N1}, Model1b, Set{N2}, Model2b) =
    (
        Set{Refresh(N1, Model1b, Model2b)}
    ,
        GenerateTrace(N1, N1)
    ,
```

```
    GenerateTrace(N2, N1)
) [owise] .
```

where the *Resolve* operator produces a tuple of three elements. In this tuple: the first element is the merged element, which coincides with the element of the first model (the preferred one); the second results in a trace that is generated between the source element N1 and the resulting element (this trace belongs to the trace model that is generated between the first model and the resulting merged model); the third results in a trace between the source element N2 and the resulting element N1 (this trace belongs to the trace model that is generated between the second model and the resulting merged model).

The semantics of the *Resolve* operator can also be customized for a specific metamodel by means of enforced QVT Relations. This feature is explained in more detail in Section 4.

3.4.3 Refreshments

Refreshments are needed to copy non-duplicated elements into the merged model in order to maintain its references in a valid state. If we merge models B and C in our case study, taking model B as the preferred one, the reference Submission of the class *PCMember* of model C is copied to the merged model. As the class Submission of model C has been replaced by the one from model B, the reference, which points to the class Submission of model C, is no longer valid. Thus, this reference must be updated. The update of a specific metamodel construct term is embodied by the operator Refresh.

3.4.4 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 and two models of traces that relate the elements of the input models to the elements of the output merged model. Therefore, these traces, which are automatically generated by the *Merge* operator, provide full support for keeping traceability between the input models and the new merged one. The declaration of the operator is as follows:

```
op Merge : Set{X} Set{X} -> Tuple{ X, TraceabilityMetamodel, TraceabilityMetamodel } .
```

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 a merged element, which is then added to the output model. The elements that belong to only one model, without being duplicated in the other one, are refreshed and directly copied into the merged model. The equational specification of the *Merge* operator is presented in Appendix C.

4 Specific Semantics for the Ecore Metamodel to *Merge* Uml Class Diagrams

In this section, we briefly describe how the user can add specific semantics to the *Merge* operator to integrate UML Class Diagrams, which are implemented in the EMF by means of the Ecore metamodel. To fulfill this goal, the user only has to add specific axioms for the operators *Equals* and *Resolve*. This is achieved by using the QVT Relations language.

4.1 Equivalence Relation Specialization

To define an equivalence relation among the elements of a model, 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, both domains are the Ecore metamodel and 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 Ecore models. These QVT Relations are also used in the merging process to check when two elements are equivalent in order to eliminate duplicates.

In the example, the following relation can be defined to indicate that two classes are the same if they belong to the same package and they have the same name[8]:

```
top relation EClassEquivalence {
    className: String;
    checkonly domain ecoreDomain1 c1: EClass {
        ePackage = p1:EPackage {},
        name=className
    };
    checkonly domain ecoreDomain2 c2: EClass {
        ePackage = p2:EPackage {},
        name=className
    };
    when {
        EPackageEquivalence(p1, p2);
    }
}
```

where the EPackageEquivalence is another QVT Relation defined within the same transformation, describing when two EPackage 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, where *EClass* instances and EPackage instances are used to check whether two classes are equivalent or not.

The checkonly QVT transformation, which is constituted by checkonly QVT Relations of this kind, is then compiled into equations for the *Equals* operator. The specific semantics for the operator *Equals* are obtained by compilation from a QVT Relations transformation. The equation that is obtained for the QVT Relation of the example is as follows:

```
ceq Equals( Set{N1,M2}, Model1, Set{N2, M2} Model2 ) =
```

[8] 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.

```
    if (
        ((N1.name) == (N2.name)) and
        (Equals(
            (N1 :: ePackage(Model1)) -> asSet , Model1,
            (N2 :: ePackage(Model2)) -> asSet , Model2
        ))
    ) then
        Equals( Set{M1}, Model1, Set{M2}, Model2 )
    else
        false
    fi
if (N1 :: oclIsTypeOf ( ? "EClass"; Model1))
    and (N2 :: oclIsTypeOf( ? "EClass" ; Model2)) .
```

where the expression *(N1 :: ePackage(Model1))* permits navigating the role *ePackage* from the class N1 and *(N1 :: oclIsTypeOf (? "EClass"; Model1))* checks if the element N1 is instance of the class *EClass*.

During the merging process, this equation permits checking that the Submission classes of the partial models B and C are equivalent so that they will be merged. In the 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. For the example, a thesaurus function can be easily implemented in this way to indicate that the *userid* attribute of the class *User* in model A is equivalent to the *login* attribute of the *PCMember* class in model D. Thus, the user only has to know the standard QVT Relations language and the domain-specific knowledge on the grounds that the underlying formalism that is used in our approach remains completely hidden.

4.2 Conflict Resolution Strategy Specialization

To refine the Merge operator, the conflict resolution strategy can also be specialized. 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 refine the conflict resolution strategy in the same way a checkonly QVT transformation is used to refine the generic equivalence relation.

An additional equation completes the generic semantics of the *Resolve* operator in order to be specialized for a specific metamodel:

```
eq Resolve ( Set{N1}, Model1b, Set{N2}, Model2b) =
    ModelGen( merge ; ? Set{N1} ? Model1b ? Set{N2} ? Model2b ) .
```

In this equation, the *Resolve* operator invokes a ModelGen operator that performs a model transformation, whose equational semantics are obtained automatically by means of the compilation of a QVT Transformation, which is previously defined by a domain-expert user. 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 (Ecore 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 the class Submission of model B with the class

Submission of model C, we have to integrate their respective attributes, references and operations. The following QVT Relation is intended to perform this task:

```
top relation EClassMerging {
    className: String;
    checkonly domain ecoreDomain1 c1: EClass {
        ePackage = p1:EPackage {},
        name = className
    };
    checkonly domain ecoreDomain2 c2: EClass {
        ePackage = p2:EPackage {},
        name = className
    };
    enforce domain ecoreDomain3 c3: EClass {
        ePackage = p3:EPackage {},
        name = className
    };
    when {
        EPackageMerging(p1, p2, p3);
    }
    where {
        EAttributeMerging(c1, c2, c3);
        EReferenceMerging(c1, c2, c3);
        EOperationMerging(c1, c2, c3);
    }
}
```

where the *EPackageMerging* QVT relation, which is invoked in the *when* clause, ensures that the container packages of both *EClass* intances must be equivalent in order to apply the current relation to the classes. The QVT Relations, which are invoked in the *where* clause, ensure that the merging process will go on by merging their attributes, references and operations.

The enforce QVT transformation that the user defines to specialize the conflict resolution strategy is automatically compiled into a ModelGen equation [9]. ModelGen [10] is the operator that embodies model transformations in the MOMENT Framework and is invoked by the *Resolve* operator. The compilation of the above QVT relation generates the following QVT equation for the ModelGen operator:

```
ceq ModelGen (EClassMerging ; ? Set{ N1, M1 } ? Model1 ? Set{ N2, M2 } ? Model2 )
= (
    Set{
        New("EClass")
        :: ePackage <- (p1
            (ModelGenRule ( EPackageMerging ;
                ? ((N1 :: ePackage(Model1)) -> asSet) ? Model1
                ? ((N2 :: ePackage(Model2)) -> asSet) ? Model2
            ))
        :: name <- (N1 :: name)
    } -> including (
        p1( ModelGenRule (EAttributeMerging ;
            ? Set{ N1 } ? Model1 ? Set{ N2 } ? Model2 ) )
    ) -> including (
        p1( ModelGenRule (EReferenceMerging ;
            ? Set{ N1 } ? Model1 ? Set{ N2 } ? Model2 ) )
    ) -> including (
        p1( ModelGenRule (EOperationMerging ;
```

[9] Indeed, the equation is generated for the ModelGenRule operator, which represents one transformation rule of the entire transformation. We have skipped this detail in the paper for the sake of clarity.

[10] More information on the ModelGen operator and the model transformation process in the MOMENT Framework can be found in [6].

```
            ? Set{ N1 } ? Model1 ? Set{ N2 } ? Model2 ) )
    ) -> including (
        p1( ModelGenRule (EClassMerging ;
            ? Set{ M1 } ? Model1 ? Set{ M2 } ? Model2 ) )
    ) -> flatten
,
    First trace model computation
,
    Second trace model computation
)
if (N1 :: ecore-EClass) and (N2 :: ecore-EClass) .
```

In this equation: the operator <– represents the value assignment operation; the expression *New("EClass")* creates a new empty instance of the *EClass* class; the value assigned to the reference *ePackage* is the resulting package that will contain the merged class (obtained by means of the *EPackageMerging* transformation rule); the first three includings permit the addition of the outputs of the transformation rules *EAttributeMerging*, *EReferenceMerging* and *EOperationMerging* to the final result [11] [12]; the final including permits the addition of the elements that are merged by applying the *EClassMerging* relation to the rest of *EClass* instances of SetM1 and SetM2, recursively. This equation returns a triple, where the first component represents elements of the merged model, the second component represents traces of the trace model that relates the first input model to the merged model, and the third represents traces of the trace model that related the second input model to the merged model. In the example the generation of the traces has been skipped. This process consists in creating a new trace where the domain property refers to the input element and the range property refers to the new generated element [13].

5 Merging Process

In this section, we present the merging process that is used to integrate the five partial class diagrams of the case study. The four steps followed are indicated in Table 1, where the first argument for the merge operator is the preferred one. In this table, the first column indicates the step number; the second column shows the invocation of the *Merge* operator; the third column describes some of the main conflicts that have appeared during the merging step; the fourth column indicates the partial models involved that contain the conflicting elements; and the latter indicates the solution of the conflict by the *Resolve* operator.

After each step of the merging process, two models of mappings are automatically generated. These mappings provide full support for traceability by registering the transformation applied to the elements of the source partial models and by relating them to elements of the merged model. In the MOMENT framework, a set of operators is provided to navigate mappings bidirectionally [5]: from a partial model to the merged model (providing support for the propagation of changes from

[11] As the returning value of a ModelGenRule is a triple, p1 obtains the first component, which is a set that contains elements of the resulting merged model

[12] In these relations, we assume that the semantics of the inheritance is taken into account for the merging process in Section 5.

[13] To study the traceability that we provide in the MOMENT Framework and its applicability in further detail, we refer to [5]

	Model Merging	Conflicts	Models	Resolution
1	<BC, map_{B2BC}, map_{C2BC}> = *Merge*(B, C)	The multiplicity of the attribute *keywords* (class Submission).	B - C	Multiplicity [1..5] (preferred model)
2	<DE, map_{D2DE}, map_{E2DE}> = *Merge*(D, E)			
3	<DEBC, $map_{DE2DEBC}$, $map_{BC2DEBC}$>	3.1. The multiplicity of the attribute authors (class Submission)	E - B	Multiplicity [1..*] (preferred model)
	=	3.2. Type of the attribute accepted (class Submission)	E - B	Type Boolean (preferred model)
	Merge(DE,BC)	3.3. Multiplicities of the association between the classes Submission and PCMember	D - C	Multiplicities 1..1 - 1..1 (preferred model)
4	<ABCDE, $map_{A2ABCDE}$, $map_{BCDE2ABCDE}$> = *Merge*(A, BCDE)	The attribute userid (class User) and the attribute login (class PCMember) are identified as the same, by means of a thesaurus.	A - D	The inherited feature prevails by means of the EClass axiom for the operator Resolve.

Table 1
The steps of the Class Diagram merging process.

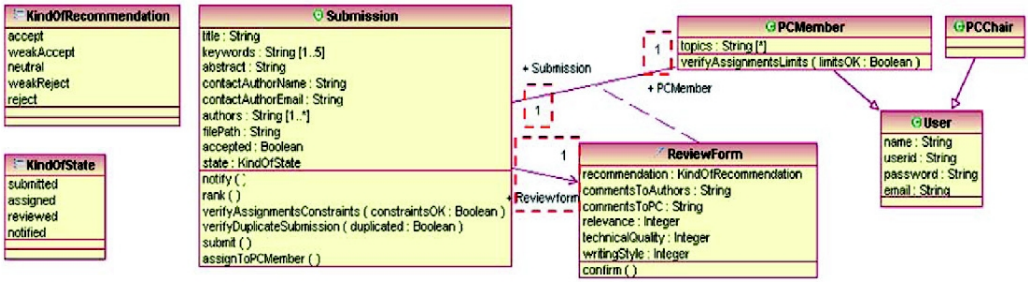

Fig. 4. Resulting merged model for the case study.

a specific use case to the merged model, as well as preserving the changes applied to the latter); or from the merged model to a partial class diagram (providing support in order to update a specific use case). Moreover, such mappings are considered as models so that generic model management operators can also be applied to them.

In Fig. 4, we show the resulting merged model resulting from step 4. Although the user describes the semantics of the *Merge* operator for a specific metamodel, as the model merging is completely automated, there might exist some undesired results in the merged model that should be fixed. In this figure, elements of this kind are highlighted by a discontinuous line. Therefore, the directed association that comes from partial model D should be deleted, and the multiplicity of the existing association between the Submission and the PCMember classes should be updated with the multiplicity that appears in partial model C.

In such cases, the user has the option to open the resulting merged model to review and update it. Merged models can be manipulated from visual editors that are integrated in the Eclipse platform. Although the merged model might be modified, the traces that are generated by the Merge operator can be updated by means of other model management operators, such as the Match operator. This operator infers a trace model between two models that conform to the same metamodel by taking into account the checkonly QVT transformation that is used to refine the

semantics of the Equals operator [14] .

6 Related Work

In [22], several approaches for model merging are presented. The *Merge* operator is a model management operator that was proposed in [3] and further developed in [4] afterwards. The specification of this *Merge* operator is provided in terms of imperative algorithms so that the *Merge* operator is embodied by a complex algorithm that mixes control logic with the functionality. Although the operator is independent of any metamodel, it depends on an external operator to check the constraints of a specific metamodel. Therefore, it might generate inconsistent models and requires an auxiliary operator to work properly. Moreover, as shown in [4], the algorithm may be changed depending on the metamodel. In MOMENT, the *Merge* operator remains completely reusable for any metamodel. To consider new metamodels, the operators *Equals* and *Resolve* can be customized by simply adding axioms to their respective semantic definition.

Another approach that provides the *Merge* operator from a model management standpoint is presented in [24] by using graph theory. The *Merge* operator is denotationally defined by means of production rules. In both operational and graph-based approaches, the *Merge* operator receives a model of mappings as input. This model indicates the relationships between the elements of the models that are going to be merged. These mappings have to be defined manually or can be inferred by a operator *Match* that uses heuristic functions [20] or historical information [19]. Our *Merge* operator does not depend on mappings since the equivalence relation is defined between elements of the same metamodel by means of the QVT Relations language, at a higher abstraction level. Another inconvenience of both model management approaches is that they are not integrated in any visual modeling environment. Therefore, they cannot be used in a model-driven development process in the way that the MOMENT framework is able to do through the Eclipse platform.

The Generic Model Weaver AMW [13] 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 [20]. This tool constitutes a nice solution when the weaving metamodel can change. It also provides the basis for a merge operator on the grounds that a weaving model, which is defined between two models, can be used as input for a model transformation that can obtain the merged model (as mentioned in [13]). In MOMENT, model weavings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators [5].

An interesting operation-based implementation of the three-way merge is presented in [1]. The union model that permits this kind of merging is built on top of a

[14] The definition of complex model management operators and trace model navigation is out of scope of this paper. We refer to [5] for further details.

difference operator. The difference operator is based on the assumption that all the elements that participate in a model must have a unique identifier. This operator uses the identifiers in order to check if two elements are the same. Our *Merge* operator is a state-based [15] implementation of the two-way merge so that it does not need a common base model in order to merge two different models. In our approach the operator *Equals* permits the definition of complex equivalence relationships in an easy way. The three-way merge can be specified as a complex operator in the Model Management arena, as described in [4].

More specifically to the problem presented in the case study, UML CASE tools permit the arrangement of Use Cases and their corresponding partial Class Diagram into the same package. Nevertheless, no option is provided to obtain the global Class Diagram from the partial ones. The Rational Rose Model Integration [16] is a tool that provides an ad-hoc solution to merge UML models by basically using the name of the element to determine equivalences, and using the preferred model strategy to obtain the merged model. The equivalence relation and the conflict resolution strategy cannot be customized by the user like in MOMENT. Moreover, once the merged model is generated, there is no way to relate the obtained model to the partial source models in order to keep some degree of traceability.

7 Conclusions

In this paper, we have presented a state-based automated approach for model merging from a model management standpoint. We have briefly introduced how we deal with algebraic models from a visual modeling environment, and we have described the generic semantics of our *Merge* operator.

The *Merge* operator can also be specialized to a metamodel by simply adding new axioms to the operator in an ad-hoc and more accurate way, taking advantage of both complementary standpoints: generic infrastructure and domain-specific knowledge. This specialization can be performed by means of standard QVT transformations that are then compiled into equations of the operators *Equals* and *Resolve*, which are used by the *Merge* operator. An example to specialize the *Merge* operator has been provided for the Ecore metamodel, in order to solve the integration of the partial class diagrams proposed in the case study. The operator takes advantage of the reusability and modularity features of the algebraic specifications. It becomes a scalable operator that can be easily specialized to a specific metamodel and that can be intuitively used with other operators. Thus, the user only has to know the standard QVT Relations language and the domain-specific knowledge on the grounds that the underlying formalism that is used in our approach remains completely hidden.

In the current version of the MOMENT framework, the specific semantics of the *Merge* operator is directly introduced using Maude code. The version of the *Merge*

[15] Software merging techniques can be classified as state-based or change-based [22]. State-based techniques only take into account the information that is embedded in the input software artifacts to be merged, while change-based techniques also use information about the precise changes that were performed during the evolution of the software.

operator that is presented in this work does not take into account the order that is defined among the elements of a model by means of ordered references. Thus, when two models are merged the elements of the merged model do not keep any order with regard to those of the input models. Currently we are extending the generic semantics of the Merge operator to preserve this order relation and we are developing a QVT Relations language compiler that targets Maude code, as shown in this paper and in [6].

8 Acknowledgments

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

References

[1] Alanen, M. and I. Porres, *Difference and union of models*, in: P. Stevens, J. Whittle and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, LNCS **2863** (2003), pp. 2–17.

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

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

[4] Bernstein, P. A. and R. A. Pottinger, *Merging models based on given correspondences*, in: *Proceedings of the 29th VLDB Conference*, Berlin, 2003.

[5] Boronat, A., J. A. Carsí and I. Ramos, *Automatic support for traceability in a generic model management framework.*, in: A. Hartman and D. Kreische, editors, *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*, Lecture Notes in Computer Science **3748** (2005), pp. 316–330.

[6] Boronat, A., J. A. Carsí and I. Ramos, *Algebraic specification of a model transformation engine.*, in: *FASE*, 2006, pp. 262–277.

[7] Boronat, A., J. Oriente, A. Gómez, J. A. Carsí and I. Ramos, *An algebraic specification of generic ocl queries within the eclipse modeling framework.*, in: *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006*, Lecture Notes in Computer Science **4066** (2006), pp. 316–330.

[8] Boronat, A., J. Pérez, J. A. Carsí and I. Ramos, *Two experiences in software dynamics*, Journal of Universal Computer Science **10** (2004), pp. 428–453.

[9] Budinsky, F., S. A. Brodsky and E. Merks, "Eclipse Modeling Framework," Pearson Education, 2003.

[10] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "Maude Manual (Version 2.2)," SRI International (2005).
URL http://maude.cs.uiuc.edu/maude2-manual/

[11] Clavel, M., F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. L. Talcott, *The maude 2.0 system*, in: R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, Lecture Notes in Computer Science **2706** (2003), pp. 76–87.

[12] Czarnecki, K. and U. W. Eisenecker, "Generative programming: methods, tools, and applications," ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[13] Didonet Del Fabro, M., J. Bézivin, F. Jouault, E. Breton and G. Gueltas, *Amw: a generic model weaver.*, in: *Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM05)*, 2005.

[14] EMF, E. O., *The eclipse modeling framework web site.*
URL http://www.eclipse.org/emf/

[15] Frankel, D., "Model Driven Architecture: Applying MDA to Enterprise Computing," John Wiley & Sons, Inc., New York, NY, USA, 2002.

[16] IBM, *Rational suite.*
URL http://www-306.ibm.com/software/sw-atoz/indexR.html

[17] Kruchten, P., "The Rational Unified Process: an introduction," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[18] Larman, C., "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process," Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[19] Madhavan, J., P. Bernstein, K. Chen, A. Halevy and P. Shenoy, *Corpus-based schema matching*, in: *Workshop on Information Integration on the Web, at IJCAI'2003*, 2003, pp. 59–66.

[20] Madhavan, J., P. A. Bernstein and E. Rahm, *Generic schema matching using cupid*, in: *Proc. VLDB 2001*, 2001, pp. 49–58.

[21] Mellor, S. J., S. Kendall, A. Uhl and D. Weise, "MDA Distilled," Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[22] Mens, T., *A state-of-the-art survey on software merging*, IEEE Transactions on Software Engineering **28** (2002), pp. 449–462.

[23] OMG, O. M. G., *Mof 2.0 qvt final adopted specification (ptc/05-11-01)* (2005).

[24] Song, G., K. Zhang and J. Kong, *Model management through graph transformation*, in: *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)* (2004), pp. 75–82.

[25] The ISSI Research Group, *The MOMENT Project.*
URL http://moment.dsic.upv.es

A Maude Notation for Operators and Conditional Equations

In a Maude module, an operator is declared with the keyword *op* followed by its name, followed by a colon, followed by the list of sorts for its arguments (called the operator's arity), followed by ->, followed by the sort of its result (called the operator's coarity), optionally followed by an attribute declaration, followed by white space and a period. Thus the general scheme has the form

op ⟨*OpName*⟩ : ⟨*Sort-1*⟩ ... ⟨*Sort-k*⟩ -> ⟨*Sort*⟩ [⟨*OperatorAttributes*⟩] .

Equations are declared using the keyword eq, followed by a term (its lefthand side), the equality sign =, then a term (its righthand side), optionally followed by a list of statement attributes enclosed in square brackets, and ending with a space and a period. Thus the general scheme is the following:

eq ⟨*Term-1*⟩ = ⟨*Term-2*⟩ [⟨*StatementAttributes*⟩] .

Conditional equations are allowed, where the equational conditions are made up of individual equations $t = t'$ and memberships $t : s$ (indicating a boolean function that holds true if the term t belongs to the sort s). A condition can be either a single equation, a single membership, or a conjunction of equations and memberships using the binary conjunction connective /\ which is assumed associative. Thus the general form of conditional equations and memberships is the following:

ceq ⟨*Term-1*⟩ = ⟨*Term-2*⟩
if ⟨*EqCondition-1*⟩ /\ ... /\ ⟨*EqCondition-k*⟩ [⟨*StatementAttributes*⟩] .

B Generated Signature for the Ecore Metamodel

fmod *sigecore* ***is***
pr DATATYPE .

sorts *ecore-EAttribute ecore-EAnnotation ecore-EClass ecore-EClassifier ecore-EDataType ecore-EEnum ecore-EEnumLiteral ecore-EFactory ecore-EModelElement ecore-ENamedElement ecore-EObject .*
sorts *ecore-EOperation ecore-EPackage ecore-EParameter ecore-EReference ecore-EStructuralFeature ecore-ETypedElement ecore-EEnumerator ecore-EFeatureMap ecore-EFeatureMapEntry ecore-EMap ecore-EResource ecore-EResourceSet ecoreNode .*

subsorts *ecore-EObject ecore-EStringToStringMapEntry < ecoreNode .*
subsorts *ecore-EAnnotation ecore-EFactory ecore-ENamedElement < ecore-EModelElement .*
subsorts *ecore-EEnum < ecore-EDataType .*
subsorts *ecore-EModelElement < ecore-EObject .*
subsorts *ecore-EAttribute ecore-EReference < ecore-EStructuralFeature .*
subsorts *ecore-EClass ecore-EDataType < ecore-EClassifier .*
subsorts *ecore-EOperation ecore-EParameter ecore-EStructuralFeature < ecore-ETypedElement .*
subsorts *ecore-EClassifier ecore-EEnumLiteral ecore-EPackage ecore-ETypedElement < ecore-ENamedElement .*

***** **op** *ecore-EAttribute: Qid, name, ordered, unique, lowerBound, upperBound, many, required, changeable, volatile, transient, defaultValueLiteral, defaultValue, unsettable, derived, iD, eAnnotations, eType, eContainingClass, eAttributeType,*
op *'(ecore-EAttribute _') : Qid String Bool Bool Int Int Bool Bool Bool Bool Bool String String Bool Bool Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EAttribute [ctor] .*

***** **op** *ecore-EAnnotation: Qid, source, eAnnotations, details, eModelElement, contents, references,*
op *'(ecore-EAnnotation _ _ _ _ _ _ _') : Qid String OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EAnnotation [ctor] .*

***** **op** *ecore-EClass: Qid, name, instanceClassName, instanceClass, defaultValue, abstract, interface, eAnnotations, ePackage, eSuperTypes, eOperations, eAllAttributes, eAllReferences, eReferences, eAttributes, eAllContainments, eAllOperations, eAllStructuralFeatures, eAllSuperTypes, eIDAttribute, eStructuralFeatures,*
op *'(ecore-EClass _') : Qid String String String String Bool Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EClass [ctor] .*

***** **op** *ecore-EDataType: Qid, name, instanceClassName, instanceClass, defaultValue, serializable, eAnnotations, ePackage,*
op *'(ecore-EDataType _ _ _ _ _ _ _ _') : Qid String String String String Bool OrderedSet {QID} OrderedSet {QID} -> ecore-EDataType [ctor] .*

***** **op** *ecore-EEnum: Qid, name, instanceClassName, instanceClass, defaultValue, serializable, eAnnotations, ePackage, eLiterals,*
op *'(ecore-EEnum _ _ _ _ _ _ _ _ _') : Qid String String String String Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EEnum [ctor] .*

***** **op** *ecore-EEnumLiteral: Qid, name, value, instance, eAnnotations, eEnum,*
op *'(ecore-EEnumLiteral _ _ _ _ _ _') : Qid String Int String OrderedSet {QID} OrderedSet {QID} -> ecore-EEnumLiteral [ctor] .*

***** **op** *ecore-EFactory: Qid, eAnnotations, ePackage,*
op *'(ecore-EFactory _ _ _') : Qid OrderedSet {QID} OrderedSet {QID} -> ecore-EFactory [ctor] .*

***** **op** *ecore-EObject: Qid,*
op *'(ecore-EObject _') : Qid -> ecore-EObject [ctor] .*

***** **op** *ecore-EOperation: Qid, name, ordered, unique, lowerBound, upperBound, many, required, eAnnotations, eType, eContainingClass, eParameters, eExceptions,*
op *'(ecore-EOperation _ _ _ _ _ _ _ _ _ _ _ _ _') : Qid String Bool Bool Int Int Bool Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EOperation [ctor] .*

***** **op** *ecore-EPackage: Qid, name, nsURI, nsPrefix, eAnnotations, eFactoryInstance, eClassifiers, eSubpackages, eSuperPackage,*
op *'(ecore-EPackage _ _ _ _ _ _ _ _ _') : Qid String String String OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EPackage [ctor] .*

***** **op** *ecore-EParameter: Qid, name, ordered, unique, lowerBound, upperBound, many, required, eAnnotations, eType, eOperation,*
op *'(ecore-EParameter _ _ _ _ _ _ _ _ _ _ _ _') : Qid String Bool Bool Int Int Bool Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EParameter [ctor] .*

***** **op** *ecore-EReference: Qid, name, ordered, unique, lowerBound, upperBound, many, required, changeable, volatile, transient, defaultValueLiteral, defaultValue, unsettable, derived, containment, container, resolveProxies, eAnnotations, eType, eContainingClass, eOpposite, eReferenceType,*
op *'(ecore-EReference _') : Qid String Bool Bool Int Int Bool Bool*

Bool Bool Bool String String Bool Bool Bool Bool Bool OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} OrderedSet {QID} -> ecore-EReference [ctor] .

```
    *** op ecore-EStringToStringMapEntry: Qid, key, value,
    op '(ecore-EStringToStringMapEntry _ _ _') : Qid String String
        -> ecore-EStringToStringMapEntry [ctor] .
endfm
```

C Generic Semantics for the *Merge* Operator

The *MergeRule* operator constitutes a rule of the *Merge* operator and indicates how to merge two elements when they are equivalent. The declaration is as follows:

```
op MergeRule : Set{X} Set{X} Set{X} Set{X}
    -> Tuple{ X, TraceabilityMetamodel, TraceabilityMetamodel } [memo] .
```

where the first two arguments represent the first input model to be merged (the first argument is traversed by recursion), and where the last two arguments represent the second input model to be merged (the third argument is also traversed by recursion). The output of the operator is a triple, where the first component is the model of merged elements, the second component is a trace model, whose traces relate instances of the first input model to intances of the merged model, and the third component is another trace model, whose traces relate instances of the second input model to intances of the merged model.

The MergeRule operator equations[16] specify the generic semantics of the merging process in the MOMENT Framework. The following equation applies the conflict resolution strategy (specialized if exists or generic by default) to all the elements that are equivalent in order to eliminate duplicates:

```
ceq MergeRule ( Set{N1,M1}, Model1, Set{N2,M2}, Model2) =
    (Resolve ( Set{ N1 }, Model1, Set{N2}, Model2))
        -> including (MergeRule ( Set{M1}, Model1, Set{M2}, Model2))
if (Equals( Set{N1}, Model1, Set{N2}, Model2)) .
```

The following equation constitutes the base case of the recursion when the models to traverse are empty:

```
eq MergeRule ( (empty-set).Set{X}, Model1, (empty-set).Set{X}, Model2) =
    (
        (empty-set).Set{X} ,
        (empty-set).Set{TraceabilityMetamodel} ,
        (empty-set).Set{TraceabilityMetamodel}
    ) .
```

The elements that belong to only one model, without being duplicated in the other one, are refreshed and directly copied into the merged model. The following equations perform this task:

```
eq MergeRule ( Model1, Model1b, (empty-set).Set{X}, Model2) =
    Refresh(Model1, Model1b, Model2) .

eq MergeRule ( (empty-set).Set{X}, Model1, Model2, Model2b) =
    Refresh(Model2, Model2b, Model1) .

eq MergeRule ( Set{M1}, Model1, Set{M2}, Model2) =
    (
        p1(Refresh(Model1, Model1b, Model2))
        -> including ( p1(Refresh(Model2, Model2b, Model1)) ) -> flatten ,
        p2(Refresh(Model1, Model1b, Model2))
            -> including ( p2(Refresh(Model2, Model2b, Model1)) ) -> flatten ,
        p3(Refresh(Model1, Model1b, Model2))
            -> including ( p3(Refresh(Model2, Model2b, Model1)) ) -> flatten
    ) [owise] .
```

The *Merge* operator takes two models that conform to the same metamodel as inputs. The outputs of the *Merge* operator are a merged model and two models of traces that relate the elements of the input models to the elements of the output merged model. The Merge operator invokes the MergeRule operator over the elements of the input elements and finally complete the obtained trace models:

```
eq Merge (Model1, Model2) =
    (
        p1(MergeRule ( Model1, Model1, Model2, Model2) )
    ,
        p2(MergeRule ( Model1, Model1, Model2, Model2) ) -> including (
```

[16] In the equations, the operators p1, p2 and p3 are projection operators that obtain the first, second and third element of a triple, respectively.

```
        Set{New("TraceabilityModel")
            :: operator <- "Merge"
            :: links <- ((
                (p2(MergeRule ( Model1, Model1, Model2, Model2)))
                -> asOrderedSet) :: OID)
        }
    ) -> flatten
,
    p3(MergeRule ( Model1, Model1, Model2, Model2) ) -> including (
        Set{ New("TraceabilityModel")
            :: operator <- "Merge"
            :: links <- ((
                (p3(MergeRule ( Model1, Model1, Model2, Model2)))
                -> asOrderedSet) :: OID)
        }
    ) -> flatten
) .
```