Formal Model Merging Applied to Class Diagram Integration

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

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. We focus on the definition of our operator *Merge*, applying it to Class Diagrams integration.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: *Computer-aided software engineering, Evolutionary prototyping, Object-oriented design methods*

D.2.7 [**Distribution, Maintenance, and Enhancement**]: *Restructuring, reverse engineering, and reengineering*

D.2.13 [Reusable Software]: *Reuse models*

I.1 [**SYMBOLIC AND ALGEBRAIC MANIPULATION**]: I.1.4 Applications

I.6.5 [**Model Development**]: *Modeling methodologies*

General Terms

Design, Experimentation, Languages.

Keywords

Model-Driven Engineering, Model Management, model merging, conflict resolution.

1. INTRODUCTION

The Model-Driven Development philosophy [1] 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 [2]. In this process, models constitute software artifacts that experience refinements from the problem space (where they capture the requirements of the application) 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 [3] 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 [4], 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) [22], which is embedded into the Eclipse platform [5] and that provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF) [6]. 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 [7]. 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 informally 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

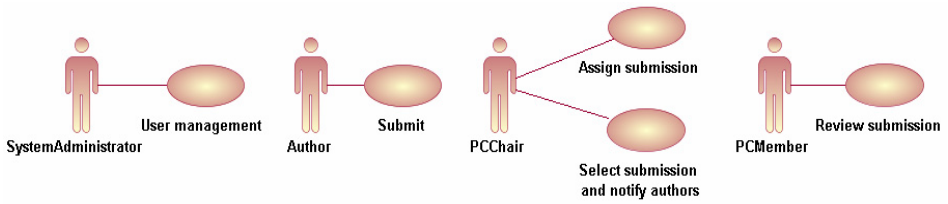

Figure 1. Use Case Model

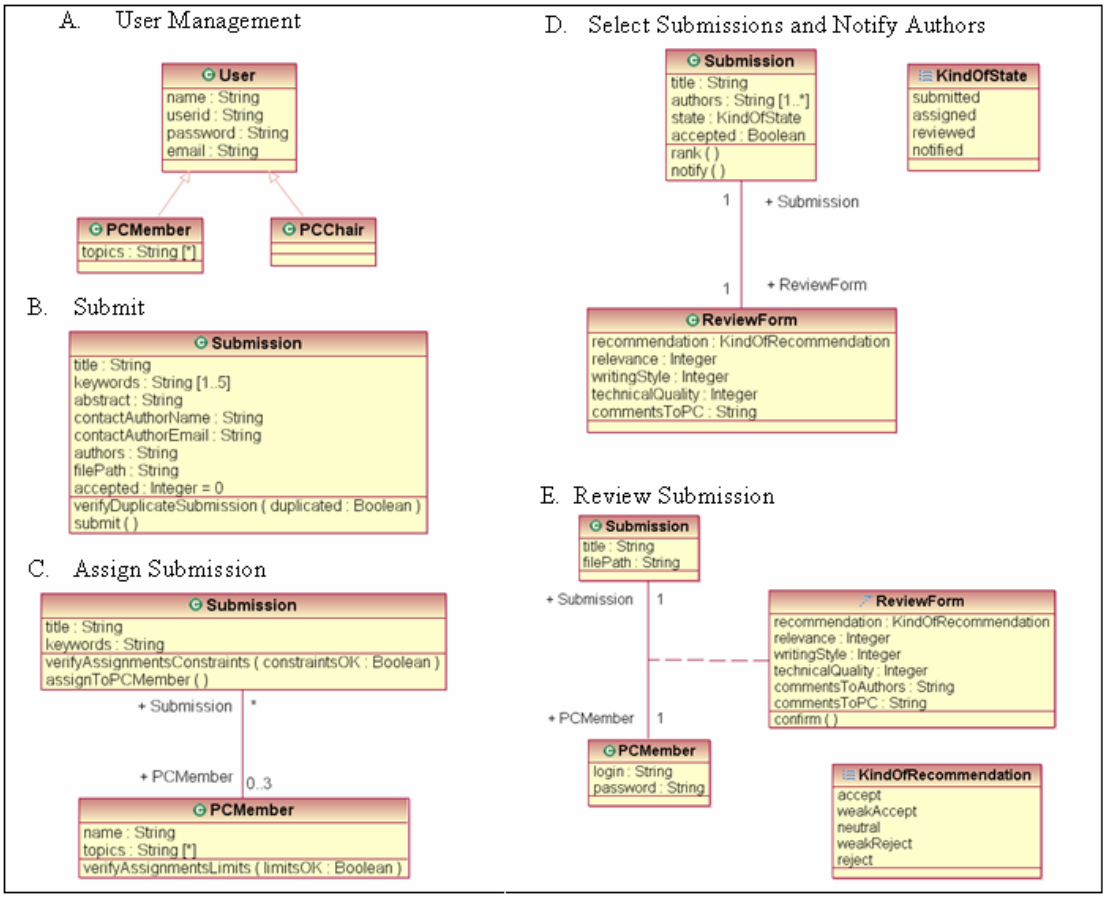

Figure 2. Partial models associated to the corresponding Use Cases

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 [8, 9]. 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 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 Figure 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. Figure 2 shows the Class Diagrams that support the functionality required for the corresponding Use Case.

3. THE GENERIC SEMANTICS OF THE OPERATOR *Merge*

In a Model-Driven Development context [10], 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 of the metamodel. To fulfill this in our framework, the Ecore metamodel can be broken down into three well-distinguished parts:

1. A parameterized module called *MOMENT-OP*, which provides our generic model management operators independently of any metamodel (the operator *Merge* is one of them). This module also provides the needed constructors to specify a model as a set of elements, based on an axiomatized specification of a set theory.
2. A signature called *sigEcore*, which provides the constructors of a specific metamodel which is specified in order to represent a model by means of algebraic terms. For example, in the *Ecore* metamodel algebraic specification, we have the constructs that define a class, an attribute, an operation, and so on. This signature is automatically generated from a metamodel (see [23] for further details), which is specified by using the meta-metamodel Ecore in the EMF. In this case, the proper metamodel Ecore has been used to define itself. This signature constitutes the actual parameter for the module MOMENT-OP.
3. A module called *spEcore*, which instantiates the parameterized module *MOMENT-OP* by passing *sigEcore* as actual parameter. In the instantiation process, the generic operators are customized to the constructs of the metamodel. This provides the constructors that are needed to specify a model of this metamodel as a set of elements. This fact also provides the generic operators that can be automatically applied to models of this kind. In this module, the specification of the operator *Merge* can also be customized to a metamodel by simply adding new axioms to the operators. This module constitutes the algebraic specification of a metamodel in MOMENT. To enable the manipulation of UML models, they have to be represented as terms of the spEcore algebraic specification. This task is automatically performed by MOMENT from models expressed in Ecore format in the EMF.

In MOMENT, the operator *Merge* 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 operator can also be customized in an ad-hoc and more accurate way, taking advantage of both complementary standpoints.

3.1 The Generic Semantics of the Operator *Merge*

The operator *Merge* 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 operator *Merge* 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 operator *Merge*, we need to introduce three concepts: the equivalence relationship, the conflict resolution strategy and the refreshment of a construct.

First, a semantic equivalence relationship is a bidirectional function between elements that belong to different models but to the same metamodel. This indicates that they are semantically the same software artifact although they may differ syntactically. This relation is embodied by the operator *Equals*. The generic semantics of *Equals* coincides with the syntactical equivalence, although this generic semantics can be enriched by means of OCL expressions that take into account the structure and semantics of a specific metamodel.

Second, we have to deal with conflicts. During a model merging process, when two software artifacts (each of which belongs to a different model) are supposed to be semantically the same, 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 operator *Merge* 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 semantically equivalent due to the *Equals* morphism, although they differ syntactically, the elements of the preferred model prevail. The semantics of the Resolve operator can also be customized for a specific metamodel in the same way that we can do with the Equals operator.

Third, 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*.

The operator *Merge* uses the equivalence relationship 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

	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	<BCDE,$map_{DE2BCDE}$, $map_{BC2BCDE}$ > = Merge(DE, BC)	3.1. The multiplicity of the attribute *authors* (class *Submission*)	B – E	Multiplicity [1..*] (preferred model)
		3.2. Type of the attribute *accepted* (class *Submission*)	B – E	Type Boolean (preferred model)
		3.3. Multiplicities of the association between the classes *Submission* and *PCMember*	C – D	Multiplicities 1..1 – 1..1 (preferred model)
4	<ABCDE, $map_{A2ABCDE}$, $Map_{BCDE2ABCDE}$> = Merge(A, BCDE)	4.1. The attribute *userid* (class *User*) and the attribute *login* (class *PCMember*) are identified as the same, by means of the 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

```
ceq Equals N1 Model1 N2 Model2 =
  if ((N1.name) == (N2.name)) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
  else if (Synonym (N1.name) (N2.name)) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
    else if (Similar (N1.name) (N2.name) 95.0) and (Equals (N1.eContainingClass(Model1)) Model1 (N2.eContainingClass(Model2)) Model2) then true
      else false
  fi fi fi
if (N1.oclIsTypeOf(.?."EAttribute"). Model1)) and (N2.oclIsTypeOf(.?."EAttribute". Model2))          *** Condition
```

Figure 3. Equivalence relationship for the EAttribute primitive

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 outputs of the operator *Merge* are a merged model and two models of mappings that relate the elements of the input models to the elements of the output merged model. Therefore, these mappings, which are automatically generated by the operator *Merge*, provide full support for keeping traceability between the input models and the new merged one.

4. SPECIFIC SEMANTICS FOR THE ECORE METAMODEL TO MERGE UML CLASS DIAGRAMS

In this section, we present the specific semantics of the operator *Merge* to integrate UML Class Diagrams, which are implemented in the EMF by means of the Ecore metamodel. To define the specific semantics for the Ecore metamodel, we only have to add specific axioms for the operators *Equals* and *Resolve*. Consequently, the axiomatic definition of the operator *Merge* remains the same. The equivalence relationships that relate two elements of the metamodel Ecore take into account the type of a construct, its name, and its container. In an equivalence relationship, the names of two instances that have the same type are analyzed in three steps[1]: two instances may be equal if they have exactly the same name; if not, two instances may be equal if their names are defined as synonyms in a thesaurus; if not, they may be equal if a heuristic function for comparing strings establishes that they are similar within an acceptable range.

Moreover, almost all the relationships add a container condition. This means that two instances of the same type are equal if, in addition to the name condition, they have an equivalent container instance. In Figure 3, we provide the conditional equation that represents an equivalence relationship for EAttribute primitive in MAUDE notation. This equation is automatically obtained by compilation of an OCL expression [23]. In the merging of the partial models B and C of our case study, when this equivalence relationship is applied, the attribute *title* of the class *Submission* (of model B) is equivalent to the attribute *title* of the class *Submission* (of model C) because they have the same name and they belong to equivalent classes.

Several axioms have also been added to the *Resolve* operator in order to take into account the constructs of the Ecore metamodel that contain other elements. For example, a class can contain attributes, references and operations, among others. 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.

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 operator *Merge*; the third column describes some of the main conflicts that have appeared during the merging step; the fourth column indicates the partial models

[1] We have chosen these principles 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 the determine which elements are equals or not

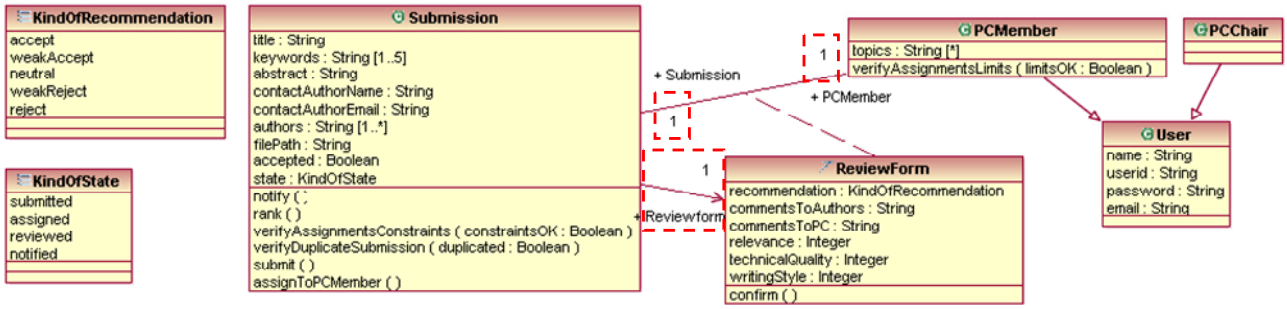

Figure 4. Resulting merged model for the case study.

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: from a partial model to the merged model (providing support for the propagation of changes from 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 Figure 4, we show the resulting merged model resulting from step 4. Although the user describes the semantics of the operator *Merge* 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 merged model to review and update it. Merged models can be manipulated from visual editors that are integrated in the Eclipse platform, such as the EMF tree editor or the Omondo tool [11], which provides a visual environment to edit UML models based on the Ecore metamodel. Other industrial modeling environments can be used to manipulate resulting models such as Rational Software Architect [12]. These environments provide the added value of code generation and the integration with other IBM software development tools. Furthermore, in the MOMENT framework, the operators that work on mappings can be used to follow the performed merging process in order to automatically detect changes in properties of elements like cardinality, type, addition of attributes to a class, etc.

6. RELATED WORK

In [21], several approaches for model merging are presented. The operator *Merge* is a model management operator that was proposed in [13] and further developed in [14] afterwards. The specification of this operator *Merge* is provided in terms of imperative algorithms so that the operator *Merge* 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 [14], the algorithm may be changed depending on the metamodel. In MOMENT, the operator *Merge* 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, preserving monotonicity.

Another approach to provide the operator *Merge* from a model management standpoint is presented in [15] by using graph theory. The operator *Merge* is denotationally defined by means of production rules. In both operational and graph-based approaches, the operator *Merge* 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 [16] or historical information [17]. Our operator *Merge* does not depend on mappings since the equivalence relation has been defined axiomatically between elements of the same metamodel in the operator *Equals*, 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 [19] 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 [16]. 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 [19]). In MOMENT, model weavings are generated by model management operators automatically in a traceability model, and can be manipulated by other operators.

An interesting operation-based implementation of the three-way merge is presented in [20]. The union model that permits this kind

of merging is built on top of a 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 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 [14].

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 [18] 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 operator *Merge*. A customization of the operator has been performed 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. Therefore, it becomes a scalable operator that can be easily specialized to a specific metamodel and that can be intuitively used with other operators. As we have shown in our case study, our approach provides support for maintaining traceability between the Use Case model and the Analysis model. The MOMENT framework offers other operators that enable the synchronization of changes between both models, wherever the changes occur, either in the partial models or in the merged models.

In the current version of the MOMENT framework, the specific semantics of the operator *Merge* is directly introduced using the Maude syntax. In future work, we plan to develop visual interfaces to define the axioms needed to customize the MOMENT operators in order to improve the usability of our tool.

8. REFERENCES

[1] Frankel, D. S.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons OMG Press.

[2] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications.Addisson-Wesley (2000). ISBN 0-201-30977-7, pag. 267-304.

[3] Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.

[4] Boronat, A., Pérez, J., Carsí, J. Á., Ramos, I.: Two experiencies in software dynamics. *Journal of Universal Science Computer*. Special issue on Breakthroughs and Challenges in Software Engineering. Vol. 10 (issue 4). April 2004.

[5] Eclipse site: www.eclipse.org

[6] The EMF site: http://download.eclipse.org/tools/emf/scripts/home.php

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

[8] Kruchten P. The Rational Unified Process: An Introduction. Addison-Wesley Professional. 2003.

[9] Larman C. Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall. 2004.

[10] Mellor, S. J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley (2004). ISBN 0-201-78891-8.

[11] The Omondo site: www.omondo.com

[12] IBM Rational Software Architect: http://www-306.ibm.com/software/awdtools/architect/swarchitect/

[13] Bernstein, P.A., Levy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. MRS Tech. Rep. MSR-TR-2000-53, (in SIGMOD Record 29, 4 (Dec. '00)).

[14] Pottinger, R.A., Bernstein, P. A.: Merging Models Based on Given Correspondences." VLDB 2003.

[15] Song, G., Zhang, K., Kong, J.: Model Management Through Graph Transformation. IEEE VL/HCC'04. Rome, Italy. 2004.

[16] Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. VLDB 2001.

[17] Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P.: Corpus-based Schema Matching," Workshop on Information Integration on the Web, at IJCAI'2003,pp.59-66.

[18] Rational Suite: http://www-306.ibm.com/software/sw-atoz/indexR.html

[19] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, and Gueltas, G : AMW: a generic model weaver. Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). 2005.

[20] Alanen, M, and Porres, I: Difference and Union of Models. In UML 2003 - The Unified Modeling Language, Oct 2003.

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

[22] The MOMENT site: http://moment.dsic.upv.es/

[23] Boronat, A., Ramos, I., Carsí J.A.: Definition of OCL 2.0 Operational Semantics by means of a Parameterized Algebraic Specification. 1st workshop on Algebraic Foundations for OCL and Applications (WAFOCA'06) Valencia, Spain, march 22nd, 2006.