Model Comparison: A Strategy-Based Approach

Kleinner Oliveira, Toacy Oliveira
Informatics Faculty
Pontifical Catholic University of Rio Grande do Sul
Ipiranga Avenue 6681 - Building 32 - ZIP 90619-900
Porto Alegre - Brazil
{ksoliveira,toacy}@inf.pucrs.br

***Abstract*— With the emergence of Model Driven Architecture (MDA), the role of model composition has become very important. One challenge of model composition is specifically to merge models expressed in the Unified Model Language (UML) and its profiles. However, for merging it is necessary to perform an essential task: *model comparison*. In this paper, we present a model comparison technique that relies on match strategies so that input models can be merged if they are considered equivalent according to a specific *match strategy*. To put this in practice we defined a *match operator* that makes use of match rules, synonym dictionary and typographic similarity. Moreover, a guidance for model comparison was elaborated to specify the activities that go along with model comparison.**

I. Introduction

A significant factor behind the difficulty of developing complex software is the wide conceptual gap between the problem and the domains of discourse [3], [4]. The model-driven approaches move development focus from third generation programming language code (e.g. Java code) to models, specifically models expressed in the Unified Model Language (UML) and its profiles [14], [16]. The goal is to manage the software at the level of its concepts in order to reduce the gap, quickly attain code and become the software development less difficult and costly. One reference to these approaches is the Model Driven Architecture (MDA) [10], an approach to Model Driven Development (MDD) from Object Management Group (OMG).

A typical MDA process involves a number of UML models to graphically represent a system's structure and behavior often defined in different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling) from a specific viewpoint and at a certain abstraction level that can be ultimately converted into the actual code by a model transformation engine. It can use models not only horizontally to describe different system aspects but also vertically, in order to be refined from higher to lower abstraction levels. Thus, the model-driven approaches make use of model transformation and model composition techniques to manipulate and manage UML models at the same and different abstraction levels. Models can represent concepts related to the system domain such as Telecom and Insurance, and also exposes the underlying execution infra-structure such as .NET or Java, which means a typical system can be represented by several models that must be somehow assembled (composed) into a cohesive unit.

The model composition can be viewed as an operation where a set of activities should be performed to merge two input models, M_A (receiving) and M_B (merged), in order to produce an output model, M_{AB}. In short, we can represent it by the equation: M_A+$M_B \rightarrow M_{AB}$. However, an important step to achieve model composition lays in the ability to compare input model elements, thus before merging M_A and M_B, it is necessary to compare to verify semantic and syntactic overlap in such models. The need to avoid such overlaps stands for the fact that the ultimate system's model should represent each concept uniquely to avoid conflicts, misinterpretation and mistransformation. For example, according to UML metamodel specification should not exist two (or more) models (e.g., two UML classes) with equal names in a same namespace, then a model composition mechanism should take in account such conditions to produce the output model, otherwise it can have conflicting names and elements with same semantic value.

In this paper we demonstrate the role and the importance of model comparison in model composition, describe the challenges that should be tackled to compare models and propose a *match operator* that is responsible for putting in practice a *strategy-based model comparison approach*. Moreover, a brief guidance for model comparison is exposed in order to specify the activities that go along with model comparison.

A. Motivating Example

We motivate our work with a composition example of two UML profiles, *Tree* and *Topology* [2] (see Figure 1) each representing a Domain-Specific Modeling Language (DSML). We have chosen UML profiles because they play a central role in the OMG's MDA approach. The Tree profile represents a common hierarchical data structure used for many computer science applications, while the Topology profile represents the connections between the elements of an Information System with a star network topology.

In the Topology profile, we have nodes (represented by stereotype Node) connected by links that can be local (LocalEdge) if they connect nodes from the same star with its central node, or remote (Edge) if they connect central nodes (MainNode) between each other [2]. Each node is identified by its position (location) and each central node has a state kind (state) that defines their availability (its values are defined by enumeration StateKind). An

end node (EndNode) is also identified by its position (position). The Tree profile has nodes (represented by stereotype Node) connected by links (Edge) to node, end node (Leaf) or root node (root) that has a state kind (state) which defines their availability (its values are defined by enumeration StateKind). Each node is determined by its name (name) and value (value). Moreover, it is possible to perform search operation (Search).

Before merging Tree and Topology, we should necessarily compare the input profiles in order to merge such profiles efficiently. To do this, we need to be able to identify correspondences among UML profile elements in a coherent manner. For example, despite the *Tree.Leaf* and *Topology.EndNode* stereotypes have different names, could they be considered domain concepts of equal semantic values?

B. Contributions of this Paper

To put model comparison in practice involves answering several model comparison questions. As stated in [9], what criteria should we use for identifying correspondences between different models? And how can we quantify these criteria? Considering two input models, should the model comparison techniques produce only one possible result that representing the correspondence among their elements? What properties of the input models should be considered in their match? What should be used so that we can compare models?

The answers for such questions are the contributions of this paper that consist in the definition of a flexible model comparison technique based on *match strategies*. The strategies are implemented by a *match operator* that uses of a range of heuristics including typographic similarities, equivalence among the semantic values of the input model elements and model signature. We propose a brief guidance to specify as conduct the model comparison process. Our approach is constituent of a UML profiles composition mechanism [12] that was shown to be an effective and flexible way for specifying correspondences among UML profiles. Moreover, we specify the approach using the formal specification language Alloy [5] and its tool (the Alloy Analyzer) in order to realize an automatic analysis of the approach.

The remainder of the paper is organized as follows. Section 2 briefly describes the background and the major challenges that researchers face when attempting to realize model comparison. Section 3 presents the our approach based on match strategies and the definition of the match operator. Section 4 presents a brief guidance for model comparison. Section 5 describes the related work. Finally, Section 6 shows some early conclusions and future works.

II. Background and Challenges

Model comparison arises as an essential activity to put the composition in practice and it can be viewed as a generic operation that varies from application to application, in which elements from M_A and M_B are compared in different forms depending of the kind of application. For example, the matching of statechart specifications [9] and of different versions of UML diagrams [11] presents particularity because the artifacts, that are being compared, have different properties, so the model comparison technique is tailored in agreement to them.

The UML specification [14] defines and presents the modelers with the *Profile mechanism* has been specifically specified for providing a lightweight extension mechanism to the UML standard. For instance, we can add semantics that is left unspecified in the metamodel, give a terminology that is adapted to a particular platform or domain and add information that can be used when transforming a model to another model or code.

However, the UML built-in composition mechanism, package merge, is not able to merge profiles or compare the input models correctly. So some research questions arise: how can we compare two profile elements? What activities should we perform to match two input models? Once we have added semantics that does not exist to a UML metamodel element, how can we compare it in a flexible manner?

To the best of our knowledge, the need for comparing models in a flexible manner neither have been pointed out nor even proposed by current model comparison techniques in the model composition mechanisms. This fact shows the pioneer side of this work.

Based on previous works [13], [12] and relevant approach studied (described in Section V), we observed and concluded that the major challenges that researches face when attempting to put into practice the model comparison in the context of MDD can be grouped into the following categories:

- The domain-specific model comparison challenge: Such challenge arises from concerns associated with providing DSMLs for creating and using domain-specific models in the MDD vision. For example, the UML supports two forms of extensions: (1) using profiles to define UML variants and (2) associating particular semantics to specified semantic variation points [14], [4]. Hence, a challenge would be how to develop support for tailoring the model comparison techniques to the semantics plugged into UML semantic variation points and the specializations of the UML metamodel specified by the profiles
- The abstraction level challenge: Once the MDD vision manipulates models in different abstraction levels, how should the model comparison techniques provide support for matching models expressed in different abstraction-level? This challenge poses its problems with respect to understanding and evolving the model comparison techniques across different modeling languages, where each one has its particularity.
- The semantic and properties challenge: As the models have a semantic value associated with it, a pair of them with the same name under matching packages could be assumed to form a match. However, what should be done if they have different semantic values or different properties? For example, two input UML

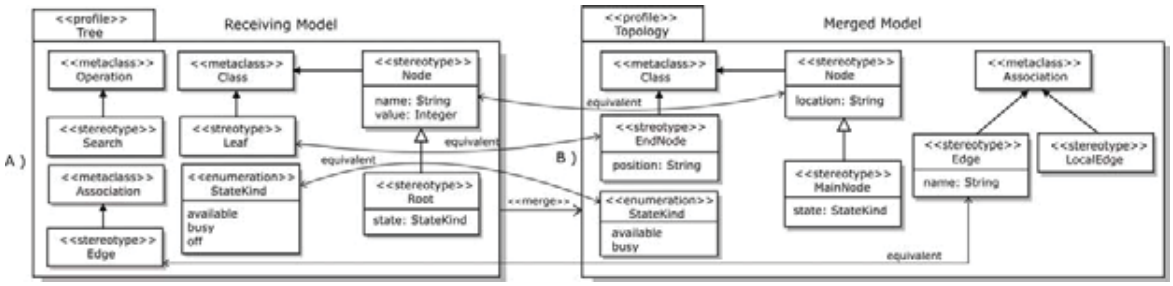

Fig. 1. Example of UML profiles comparison

classes with same name, however one is *abstract* and the other is *concrete*. While the pair of classes may still be considered a match, there is a conformance mismatch between them.

III. Strategy-Based Model Comparison

Having explained a motivation example and defined the challenges of model comparison we present, in this section, a flexible model comparison approach based on *match strategies*. We specified three strategies (i) *default*, (ii) *partial* and (iii) *complete match strategy*; however, new strategies may be created and inserted in our approach as well. We also define a *match operator* that is responsible for putting the strategies in practice together. From input models and the match strategy specification, the match operator verifies the equivalence degree among the input model elements and according to a *threshold* specifies the match models.

A. The Match Operator

The match operator is a heuristic and its goal is to find correspondences among model elements founded in static matching and to implement the *match strategies*. The static matching uses *synonym dictionary*, *model signature* and *typographic similarity* among input model elements in order to define the *equivalence degree* ($\mathcal{S}$).

With a *synonym dictionary* it is possible to make a mapping among the domain concepts that have the same semantic values. The *synonym dictionary* paves the way to the domain specialists to apply their domain expertise in the matching process, once they have defined what concepts are synonyms. Hence, this fact improves the result of the comparison. We denote by $\mathcal{D}(r,m) \rightarrow [0,1]$ the degree of similarity between receiving (r) and merged (m) model elements, it returns 0 whether r and m are synonym, otherwise it returns 1. $\mathcal{D}$ is calculated for every possible pair of (r,m). Initially, every pair (r,m) of input model elements are assumed to be not a synonym, then $\mathcal{D}(r,m) = 0$ for every pair of (r,m). For instance, according to synonym dictionary (see Table I) the stereotypes *Tree.Leaf* and *Topology.EndNode*, depicted in Figure 1(a), represent the same concepts, therefore $\mathcal{D}(Leaf,EndNode) = 1$.

The goal of *typographic similarity* is to determinate $\mathcal{T}(r,m) \rightarrow$ [0..1] to every possible pairs of receiving (r) and merged (m) model elements. The N-gram algorithm [8] is applied to assign a similarity value in [0..1] to every possible pairs of (r,m). These pairs are defined by cartesian product of (R×M), where R and M are the set of receiving and merged model elements, respectively. The result of this is the matrix shown in Figure 2. This algorithm yields a similarity degree to a pair of strings based on counting the number of their identical substrings of length N (we use N = 2).

The *signature* is defined in terms of model element syntactic properties, where a syntactic property of a model element defines its structure. The signature is a collection of values for a subset of syntactic properties in a model element's metamodel class. For example, *isAbstract* is a syntactic property defined in the metamodel class called *Class*. If an instance of a *Class* is an abstract class then *isAbstract = true* for the class, otherwise the instance is a concrete class, *isAbstract = false*. The set of syntactic properties used to determine a profile element's signature is called *signature type*, as defined in [15]. A signature that consists of all syntactic properties associated with a model element is called *complete signature type*, based on a range of syntactic properties is called *partial signature type* and the signature only based on name is called *default signature type*.

The signature is structured in comparison levels organized hierarchically. For instance, in Figure 1, a possible definition of levels for the stereotype *Tree.Node* would be: *Tree.Node* (name) (level 2), with *Tree.Node.name* and *Tree.Node.value* (tagged values) (level 1). Every profile element type has one signature which is defined for it.

TABLE I
Example of synonym dictionary

Name	Synonym
Leaf	EndNode, FinalNode
Edge	Border, Limit, Margin
Search	Research, Searching, Query

The similarity degree based on signature $\mathcal{M}$ between receiving (r) and merged (m) model element $\mathcal{M}$(r,m) is defined by computing the weighted average between the arithmetic average of the levels (see Equation 1):

$$\mathcal{M} = \frac{\sum_{i=1}^{n} p_i \cdot \left[\sum_{j=1}^{k} \frac{\varphi_{i,j}}{k} \right]}{\sum_{i=1}^{n} p_i} \rightarrow [0..1] \quad (1)$$

- n is the number of levels employed to compare the elements, where $n \geq 1$ and $n \in \mathbb{N}_+^*$.
- p_i represents the weight, being $p_i = i$, where $i \geq 1$ and $i \in \mathbb{N}_+^*$; k expresses the number of elements in each level, where $k \geq 1$ and $k \in \mathbb{N}_+^*$ (i.e. *Tree.Node* has two properties, as these properties represent a level, so $k = 2$);
- $\varphi_{i,j}$ (i and j represent the level and item of model elements that are being compared, respectively) is used to denote if an item of receiving model element (i.e., *name:Strig* in *Tree.Node*) is equivalent to another item of merged model element. It is a boolean variable and we use the match rules (described as follows) in order to assign value to it. The match rules compare items of model elements, so it returns 1 if the rule is satisfied, otherwise it returns 0. For instance, when we compare the *Tree.Root* and *Topology.MainNode* stereotypes, $\varphi_{2,1} = 0$, applying the match rule MR1, and $\varphi_{1,1} = 1$, applying the match rule MR3.

We denote by $\mathcal{S}$ the degree of similarity between receiving (r) and merged (m) model elements. To define the similarity degree it is necessary to combine the partial similarity degrees. To do this, it is calculated the average of $\mathcal{D}$, $\mathcal{T}$, and $\mathcal{M}$, as showed in Equation 2. If $\mathcal{D} = 1$, then $\mathcal{T}$ also assumes value 1 and contrariwise.

$$\mathcal{S} = \frac{(\mathcal{D} + \mathcal{T} + \mathcal{M})}{\mathcal{D} + 2} \rightarrow [0..1] \quad (2)$$

Based on the Equation 2, we compute the similarity degree of every *Tree* elements in related to *Topology* elements. The Figure 2 shows the match results. To produce a correspondence relation between the two models, we set a threshold ($t = 0.7$). So, pairs of model elements with similarity degree above threshold are considered equivalent. In short, if $\mathcal{S}(r,m) > t$, then r and m are *equivalent*. In Figure 2, we point out the similarity degree above threshold and define the profile elements are equivalent, as follows: (Tree.Node, Topology.Node), (Tree.Edge, Topology.Edge), (Tree.Leaf, Topology.EndNode) and (Tree.StateKind, Topology.StateKind)

		Topology Profile					
		Node	MainNode	Edge	LocalEdge	EndNode	StateKind
Tree Profile	Node	0,83	0,22	0	0,08	0	0
	Root	0	0,05	0	0	0	0
	Edge	0	0	1	0,22	0	0
	Search	0	0	0	0	0	0
	Leaf	0	0	0	0	1	0
	StateKind	0	0	0	0	0	0,96

similarity degree above the threshold ($t = 0.7$)

Fig. 2. Similarity degree between profile elements

B. Match rules

In order to check if two input model element are equivalent, we defined *match rules*. The match operator is responsible to execute these match rules and, according to the resulting of this execution, it defines consequently the value of $\varphi_{i,j}$, which was specified earlier. For every model element and item of model element are necessary a match rule to check if they are equivalent. This checking is based on their signature. If a match rule fails, then the models are not equivalent ($\varphi_{i,j} = 0$). Otherwise, models are equivalent ($\varphi_{i,j} = 1$). The match rules verify whether the input model element properties have the same values, and for each match strategy is defined a set of match rule according to respective signature type of the strategy.

There are three kinds of match rules: (i) *default match rules* are a set of rules that compare models based on only their name, using the default signature type; (ii) *partial match rules* are also a set of rules that compare models based on a number of syntactic properties of the models, using the partial signature type; (iii) *complete match rules* are also a set of rules that compare models based on their syntactic properties, using the complete signature type. Thus, the match operator makes use of these rules to implement the default, partial and complete match strategies. For example, the match operator makes use of the default match strategy (hence using default match rules) to produce the similarity table depicted in Figure 2.

Now, we present a short description of the default match rules used in the motivation example, as follows:

MR1. Stereotype match rule:
MatchStereotype(Stereotype rcv, Stereotype mrgd) → rcv.name = mrgd.name AND MatchAttribute(rcv, mrgd) AND MatchOperation(rcv, mrgd)

MR2. Association match rule:
MatchAssociation(Association rcv, Association mrgd) → (rcv.name = mrgd.name) AND (rcv.memberEnds = mrgd.memberEnds)

MR3. Attribute match rule:
MatchAttribute(Stereotype rcv, Stereotype mrgd) → (rcv.ownedAttribute.name = mrgd.ownedAttribute.name) AND (rcv.ownedAttribute.TypedElement = mrgd.ownedAttribute.TypedElement)

MR4. Operation match rule:
MatchOperation(Stereotype rcv, Stereotype mrgd) → (rcv.
ownedOperation.name = mrgd.ownedOperation.name) AND (rcv.ownedOperation.ownedParameter.length = mrgd.ownedOperation.ownedParameter.length) AND (∀x(rcv.ownedOperation.ownedParameter[x] = mrgd.ownedOperation.ownedParameter[x])

MR5. Enumeration match rule:
MatchEnumeration(Enumeration rcv, Enumeration mrgd) → rcv.name = mrgd.name AND MatchEnumerationLiteral(Enumeration rcv, Enumeration mrgd)

MR6. Enumeration Literal match rule:
MatchEnumerationLiteral(Enumeration rcv, Enumeration mrgd) → ∀x(rcv.ownedLiteral.name[x] = mrgd.ownedOperation.name[x])

IV. A Guidance for Model Comparison

There is little agreement on requirements, activities and steps that should be followed in order to accomplish the model comparison, and even less on good practices to avoid errors during matching. Several works (e.g., see [7], [11]) have been proposed to tackle the problems found in model comparison, but none of them, as yet, was defined as standard. In [14], the UML built-in model comparison technique does not present a task flow to help the comparison specification of UML models, does not present a good documentation, and does not define how model comparison should be performed.

We previously identified and delegated activities to the match operator. We aim to successfully order and provide a flow of how such activities are accomplished. Such flow can be used as a *guidance* to compare models, and it aims to represent good practices and become as comprehensive as possible the match operator role in the model comparison process.

The guidance is organized in two phases: (1) *initial* and (2) *comparison phase*. The *initial phase* is started up when the matching operator receives the input models. The match operator analyzes the models in order to know each type (i.e. Stereotype, Class, Association, etc). Such models are separated and grouped according to their types. For example, Stereotypes (Tree.Node and Topology.Node) and Association (Tree.Edge and Topology.Edge) are identified and grouped according to their types.

The goal of the *comparison phase* is to define what input model elements are equivalent. It is initially realized as an analysis of the input models and a signature is defined for every model element type. The next step is to specify the match strategy that determines how the comparison will be accomplished. The match operator defines the similarity degree ($\mathcal{S}$) for every receiving and merged model element, and based on a threshold (t) finally it determines model elements are equivalent. The phase is finished as soon as the matching models, no-matching models and matching description are specified. The next step is to merge the models, however this activity is not the focus of this paper.

V. Related Work

The model comparison is applied in different domains and contexts, and plays a central role in numerous applications, such as model composition, schema integration, schema evolution and migration, merging of source code, application evolution, database integration, differences between XML documents, and differences between versions of UML diagrams. Thus, previous research works have proposed many techniques to tackle the inherent problems related to matching, and achieved an automation degree of the match operation for specific application domains. We

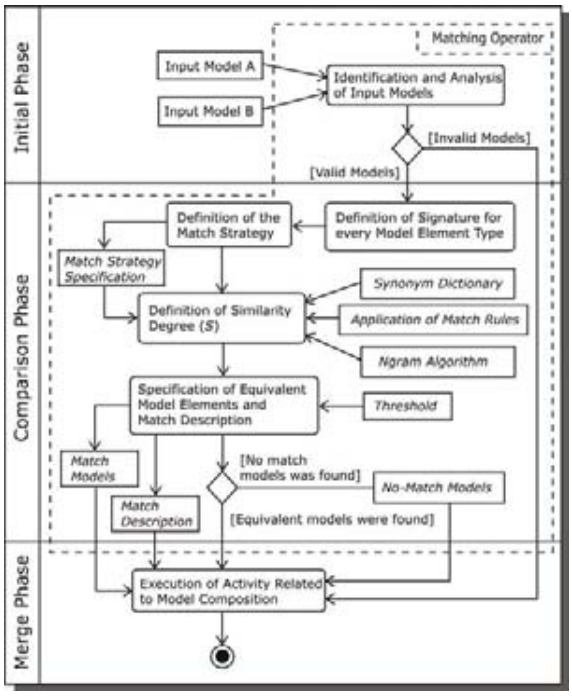

Fig. 3. A guidance for model comparison

give an overview on other relevant approaches related to our goals of putting flexibility into the model comparison process and analyze others that make use of model comparison to merge models. To do this, the main focus of each approach is summarized briefly, followed by pointing out similarities and differences to our own approach (see Figure 4).

Model Composition Semantics. S. Clarke [1] introduces composition semantics for UML class diagrams. The approach defines a new design construct, called *composition relationship* that supports the specification of how design models should be composed. With this *composition relationship* it is possible to: (*i*) identify and specify overlapping and non-overlapping concepts; (*ii*) specify how models should be integrated, and how conflicts in equivalent elements are reconciled. The identification of the overlapping parts is based on the name of the input models; it is a weakness of the approach.

Model Composition Directives. Reddy et al. [15] present a model composition technique relies on signature matching, in which model elements are merged if their signatures are correspondent. However, the match operator, in our work, makes use of a static matching approach based on synonym dictionary, typographic similarity and model signature in order to define the degree of similarity between two models elements.

Package Merge. It is the composition mechanism of the UML [14] and is defined by *match rules*, *constraints* and *transformation* (the merge rules). The major application is in the implementation of the UML compliance levels. In principle, their *match rules* are similar to *match* used by our *match operator*. However, its *match rules*

are expressed in natural language and the match process consider only the name of the models. Moreover, the definition of Package Merge is incomplete, ambiguous and inconsistent.

Epsilon Merging Language. EML [6] is a metamodel agnostic language for expressing model composition. It includes a model comparison and model transformation language as subsets. The model comparison is only based on syntactic criterion. However, the match, in our approach, is founded in synonym dictionary, typographic similarity, syntactic properties and match strategy.

Difference between Models. It presents an approach of the how to detect and visualize differences between versions of UML documents such as class or object diagrams. It produces a unified document which contains the common and specific parts of two base documents, where the specific parts are highlighted [11]. While our approach tackles a range of very difficult problems related to dealing with comparison of semantics values in a flexible manner, it is primarily concerned with the comparison and manipulation of models from the same domain and with equal semantic values; without any flexibility during the comparison.

Approaches	Assessment Criteria: Guidance	Strategies	Typography	Name	Structural	Semantic	Rules	Operator
Model Composition Semantics				X				
Model Composition Directives				X	X			
Package Merge				X			X	
Epsilon Merging Language					X			
Difference between Models				X	X			

Legend:
X – supported by the approach
☐ – not supported by the approach

Fig. 4. Comparison of related approaches

VI. Conclusions and Future Work

In this paper we discussed the importance of model comparison for the task of model composition, its problems and challenges involved in its implementation. Our approach provides a flexible form of realizing the model comparison founded on match strategies by defining the match operator and by specifying its responsibility. Moreover, we consider that the range of different forms for matching models improves and assures a better performance to the comparison process and the use of guidance in order to provide a clear and easy manner to perform the comparison helps its improvement and evolution.

The problems and challenges outlined throughout the paper should encourage researchers to cope with the ever-present problem of matching models so that new generation of the application can enjoy the use of better techniques. Our approach has some limitations that should be investigated further. When models are defined, it is possible to associate them semantics constraints. These constraints should be considered and respected when it is necessary to perform the composition so that the specified semantic is not disrespected. Thus, our approach is not able, as yet, to compare these constraints. We claim to enhance the functionality of the match operator by creating new match strategies and improving the match rules. Another extension of our approach would be the use of ontology to improve the handle of the models' semantic values.

Even through our approach has been implemented and integrated to a profile composition mechanism demonstrating feasibility [12], empirical studies are necessary to validate the approach in real world design settings of model comparison and verify its performance and applicability in different application domains. Finally, we observed improvement in model comparison is absolutely necessary to the model engineering evolution and to allow model engineering to become an industrial reality.

References

[1] S. Clarke, "Composition of Object-Oriented Software Design Models," Ph.D. dissertation, School of Computer Applications, Dublin City University, Dublin, Irland, January 2001.

[2] L. Fernndez and A. Moreno, "An Introduction to UML Profiles," in *The European Journal for the Informatics Professional*, vol. 5, no. 2, April 2004, pp. 6–13.

[3] R. France, S. Ghosh, and T. Dinh Trong, "Model Driven Development Using UML 2.0: Promises and Pitfalls," *IEEE Computer Society*, vol. 39, no. 2, pp. 59–66, February 2006.

[4] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering (FOSE'07) co-located with ICSE'07*, Minnesota, EUA, May 2007, pp. 37–54.

[5] D. Jackson, "Alloy: a Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[6] D. Kolovos, "Epsilon Merging Language Project Page," http://www.eclipse.org/gmt/epsilon/.

[7] D. Kolovos, R. Paige, and F. Polack, "Model Comparison: a Foundation for Model Composition and Model Transformation Testing," in *International Workshop on Global Integrated Model Management*. New York, NY, USA: ACM Press, 2006, pp. 13–20.

[8] C. Manning and H. Shütze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[9] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *ICSE'07*, Minnesota, EUA, May 2007, pp. 54–64.

[10] Object Management Group, *MDA Guide Version 1.0.1*, 2003, http://www.omg.org/docs/omg/ 03-06-01.pdf.

[11] D. Ohst, M. Welle, and U. Kelter, "Differences between Versions of UML Diagrams," in *9th European Software Engineering Conference*. ACM Press, 2003, pp. 227–236.

[12] K. Oliveira, "Composition of UML Profiles," Master's thesis, Informatics Faculty, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil, February 2008.

[13] K. Oliveira and T. Oliveira, "A Guidance for Model Composition," in *International Conference on Software Engineering Advances (ICSEA'07)*, 2007, pp. 27–32, IEEE Computer Society.

[14] OMG, *Unified Modeling Language: Infrastructure version 2.1*, Object Management Group, February 2007.

[15] Y. Reddy, R. France, G. Straw, N. M. J. Bieman, E. Song, and G. Georg, "Directives for Composing Aspect-Oriented Design Class Models," *Transactions of Aspect-Oriented Software Development*, vol. 1, no. 1, pp. 75–105, 2006.

[16] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.