Electronic Communications of the EASST
Volume 50 (2011)

Recent Advances in Multi-paradigm Modeling (MPM 2011)

A hybrid approach for multi-view modeling

Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque

12 pages

Guest Editors: Vasco Amaral, Cécile Hardebolle, Hans Vangheluwe, László Lengyel, Peter Bunus
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A hybrid approach for multi-view modeling

Antonio Cicchetti[1], **Federico Ciccozzi**[1], **Thomas Leveque**[2]

[1] School of Innovation, Design and Engineering - MRTC
Mälardalen University, Västerås, Sweden
firstname.lastname@mdh.se
[2] Orange Labs - Meylan, France
thomas.leveque@orange.com

Abstract: Multi-view modeling is a widely accepted technique to reduce the complexity in the development of modern software systems. It allows developers to focus on a narrowed portion of the specification dealing with a selected aspect of the problem. However, multi-view modeling support discloses a number of issues: on the one hand consistency management very often has to cope with semantics interconnections between the different concerns. On the other hand, providing a predefined set of views usually results as too restrictive because of expressiveness and customization needs. This paper proposes a hybrid solution for multi-view modeling based on an arbitrary number of custom views defined on top of an underlying modeling language. The aim is to benefit from the consistency by-construction granted by well-defined views while at the same time providing malleable perspectives through which the system under development can be specified.

Keywords: Multi-view modeling, separation of concerns, model-driven engineering, model synchronization

1 Introduction

Nowadays software systems are employed in any kind of applicative domain, ranging from a tiny music player to the management of nuclear plants or air traffic control. The growth of their complexity is never ceasing demanding adequate techniques to face their development. Model-driven engineering (MDE) [Sch06] has been conceived as a way to face such difficulties by means of models, that is precise abstractions of real-world phenomena highlighting the salient details with respect to the system under study [BÓ5]. Moreover, models are no more considered as mere documentation but exploited as the specification of the application itself.

Because of the aforementioned complexity of software systems, that also tend to mix heterogeneous domains, a problem is typically decomposed by different viewpoints, each of which approaches the solution from a domain-specific perspective. Multi-view modeling mechanisms are usually distinguished between [ISO07]:

- **synthetic**: each view is implemented as a distinct metamodel and the overall system is obtained as *synthesis* of the information carried by the different views;
- **projective**: end-users are provided with *virtual* views made up of selected concepts coming from a single base metamodel by hiding details not relevant for the particular viewpoint taken into account.

The former is a powerful solution to multi-view modeling as it can exploit the expressive power of disparate metamodels, each of which dealing with a particular aspect of the system under study [BÓ5]. However, the use of a constellation of domain-specific metamodels opens up a number of problems, mainly related to consistency management: in fact, modifications operated within one view can have impacts on other views, often pertaining to the semantics of the considered domains, demanding a thorough specification of interplays between the different views. Moreover, it is worth noting that such a problem grows with the number of adopted views and languages [MV09].

Technically, the projective solution relies on a single underlying metamodel to ease the consistency management; even if end-users[1] work virtually on multiple views changes are operated on the same shared model. This often amounts to being too restrictive because either the metamodel is too generic (i.e., with scarcely specified semantics) or the views are too specific to be reused in several development contexts [CCK+11]. Moreover, user interaction raises a number of issues especially when cross-view constraints exist, since in general the base metamodel has no concept of view embedded in the language making it difficult to express, for example, that some editing operations are only allowed in a specific view [MV09].

This work aims at providing an automated mechanism representing a hybrid technique for multi-view modeling: it is based on the definition of multiple views on top of a metamodel, each of which entails a corresponding subportion of the original metamodel. For this purpose, we first define a set of desirable features a multi-view modeling environment should support, and then provide a solution in order to satisfy such demands. The final goal is to obtain a good trade-off between both the synthetic and projective techniques for multi-view modeling implementation. A prototypical implementation has been realized on the Eclipse platform.

The structure of the paper is as follows: the next section discusses related solutions available for both the synthetic and the projective implementations of multi-view modeling. Then, Section 3 illustrates a set of basic features that a multi-view solution should provide and these features are implemented as described in Section 4. Section 5 discusses the current status of the work and limitations, future investigation directions, and draws some conclusions.

2 Background and Related Works

Separation of concerns is not a novel concept; it is, in fact, the basic principle prescribing to reduce problem complexity by tackling it from different perspectives. The IEEE 1471 standardized a set of recommended practices for the description of software-intensive systems' architectures that have been adopted as standard by ISO in 2007 [ISO07]. In particular, architectural descriptions are conceived as inherently multi-view, since an exhaustive specification of a system can only be provided by means of different viewpoints. In particular, a *viewpoint* is a set of concerns and modeling techniques to define the system from a certain perspective, and a *view* is the corresponding instance of the viewpoint taken into account for the system under development. As distinguished in the ISO specification and in other works [MV09, BJHR10], multi-view approaches can be categorized in synthetic and projective. Depending on the kind of approach, different techniques have been developed to support development and maintenance of the system specification.

[1] "Developer" represents the person creating the views, while "end-user" the one using them.

For the synthetic approach, a constellation of (typically) distinct metamodels is used to describe different features of the system depending on the domain the system is studied in. For instance, a web modeling language based on the Model-View-Controller pattern [Con99] allows to specify a web application by considering the data underlying the application, the business logic, and the user presentation as three different concerns that are modeled on their own. Then, in order to obtain the blended application, those concerns have to be synthesized (or woven) toward a resulting system [DBJ+05, EAB02]. Analogously it happens with, for example, embedded systems, whose development is made by separating hardware from software characteristics, and in turn functional from extra-functional features, and so forth [BJHR10]. In these and similar cases, interplays between the different viewpoints have to be explicitly defined in order to allow the synthesis of the resulting system. In other words, it must be clarified how the different viewpoints can be merged (the matches between entities in different models), and the semantics of overlaps. In general such relationships can be defined by means of transformations that embed the semantics behind views interplays [RJV09]. Alternatively, all the views can be reduced to a *common denominator* through which it is possible to synthesize the information carried by the different viewpoints and derive the resulting system specification [Van00].

The main issue related to synthetic approaches is consistency management: since semantics is involved in the relationships across models, interconnections have to be carefully defined, a task that grows with the number of exploited views. Moreover, adding or updating views, especially if not orthogonal to the existing ones, demands a revision of the current consistency rules as well as synthesis mechanisms.

In order to partially overcome the problems mentioned above, a possible solution is to build up views on top of a single base metamodel. In this way, end-users can be provided with a set of views allowing the specification of the system from different perspectives. At the same time, consistency management can be obtained for free by construction, since all the changes boil down to manipulations of the same model, even if virtually operated from different viewpoints [MV09]. Despite an easier consistency management, such projective approaches demand a well defined semantics of the base language. For instance, synchronization of UML diagrams [CLN+09] poses several issues, even if developers are operating on the same model, because of ambiguities in the formalization of such language. Moreover, projective solutions suffer a limited customizability because of the fixed base language and the predefined set of views. It is worth noting that in general the base language has no knowledge of views, therefore implementing cross-checks between user operations or providing editing rights within each view to drive the application design either require language extensions [Nas03] or have to be hard-coded in the supporting tool [CCK+11].

The contribution presented in this paper aims at reaching a good trade-off between synthetic and projective approaches. The main idea is to start from a base metamodel (referred to as overall metamodel in the paper) and to allow the developer to create views through an extensive set of customization opportunities. After the creation of a view, end-users can model the system from the corresponding perspective as a metamodel for the purpose was created. In fact, a view creation entails the generation of a new metamodel together with the essential equipment to create models conforming to it. Moreover, automated synchronization procedures are derived in order to maintain consistency among the different views. As it will be explained later on in the paper, view creation is assisted by a creation wizard and has no restrictions in terms

of numbers, overlapping with existing ones and creation's point of time. In this respect, the closest approach to our proposal is provided by the Obeo Designer[2]: it allows to create views on top of a metamodel, to select the elements to show for each view, and even to customize the graphical rendering of model elements and operations on them. However, the implementation mechanism does not rely on the creation of a proper metamodel for each view, and view creation is a preliminary step operated before the system development begins. Moreover, editing rights granted on elements pertaining to each view cannot be controlled.

The main distinction with synthetic solutions is the consistency and synthesis management: interdependencies between views are directly derived when a view is built, as well as synthesis mechanisms, that can be automatically generated to keep the different perspectives up-to-date. Compared to projective approaches, the proposal described in this work introduces a technique that creates proper metamodels, one for each view, instead of simply hiding elements that do not matter a particular viewpoint. Moreover, views are not predetermined and can be introduced at any time of the development together with manipulation rights. To summarize, this proposal aims at providing better customization features while preserving consistency automation.

In the reminder of the paper, basic requirements for views are described and they underpin creation and management features illustrated in detail in the following sections.

3 Basic features for view customization

As discussed so far, the aim of this work is to provide a good trade-off between synthetic and projective solutions to support multi-view modeling. In this respect, the following clarifies: (i) the set of features we considered as basic needs when specifying and using a particular view and (ii) a set of rules for the Eclipse Modeling Framework (EMF)[3] views and overall metamodels to be consistent and their respective models to be synchronizable.

3.1 A basic set of multi-view editing support features

The following list highlights view features by distinguishing them in characteristics relevant at specification time (view definition) and those intended to support modeling tasks (editing facilities and synchronization management):

1. **View definition**
   - arbitrary selection of subportions of the overall metamodel, meaning that there should not be limitations about the sub-metamodel considered for the view under definition;
   - support of variable number of views, i.e. there should not be constraints about the number of defined views;
   - support of overlapping metamodel portions, meaning that different views can be built on top of (partially) overlapping sub-metamodels;
   - management of well-formedness issues, i.e. there should exist appropriate support driving the developer in a correct selection of subportions of the overall metamodel;
2. **Editing facilities**
   - arbitrary editing rights, entailing that each view should carry with it a set of modification rights on its elements coherent with the perspective it pertains to;

[2] http://www.obeodesigner.com/
[3] http://www.eclipse.org/modeling/emf/?project=emf

- support of customized editors, i.e. each view should carry with it a proper palette of editing tools, appropriate to the perspective it is related to;

3. **Synchronization management**
   - transparent merge of separate views while editing, meaning that consistency management across views should happen in background without any end-user intervention;
   - non-blocking management of concurrent manipulations for overlapping views, i.e. modifications can be operated concurrently on overlapping portions of the overall metamodel.

The group of requirements related to view definition are tailored to guarantee high customization opportunities during view creation; editing facilities are devoted to make the view as much "domain-specific" as possible, both by allowing only narrowed manipulation features and through the definition of appropriate concrete syntaxes. Finally, the satisfaction of synchronization management requirements is needed to make the end-users able to operate on separate views independently of one another.

3.2 Requirements for Consistency and Synchronization

The selection of model elements for generating a customized view from the overall metamodel must follow a set of rules in order for the views and the overall metamodel to be consistent and their respective models to be synchronizable. The following rules, expressed by means of Ecore metamodeling language[4], have been identified for such purposes:

1. **Required EAttribute elements:** if an EClass element selected to be included in the new view has EAttribute elements with $lowerbound > 0$ (i.e., at least one instance of them has to exist) they have to be included, otherwise the EClass element would not be correctly created;
2. **Required EReference elements:** if two EClass elements chosen to be included in the view are linked by a non-selected EReference element which has $lowerbound > 0$ (i.e., the reference demands the existence of related classes), this EReference element must be included in the selection in order to enable consistent creation of such EClass elements;
3. **Containment EReference elements:** if two included EClass elements are linked by a non-selected EReference element which is a *containment*, this EReference element must be part of the view since such EClass elements cannot be edited ignoring such relationship;
4. **ESuperType elements:** if an included EClass specializes one or more EClass elements (as ESuperTypes) those have to be included as well, since by removing or modifying them consistency can be jeopardized;
5. **Containment EClass elements:** if an EClass selected for inclusion in the view is contained by an EClass element, this must be part of the view together with the containment EReference element, since the modification of the contained EClass element can affect the containing one;
6. **Unique Identifiers:** a non-empty set of EAttribute and/or EReference elements must be selected to act as unique identifier for each selected EClass for the synchronization mechanisms to avoid inconsistencies and conflicts.

[4] For more information about Ecore the reader can refer to the EMF project pages.

Rules 1, 2, 3, 4 and 5 are needed for consistency reasons and for allowing the manipulation of selected EClass elements; those rules are automatically applied and require elements to be automatically included for consistency reasons. The developer may disable such automation feature and decide by himself what to select. However, consistency warnings will be given by the wizard when completing the view creation for the developer to modify the selection according to the warnings in order for the view to achieve well-formedness. Rule 5 is always transparent to the developer and even if elements will be part of the view metamodel, they will not be visible nor editable in the editing environment. Rule 6 is needed for each EClass to have a unique identifier to be used for synchronization purposes as described in the section 4.

4 Implementation of the Solution

The approach we propose is based on the creation of customized views starting from an initial metamodel defined in Ecore. Such views are meant to be consistent subportions (i.e. metamodels) of the total metamodel that isolate the manipulation of a certain set of interesting aspects. In this section we describe the implementation of the proposed solution in order to achieve the followings: (i) creation of customized views, from an initial Ecore overall metamodel, for achieving a consistent concern-specific metamodel still conforming to the overall metamodel, (ii) provision of automatically generated synchronization mechanisms for maintaining consistency among the customized views and the original metamodel, and (iii) provision of an automatically generated ad-hoc Eclipse environment for managing such views.

4.1 View Generation Process

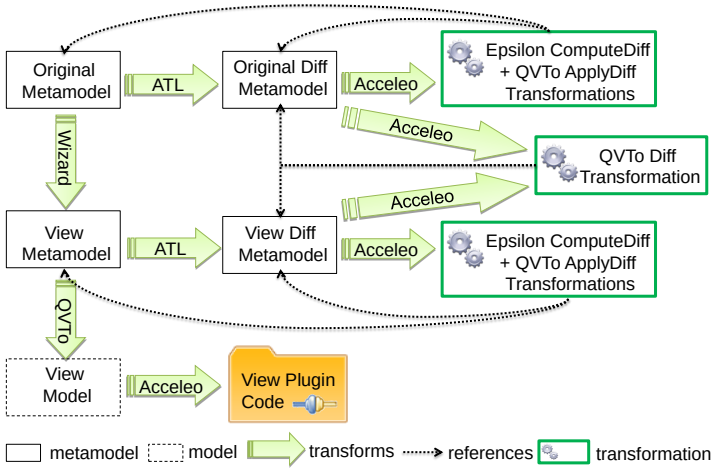

Figure 1: View Generation Process

An overview of the view generation process is shown in Fig. 1.

1. First, the view metamodel, which is a subportion of the original metamodel, is produced through the creation wizard described in the next sections;
2. Then through an ATL model-to-model transformation, the difference metamodels, which represent model modifications, are derived from original and view metamodels;

3. A corresponding difference computation transformation and patch (difference application) in-place transformations are generated for each difference metamodel through higher-order transformations;
4. At this point, a bidirectional model-to-model transformation is produced in order to be able to convert model differences between the view models and the original model; also in this case an higher-order transformation is used;
5. Eventually, through a model-to-text transformation, an Eclipse plugin is generated and provides an editor to manipulate view models; it also includes the needed synchronization mechanisms.

In the following sections, we will refer to original metamodel to talk about the initial metamodel, to view metamodel to talk about the derived metamodel associated with a view, to view definition metamodel to represent the language used to specify a view.

4.2 View Creation Wizard

Each view is independently generated by a wizard (Fig. 2) that drives the developer through the generation process which consists of the following steps:

- **View properties selection:** the first step of the creation of the customized view is for the developer to provide general information needed for creating the view, storing it and generating a related Eclipse editor model. Such model is then used for the creation of an editor in Eclipse that provides a customized environment for editing and manipulating the newly created view. The set of information to be inserted is: (i) *View Name*, which represents the name of the new view, both for its metamodel and the related Eclipse editor, (ii) *NameSpace Prefix* for the view metamodel, and (iii) *NameSpace URI* for the view metamodel.
- **View elements selection:** the elements constituting the overall metamodel are shown and the developer is able to select each element (EClass and EAttribute) that is going to be part of the new view (Fig. 2). Every Ecore model element is considered;
- **Unique identifiers selection:** in order to allow synchronization, as described later on in this section, for each selected EClass element a non-empty set of its EAttribute and/or EReference elements must be selected to act as its unique identifier;
- **Editing rights selection:** once the view is populated, desired editing rights are selected for each of the selected elements among two possibilities: (1) read only, meaning that the element will be part of the view only as visible but not editable nor creatable, (2) editable, the element is both readable and editable if the element is not of EClass type, in which case the element would be readable and creatable, while its editability will depend on whether attributes/references of the EClass are themselves selected as editable parts of the view;
- **Selected view final check:** a final summary page shows a summary of selected elements with associated editing rights. Moreover, according to the requirements described in section 3.2, further elements may appear automatically selected by the consistency checking engine to ensure the creation of a view whose models will be still consistent and conforming to the initial metamodel. Such automatically selected elements are marked with 'consistency' editing right and, even if still present in the view, will not be visible nor editable since their purpose is consistency-ensurance only.

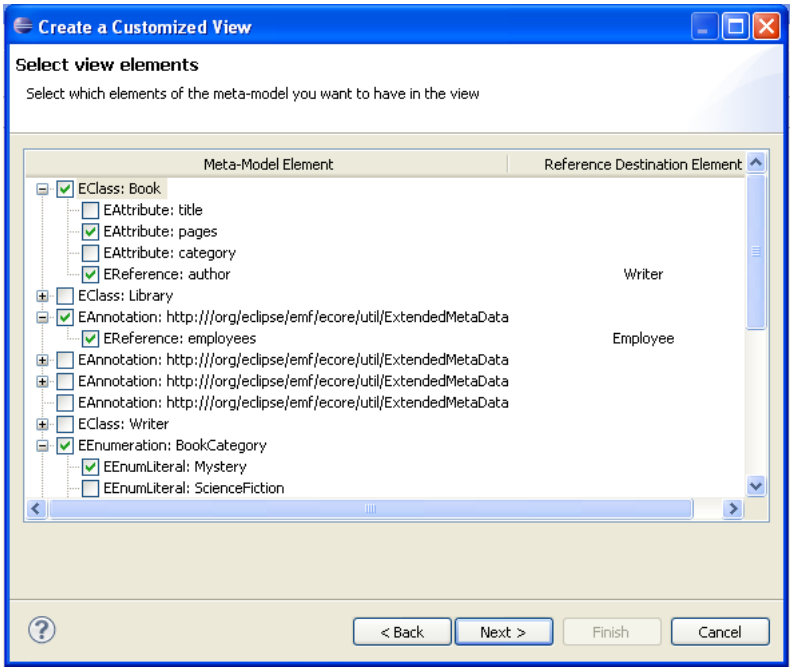

Figure 2: View Creation Wizard

The developer can decide to get the wizard itself to automatically select model elements in order to maintain consistency between the selected view and the overall metamodel. Such elements are selected according to the manually selected elements and the rules specified in section 3.2 and applied to them. In case the developer decides not to use this facility, the wizard will anyhow inform the developer about the missing elements needed to create a consistent view in a dedicated page after the selection phase.

The view is generated in terms of an Ecore metamodel composed by the elements selected through the creation wizard. Moreover, different kinds of models are automatically generated: (i) difference metamodels, to be used for modification representation, (ii) model comparison transformations, to produce the difference models, (iii) in-place model transformations which apply difference models, (iv) model-to-model transformations which convert modifications of one view to corresponding modifications on the original model for synchronization purposes, and (v) Eclipse editor model, needed for automatically generate an Eclipse editor for view models.

4.3 Implementing Synchronization among Views through Model Differencing

Fig. 3 shows how the synchronization is performed between the views and the original model. Our solution relies on the assumption that view models and original model cannot be changed concurrently in the workspace and on the fact that every evolution action is first applied to the original model and then propagated to the other views; thus, direct evolutions propagation among views is never performed. Synchronization between customized views and original model is based on model differencing and model transformations. Modification propagation follows several steps:

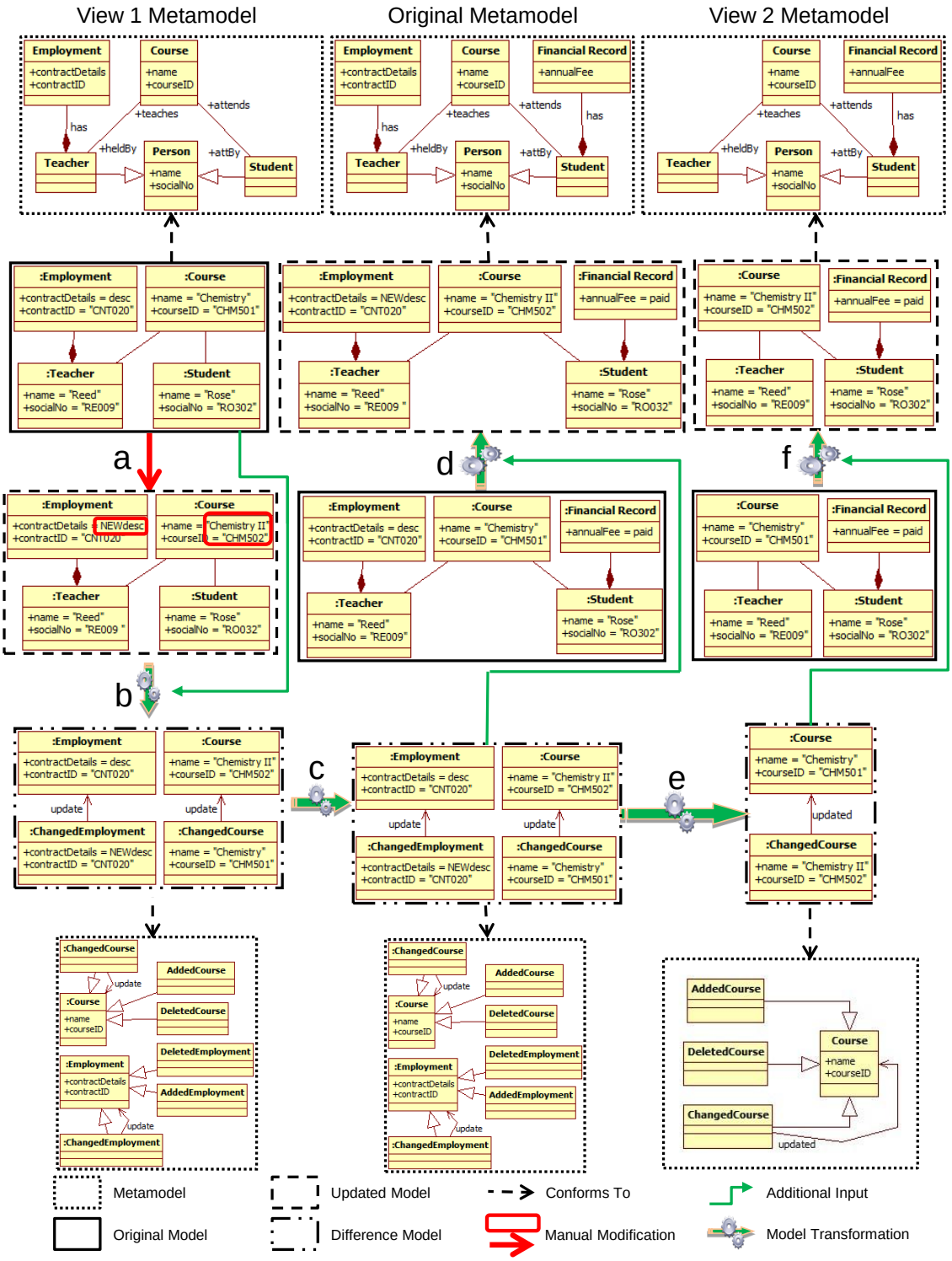

Figure 3: Synchronization

1. First, model changes are detected by listening to file changes (Fig. 3.a);
2. Then, a model differencing algorithm is applied between the old and new model version files producing a difference model representing the performed modifications (Fig. 3.b);
3. Hereafter, the difference model is transformed into another difference model representing the corresponding modifications on the original model (Fig. 3.c);
4. The resulting difference model is applied to the original model (Fig. 3.d);
5. Then, for each view, it is transformed into a difference model representing the corresponding modifications on the related view model (Fig. 3.e);
6. Eventually, the difference models are applied to the corresponding view models (Fig. 3.f).

When modifications come from the original model, steps 3 and 4 are skipped. The model difference representation is based on an existing work [CDP07] which introduces a technique to derive a difference metamodel from the original one relying on the partitioning of the manipulations into three basic operations: *additions*, *deletions*, and *updates* of model elements. For this purpose we use difference models that conform to difference metamodels automatically derived both from the overall metamodel and the view metamodel during the view creation process through the ATL transformation described in [CDP07]. The transformation takes as input a metamodel and enriches it with the constructs able to express the modifications that are performed on the initial version of a given model in order to obtain the modified version (i.e. additions, deletions and changes). These constructs are defined in terms of metaclasses that specialize the corresponding original metaclass. The computation of the difference models is based on the Epsilon comparison and merge language in a similar way as used in [CCLS11]. In order to avoid cyclic cascading of model changes, we associate a timestamp with the difference model and ensure that modifications are only applied if the timestamp is more recent than the one related to the current model file. Nevertheless, the resolution of conflicts that may arise from concurrent modifications in overlapping views is performed manually by the end-user. A partial automation of such resolution by means of suggested quick fixes is left for future work. Eclipse local history is used to keep track of previous model file versions. We add an annotation on every view model in order to specify where the original model is located.

4.4 View Model Editor

The view creation process provides also a customized editing environment for the created view. This is achieved through a model-to-text transformation implemented using Acceleo[5] and that takes as input a specific model generated during the view creation process and conforming to the editor metamodel shown in Fig. 4; it generates an Eclipse plugin implementing the Eclipse model editor associated with the created view.

The editor metamodel has the following structure:

- **Editor:** is the element representing the created editor and is composed by the attribute *name*, containing the editor name. View is composed by a non-empty collection (i.e. *contains*) of *Node* elements;
- **Node:** represents each of the EClass elements composing the view metamodel except, if present, for the consistency only ones. Each node contains the following attributes: (i) *isRoot*, which shows if the element is root in the view metamodel, (ii) *canBeCreated*,

[5] http://eclipse.org/acceleo/

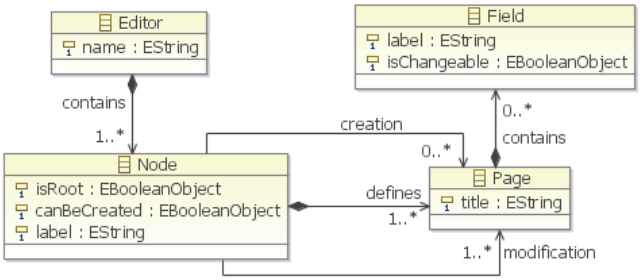

Figure 4: Editor Metamodel

which shows whether the element has been granted with the read only or editable privilege, and (iii) *label* which stores the element's name. The Node can be linked to three different pages according to the related editing rights: (1) *creation* links it to the creation page only if editable privilege is granted, (2) *defines* links to the visualization page which is always present as containment, and (3) *modification* connects the node to page for editing of its EStructuralFeature elements (e.g. EAttribute, EReference) according to the editing rights granted for them;

- **Page:** is the element representing modification, creation and definition pages for each Node. The attribute *title* represents the page title on the Eclipse view. Each page contains a set of *Field* elements;
- **Field:** represents the EStructuralFeature elements related to each EClass element in the view metamodel. The attribute *label* contains the element's name while *isChangeable* shows whether the element has been granted with the read only or editable privilege.

The editor model is automatically generated by a QVTo model-to-model transformation that takes as input the created view metamodel and other needed information inserted by the developer in the wizard. An Eclipse plugin is generated from this model using Acceleo. The resulted code implements an Eclipse editor dedicated to models conforming to the view metamodel. The editor is similar to the default EMF generated tree editor except that (i) model element creation opens a wizard which contains editing pages and (ii) fields in property views can be not editable depending on the specified editing rights. In addition, a filter is used to hide details on EClass elements that have been added to the view metamodel only for consistency purposes and a resource listener is generated in order to call the synchronization mechanism when required.

5 Conclusions

This paper presented an approach for hybrid support to multi-view modeling. We first established a set of basic needs for view customization, and then discussed later on their implementation in the Eclipse platform[6]. Despite the implementation being technology specific (i.e. based on EMF), our experiences in other research projects [CCK+11, BCF+11] make us confident that the requirements illustrated in Sect. 3 can be considered as independent of the modeling technology taken into account. Nonetheless, we do not consider them as complete, but only a set of the needs that typically come out when setting up modeling views. As a consequence, a part of future investigations will deal with some empirical studies devoted to analyze view customization

[6] For the interested reader the implemented prototype is available for download at http://www.mrtc.mdh.se/~acicchetti/multiviewProject.php

demands, and hence to discover further requirements to accommodate the creation of new views. Moreover, additional work will cope with the validation of the proposed technique in some industrial setting in order to verify feasibility and analyze possible scalability issues related to the proposed mechanism. Especially, we will have to survey the reliability of the element identification mechanism; the current version allows us an efficient element identification for difference calculation and view synchronization, but still leaves place to possible undesired duplications. Finally, investigation efforts will be devoted toward the application of our approach in the reverse direction, i.e. as a method to create a common denominator as proposed in [Van00].

Bibliography

[BÓ5] J. Bézivin. On the Unification Power of Models. *Jour. on Software and Systems Modeling (SoSyM)* 4(2):171–188, 2005.

[BCF+11] E. Borde, J. Carlson, J. Feljan, L. Lednicki, T. Leveque, J. Maras, A. Petricic, S. Sentilles. PRIDE – an Environment for Component-based Development of Distributed Real-time Embedded Systems. In *9th IEEE/IFIP Conference on Software Architecture*. IEEE, June 2011.

[BJHR10] F. Boulanger, C. Jacquet, C. Hardebolle, E. Rouis. Modeling Heterogeneous Points of View with ModHel'X. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009*. LNCS, pp. 310–324. Springer, 2010.

[CCK+11] A. Cicchetti, F. Ciccozzi, M. Krekola, S. Mazzini, M. Panunzio, S. Puri, C. Santamaria, T. Vardanega, A. Zovi. CHESS Tool presentation. In *1st TOPCASED Days, Toulouse*. 2011.

[CCLS11] A. Cicchetti, F. Ciccozzi, T. Leveque, S. Sentilles. Evolution Management of Extra-Functional Properties in Component-Based Embedded Systems. In *The 14th Int'l ACM SIGSOFT Symposium on Component Based Software Engineering*. 2011.

[CDP07] A. Cicchetti, D. Di Ruscio, A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology* 6:165–185, 2007.

[CLN+09] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, R. Salay. Relationship-based change propagation: A case study. In *MISE '09*. Pp. 7–12. IEEE CS, Washington, DC, USA, 2009.

[Con99] J. Conallen. Modeling Web Application Architectures with UML. *Comm. ACM* 42(10):63–71, 1999.

[DBJ+05] M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, G. Gueltas. AMW: a Generic Model Weaver. In *Procs. of IDM05*. 2005.

[EAB02] T. Elrad, O. Aldawud, A. Bader. Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In *Procs. of GPCE 2002, Pittsburgh, PA, USA*. LNCS, pp. 189–201. Springer-Verlag, 2002.

[ISO07] ISO/IEC/(IEEE). ISO/IEC 42010 (IEEE Std) 1471-2000 : Systems and Software engineering - Recomended practice for architectural description of software-intensive systems. 2007.

[MV09] E. Miotto, T. Vardanega. On the integration of domain-specific and scientific bodies of knowledge in Model Driven Engineering. In *Procs. of STANDRTS'09, Dublin (Ireland)*. 2009.

[Nas03] M. Nassar. VUML : a Viewpoint oriented UML Extension. In *Procs. of ASE 2003*. Pp. 373–376. IEEE Computer Society, 2003.

[RJV09] J. R. Romero, J. I. Jaen, A. Vallecillo. Realizing Correspondences in Multi-viewpoint Specifications. In *Procs. of the 13th IEEE EDOC, 1-4 September 2009, Auckland, New Zealand*. Pp. 163–172. IEEE CS, 2009.

[Sch06] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39(2):25–31, 2006.

[Van00] H. L. M. Vangheluwe. DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling. Pp. 129–134, 2000.