Diff-based model synchronization in an industrial MDD process

Kindler, Ekkart; Könemann, Patrick; Unland, Ludger

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Diff-based model synchronization in an industrial MDD process

Ekkart Kindler
Technical University of Denmark

Patrick Könemann
Technical University of Denmark

Ludger Unland
Software Design & Management, Düsseldorf, Germany

Technical University of Denmark

IMM-TECHNICAL REPORT-2008-07

June 30, 2008

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-TECHNICAL REPORT: ISSN 1601-2321

Diff-based model synchronization in an industrial MDD process*

Ekkart Kindler[1], and Patrick Könemann[1], and Ludger Unland[2]

[1] Technical University of Denmark, Informatics and Mathematical Modelling
[eki|pk]@imm.dtu.dk
[2] Software Design & Management, Düsseldorf, Germany
ludger.unland@sdm.de

Abstract

Models play a central role in model-driven software engineering. There are many different kinds of models during the development process, which are related to each other and change over time. Therefore, it is difficult to keep the different models consistent with each other. Consistency of different models is maintained manually in many cases today. This paper presents an approach for automated model differencing, so that the differences between two model versions (called delta) can be extracted and stored. It can then be re-used independently of the models it was created from to interactively merge different model versions, and for synchronizing other types of models. The main concern was to apply our concepts to an industrial process, so usability and performance were important issues.

Keywords. MDA, MDD, model differencing, model merging, model transformation, model synchronization.

1 Introduction

Models play an increasingly important role in software development. Different types of models are used, which are produced in different phases of the software development process, with different objectives, and on different levels of abstraction. The models in the later phases of the development process can often be used to generate large parts of the code automatically. In terms of model-driven software development (MDD), a well-known approach is the model-driven architecture (MDA) [2], which distinguishes between a platform independent (PIM) and a platform specific model (PSM). The platform specific model is then used to generate at least parts of the code for the final software. Many industrial development processes distinguish between analysis and design models; our paper refers to such a process of sd&m [3].

There need to be tools that support models and code generation in an MDD process. Several conceptual problems must be solved to implement those tools. Some of them are of quite technical nature and have been long solved for textual representations; examples are versioning of models and merging changes into different models. Other problems are more conceptual in nature: keeping different models in different phases and for different purposes consistent with each other. This is, in general, called *model synchronization*, and there are already many

*Major part of this work was done at the University of Paderborn, Germany, [1].

concepts, methods, and tools that support some kind of model synchronization (see [4, 5, 6, 7] for examples). But up to now, there is no technology that fits all purposes.

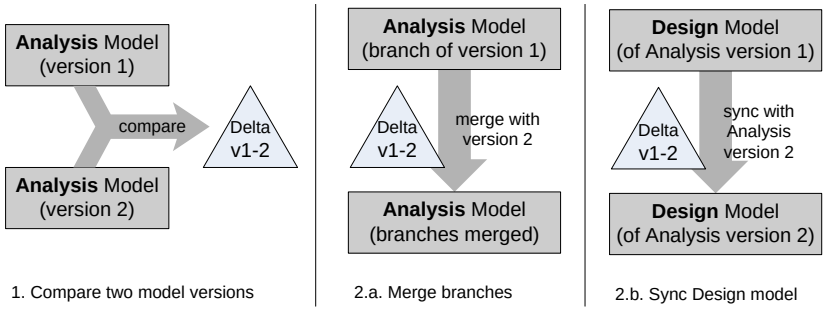

Figure 1: Diff and merge to synchronize models

In this paper, we discuss differencing, merging, and synchronizing models in the analysis and the design phase of an industrial software development process. Figure 1 gives a brief overview of these scenarios. In step 1 the delta between two model versions, analysis models in this case, is extracted for documentation and later reuse. Step 2.a uses this delta to merge version 2 with another branch of version 1. Furthermore, step 2.b uses the same delta to transfer the delta to another model type, a design model in this case. These steps are often performed manually today, so our goal is to automate them.

Our contribution consists of two parts: First, we describe how a delta between two model versions can be computed automatically and stored as a *change list*. Second, we deal with the visualization of change lists independently of the model versions it was created from. Third, a delta or parts of it can be transferred semi-automatically[1] to another model version or another model type to synchronize them. Our main concern is to apply these concepts to a real project with big models. So we finally report on the experiences we made connecting different technologies.

The rest of the paper is structured as follows: After presenting some related work in Sect. 2, Sect. 3 explains the process and tasks that will be automated by our approach. Sect. 4 presents the concepts, and Sect. 5 discusses implementation and performance issues. We conclude with a summary, compare our concepts with related approaches, and give an outlook for future work.

2 Related work

This section introduces some important terms and gives an overview of related work and tools. A comparison between our work and other approaches is given later in Sect. 6.1. In [8], Förtsch et al. introduce some categories for diff and merge based approaches and compare some of them. The most important categories are symmetric vs. directed delta and unique identifiers. Furthermore, we distinguish between model-dependent and model-independent deltas.

[1] This step can not be fully automated since conflicts may occur during a merge.

A *symmetric delta* between two model versions contains all differences between them such that it can be used to construct either version out of the delta and the other version. In contrast, a *directed delta* can only be used to construct one version from the other, i.e. not the other way round. A *model-independent delta* can be applied to another model without having the models available it was created from, and to view and even modify it independently of a model. If *unique identifiers* are used, each model element has a unique id, which does not change over the life-time of the element. Non-id-based approaches exploit similarities in the structure of a model to guess which elements of two different versions actually are the same. This has a big influence on the algorithms, since *heuristics* are less efficient than matching elements by their ids. In contrast, algorithms which use identifiers are called *exact*. Our approach is id-based and uses a model-independent symmetric delta.

Traditionally, two text files are merged by analyzing them line-by-line (*two-way merge*). However, this may lead to unprecise results if conflicts occur (see [9] for details). It is much more precise to use the common ancestor to identify conflicts between the two evolved versions, since the changes made in each of the versions are calculated prior to the actual merge. This merging technique is called *three-way merge*.

The *EMF Compare Project* [10] recently started and is not yet finished. It provides a meta-model-independent and a meta-model-specific part to compare and merge different models, even a three-way merge will be supported. It does not rely on unique identifiers but rather uses a distance relation to match the same elements. The delta is symmetric and model-dependent. This approach is applicable to arbitrary EMF models, but presumably less efficient than id-based approaches as mentioned above.

Difference and Union of models [11] is id-based and stores changes model-independently in a set of commands. A command describes a change of a model; "del('id1',Class)", for instance, removes the class with the id 'id1'. The idea is that the list of commands applied to the original version gives the second version—so the delta is model-independend and directed.

UMLDiff [12] is based on object-oriented programming code (they used Java) and looks for differences in two versions. It is non-id-based and uses a distance relation that considers structure and names.

A Metamodel Independent Approach for Difference Representation [6] offers a generic way to compare different model versions, to create deltas and to apply them. It transforms the meta model of the compared model into an extended meta model that contains three additional subclasses for each original class (one for an added, a removed, and a changed element). A delta is described using this extended meta model. A QVT transformation can then be performed to apply a delta to another target model.

The *Rational Software Architect* also supports model differencing and merging, even in a graphical way. It relies on unique identifiers and only performs in-memory merges; that means that a delta cannot be stored and reused. However, the difference presentation is nicely done in a graphical way as well as a tree, similar to EMF Compare.

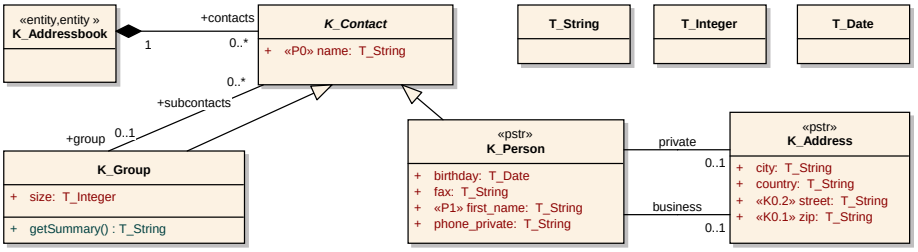

Figure 2: The analysis model for the example address book application

3 The reference process

The concepts will be integrated into the process described in this section. It was set up in the company sd&m about six years ago [3]. It uses UML class diagrams for two types of models, the *analysis model*, which shows the application or its underlying concepts from customer's point of view, and the *design model*. The latter is closer to the implementation and used by a proprietary code generator. Our concepts are based on but not limited to this process.

3.1 The models

Our example model is artificial to keep it small and simple, because the real models would be too big and too complex. However, the characteristics and modeling techniques of the real project are adopted in our simplified example. Figure 2 shows the analysis model for a simple address book application. It shows contacts, which can either be persons or groups; a person may have two addresses, a private and a business address. The prefix K describes a class in the analysis, the prefix T a datatype. Except for some extra information, which will be discussed below, this is a standard UML class diagram.

Figure 3 is a *design delta* that shows how the design model extends the analysis model. The design classes are named without a K or T prefix. Their relation to analysis classes is indicated by a UML dependency with the stereotype A2D, which stands for *analysis to design*. It expresses that design classes have all the attributes and associations of the corresponding analysis classes, unless stated otherwise. However, design classes can add or refine attributes or associations.

The main reason for sd&m to make the design model an extension of the analysis model rather than having two separate models is the following: There is less redundancy in these models since every concept is, in principle, stored only once. This way, it is easier to maintain the relation between the analysis and the design model. This presentation makes it easy to keep them manually consistent, since the models are maintained by hand.

But this choice has also some disadvantages. The design delta in Fig. 3 has almost twice as many classes as the explicit design model would have. Moreover, software engineers need to build the actual design model from the delta in their mind. To work on the design, it would be much easier to have an explicit diagram of the design. As we will see in Sect. 3.3, we can split up the analysis and the design model without loosing the advantages of the current approach.

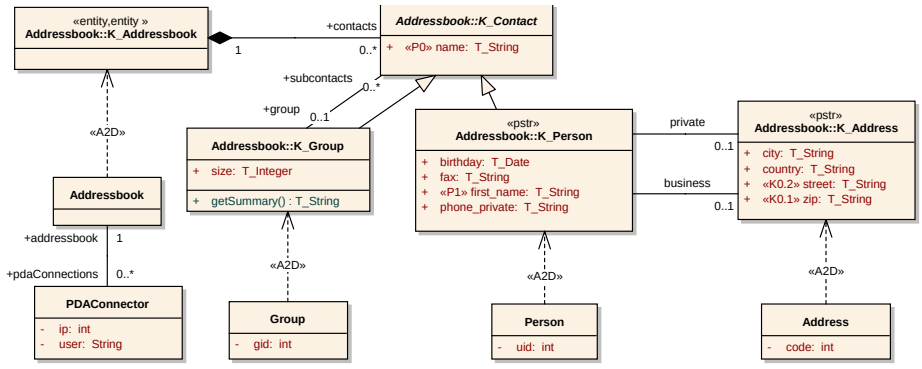

Figure 3: The design model for the example the address book application

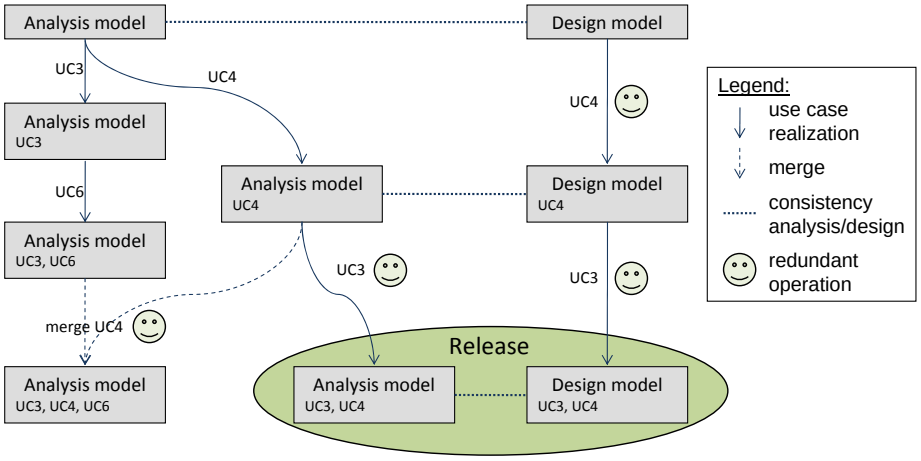

Figure 4: Extract of a development process

3.2 Scenarios: Working with these models

Next, we will briefly discuss the way these different models are used during the development process to identify some typical synchronization scenarios. The process is *incremental* and *iterative*, which means that individual features of the application are developed more or less independently of each other. Features correspond to *use cases* (UC for short) that are specified in requirement documents earlier in the process. Please keep in mind that we deal with industrial models which may be very big, e.g. more than 1000 classes.

Figure 4 shows an excerpt of the real development process for some features, called UC3, UC4, and UC6. At the beginning, which is shown at the top in the diagram, the analysis and the design correspond to each other, i.e. they are synchronized. Then, two teams start working on the analysis model independently of each other to develop UC3 and UC4 respectively. This gives rise to a *branch* of the analysis model (we indicate the covered use cases of a model version in the lower part of a box). Later a team works on UC6 integrating it into the analysis model covering UC3. After UC4 was integrated into the analysis model, the process also requires its realization in the design. Many of

the changes that are made in the design were already made in the analysis (i.e. extentions and modifications). The realization in the design needs to be done for all use cases.

In the end, there are two different versions of the analysis model, one covering UC3 and UC6, the other covering UC4 only. Both versions need to be merged. Another common scenario is the creation of a release that covers only parts of the final application. A reason might be that the customer explicitley requested such a release while some other use cases (e.g. UC6) are not yet realized. In order to create a release, an analysis and a design model are needed that cover the requested use cases, UC3 and UC4 in our example.

Many of the previously described operations were already performed in the same or a slightly different way and may be automated—so they are marked with a 'smiley' in Fig. 4. The redundant operations within the analysis seem to be straightforward, but conflicts need to considered similar to commonly known text-based merges. The redundant operations on the design model are more complicated because the structure of the design slightly differs. All these operations were performed manually by the developers until now. The models are of course stored without much redundancy, as we have seen in Sect. 3.1. Nevertheless the design has to be updated for each change. Since all these operations are performed and documented by hand they are error-prone.

Our intention is to improve and automate the redundant operations in this process in order to simplify it and to make it less error-prone. The following steps are schematically shown in Fig. 1. The key for our approach is the change list format. In step 1 we compare two model versions and store the diff fully automatically in a change list; e.g. a change list is created that contains all (and only) changes made for UC4. Then we use that change list later to *merge* UC4 into the analysis version UC3,UC6 (step 2.a). This step can obviously not be automized completly because conflicts may occur and need to be resolved. Furthermore, we can use the same change list to transfer the changes to the design in order to synchronize it with the analysis (step 2.b); e.g. the design model including UC4 can semi-automatically be produced.

The same steps enables us to produce the requested Release containing UC3 and UC4. A change list for UC3 can be produced from the original analysis model version and the one containing UC3. That list can then be used to replace the redundant operations in Fig. 4 creating analysis and design model versions covering UC3 and UC4.

3.3 Improvements

We now summarize some problems with the current development process, and show how we improve the situation by using model synchronization.

Up to now, the consistency among different versions of the models and the consistency between the analysis and the design models were maintained manully. Every change made in a model was recorded manually in a list. When some changes needed to be transfered into another model, some engineer went through this list and made all the necessary changes in the other models manually.

Our tool is able to identify all changes between two models fully automatically and compile these changes into a *change list*. So changes between two models can be documented fully automatically. Furthermore, that list is in-

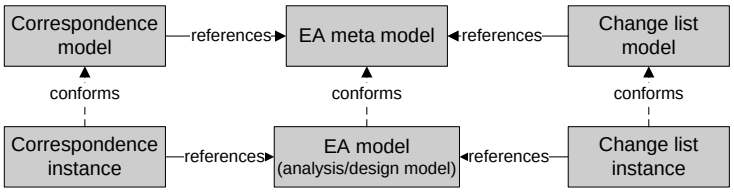

Figure 5: The meta levels for change lists and correspondences

dependent from the model versions it was created from – i.e. it can be reused without having the original model versions available. It can then be applied to other models semi-automatically, where the degree of necessary interaction depends on the scenario.

Moreover, we will make the analysis and the design model explicit, instead of working with the design delta only. This, of course, can easily result in inconsistencies between the two models. To this end, we create a first version of the design model as a copy of the analysis model, and store the relation between the two models as separate correspondences. These correspondences enable us to transfer changes to other models to keep them consistent.

4 Concepts

This section explains the concepts of change lists and their usage for model synchronization. Some issues concerning the implementation of the tool as well as its performance will be discussed later in Sect. 5.

sd&m uses the modelling tool Enterprise Architect (EA) in the process. Fig. 5 gives an overview of how a change list relates to EA models. The center column shows the EA meta model and instances, the actual EA models. Since change lists (instances) refer to EA models, the model for change lists refers to the EA meta model. The relation between correspondences and EA models is similar and will be discussed later in Sect. 4.4.

Every element of a diagram within the EA has a Global Unique IDentifier (GUID) [14] that is contained in the common superclass *EANamedElement* in the EA meta model. The GUID does not change over the lifetime of an object, so we can exploit this GUID for our concepts: we use it to identify changes of an element as well as to maintain the relation between the analysis and the design model.

4.1 Change lists

As already indicated in Sect. 3.3, the *change list* is at the core of our approach. It is for models what the result of a diff operation is for texts. The change list exactly describes all changes between two models. It serves different purposes:

1. It should be possible to reproduce the changed model from the original model and the change list; and vice versa, it should be possible to reproduce the original model from the changed one and the change list.

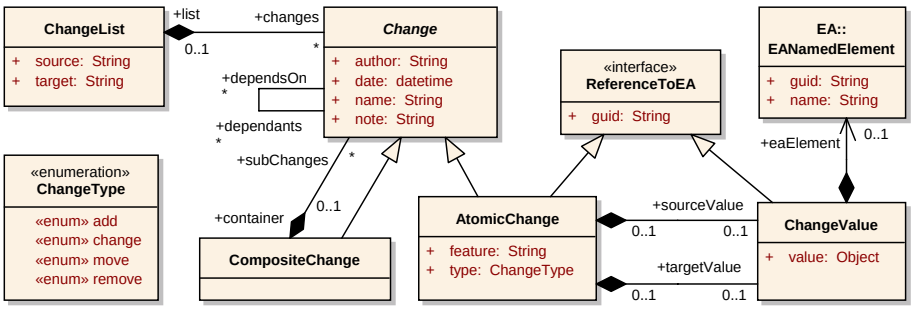

Figure 6: A model for change lists

2. It should be possible to easily merge two change lists[2].

3. It should be used for documentation and visualization of changes.

Figure 6 shows the model for change lists. This model, among other purposes, helps to easily access and manipulate change lists by other tools. With respect to documentation purposes, it should be possible to structure a change list. Therefore, the model distinguishes between *CompositeChange*s and *AtomicChange*s. Every *Change* has a name, an author, a date, and a note.

In the context of this paper, the most interesting issue are the atomic changes. There are four types of atomic changes: *add*, *change*, *move*, *remove*, which are defined in the *ChangeType*. Note that most other approaches have only three types; a move is then either represented as a *change* or as a *remove* and an *add* [12, 6, 11, 15]. But a move of a whole package, for instance, is exactly one modification in the model; a representation with remove/add would be a whole bunch of changes in the list! Furthermore, moves are required to preserve the GUIDs which are used later for incremental code generation [3]. The attribute *type* of an AtomicChange indicates its type. The attribute *feature* refers to the element that was changed; the *sourceValue* and the *targetValue* define the values before resp. after the change, if applicable for the type of change. We also need dependencies between changes to selectively transfer changes to other models (see Sect. 4.3 for details). They are stored using the association *dependsOn - dependencies*.

Note that the atomic change as well as the ChangeValue actually refer to an element of the EA model, which is why they are derived from *ReferenceToEA*. But this reference to a model element is symbolic; it refers to a GUID only. The reason for that is, that we do not want to attach a change list to a concrete EA model since it should be possible to use the change list on different models and different versions of a model. For the very same reason, the reference to the source and the target of the complete *ChangeList* is symbolic only.

4.2 Creating a change list

Next, we present the algorithm for creating a change list from a source and a target model, where the target model is a changed version of the source model.

[2]This feature is planned for the future, but not implemented yet in our tool.

Type	Feature	Description	Old value	New value	Apply	Valid	Details	Result	Apply	Valid
add	attributes	url			apply	true			apply	true
add	connectors	picture			apply	true			apply	true
add	elements	K_Picture			apply	true			apply	true
add	packages	Datatypes			applied	true			applied	true
change	name	name	K_Person	K_Individualcontact	apply	true			apply	true
change	stereotype	stereotype	P1	K1	failure	false	Exist...		ignore	lse
move	elements	T_Date	Addressbook	Datatypes	apply	true			apply	ue
move	elements	T_Integer	Addressbook	Datatypes	apply	true			ignore	ue
move	elements	T_String	Addressbook	Datatypes	apply	true			apply	true
remove	methods	getSummary			apply	true			apply	true

Figure 7: Left part: extract of a change list; right part: validation result of that list with one conflict

The algorithm is id-based, exact (id-based algorithms are usually not heuristic), model-independant, and it creates a symmetric delta[3]:

1. Preload all GUIDs of both models for fast access.
2. Iterate over all elements of the target model version. Look for GUIDs that did not occur in the source model; this indicates a change of type *add*. Moreover, compare elements that exists also on the source model and check for changes; this indicates a change of type *change* or *move*.
3. Iterate over all elements of the source model and check whether their GUID does not occur in the target model; this indicates a change of type *remove*.
4. Calculate the dependencies between the identified changes. See tab. 1 for details.

This simple algorithm identifies and stores all differences between two model versions. The left part of Fig. 7 shows the most important columns of a change list, the type, the changed feature, a description, and the old and new value (if they exist). A change of type *add*, for instance, does not contain a String as a new value; instead it contains a copy of the newly created element (cf. reference *eaElement* in Fig. 6), which is not visible in the table view. The right part of the figure is discussed in the next section.

The dependencies are most important in the calculation; they are later used for validation and change application order. A change may depedent on elements as shown in Tab. 1. A dependency between two changes is defined as follows: Change C_A depends on change C_B if C_A depends on an element that is changed by C_B and C_B is of type *add* or *remove*. Figure 7 shows an example: A new package 'Datatypes' was added (row 4), and some classes are moved to that package (rows 7-9). So all these changes of type *move* depend on the change in row 4. For validation later on, if (for any reason) the change describing the new package is invalid, then the changes describing the moved classes are also invalid. When the changes are applied to another model, the order of application is important—of course, the classes cannot be moved until the package is created.

[3]See [8] for more information about categories for diff algorithms.

Change type	Change depends on elements
change	none (besides the changed element)
add	The parent of the added element
remove	The parent of the removed element
move	The old and the new parent of the moved element

Table 1: Description of the dependent elements against change types

4.3 Applying a change list to another model version

In this section, we describe the way change lists can be used to merge changes into another version, e.g. for merging two branches (cf. Fig. 4 on page 5). The challenge is how to deal with conflicts: (1) All changes are validated against the target model version. (2) Suggestions are made how to handle each change and presented to the user. (3) Users may either accept the suggestions and apply the changes, in case no conflicts occurred. But if the users wants to change the suggestions or if conflicts occurred, they have to change some status flags and continue with step (1).

We use the flag *Valid* to express whether a conflict exists or not. The flag *Apply* expresses the suggestions and may be modified by the user.

We consider two causes for conflicts: Either elements on which some changes were made according to the change list might not exist any more, or these elements might have changed differently. In the first case, the change cannot be applied at all and must be ignored. In the second case, the conflict must either be resolved explicitely or the change must be ignored. The validation (conflict detection and suggestion making) is performed in the following way:

- If the change depends on an invalid change, this change is also invalid: *valid = "false"*, and *apply = "failure"* (may be changed to *"ignore"* by the user).
- If the state *before* the change is found in the model, the change is valid and applicable: *valid = "true"*, and *apply = "apply"* (may be changed to *"ignore"* by the user).
- If the state *after* the change is found in the model, the change is valid was already applied: *valid = "true"*, and *apply = "applied"* (cannot be changed by the user).
- In all other cases, in particular when the element on which the change applies is not found, a conflict is found: *valid = "false"*, and *apply = "failure"* (may be changed to *"ignore"* by the user).

Users have different choices to solve a conflict: Either they decide to ignore the change (and possibly modify the resulting model manually afterwards) or to make some modifications on the model or the change list first so that the change can be applied. A change list can be applied only if all conflicts are solved, i.e. all changes are either valid or ignored.

Disregarding technical details, applying valid changes is straightforward.

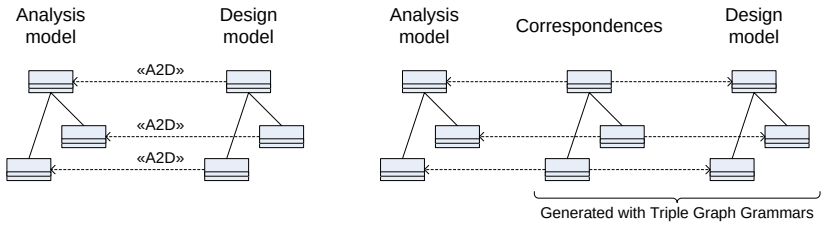

Figure 8: Left part: old relation between analysis and design; right part: correspondences maintain the relation between analysis and design

Example

The right part of Fig. 7 shows a sample validation result of the previously discussed change list. Column *Apply* indicates that one change is already applied, and one conflict was found; all other changes are ready to apply. A brief description of the conflict is given in column *Details*. The result is not set until change list application. Next, the user has to resolve the conflict. This can either be done by ignoring the invalid change (as shown in the figure), or by adjusting either the model to match the old value or by adjusting the changelist's old value to match the model. In any case, a revalidation is required.

4.4 Applying a change list to a design model

As discussed in Sect. 3.3, the development process requires that changes made in an analysis model must also be transferred to a corresponding design model. The actual transfer is very similar to a merge of two versions of the analysis model, once it is clear how the elements of the analysis model correspond to the elements of the design model. In the previous section, elements were matched using their GUIDs—that works because just a different version of the *same* model is used. The design model is another model with new GUIDs[4], so we need another strategy to match design elements to the corresponding analysis elements.

Correspondences

Until now, the relation between the analysis and the design was maintained with stereotyped UML dependencies, as indicated in the left-hand side of Fig. 8. From now on, we would like to have the design model independent from the analysis model; the relation between both models will be maintained in a separate data structure: in correspondences. We use Triple Graph Grammars (TGG) [16, 17] to build correspondences and the new design model. This model transformation and its rules are described in [1].

A correspondence element usually points to an analysis element and a design element to express the relation between them. Figure 9 shows three correspondences, represented as a chain segment. The first one describes such a relation. The second one points only to a class from the analysis model meaning that there is no corresponding design element. The last one expresses that there

[4]EA creates its own readonly GUIDs for all elements, they cannot be changed.

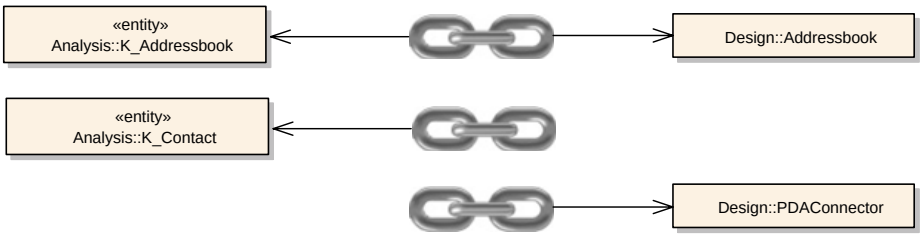

Figure 9: A correspondence element points to one analysis element and to two design elements

Figure 10: The correspondence model

is no corresponding analysis element for the design element. Thus, correspondences store whether an element does or does not have a corresponding element in the other model.

The correspondence model is shown in Fig. 10; it describes the general structure of correspondence elements (called *EA2EA*), that are represented as chain segments in Fig. 9. The reference *source* points to an element from the analysis, the reference *target* to an element from the design. In addition, the GUIDs of all elements are stored as a symbolic reference.

Applying the change list

The algorithm for applying a change list is very similar to the one in Sect. 4.3. But instead of using GUIDs stored in the change list to match elements in the design model, we need correspondences to look up the design elements.

In addition to the conflicts that are already described in Sect. 4.3, the correspondences need to be considered as well. Therefore, we use two flags for the status of a change: *transfer* specifies (changeable by the user) the scheduled action (similar to *apply* in Sect. 4.3), *correspondence* indicates (not changeable) the update of the correspondences. Furthermore, the *valid* flag is determined the same way as before.

The *transfer* flag can take one of the following values:

transfer: The change will be transferred to the design

transferred: The change is already transferred (the new value already prevails)

transfer_not: The change shall *not* be transferred to the design

ignore: Ignore this change

The status *transfer_not* and *ignore* seem to be similar but there is an important difference between them. If a change has the status *transfer_not* it will not

	add	***remove***	***change***	***move***
transfer	*create*	*remove*	*synchronize*	*synchronize*
transfer_not	*create*	*remove* *design_only*	*synchronize*	*synchronize*
transferred	*create* *analysis_only*	*remove* *design_only*	*synchronize*	*synchronize*

Table 2: The correspondence flag depends on the change type (columns) and the transfer flag (rows)

Type	Feature	Description	Old value	New value	Transfer	Correspondence	Valid	Details	Result
add	attributes	url			transfer	create	true		
add	connectors	picture			transfer_not	analysis_only	true		
add	elements	K_Picture			transfer_not	analysis_only	true		
add	packages	Datatypes			transfered	create	true	Element...	
change	name	name	K_Person	K_Individualcontact	transfer	synchronize	true		
change	stereotype	stereotype	P1	K1	transfer	synchronize	true		
move	elements	T_Date	Addressbook	Datatypes	transfer	synchronize	true	Depend...	
move	elements	T_Integer	Addressbook	Datatypes	transfer	synchronize	true	Depend...	
move	elements	T_String	Addressbook	Datatypes	transfer	synchronize	true	Depend...	
remove	methods	getSummary			transfer	remove	true		

Figure 11: Transfer changes from a change list to a design model

be transferred to the design, but a correspondence element will be created that stores this decision (cf. the middle correspondence in Fig. 9). So if the change list is validated again, this particular change will not be scheduled for transfer again. But if you have choosen *ignore* before, the correspondences will be kept untouched and the particular change will be scheduled for transfer again.

The correspondence flag depends on the transfer flag as shown in table 2; it can take one of the following values:

create: A new correspondence element needs to be created

remove: The correspondence element needs to be removed

design_only: The correspondence element only points to a design element

analysis_only: The correspondence element only points to an analysis element

synchronize: The correspondence element does not need to be changed

This time changes from a change list will be transferred to a design model. Figure 11 shows the same change list as before, the right-hand side shows the flags for the synchronization. The first four changes are newly created elements, with different synchronization intentions: The first one is scheduled for transfer, the second two will not be transferred (a correspondence element only points to the analysis elements), the next one does already exist in the design (a correspondence element will be created for future synchronizations). The remaining transfer status are self-explanatory. Note again that there is no analysis model needed to transfer the changes—only the design model, the correspondences and the change list itself are required.

4.5 Summary

In this section, we have seen how two different model versions can be compared, and how differences may be identified and stored in a change list. This change

list can then be used to view and to transfer changes to other models—even *without* having the two models from which the change list was created available. Furthermore, a change list in combination with a correspondence model can be used to synchronize with another kind of model, a design model in our case. The key to this flexible model synchronization are the different status flags and, for practice, the flexible tool for viewing and manipulating change lists and their validation results. The main challenge was the presentation of changes and conflicts to the user. See [1] for details, and http://paphko.de/ea for our tool and additional documentation.

5 Implementation

In this section, we discuss the implementation of the tool which supports our approach. As already mentioned, sd&m uses the CASE tool *Enterprise Architect* (EA) for creating and maintaining different models of a process. We decided to implement our tool as an Eclipse plugin for two reasons: First, many tools for model transformation and model synchronization are available in Java, in particular on the Eclipse platform [18, 5, 19]. Second, the EA does not support Java plugins. However, it provides a Java API that allows connecting to the EA via its *Automation Interface* [14].

This section describes the access to EA models from Eclipse via our adapter as well as the GUI for change lists. In the end, we discuss the performance of our approach.

5.1 Adapter to access EA models with EMF

The model transformation engine [5] we used for transforming the design model works natively on models of the Eclipse Modeling Framework (EMF). The EA provides a Java interface for accessing its models. We extended the EA meta model from [4], adjusted the EMF code generation templates, and generated most parts for a transparent adapter. To be more precise, EA models can now be opened, viewn, and modified from Eclipse as if they were pure EMF models. Fig. 12 illustrates this architecture. The *DLL-JAR* bridge includes access to the Automation Interface. The actual models in Eclipse are instances of our meta model and use the adapter code to wrap access to all features of the real models in the EA project.

There are already some adapters for accessing EA models from Java, but they do not meet our requirements [4, 19]: The process described in Sect. 3 uses EA specific features (*tagged values* and the *note* property [14]) that are not covered in other adapters. A round trip with those tools may even unintentionally destroy some of these information. Furthermore, our wrapper models can be parameterized in different ways. For instance, only specific sub-packages of the model may be visible or some parts of the model are hidden in order to make the access more efficient.

Figure 13 shows this adapter in action. The left-hand side shows the EA with an analysis model. On the left-hand side, the standard tree editor for EMF models in Eclipse shows the same EA model; Changes on either side will be visible in the other editor.

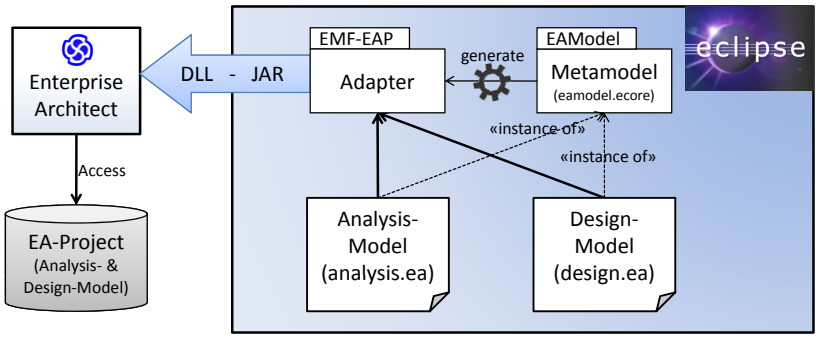

Figure 12: Accessing EA models with EMF

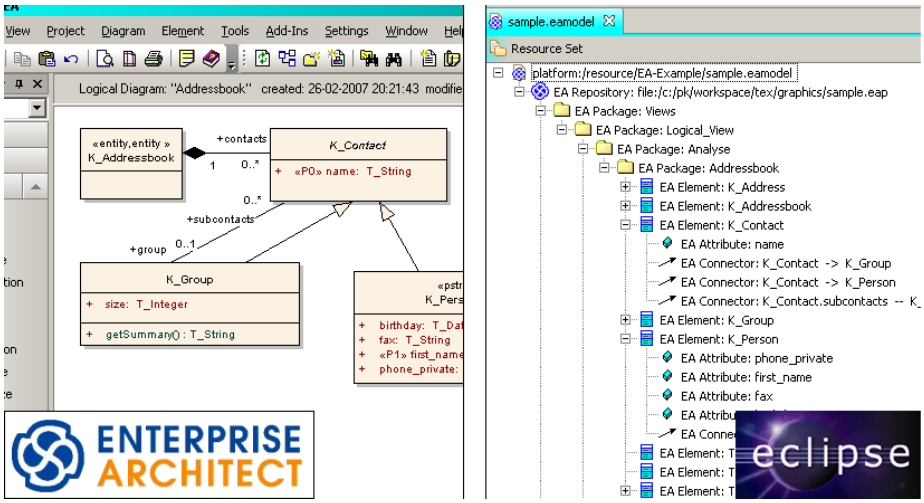

Figure 13: The adapter in action

5.2 User Interface for change list validation

The user interface for the process described in Sect. 3 consists of the EA for editing the models, and a new GUI for working with change lists. So the adapter GUI as shown on the right-hand side of Fig. 13 is not needed in the process, even though it can be used for other purposes[5]. On the Eclipse side, we have a graphical user interface for creating, presenting, and applying change lists. This GUI allows us to select two files which contain subsequent versions of *the same* EA model (an analysis model, for example). The change list is then created fully automatically, and will be saved in a file in the end. Fig. 14 shows the two different ways a change list can be presented: The left-hand side shows a table containing several changes, the right-hand side shows a tree in which all changes are hierarchically organized. Different kinds of filters and sorting as well as direct editing support are available in these views.

To apply a change list, the target model needs to be selected as shown at the bottom of Fig. 14. The panel provides functions for change list validation

[5]For instance, we used it to quickly collect statistics of some models.

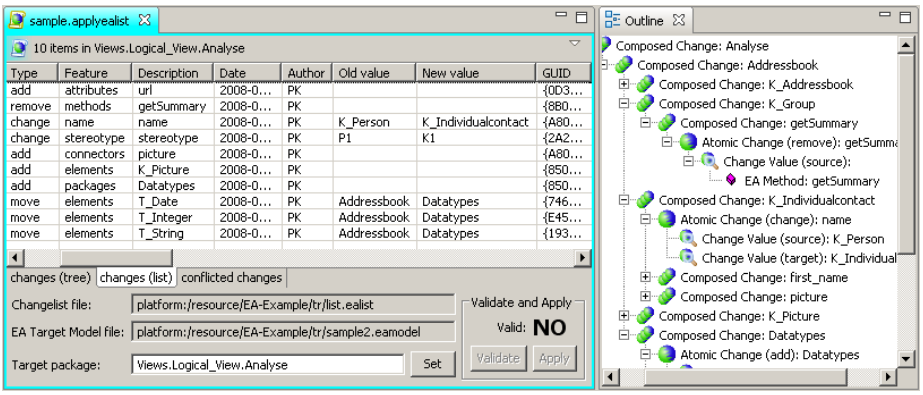

Figure 14: A change list: presented as a list and as a tree

against the target model. In this example, the target model is located in package *Views.Logical_View.Analyse* in the file *sample3.eamodel*. After validation, each change gets a *valid* and an *apply* status, as discussed in Sect. 4.2. The result of the validation is already shown in Fig. 7 on page 9.

5.3 Performance

One of the main questions of our project was, whether existing or new model transformation and synchronization technologies would be usable in industrial practice from a performance point of view. In contrast to toy examples, the industrial analysis models contained about 11.000 elements[6], and the design model even some more. Table 3 shows the times the different algorithms took with these models on a fast machine (3 GHz and 3 GB RAM). All times are divided into initialization time (loading GUIDs and building the object tree) and the actual time the algorithm needs. The bottleneck is definitely the connection between our adapter and the EA—in particular, the Automation Interface of the EA, which is very slow but stable and easy to use. For instance, we reduced the access to the Automation Interface by caching the model on the java side. That yields an immense speed-up for our algorithms which perform their work quite fast as soon as the initialization is completed.

Some of these delays, especially those with user interaction, are too high to integrate the according operations into the process. But most time-consuming operations can be performed offline without user interaction. For example, the change list creation can automatically be triggered every time a new model version is committed in the versioning system. Moreover, when applying a change list, the initialization takes most of the time. But once the initialization is done, the validation, conflict resolution, and change list application take only a few minutes. Altogether, the processing times can be distributed in a way, that the concepts can smoothly be integrated into the development process—though better performance is desirable.

[6]Elements are packages, classes and interfaces, attributes, associations, methods and tagged values (EA specific).

Operation	Time
Change list creation of two versions of the analysis model (Initialization / model comparison)	66min (2x 26min / 14min)
Applying a change list (1521 changes) to another version (Initialization / applying the list)	30min (26min / 4min)
Initial TGG Transformation of analysis model (Initialization / transformation)	69min (27min / 42min)

Table 3: Performance of different operations

6 Conclusion

The main contribution of this paper is a model synchronization concept using change lists, as well as experiences with industrial sized models. The proposed process has different scenarios that are time-consuming and error-prone due to redundant manual work. So we introduced change lists which describe differences between two model versions. They are created automatically and can then be used for model difference representation, branch merging, and model synchronization. A change list is model-independent, i.e. it can be used without having the models available it was created from.

We have developed an adapter to access models of the CASE tool Enterprise Architect (EA) from the Eclipse Modeling Framework (EMF). This made it easy to extend and use tools for model synchronization from academia, which are often based on EMF, in the industrial setting. The adapter can also be used for other purposes since it provides a generic access to models from the EA. The adapter and the tools can be obtained from http://paphko.de/ea/.

Next, we compare our approach to related work, evaluate the performance, and discuss future work.

6.1 Comparison to other approaches

None of the other approaches fits our needs to store the delta symmetric and model-independent, and are able to work with EA models. Especially the transfer of changes to another model type is unique, although [6] has a similar approach. Next, we discuss the differences to our ideas.

The EMF Compare project [10] plans to use three-way merging to identify conflicts more precisely. The validation of change lists is similar to three-way merging, because the states borfore the changes can be seen as the common ancestor. But there are some differences between merging with change lists and three-way merging: Firstly, three-way merging always needs three models[7], whereas a change list stores all necessary information with no need to have the other two models (the change list was created from) available. Secondly, change lists were designed to also transfer changes to other model types. This is not possible (not even intended) with either two- or three-way merging.

[11] distinguishes three different change types: *add*, *remove* and *change*. A moved element will be represented as by a delete and created command. But as a set of commands, this format of differences is not feasible for the use with change lists.

[7]For instance versions 1 and 2 and the branch of version 2 in Fig. 1 on page 2, so Version 1 is the common ancestor of the other two versions.

The meta model of [12] for representing differences between models is similar to our approach, unfortunately it works on code and not on models. Experiments with a lot of code showed a bad performance (e.g. creating a change list for an example model with about 800 classes took—depending on the differences—between 30 and 50 minutes).

Indeed, [6] covers most of our criteria, but unlike our approach, it cannot transfer changes selectively. The reason is that they do not maintain dependencies between changes. Furthermore, the visualization of changes is not clear arranged for big models and does not support filtering.

The rational architect [20] can only compare two versions and merge changes from the one version into the other. It relies on unique identifiers, so the algorithm for model comparisons is straight forward. Conflicts are highlighted nicely as a tree or in a small diagram excerpt. But it is not possible to store deltas for later reuse or to synchronize other models. Furthermore, it does neither support model-to-model transformation nor model synchronization—however, it supports synchronization of code with a corresponding UML model.

6.2 Performance

The experiments with industrial models show that there is some need for improving the efficiency of the tools. The bottleneck is the Automation Interface of the EA, which makes model access easy at the cost of speed. At least we found a way to integrate our concepts into the process with accaptable delays, for instance by automatically scheduled change list creation. In addition, we have some ideas on how to improve the performance—but this is future research. Of course, it would be much nicer to integrate our tool into EA, but Java is not supported by EA yet. However, the API is mostly the same, so there will not be a remarkable increase of performance.

6.3 Furture work

We achieved most of our goals, except for the following. Change lists cannot be concatenated easily, because the dependencies between changes are consistent within one change list only. If other changes are added to a change list, they might conflict with existing changes. So to support change list concatenations, the resulting change list must be checked for conflicts and the dependencies need to be re-calculated.

Our work gives also rise to interesting conceptual questions: The tool allows us to identify differences between an analysis and a design model in an MDD process. Conceptually, these changes are—or at least document—the design decisions taken in the project. These changes are currently maintained on a very low technical level so that they are not of much help for documenting the design decisions. But we believe that from this low-level information, the design decisions on a higher level can be generated automatically, and presented to the developer. This, however, is subject to future research.

Acknowledgement. We would like to thank sd&m for the insight into their processes and the support we got from them. Many thanks to Johannes Jakob and Alexander Königs for their Enterprise Architect meta model [4].

References

[1] Könemann, P.: Verbesserung eines modellbasierten Softwareentwicklungsprozesses mit Hilfe von Modellsynchronisation. University of Paderborn, Software Engineering Group, Germany, Master thesis (2007)

[2] Object Management Group: MDA Guide v1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01 (2003)

[3] Unland, L., George, T.: MDA richtig einsetzen: Klassische und innovative Rezepte. OBJEKTspektrum **6** (2005) 41–44

[4] Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink, A., Warmer, J., eds.: Model Driven Architecture – Foundations and Applications: Second European Conference. LNCS 4066, Springer, (2006) 361–375

[5] Greenyer, J.: A study of model transformation technologies: Reconciling TGGs with QVT. University of Paderborn, Software Engineering Group, Germany, Master thesis (2006)

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

[7] SmartQVT http://smartqvt.elibel.tm.fr/ (2008)

[8] Förtsch, S., Westfechtel, B.: Differencing and Merging of Software Diagrams – State of the Art and Challenges. In Filipe, J., Helfert, M., Shishkov, B., eds.: International Conference on Software and Data Technologies (ICSOFT), Setubal (Portugal). Volume 2., Institute for Systems and Technologies for Information, Control and Communication (2007)

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

[10] Toulmé, A.: Presentation of EMF compare utility. In: EclipseCon. (2007)

[11] Alanen, M., Porres, I.: Difference and union of models. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 – The Unified Modeling Language. LNCS 2863, Springer (2003) 1–17

[12] Xing, Z., Stroulia, E.: UMLDiff: An algorithm for object-oriented design differencing. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, ACM (2005) 54–65

[13] Greenyer, J., Kindler, E.: Reconciling TGGs with QVT. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 16–30

[14] Sparx Systems Pty Ltd Victoria, Australia: EA User Guide (2008)

[15] Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2003) 227–236

[16] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Tinhofer, G., ed.: WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science. LNCS 903, Springer (1994) 151–163

[17] Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn (2007)

[18] ATLAS Transformation Language http://www.eclipse.org/m2m/atl/ (2008)

[19] openarchitectureware.org http://www.openarchitectureware.org/ (2008)

[20] IBM: Rational Software Architect http://www-306.ibm.com/software/awdtools/architect/swarchitect/ (2008)