Towards a Framework for Distributed and Collaborative Modeling*

Antonio Cicchetti
School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
antonio.cicchetti@mdh.se
Henry Muccini, Patrizio Pelliccione, Alfonso Pierantonio
Dipartimento di Informatica, Università degli Studi dell'Aquila, L'Aquila, Italy
{henry.muccini,patrizio.pelliccione,alfonso.pierantonio}@di.univaq.it

Abstract

Increasingly, models are becoming first class core assets, and model-driven engineering requires novel techniques, tools, and practices to face the globalization of software development in the (always more) pervasive IT world.

This paper proposes a framework for synchronous and asynchronous concurrent and collaborative modeling. Synchronous collaborative modeling offers services for sharing the modeling space, models, documentation, and configuration, while asynchronous collaborative modeling offers services for supporting merging of models modified and edited separately by different software engineers. Our approach is based on the observation that it is in general more convenient to store differences between subsequent versions of a system than the whole models of each stage.

1 Introduction

In today's society individuals and organizations deal with an every growing load and diversity of information and content. The content is becoming of multimedia nature, such as diagrams, pictures, photos, video, etc. and it is produced in a distributed way. World globalization requires collaborative services, especially for non-stable, volatile, self-organizing communities that contribute to content production [4]. This implies that existent technologies for collaboration based on text documents are becoming more and more obsolete and a new way to uniformly represent these different pieces of content should be devised. Model-driven development has recently become a reality, thanks to main forces such as the Unified Modeling Language (UML) [15] and the Model-Driven Engineering (MDE) [18]. Consequently, models are primary artifacts retained as first class entities that can be analyzed and manipulated by means of automated tools. In general we can assume that the content and information can be abstractly represented by means of models. Models permit to easily relate the knowledge that can be represented in several different ways, and to automate several aspects of the knowledge sharing.

In this paper we propose DiScom, a framework for distributed and collaborative modeling. DiScom supports both synchronous (also referred as real-time) and asynchronous collaborative modeling. Synchronous collaborative modeling offers means for sharing information and multimedia content (expressed in terms of models), while asynchronous collaborative modeling offers means for supporting merging of models modified and edited separately by different players. During synchronous distributed collaborative work, many users collaborate together on the same artefact/model, they can switch from a model to another, and they can take the lock to some portions of the model according to some defined rules. During asynchronous distributed modeling, different users can take the model from the repository and apply changes in parallel. As usual, the problem to be managed here consists in comparing the different models for identifying differences and in merging the different versions of the models edited separately by different users. These operations must take into account dependencies among the different models. Semantic relations among models must be defined and taken into account to calculate models dependencies. In general, differences represent a profitable mechanism each time multiple versions of documents must be stored, transmitted, or processed, and version management is one of the contexts in which they are commonly exploited [6]. Therefore, starting from an initial version of a model, we propose to calculate the differences between the initial version and its updates, and subsequently to represent them in delta documents.

The paper is organized as follows: Sect. 2 presents the DiScom framework. In particular, Sections 2.2, 2.3, and 2.4 describe the theoretical results that allow us to man-

*Partially supported by they European Community's 7th Framework Programme (FP7/2007–2013), http://www.mancoosi.org Mancoosi project, grant agreement n°214898.

age model versions. Section 3 compares DiScom with related researches for model versioning, while Sect. 4 concludes the paper and outlines future work.

2 Engineering Distributed and Collaborative Modeling

Figure 1 schematizes DiScom: a centralized server contains the modules enabling conflicts management (i.e., the *Conflict manager* component), an optimizer that manages the modifications storage (i.e., the *Optimizer* component), and version management (i.e., the *Versions manager* component). The different versions are stored in a *Database*. Borrowing the lessons learnt in version management experiences for text documents, our approach is based on the observation that it is in general more convenient *to store differences between subsequent versions of a system* than the whole models of each stage. In fact, on one hand it is possible to save storage space; on the other hand, it is more evident and understandable which changes the models undergone and the underlying objectives.

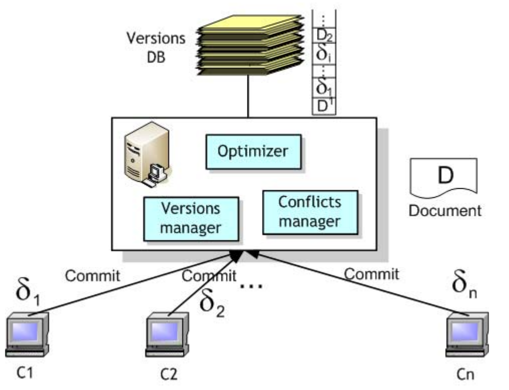


Figure 1. An overview of DiScom

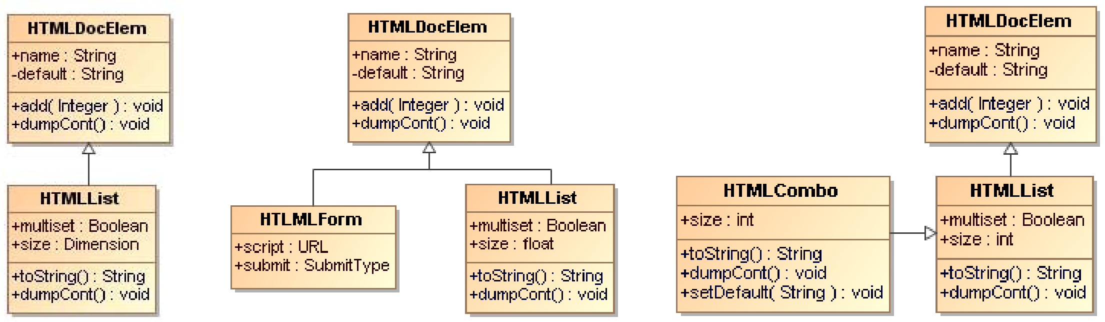
Thus, starting from an initial version of a system model, we propose to calculate the differences between the initial version and its updates, and subsequently to represent them in *delta documents*. Hence, differences are stored in the database (in figure represented with the symbol δ) instead of storing the different versions of the document. Sometimes, when there are a lot of small and consequent modifications stored, some of them could be collapsed to one single difference to reduce the amount of space needed and to optimize the computation of a particular version when asked for. This is the case of the synchronous or realtime collaborative modeling in which the commit operation (i.e., the operation that communicates to the shared view of the document the operation performed) is performed very frequently. In particular situations, when a version is considered stable, it could be preferable to save a new model version instead of a new difference; in this way it is possible to reduce the inefficiency of having sequences of multiple small changes, for instance. The computation that resolves these different solutions and that decides what is the better solution in a particular situation is performed by the *Optimizer* component.

The manipulations performed by each developer are encoded as difference models. When multiple changes are operated by different users, the framework will compose the initial model with the different changes, so to create a modified model while identifying possible conflicts. Following what described above, in general the actual version of the document, called D in Fig. 1, is obtained by applying all the modifications, following the defined order, to the last version of the document stored in the repository.

This computational model works both for *synchronous* and *asynchronous* collaborative modeling. The difference between these two situations is on the arrangement of the commit operations. In case of synchronous collaborative modeling the commit operation is performed at each operation so to simulate a shared screen. Thus, conflicts cannot appear since, on the server side, the operations are totally ordered. Different situation is for asynchronous collaborative modeling. In this case it is not possible to totally order the modifications. Let us suppose that two clients, *C1* and *C2*, are modifying the same version of the document, i.e., D. Let us suppose that *C1* and *C2* have produced the difference δ_1 and δ_2, respectively, off-line. It is typically impossible to decide which is the modification order. Thus, let us suppose that *C1* is the first one to perform the commit of δ_1. Now the document in the server is D+δ_1. When *C2* tries to perform the commit it is noticed by the system that the operation is not allowed since the version of the document that has been modified by *C2* is old with respect to the one that is currently in the server. Thus, *C2* is asked to update the document and possible conflicts are calculated. In case of conflicts *C2* has three possible choices:

- $D+\delta_1+\delta_2$: δ_2 is considered consecutive to δ_1
- $D+\delta_2+\delta_1$: δ_1 is considered consecutive to δ_2
- $D+(\delta_2\|\delta_1)$: in this case δ_1 and δ_2 are considered parallel modifications and then they are merged and then applied to D.

Conflicts can be due to modifications performed in a specific order [14], and in that case it makes sense to focus on the first two choices. A possible alternative is to associate roles to users so that depending on the user's role, a policy among the three presented above is chosen. The idea is that users that have more rights perform operations that in case of conflict overwrite what done by other users with less rights, and it is easily obtained by properly ordering manipulations.



a) b) c)

Figure 2. The sample a) initial model and b), c) two possible revisions.

In other situations conflicts are not caused by a specific revision sequence, thus demanding for appropriate techniques for detection and reconciliation [14]. In this proposal, conflicts identification and resolution exploits a model-based composition engine [8]. It tries to merge δ_i couples by unifying parallel independent modifications which can be considered as sequential changes on the same model. Then, when it finds diverging updates involving the same element it keeps them separated. Subsequently, reconciliation criteria can be given to solve each issue. Otherwise, the remaining conflicts are left unsolved and delegated to the user manual intervention.

The next sections introduce an example application in Sect. 2.1 and then detail the aspects of the version manager, the optimizer, and the conflict manager components (in Sections 2.2, 2.3, and 2.4, respectively) that constitute the technical base the proposed framework relies on.

2.1 Example Application

Let us consider a typical collaboration among members of development and verification&validation teams [2]: the software analyst, the software designer, and the software verifier. The collaboration assumed in [2] is regulated in a loop of actions that avoids concurrent tasks. By removing this assumption we allow concurrent modeling.

Figure 2 shows how collaborative work on the same model can cause conflicts between the changed models. More precisely, Fig. 2.a shows the initial model. It represents a HTML document (`HTMLDoc`) composed of HTML elements (`HTMLDocElem`), which have a particular type `HTMLList`. Let us suppose that two different teams are collaborative working on the model, M_1, and that they produce two different revisions, M_2 and M_3, respectively. There are two possible scenarios: (i) the revisions are temporarily ordered and then M_3 is build on M_2 or M_2 is build on M_3 (accordingly with the order) and (ii) the revisions are not ordered and then M_2 and M_3 must be merged, thus producing a new model M_4. Focusing on the latter scenario, Fig. 2.b and Fig. 2.c show two possible revisions. In particular, the M_2 model (in Fig. 2.b) has added a new subclass `HTMLForm` of `HTMLDocElem` and the type of the size attribute is changed in `float`. Simultaneously, the other developer produce the M_3 model (in Fig. 2.c) in which a new class `HTMLCombo` is added as subclass of `HTMLList` and the type of the size attribute is updated in `int`.

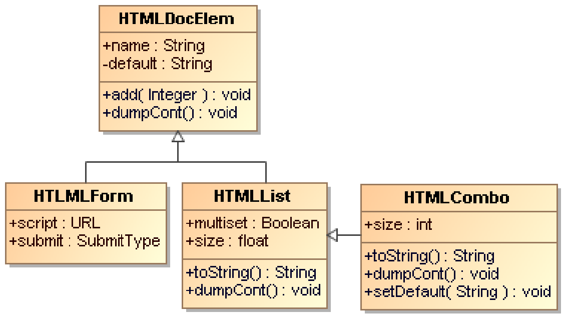


Figure 3. The result of a merging strategy.

These two revisions need to be merged and Fig. 3 shows the new model obtained after the merge. The obtained model contains both the `HTMLForm` and `HTMLCombo` classes and the conflict on the type of the size attribute of `HTMLList` is set to `float` since an integer can be obtained from a float without information loss, but the vice versa is not true.

2.2 Versions Manager

What described so far can be applied also to concurrent versions systems for textual documents and depicts a framework that is very similar to Concurrent Versions System (CVS). However, it is important to note that despite a number of concurrent versions systems are available for textual documents, the hierarchical nature of models requires specific tools and techniques for model comparison, model composition and model version management [12, 10]. In fact, even if there exist some serialization mechanisms able to map models toward corresponding structured documents (see for example XMI [16]), by decreasing the level of abstraction for the comparison the result will face a precision degradation. In other words, the abstraction level mismatch between text-based comparison and model-based design prevent the possibility for the former to return differencing results at the same abstraction level (and related semantics) of the corresponding models.

In MDE models are not considered as merely documentation but precise artifacts that can be understood by computers and can be automatically manipulated. In this scenario metamodeling plays a key role. It is intended as a common technique for defining the abstract syntax of models and the interrelationships between model elements. Metamodeling can be seen as the construction of a collection of *concepts* (things, terms, etc.) within a certain domain. A model is an abstraction of phenomena in the real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. This model is said to *conform to* its metamodel like a program conforms to the grammar of the programming language in which it is written.

As described so far, designers work on the same document stored in a shared repository. Each time they commit a new version, a document representing the changes is sent to the repository to update the current version of the artefact and to inform the other developers about the manipulations. Regardless the kind of concurrent development process (i.e., synchronous or asynchronous) it relies on differences. Consequently, an agreement on the representation of modifications is needed. In [7] a metamodel independent approach has been proposed to deal with such task. Given two models that conform to a given metamodel (namely MM), their difference has to conform to another metamodel (MMd) that can be automatically derived from MM. In particular, MMd has to provide the constructs capable to express the modifications that have to be performed on the initial version of a given model in order to obtain the last one. While D is a model conforming to MMd, each δ_i is an instance of the meta-model of the differences. Having the meta-models assures that each model obtained by making modifications conforms to the meta-model, and the same holds for the differences that conform to the corresponding meta-model.

Since there is not a unique metamodel for representing the differences between the models that conform to any metamodel, according to the proposed approach each metaclass `MC` of a given metamodel, gives place to the `AddedMC`, `DeletedMC` and `ChangedMC` metaclasses that enable the representation of additions, deletions and changes of elements conforming to the `MC` metaclass, respectively. As a consequence, the approach is said to be metamodel independent meaning that it can deal with any kind of metamodel; however, once the source metamodel MM has been given, all the environment is built up (and hence fixed) as tailored to the input metamodel MM.

Such representation mechanism has a number of properties, several of which can be profitable in the context of collaborative modeling and are recalled in the following[1]:
- *minimalistic* and *self-contained*, each difference model stores the minimum information to represent the evolution neglecting the context in which manipulations have been operated;
- *transformative*, a difference model induces a transformation, such that whenever applied to the initial model yields the final one. Moreover, the transformation can be applied to arbitrary input models (conforming to the base metamodel MM) giving place to patch-like updates. In this latest scenario, *minimalistic* and *self-contained* properties ensure the right behaviour;
- *invertible*, each difference model can be inverted, such that whenever applied to the final model nullifies changes thus getting the initial model;
- *compositional*, the result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations.

It is worth noting that a model-based representation of differences enables the integration between different modeling tools. In fact, once agreed on the common evolution description, developers can still work by using their own environments and producing as a result manipulations documented through the shared representation formalism. In the same way, updates can be retrieved from the shared format and applied in the corresponding development platform. Moreover, minimality and self-containedness make it possible to focus on the entities which have been manipulated and the corresponding change characteristics, thus allowing an easier management of distributed evolution. Finally, invertibility and compositionality permit to support the common operations in CVSs.

2.3 Optimizer

The evolution of a model consists of the original model and a number of difference models in such a way the final model is obtained by applying all the modifications to the original one. A useful construction would be the compositions of delta documents, like the *sequential merging* of several versions; in fact, it can be exploited to group two or more subsequent modifications in a single difference model.

For the sake of simplicity, let us consider only two subsequent modifications over the original model. The sequential

[1]Due to space limitations a complete description of those properties is not provided. The interested reader is referred to [6] for further details.

composition of such manipulations would mean to merge the modifications conveyed by the first document and then, in turn, by the second one in a resulting difference model containing a minimal difference set, i.e., only those modifications which have been not overridden by subsequent modifications. Given a couple of subsequent modifications affecting the same element, the optimization management will behave as summarized in Table 1. Going deeper, when a (`added`, `deleted`) sequence occurs it is possible to ignore both the manipulations being one the dual of the other, while in the case of a (`changed`, `deleted`) it is possible to perform only the deletion of the element since the changes would be lost anyway. To compact a (`deleted`, `added`) couple an update should be built which changes the version of the element depicted in `deleted` with the re-added one in `added`. In the situations where a (`added`, `changed`) or a (`changed`, `changed`) occur, it is possible to group the manipulations in a single `added` and `changed` delta, respectively. In particular, in the former case the addition can be completed with the subsequent changes, whereas in the latter the updates can be composed in a single merged one. Finally, the other couples can be simply ignored because it is not possible they could occur; for instance, it is not possible to update an element before creating it (`changed`, `added`) or to modify a previously deleted element (`deleted`, `changed`). It is worth noting that this simple optimization procedure is enabled by the abstraction level at which model comparison is computed and its result represented.

$\delta_1 \setminus \delta_2$	added	changed	deleted
added	\	added $\oplus$ changed	
changed	\	changed $\oplus$ changed	deleted
deleted	added	\	\

Table 1. Optimization cases.

2.4 Conflict Manager

In a distributed environment modifications can be operated also diverging from the same ancestor in parallel. In case both modifications are not affecting the same elements (or in other words are parallel independent) their composition is obtained by merging the difference models. This property can be easily shown by performing the parallel independent modifications by interleaving the single changes and assimilating it to the sequential composition. Unfortunately, the result of two parallel modifications can give place to conflicting results, i.e., elements in the original model which are changed by both difference models without converging to a common result. In this case, conflicting modifications either have to be resolved by the corresponding designers (see for instance [1]), or they need some mechanisms to support such a task [8].

In case of synchronous development, the mechanism of commits enforces the resolution of the mentioned conflicts by arranging the updates in a sequence (therefore the last commit will replace the previous one). In an asynchronous scenario, there is the need to choose one of the illustrated modifications. As said before, this process can be aided by some automation if criteria have been specified (like developer role priority); otherwise, the reconciliation is delegated to manual intervention of the architects.

3 Related Works

Version management has been largely investigated giving place to a number of tools, like CVS[2] and SubVersion[3] to mention a few. However, those text-based systems tend to be less efficient with models because of the hierarchical nature of the latter ones; in fact, they are not able to fully detect structural manipulations.

Nonetheless, there exist several modeling tools based on textual differencing, like Borland Together[4] and MagicDraw[5] which mainly rely on the XMI serialization format of models [16]. Unfortunately, the difference calculation and representation are tool specific; moreover, they are often based on the internal storage mechanisms of each tool which imply the use of unique identifiers. Those issues tend to lock the development process to the particular tool and pose several problems to document exchange [1]. In the same way, other tools lock the development to a particular metamodel, that is a fixed domain specific language; for example, Rational Software Architect[6] and Enterprise Architect[7] offer an interesting set of functionalities for managing changes in modeling files; however, those features are restricted to the UML metamodel. On the contrary, this work is based on metamodel independent techniques which ease document exchange and tool integration.

Reference [11] offers a tool integration infrastructure with versioning and synchronization facilities. In other words, starting from different models conforming to their own metamodels, a set of mappings is provided to build a chain of models by translating a source model to a target one. Whenever some change occurs, a synchronization mechanisms updates the models involved by such updates. From the point of view of this paper, the mentioned work can be considered as a useful view integration methodology which could be exploited to update diagrams (or views) stored in the centralized repository. However, it is provided neither any support for distributed development of the same

[2]CVS web site: http://www.nongnu.org/cvs.

[3]Subversion web site: http://subversion.tigris.org.

[4]Borland Together: http://www.borland.com/us/products/together.

[5]MagicDraw: http://www.magicdraw.com.

[6]Rational Software Architect: http://www-01.ibm.com/software/awd-tools/swarchitect/websphere/.

[7]Enterprise Architect: http://www.sparxsystems.com.au/products/ea.html

model, nor a manipulation sharing technique based on differences.

Finally, approaches like [5, 13, 3, 9] provide the support to represent the history of changes in a model-based fashion. However, this aspect can be considered orthogonal to the one related to difference representation.

4 Conclusions and Future Work

In this work we presented a step towards the definition of a framework to support collaborative and concurrent modeling. In this respect, several problems need to be faced, like version management, optimization of stored versions and conflict resolution. The framework is based on metamodel independent techniques, or in other words each mechanism is defined in a general way and can be automatically adapted to the specific metamodel given as input.

On the future work side we plan to further investigate and implement the techniques here introduced. In particular the identification and the resolution of conflicts between models opens several problems in automatizing and supporting the merging. In fact, collisions can be also originated by interconnections between different models, which need to be specified and handled through appropriate mechanisms. In order to deal with the intrinsic complexity of that problem, those techniques should at least encompass transaction management and cyclic relationship checks [17]. In any case, those approaches will exploit the representation of model evolution by means of differences. Finally, future work encompasses also real world case studies in order to improve the illustrated set of techniques in terms of scalability, performance and usability.

References

[1] M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.

[2] M. Angelaccio and A. D'Ambrogio. A model transformation framework to boost productivity and creativity in collaborative working environments. In *proceedings of CollaborateCom 2007*, pages 464–472, Nov. 2007.

[3] Auckland Bioengineering Institute at the University of Auckland. CellML Model Repositories Project. http://www.cellml.org/wiki/CellMLModelRepositories.

[4] J. A. B. Blaya, I. Demeure, P. Gianrossi, P. G. Lopez, J. A. M. Navarro, E. M. Meyer, P. Pelliccione, and F. Tastet-Cherel. Popeye: providing collaborative services for ad hoc and spontaneous communities. *Service Oriented Computing and Applications*, 3(1), 2009.

[5] M. Carey. Data delivery in a service-oriented world: the bea aqualogic data services platform. In *Proceedings of the ACM SIGMOD'06*, pages 695–705. ACM Press, 2006.

[6] A. Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, University of L'Aquila, Computer Science Dept., 2008.

[7] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Representation of Model Differences. *Journal of Object Technology Special Issue*, 6(9), 2007.

[8] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *Procs. of the 11th Int. Conf. MoDELS '08*, pages 311–325, 2008.

[9] H. B. et al. *The FAMOOS Object-Oriented Reengineering Handbook*, 1999. http://www.iam.unibe.ch/ famoos/handbook/4handbook.pdf.

[10] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Procs of GaMMa '06*. ACM Press, 2006.

[11] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Procs of GaMMa '06*, pages 43–46. ACM Press, 2006.

[12] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA*, 2004.

[13] D. Matheson, R. France, J. Bieman, R. Alexander, J. DeWitt, and N. McEachen. Managed Evolution of a Model Driven Development Approach to Software-based Solutions. In *OOPSLA*, 2004.

[14] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electr. Notes Theor. Comput. Sci*, 127(3):113–128, 2005.

[15] Object Management Group. UML 2.0 Infrastructure Final Adopted Specification, 2003. OMG document ptc/03-09-15.

[16] Object Management Group. XMI 2.0 XML Metadata Interchange, 2003. OMG document formal/2003-05-02.

[17] R. Paige, P. Brooke, and J. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.*, 2007.

[18] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.