Technical University of Denmark

Model-Independent Diffs

Könemann, Patrick

Publication date:
2008

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

Link back to DTU Orbit

Citation (APA):
Könemann, P. (2008). Model-Independent Diffs. Lyngby: Technical University of Denmark, DTU Informatics, Building 321. (D T U Compute. Technical Report; No. 2008-20).

Model-Independent Diffs

Patrick Könemann
Technical University of Denmark

Technical University of Denmark

IMM-TECHNICAL REPORT-2008-20

December 19, 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

Model-Independent Diffs

Patrick Könemann

Technical University of Denmark, Informatics and Mathematical Modelling
pk@imm.dtu.dk

Abstract

Computing differences (diffs) and merging different versions is well-known for text files, but for models it is a very young field. Text-based and model-based diffs have different goals and different starting points, because the semantics of their structure is fundamentally different. Text files just contain a list of strings, one for each line, whereas the structure of models is defined by their meta models. There are tools available which are able to compute the diff between two models, e.g. RSA or EMF Compare. However, their diff is not model-independent, i.e. it refers to the models it was created from. We present concepts for model-independent diffs, how it can be created and used. In the end, we present an idea of how the diff could be generalized, e.g. many atomic differences are merged to a new, generalized diff – similar to a patch for text files. The advantage of such a generalized diff is that it is applicable to a higher variety of models.

Keywords. model differencing, emf compare.

1 Introduction

Text-based diff & merge[1] is used to compute differences between two different versions and merge them, e.g. if many developers are working on the same files. It is well-known in software development, but only applicable to text files. In the time of model-driven software engineering, models are used for software development and, in particular, for generating code. This makes diff & merge for models desirable as well.

A textual diff can be stored as a *patch*, which is a self-containing file describing all differences between two versions of one or more text files. Its intention is to store the diff and make it applicable to other files, maybe on another workspace on which the original text files are not available. Moreover, a patch describes both states, before and after the change. This makes it possible during application of a patch to identify whether it was already applied or not; in addition, a patch can also be used in reverse direction, i.e. the changes can be undone.

Some approaches like RSA [IBM08] and EMF Compare [Tou07] provide support for diff & merge of models. RSA performs it in-momory, whereas EMF Compare is also able to store the diff in a file for later re-use. However, it always refers to the models it was created from, hence it depends on the models and cannot be

[1]We use the short form *diff & merge* for the term *differencing and merging*.

used as a patch for text files. This report introduces a way to *describe model-based diffs independently from the models they were created from*, hence such a model-independent diff can be used as a patch for models.

The paper is structured as follows. Sect. 2 compare text- and model-based diffs and motivates our work. In Sect. 3, we present our concepts for model-independent diffs, whereas in Sect. 4, we propose an idea for generalized changes. We end the paper with the presentation of related work in Sect. 5 and a summary in Sect. 6.

2 Text- vs. model-based diffs

The main differences between text- and model-based diffs are shown in Table 1: it opposes the structure, referencing, comparison strategy, and accuracy of compare results. These properties of model-based diffs lead to the following challenges which need to be solved for creating model-independent diffs. Furthermore, we already propose our approach for solving them, which is described in more detail in Sect. 3.

Text-based diffing	**Model-based diffing**
A text file is represented as a *list of lines*, each containing a string.	A model is mostly represented as a *graph*, i.e. model elements may reference each other.
The only way of pointing to some place in the text file is by addressing the *line number*.	A model element can be referenced differently, e.g. by *structure* or by a *unique identifier* (ID).[2]
Comparisons are usually performed *line-by-line*, so this structure is used for comparing text files. There are different ways of representing changed lines, some examples are given in [MES03].	Models do not have a common structure (e.g. as lines in text files), but they may be *structured as a tree* or may have *unique IDs*. So the compare-strategy depends on the type of model.
Text file comparisons are *not exact*, because the contents of a text file cannot be addressed exactly.[3] Substrings are compared and heuristics are used to find the correct locations for applying a patch. Again, [MES03] gives some details concerning accuracy.	If model elements have unique IDs, then the diff is *exact*. However, if the model does not have unique IDs, then the diff is also *not exact* and usually based on heuristics. Mostly, a combination of the structure and attribute values is used to match elements.

Tabelle 1: Comparison of textual and model-based diffing

[2]If an element has a unique ID, it can be addressed uniquely during its entire life-cycle; the ID also remains the same across different versions of the model.

[3]Let us assume, a patch points to line 5 of a file; but before applying the patch, 2 new lines were added before line 5, so the patch must be applied to line 7 now. This is why some substring before and after each actual difference is stored as well, which can be used to find the relevant locations in the file.

How is a model structured?
A text-file always contains lines, each containing a string. However, models may consist of arbitrary elements, having attributes, references, and maybe other properties – so the structure of models may vary.
Our approach: In order to support comparisons for a particular kind of models, we have to agree on a common meta model which describes the structure of our models. A good choice could be the Meta Object Facility *(MOF, [OMG06a]), since it is the basis for many modeling languages used in software engineering, such as UML [OMG07] and EMF [emf08].*

How to reference changed elements?
In text files, each place can be addressed using a line number – but this is not the case for models. If we are lucky, each element can be addressed via a unique ID, but that depends whether the meta model enforces unique IDs for each element. If not, the structure of the model or the values of the elements can be used to match common elements. But it can even be worse: what if a model does not contain a sophisticated structure and maybe too few attributes to compare?
Our approach: If IDs are available, they would be the easiest way to address elements. If that is not the case, we need heuristics to match common elements, similiar as the text-based approach does it. The combination of the elements' structure and their attribute values is a very common strategy that is often used. But maybe this is not the best strategy either – we cannot decide a proper strategy for each model! Instead, we propose an interface for addressing model elements; implementations may use e.g. IDs, heuristics, or another individual strategy.

How to store the addition of a new model element?
A *patch* for text-files just contains the added lines (i.e. the strings) and some line numbers. This works, because the lines are independent or the rest of the file. But that is not so easy for models for two reasons. First, model elements may have a more complex structure than just strings. They may have different attributes, maybe even sub-elements – hence we need to store sub-models. Second, the newly added element may contain references to other model elements, which is again the previous challenge. This problem does not occur in text files, because they do not have cross-references.
Our approach: We need some kind of descriptor, that sufficiently describes a sub-model, including multiple elements and references to other elements that may not be contained in this particular sub-model. It should work to use the same kind of references described previously.

All these problems are already discussed and solved specifically for models of the tool Enterprise Architect [KKU08]. However, it is hardcoded for the meta model used, and especially for the unique IDs, of the Enterprise Architect. To transfer these concepts to other models, they need to be re-coded for the respective meta model. Our goal is to abstract from such particular meta models and to generalize this approach for arbitrary models. To do so, we formulate these concepts on a higher level, namely for a subset of *MOF* (Meta Object Facility [OMG06a]). As an implementation, we have chosen *EMF* (Eclipse Modeling Framework [emf08]); one reason was the sub-project EMF Compare, which already contains a sophisticated diffing framework and supports most of the just presented diffing capabilities. Furthermore, a UML meta model based on EMF exists, so the concepts developed here are applicable to these models as well.

Having these technologies available, we can use the existing compare engine of EMF Compare to first create a model-dependend. Then we focus on the model-independent representation of diffs, so we do not need to consider the diff creation process. Figure 1 gives an overview of the creation and application of model-independent diffs:

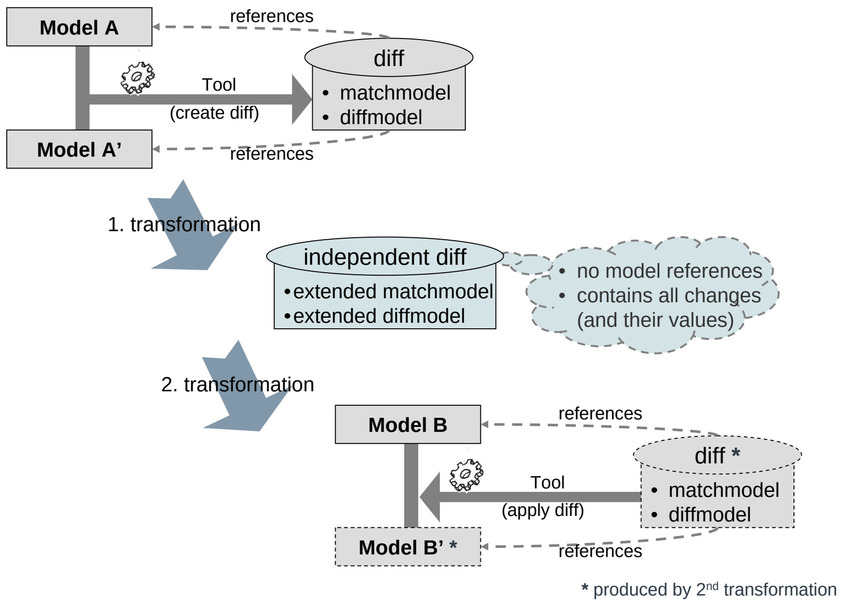

Abbildung 1: Transformations between model-dependent and model-independent diffs

1. Create a model-dependent diff from two versions of model *A* with an existing tool. The diff does not contain the actual differences, it just refers to the changed elements of the source and the target model and gives some information about the difference itself.
2. In the 1st transformation, the model-dependent diff is transformed into a model-independent diff with respect to the three main challenges described above.
3. In order to apply the diff to another model *B*, the references need to be restored and conflicts need to be identified. To do so, the 2nd transformation creates a new diff and a temporary model *B'*. Then, the very same tool from step 1 can be used to visualize and resolve the conflicts.[4]

The concepts covered in this paper focus on the properties of the first transformation and some on some thoughts how to realize the second transformation.

[4]The second transformation, however, is not described in this report but is future work.

3 Diffing models

A model-independent diff has special requirements as already mentioned previously.

- It must describe all differences independently from the originating models, i.e. it must contain the actual values which changed (EMF Compare, in contrast, just refers to the models and describes *where* something was changed – so the actual differences are implicit and have to be computed form the models on the fly).
- It must contain information to find the changed element in arbitrary models; for instance by using IDs or structural similarities.
- We need to consider at least 9 types of changes: elements, attributes, and references, each may be changed, added, or removed. A three-way-compare (see [Men02] for details) is not considered – this is why the EMF Compare meta model[5] has 15 change types.
- Conflicts are not considered, because we only want to store non-conflicting diffs. Later, if changes are applied to a particular model, there might be conflicts – but this is not of interest at the moment.

Example

We present a small example before explaining our meta model for model-independent diffs. Figure 2 shows a simple library meta model, in which books have a title and a catalogue number (which identifies a book uniquely). A model (in the middle of the figure) just contains one book titled *Galaxy*. Next, we create a copy of the model, change the name to *Guide* (right-hand side of the figure), and show how a model-independent diff could look like.

The diff is shown as a small object diagram, containing an object of type *IndepAttributeChange*, which represents the actual difference, and an object of type *IdEmfReference*, which is a symbolic reference to the changed element. In order to be model-independent, the diff must not contain a direct reference to the models (similar, a patch for text files is also independent from the text files it was created from). So we just need to store some information to address which element was changed, namely the unique identifier *catalogueNr* of the book.

The other parts of the diff are simple: The old as well as the new value of the changed attribute are stored in *oldValue* and *newValue*. The information of which attribute of the referenced element was changed, is given in a reference to the library model, which points to the according attribute in the library model. So the model-independent diff in this simple example only consists of a changed attribute. The following proposes a meta model for such diffs.

[5] In tutorial *Using the Compare Services* of EMF Compare documentation; direct link: http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.emf/org.eclipse.emf.compare/doc/org.eclipse.emf.compare.doc/tutorials/images/DiffMM.jpg?revision=1.1&root=Modeling_Project

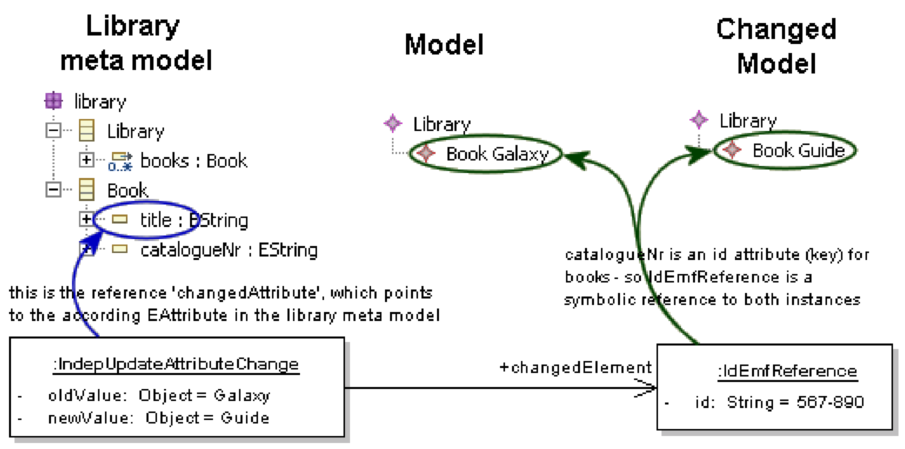
Library meta model
Model
Changed Model
library
Library
books : Book
Book
title : EString
catalogueNr : EString
Library
Book Galaxy
Library
Book Guide
this is the reference 'changedAttribute', which points to the according EAttribute in the library meta model
catalogueNr is an id attribute (key) for books - so IdEmfReference is a symbolic reference to both instances
:IndepUpdateAttributeChange
- oldValue: Object = Galaxy
- newValue: Object = Guide
+changedElement
:IdEmfReference
- id: String = 567-890

Abbildung 2: Parts of a model-independent diff of a changed attribute

Meta model for model-independent diffs

We present our meta model for model-independent diffs in three class-diagrams. The first one is shown in Fig. 3 which contains the root element for diffs, *IndependentChangeModel*, as well as several classes for the actual changes, and two interfaces (*IModelDescriptor* and *IElementReference*). All classes colored cyan (subclasses of *IndepChange*) describe the actual changes (i.e. elements, attributes, or references can be added, removed, or changed), similar to the diff meta model of EMF Compare.[6] Imported classes from other packages are colored gray with the package stated in paranthesis, e.g. *EAttribute (from ecore)*. The tricky parts in this meta model are the (roughly corresponding to the challenges in Sect. 2):

1. *References from a diff to the meta model of the changed model* (in the example in Fig. 2, it is the reference from the diff to the library meta model) are realized with references to *EAttribute* and *EReference*, which are part of the EMF ECore meta model.
2. *Symbolic references* from the diff to elements in the changed model are represented by the interface *IElementReference*. It is explained in more detail in Sect. 3.1.
3. *A descriptor for added or removed model elements* (and for whole submodels) are represented by implementations of the interface *IModelDescriptor*. An important aspect is to also remember all references to other elements. This concept is explained in Sect. 3.2.

[6]Concerning an emfdiff of EMF Compare, there will be a 1:1 correspondence between a *DiffElement* and an *IndepChange* of a model-independent diff.

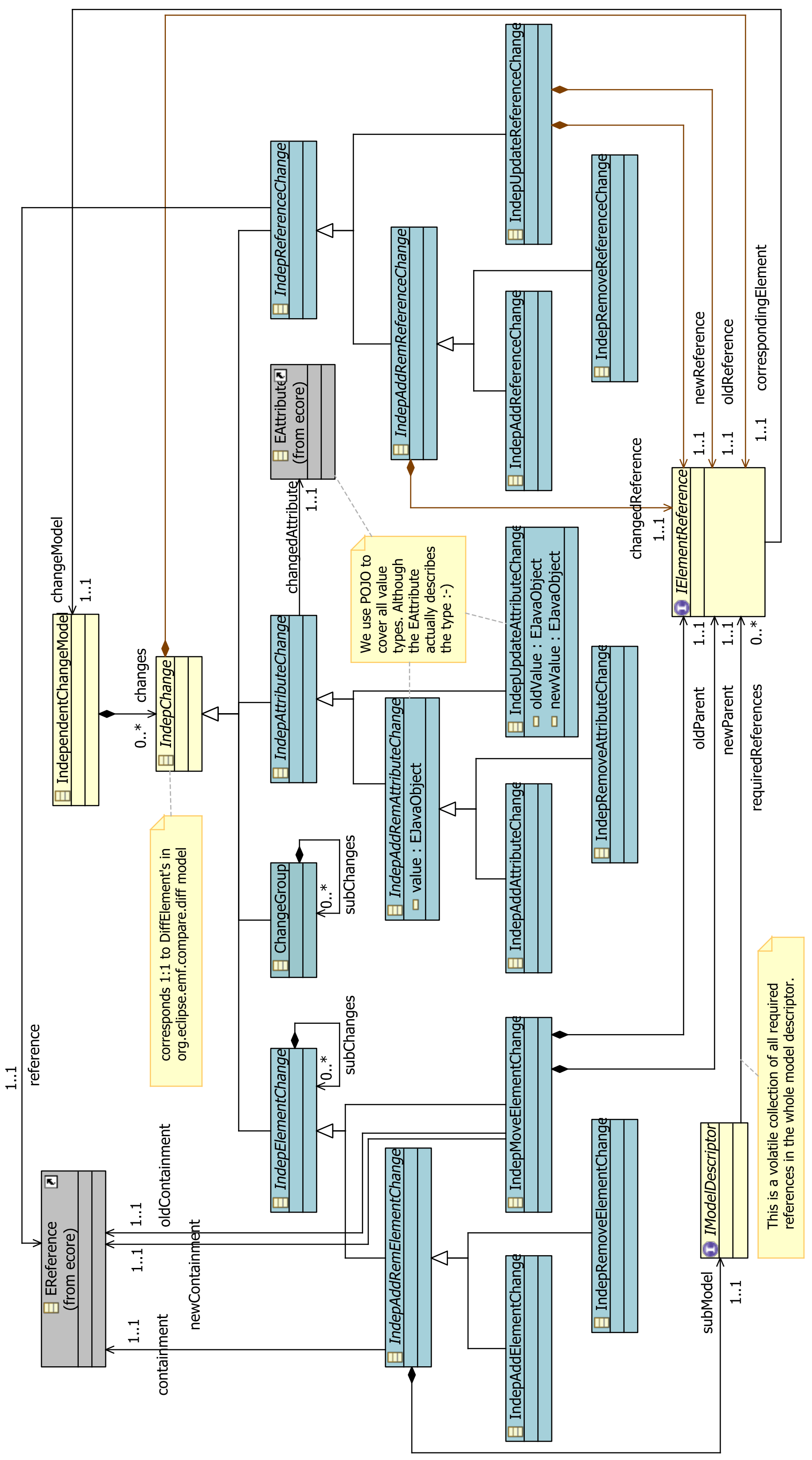

Abbildung 3: Meta model for model-independent diffs

3.1 Symbolic references

A model-independent diff needs to point to changed elements without directly referring to them. So we would like to use *symbolic references* (in literature also called *indirect references*) in order to separate the diffs from the models. But what is a symbolic reference? We found a nice explanation in the book "Inside the Java Virtual Machine":

"A symbolic reference is a character string that gives the name and possibly other information about the referenced item – enough information to uniquely identify [it]." [7]

Unlike direct references, symbolic references do not require the referenced items to be available; however, a symbolic reference can be resolved to a direct reference which can be seen as a direct pointer to the de-referenced item. The example in Fig. 2 already motivated the need of such references. Instead of using a *character string*, we use the following meta model for describing symbolic references in model-independent diffs.

Meta model for symbolic references

Symbolic references are used in the meta model for independent diffs (Fig. 3) in many places (interface *IElementReference*). As explained before in Sect. 2, there might be different ways of pointing to model elements. Let us first distinguish between model elements that have a unique identifier, and those which do not. In the first case, symbolic referencing can easily be done using the unique identifier, which – by definition – identifies the element uniquely during its entire life-cycle; the ID also remains the same across different versions of the model. In the other case, we need *some other information about the referenced item*, for example its attribute values, some structural information, or its neighbour elements. The diagram in Fig. 4 shows three possible implementations for symbolic references in EMF models.

The class *IdEmfReference* can be used to refer to elements which have an attribute marked as a unique identifier. For all other classes, we need to store *some other information*.The *ElementSetReference* has a set of conditions, e.g. in OCL (Object Constraint Language [OMG06b]), which can be used to identify one or a set of elements. We will see an example for that later in Sect. 4.1 on page 15. The *StructureEmfReference*, on the other hand, contains a sub-model, which is supposed to contain sufficiant information to identify that particular item in a model. Then, the reference *elementDescriptor* points to the referenced item in the sub-model. Next, we present an example for a *StructureEmfReference*.

Example for symbolic references

The object diagram in Fig. 5 shows two symbolic references from the previously used example. The reference to the book is ID-based, whereas the reference to *HitchikerTricks* is a *StructureEmfReference* and thus not ID-based.

[7] Available at `http://www.artima.com/insidejvm/ed2/securityP.html`

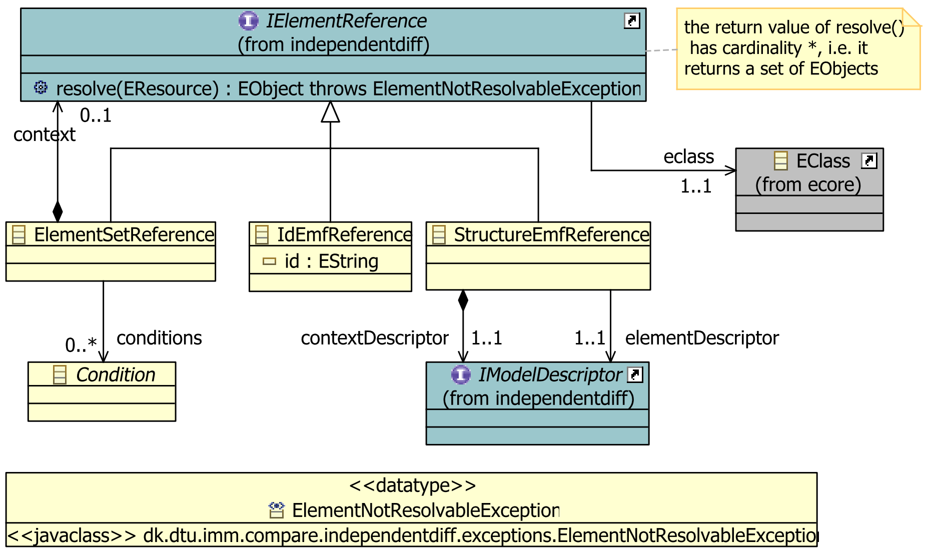
IElementReference
(from independentdiff)
resolve(EResource) : EObject throws ElementNotResolvableException
the return value of resolve() has cardinality *, i.e. it returns a set of EObjects
0..1
context
eclass
1..1
EClass
(from ecore)
ElementSetReference
IdEmfReference
id : EString
StructureEmfReference
0..*
conditions
Condition
contextDescriptor
1..1
1..1
elementDescriptor
IModelDescriptor
(from independentdiff)
<<datatype>>
ElementNotResolvableException
<<javaclass>> dk.dtu.imm.compare.independentdiff.exceptions.ElementNotResolvableExceptio

Abbildung 4: Meta model for symbolic references

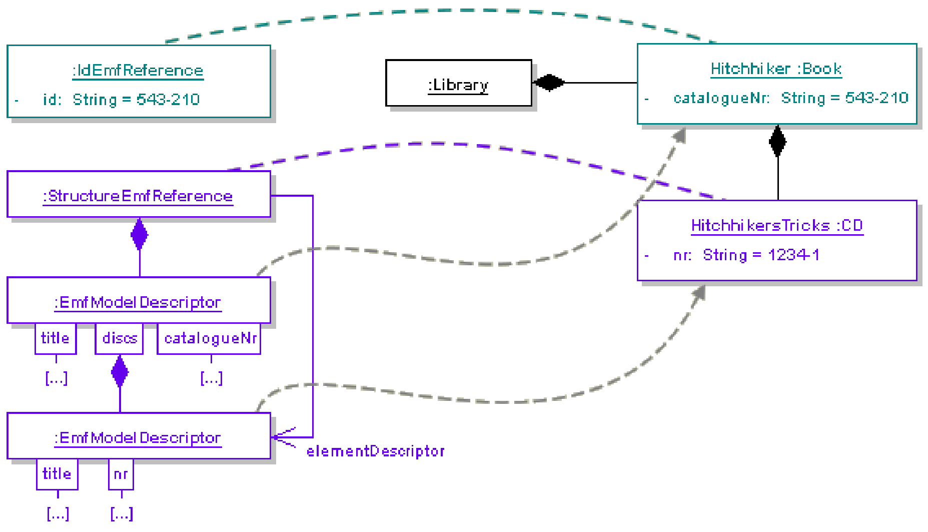

Abbildung 5: Example for symbolic references

The right-hand side shows our example again, whereas the left-hand side shows the two symbolic references mentioned. *IdEmfReference* uses the catalogueNr attribute of the book as a unique ID. The CD does not have a unique ID defined in its meta model, hence the *StructuredEmfReference* is used as a symbolic reference. It contains a model descriptor containing the attributes of the element itself as well as its parent. Furthermore, *elementDescriptor* points to the wanted element in the sub-model. This gives a lot of information which can be used to resolve the symbolic reference, which is covered next.

Resolving symbolic references of type *IdEmfReference*

Considering the scenario that the model-independent diff from the previous example should be applied to another model, then both symbolic references the need to be resolved. The *IdEmfReference* is easy, because we just need to 'ask' the model for the element with the specified unique ID. Fig. 6 shows the symbolic reference on the left, a temporary model in the center, and another model, for which the symbolic reference should be resolved, on the right. The symbolic reference to the CD is not ID-based, so we cannot simply 'ask' the other model for a reference to the element. Instead, we suggest an idea of how de-referencing could be realized:

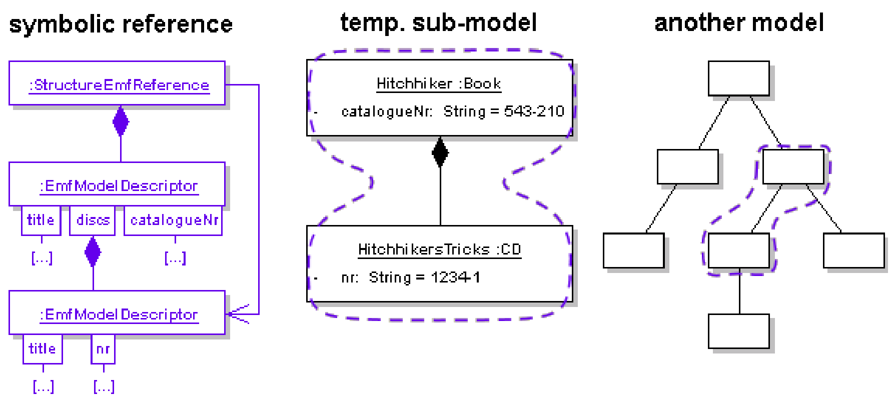

Abbildung 6: Resolving symbolic references

1. Use the model descriptor of the symbolic reference to build a temporary model (in the middle of Fig. 6).
2. Use a matching engine in EMF Compare to compare the temporary model with the other model. Let us assume that the marked elements on the right-hand side are matched by the match engine.
3. The element in the other model that matches the requested element in the temporary model is then the resolved symbolic reference.

An important issue is, how much information is needed in order to provide a sub-model that is sufficiant to uniquely identify the element. Here, we only use the element itself and the parent, but maybe it would be a good idea to as well add all elements directly connected to the referenced element.

Notes

We have shown ideas how to implement symbolic references for elements with and without unique IDs. Sect. 4.1 explains the idea of de-referencing sets of elements in more detail. However, we focus on ID- and condition-based symbolic references at the moment, to keep things simple. But the interface is built in a way such that other strategies for symbolic referencing are possible as well.

3.2 Descriptor for sub-models

One of the challenges is the description of a sub-model in a model-independent diff. The example in Fig. 7 shows a simple object diagram, a library containing two books titled *Galaxy* and *Universe*. There are several elements stereotyped with «add», which means that they have been added to the model: a book titled *Hitchhiker* with a CD *HitchhikersTricks*, and two references between the new and the existing books.

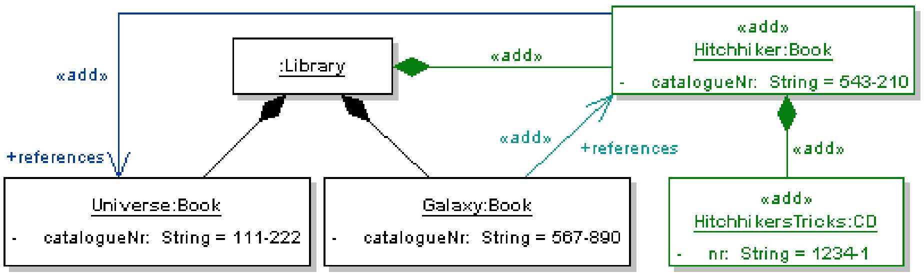

Abbildung 7: Two elements are added as a sub-model

There are actually two logical changes made to the model: first, a new book with a CD was added; second, a reference from the book *Galaxy* to the new book was added. Next we will see how these changes are expressed in a model-independent diff.

Additions in model-independent diffs

Regarding two versions of a model, a newly created element exists in the later version but not in the ealier one. In order to define an addition, we need to know what *exists* means. Concerning model-driven development in general, there might be many models involved, also referencing each other. With this regard, we say that elements *exist* in a model, if they are *contained* in that particular model. Do we consider containments as the main structure for all models, as it is the common case for EMF models. This has two important consequences: First, all diffs concerning elements are based on containments, hence the addition, deletion, and movement of model elements. Second, although containments are a special type of references, they are not covered as reference changes.

With regard to the challenge as described in Sect. 2, there are several aspects we have to consider:

1. We have to store an entire hierarchy of model elements,
2. including their attribute values,
3. and also their references to other model elements.

According to our meta model, all green elements in the example above (classes *Hitchhiker* and *HitchhikerTricks*) would be part of an *IndepAddElementChange* (cf. Fig. 3). An implementation of *IModelDescriptor* is then responsible for describing all green elements independently of the actual model. Furthermore, it has to store the reference from *Hitchhiker* to *Universe*, because it is part of one of the newly added elements. Fig. 8 shows our meta model for such model descriptors.

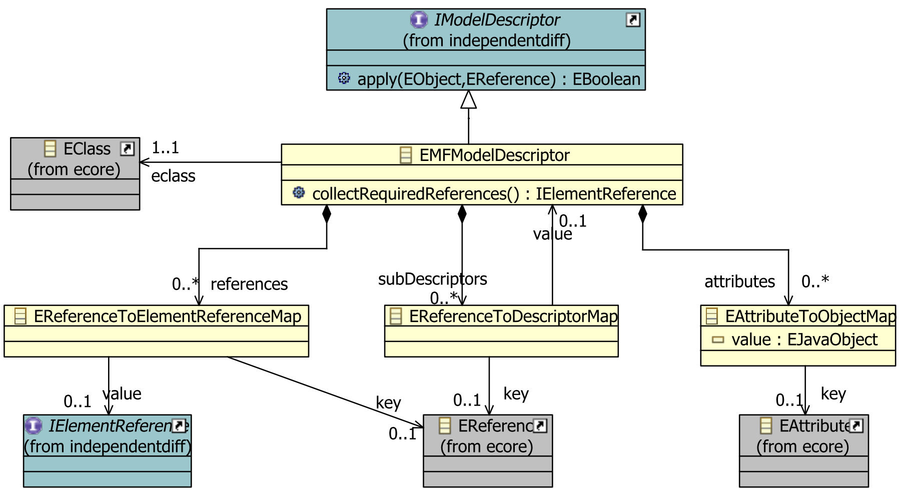

Abbildung 8: Meta model for sub-models

The reference to the EClass points to the type the particular model descriptor represents. In this case it would point to the EClass *Book* of the library model (cf. Fig. 2). Moreover, there are three maps for a model descriptor:

1. The *ERerefenceToElementReferenceMap* takes EReferences as a key (the reference from *Hitchhiker* to *Universe*, for example), and an IElementReference as the value. *IElementReferences* are symbolic references, which address model elements without directly referring to them (cf. Sect. 3.1).

2. The *EReferenceToDescriptorMap* takes only containment references as keys and contains other model descriptors. In the example, it contains a descriptor for the element *HitchhikersTricks*.

3. The *EAttributeToObjectMap* contains the values for all attributes of the particular element. For our example, it contains two entries: One with the *EAttribute* catalogueNr (cf. library model in Fig. 2) as a key and the String “543-210” as a value, and one with title as a key and the String “Hitchhiker” as a value.

Next, a model-independent diff is shown in abstract syntax, i.e. as an object diagram, in the bottom part of Fig. 9 which describes all added elements in the example from Fig. 7: the *IndepAddElementChange* for the library element (the *StructureEmfReference* again is an implementation of the *IElementReference* interface) contains a model descriptor for all added elements. The relation between the independent diff and the library model is shown in Fig. 9 using curved arrows. The solid blue arrows (pointing to the library meta model) are references to the library model as required by the descriptor. Values in the three maps are represented as qualified associations, using the blue dashed library model elements as keys. The green arrows (pointing to the library models) again represent symbolic references.

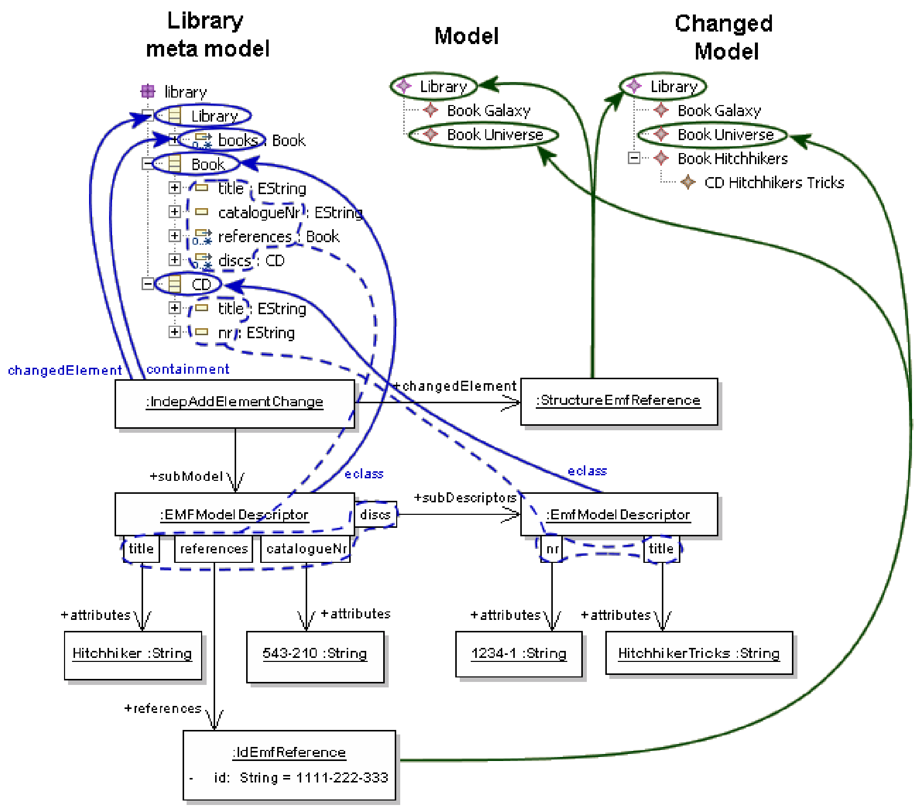

Abbildung 9: Connection between a model-independent diff (in abstract syntax) and the actual models

Please note that the reference from *Galaxy* to *Hitchhiker* is added to an already existing element, so it is not part of this particular change. But it depends on the just explained change! We will discuss this issue later in Sect. 3.3.

Removals are handled the very same way, this is why sub-models are contained in the abstract class *IndepAddRemElementChange*.

3.3 Change dependencies lead to groups

Unlike textual diffs, changes in models may *depend on each other*. To understand this, it is important to understand the meaning of the terms *diff* and *change*. As the name says, a *diff* just contains and describes the structural differences between two files or models. In contrast, a *change* refers to the action of changing something (resulting in a diff).

If we take the example from before (Fig. 9 on page 13), there are two changes made: the first includes a newly added sub-model containing the elements *Hitchhiker*, *HitchhikerTricks*, and a reference to the book *Universe*; the second just includes a reference from the existing book *Galaxy* to the new book. The second change obviously depends on the first one, because it uses one of its elements – consequently, the second change is useless if the first one is not applied. We decided to use these dependencies to logically group changes.

This is, of course, only a very simple example. But if we consider larger models with a lot of changes that depend on each other, then it would be nice to see structured groups which logically represent sets of independent changes. Using the previously proposed meta model, it would be easily possible using *ChangeGroups*. The idea is that *ChangeGroups* do not interfer with each other. This of course opens up potential for further optimizations, e.g. sub-grouping.

4 Generalization

Normal diffs contain all differences between two models, usually as a whole bunch of atomic changes. The example in Fig. 10 again shows the library with some books. There are three changes which are structurally similar and each of them produces an entry in a diff. Three new references point from a customer to books, marked with the stereotype «add».

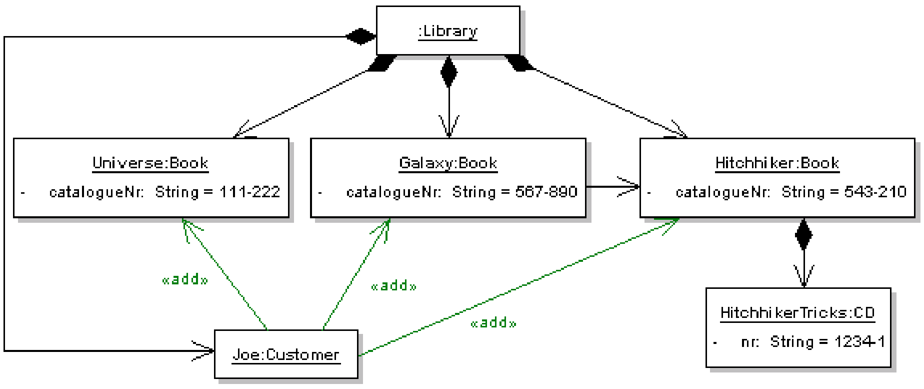

Abbildung 10: Similar changes: three new references

However, if this diff should be applied to another model (i.e. similar to a patch), it requires exactly these three books. So it is not possible to apply this diff to another model with different books, e.g. with different attributes. It might be useful in some cases to precisely refer to particular model elements, but in some cases it might be useful to weaken the precision of the target model elements in order to create a diff that can be used similar to a patch for text-files. For example, we could re-phrase our three changes to: *new references were added from the customer 'Joe' to all books in the library the customer belongs to.*. Thus we weaken the description of the changes in such a way that it fits for all atomic changes at once, in order to find a more concise and adequat represenation of these changes. Consequently, we can probably apply this change to other models with other libraries even with different books. We call this kind of weakening of changes *generalization of changes.* Next, we briefly mention some other scenarious, and explain how we use symbolic references to obtain this generalization.

More examples for generalizations

Starting from the example above, we would like to have *one* change describing *many* references from a customer to *many* books. This is the first case of the following list of possible generalized changes, which is not exhaustive but should be enough for now:

- adding / removing *many* references to / from an element
  (we have just seen an example of this case)
- adding / removing / changing a reference of *many* elements
- adding / removing / changing an attribute of *many* elements
- adding / removing a sub-model to / from *many* elements
- moving *many* elements from one place to another

The obvious advantage would be that such a change is more general and hence applicable to much more models than just atomic changes. It is also more concise and user-friendly than many atomic changes actually describing the same intentional change.

On the other hand, generalizations do also have some drawbacks! If a change addresses many elements, this set of elements might include some, which are not intended to be part of the change – for example, there could be another book in Fig. 10 for which there was not added a references. Thus, it might be a good idea to specify exceptions, or in general, to have some kind of flexible language to describe such sets. Moreover, bidirectionality might be lost – in contrast to textual diffs: that is, it cannot easily be determined, whether and to which elements such a generalized change has already been applied. The next section proposes an idea of how a symbolic reference can be used to generalize such a change.

4.1 Referencing sets of elements

In the example, three references were added – from the customer 'Joe' to each book in the library. Another way of describing that would be: *New references were added from the customer 'Joe' to all books in the library the customer belongs to.* That statement explains the very same change but on a more general level. The intention is to apply this change to more than only one concrete model, e.g. for any book available in the library. Furthermore, the change is described in a more compact, concise, and intuitive way.

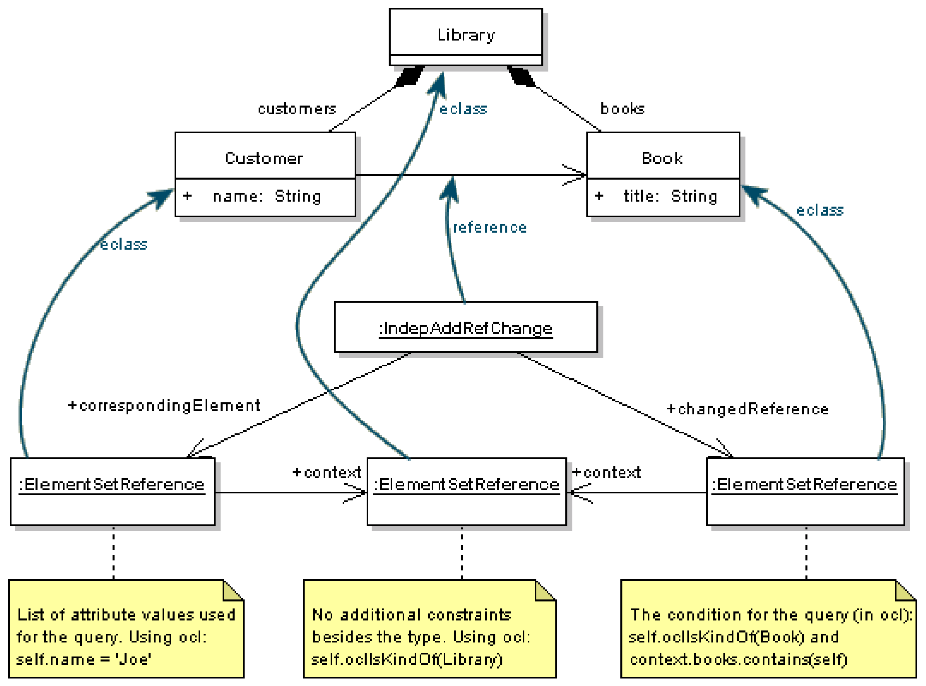
Library
customers
eclass
books
Customer
+ name: String
Book
+ title: String
eclass
reference
eclass
:IndepAddRefChange
+correspondingElement
+changedReference
:ElementSetReference
+context
:ElementSetReference
+context
:ElementSetReference
List of attribute values used for the query. Using ocl: self.name = 'Joe'
No additional constraints besides the type. Using ocl: self.oclIsKindOf(Library)
The condition for the query (in ocl): self.oclIsKindOf(Book) and context.books.contains(self)

Abbildung 11: Symbolic references for resolving sets of elements

The change needs to resolve more than only one element, in particular all books in the library. The example in Fig. 11 shows an *IndepAddRefChange*, which uses *ElementSetReferences* as an implementation of symbolic references for this purpose (see meta model in Fig. 3 for this context). The three classes at the top are part of the library meta model. All other elements are concrete objects of a change model as a UML object diagram which describe this generalized change. The cyan colored references (curved arrows) are described in the meta model of independent diffs. The left-most *ElementSetReference* contains an OCL condition, which, if resolved via the context, returns the customer Joe. The context is yet another *ElementSetReference*, which resolves a library – in our example just one library exists. The right-most *ElementSetReference* resolves all books of that particular library, as described by its OCL condition *"self.oclIsKindOf(Book) and context.books.contains(self)"*, where *books* is the containment between the library and books at the top.

Implications for generalized changes

The example we have just seen emphasizes some assumptions and requirements for generalized diffs:

- Symbolic references need to refer to a set of elements, not only one element! This is why ID-based symbolic references do not work here and we introduced *ElementSetReferences*.

- We somehow need a language to describe sets of elements. In our example, we have used OCL conditions to describe the constraints the particular elements have to fulfill.[8]
- However, not all uses of symbolic references are applicable for referring to sets of elements: the symbolic references for describing a movement (*oldParent* and *newParent* in the meta model) must not be sets of element, but just one element.

4.2 Benefits and drawbacks

Is it worth the effort introducing yet more concepts for describing changes in a model? Before answering this question, let us discuss the benefits for using generalized changes:

- There is only one general change for many similar changes, which leads to a more concise and compact form.
- General changes are applicable to much more models, because the language for symbolic references describing sets of elements is powerful enough to resolve elements even for different contexts.

However, there are some drawbacks:

- There is of course an overhead for constructing these changes: user interaction is probably required to correctly identify similar changes and to combine them.
- We loose bidirectionality! We cannot say whether / how many of the generalized changes have already been applied to a particular model.
- If a change has more than one element set, the meaning of combining these sets is ambiguous.E.g. what if the symbolic reference to the customer in our example returns a set of elements as well? Do we want a cartesian product? Or only specific customer-book pairs? So if we want to consider multiple element sets, we probably need to define a relation for combining these sets. For now, we leave this aspect open and only concentrate on single element sets.
- Even if there is only one element set involved, the result can be interpreted in different ways: e.g. a new element A should be aggregated to a set of elements – will there be *one A in total* that is aggregated to all elements in the set, or will there be *one A for each element* in the set? Again, we leave this aspect open at this point.

[8]The EMF Query subproject offers an API to perform these kinds of queries on arbitrary EMF models.

Conclusion

Generalized changes provide two very interesting improvements over atomic changes: namely a more compact and concise form of describing sets of elements, as well as much broader application scenarious due to a powerful language describing element sets. On the other hand, there are some important drawbacks which may not be desirable in some cases – e.g. the loss of bidirectionality. To conclude, one needs to decide when to use which constructs for describing changes, depending on the context.

The next step is the application of generalized changes to other models, as already denoted in Sect. 4.1. It is part of the second transformation of the process outlined in Fig. 1 on page 4 and subject of future work.

5 Related Work

In [KKU08], we already presented and implemented such a model-independent diff for a particular tool, namely the Enterprise Architect [Spa08]. However, the diff was only based on unique IDs and implemented only for the meta model of this particular tool. It supports automatic diff creation as a list of changes, and a semi-automatic transfer (with conflict resolution) to other model versions. Futhermore, two different types of models are supported, an early-phase analysis and a platform specific design model; in parcitular, they can be synchronized using these change lists.

The main purpose of EMF Compare [Tou07] is the support of diff & merge for arbitrary models based on EMF. It provides a GUI for comparing and merging models, including three-way-merge and conflict resolution. However, support for patches (which can be seen as model-independent diffs) is not yet included but scheduled for the next release.

The Rational Software Architect [IBM08] is also capable of diff & merge for UML models which includes difference detection, visualization, conflict resolution, and merging. Nevertheless, it performs these operation only in-memory, hence no persistence support exists. Moreover, logical grouping is missing – a new association, for instance, produces three changes instead of one: the association itself as well as two association ends.

[CRP07] has a similar goal but a different strategy. First, they extend each class in the meta model of the compared models with three new classes for the addition, deletion, and change of model elements. Second, the diff between two models is computed and stored accordingly to the extended meta model. Third, they create higher-order model transformations on these extended metamodels to transfer diffs to other models. So their approach works for arbitrary models. However, they did not consider conflicts so far, and their difference representation is not generalized.

6 Summary

In this paper, we first discussed the main differences between text- and model-based differencing. Then we pointed out the main challenges for model-based diffing, as well as an approach of model-independent diffs. It uses sub-model descriptors to describe sub-models that have been added or removed (cf. Sect. 3.2), and symbolic references to point to other model elements (cf. Sect. 3.1). These constructs make the diff model-independent, i.e. the diff can be viewed and used without having the models available it was created from.

In addition to that, we presented concepts for generalizing changes in order to make them more concise and compact, and applicable to a broad scenario of models. This has some advantages and disadvantages, as we have discussed in Sect. 4.2.

Concerning the overall procedure described in the introduction (Fig. 1 on page 4), we covered the concepts of model-independent diffs as well as the first transformation. The second transformation, however, is future work.

Acknowledgements

I would like to thank my supervisor Ekkart Kindler for many helpful discussions and advices.

Literatur

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[emf08] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf, 2008.

[FW07] Sabrina Förtsch and Bernhard Westfechtel. Differencing and Merging of Software Diagrams – State of the Art and Challenges. In Joaquin Filipe, Markus Helfert, and Boris Shishkov, editors, *International Conference on Software and Data Technologies (ICSOFT), Setubal (Portugal)*, volume 2. Institute for Systems and Technologies for Information, Control and Communication, 2007.

[IBM08] IBM. Rational Software Architect. http://www.ibm.com/software/awdtools/architect/swarchitect, 2008.

[KKU08] Ekkart Kindler, Patrick Könemann, and Ludger Unland. Diff-based model synchronization in an industrial mdd process. Technical Report IMM-Technical Report-2008-07, Technical University of Denmark, June 2008.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[MES03] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd., 2003.

[OMG06a] Object Management Group. Meta Object Facility (MOF) Core Specification. http://www.omg.org/cgi-bin/doc?formal/2006-01-01, January 2006.

[OMG06b] Object Management Group. Object Constraint Language Specification, Version 2.0. http://www.omg.org/technology/documents/formal/ocl.htm, May 2006.

[OMG07] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. Object Management Group, November 2007.

[Spa08] Sparx Systems Pty Ltd, Victoria, Australia. *EA User Guide*, 2008. Referring to version 7.5.

[Tou07] Antoine Toulmé. Presentation of EMF Compare Utility. In *EclipseCon*, March 2007. http://www.eclipse.org/emft/projects/compare.