Making Sense of UML Class Model Changes by Textual Difference Presentation

Harald Störrle
Department of Informatics and Mathematical Modeling,
Technical University of Denmark
Richard Petersens Plads, 2800 Lyngby, Denmark
hsto@imm.dtu.dk

ABSTRACT

Understanding the difference between two models, such as different versions of a design, can be difficult. It is a commonly held belief that the best way of presenting a model difference is by using graph or tree-based visualizations. We disagree and present an alternative approach where sets of low-level model differences are abstracted into high-level model differences that lend themselves to being presented textually. The results of preliminary user studies support our claim.

1. MOTIVATION AND APPROACH

Version control operators for models have attracted great attention over the last years. For instance, the on-line bibliography "Comparison and Versioning of Software Models" (see [5]) records over 400 publications in the area in the last 20 years, 250 of which have been published in the last five years. However, only less than ten out of these 250 publications consider the presentation of differences, even though the best difference computation is little use if the modeler cannot make sense of the change report.

Apparently, it is a commonly held belief in the model differencing community that the best way of presenting a model difference is by graph or tree-based visualizations, while text-based difference reports are considered inferior. For instance, Ohst et al. maintain that "*the concept of [side-by-side presentation] works well with textual documents, [...but] does not work well with graphical documents such as state charts, class diagrams, etc.[...].*" (cf. [3, p. 230]). Schipper at al. believe "*that there is a real need for a visual comparison*" (cf. [4, p. 335]). Wenzel even claims that "*The textual presentation [of differences] [...] is very difficult, or even impossible, to be read by human readers*" (cf. [10, p. 41]).

We disagree with this opinion. It is certainly true that high-accuracy difference computations leads to very large numbers of low-level differences, and simply dumping these to the user is not very helpful: modelers will be overwhelmed by the amount of information. There is, however, no reason, why we cannot try and find a more abstract textual difference representation that is equally accurate but less detailed, and thus easier to understand for modelers.

In this paper we propose an approach to compute model differences with maximum accuracy, automatically derive high-level explanations from them to reduce the level of detail, and present these high-level differences in user-friendly textual way. We propose a new approach to model version control that looks at models as knowledge bases providing an abstract view into some application domain.

2. RELATED WORK

There are mainly two approaches to presenting model differences, both of which are primarily visual. On the one hand, model differences may be visualized by color-highlighting different change states in the diagrams used for presenting the model (see e.g. [2]). While initially quite appealing, this approach has some severe limitations. First, using colors to differentiate element status is limited by the number of colors humans effectively (i.e.: pre-attentively) distinguish in a diagram (at most 5, see [1]). Second, the relatively widespread occurrence color vision deficiencies limits the effectiveness of this approach (up to 10% in males have total or partial color blindness). Third, only changes to elements that are presented in some diagram can easily be represented by color highlighting. Changes to the model structure, say, have to be presented in different ways. Finally, even those model changes that are presented in a diagram might be difficult to present when they affect more than one diagram. For instance, consider the changes done to a model as part of the rework assignment after a model review: this is likely to be spread out all over the model.

On the other hand, model differences may be visualized by side-by-side presentations of containment trees of models, possibly enhanced by color coding or connecting lines for movements (see e.g. the treatment in EMFcompare). This way, some of the limitations inherent in the first approach are avoided: issue relating to color vision are less important or can be neglected altogether. Also, all changes can be displayed uniformly, whether the elements affected are presented in a set of diagrams, a single diagram, or no diagram at all. However, this approach does not offer a satisfactory solution for large change sets: if a model difference results in a large number of low level changes, modelers can easily be overloaded by the amount of information, resulting in confusion and errors. In order to overcome these problems, we propose a new approach aiming to support modelers in making sense of model change reports and help the modeler deal with model changes.

This paper is based on the difference computation we have proposed before (see [7, 6]), but adds the abstraction from low-level changes into high-level model differences, and their presentation in a natural-language format.

Table 1: Domains for Models

Domain	Explanation
I	Identifiers are globally unique elements
S	Slot names are identifiers that are unique in a given model element, in meta-model based languages like UML they correspond to meta-attributes
V	Slot values may be of arbitrary type, including basic and complex data types, and references to (sets of) model elements
$\mathcal{B} : \mathcal{P}(\mathcal{S} \to \mathcal{V})$	Model element bodies are maps from slot names to slot values
$\mathcal{M} : \mathcal{P}(\mathcal{I} \to \mathcal{B})$	Models are maps from model element identifiers to model element bodies

3. A MODEL VERSIONING ALGORITHM

In the context of model comparison for version control, we typically want to compare two models that are subsequent versions of the same model. Thus, we can typically assume that (1) the two models have been created using the same tool, and (2) they have a large degree of overlap. This means, that both models use the same kind of internal identifiers for model elements, and that most of the model elements have the same identifiers in both versions. Thus, there is no need to align models, and matching between their elements becomes trivial. These assumptions are clearly not applicable in other, related areas, such as general comparison of models, or model clone detection (see e.g., [8]). Note that EMF Compare (see `www.eclipse.org/emf/compare`) has a wider focus and includes an explicit matching phase before computing model differences, although only identifier and hash-based matching seem to be currently available.

Mathematically speaking, we interpret a model as a finite function from model element identifiers to model element bodies, which are in turn finite functions from slot names to values, which of course may be (sets of) model element identifiers. We define the following domains (see Table 1).

We use the notation $dom(f)$ to denote the domain of a function, and $f \downarrow_X$ to denote the restriction of f to the sub-domain $X \subseteq dom(f)$. For example, if $f : A \to B$, then $dom(f) = A$ and $f \downarrow_X = \{\langle x, f(x)\rangle \,|\, x \in X, X \subseteq A\}$. The operator / denotes set difference, i.e., $X/Y = \{x \,|\, x \in X \wedge x \notin Y\}$.

3.1 Difference Computation

The domain definitions and notations introduced above allow us to formulate the difference computation as basic set-operations. Let P, and P' be two versions of a model, then the following are obvious.

Unchanged elements	$U =$	$P \cap P'$
Added elements	$A =$	$P' \downarrow_{dom(P)} / U$
Deleted Elements	$D =$	$P \downarrow_{dom(P')} / U$
Changed Elements	$C =$	$P'/(U \cup A)$

In order to compute the detailed changes of the changed elements, similar definitions apply. For every changed model element $c \in C$ with identifier i, we define the following sets.

unchanged slots	$c_u =$	$P(i) \cap P'(i)$
added slots	$c_a =$	$P'(i) \downarrow_{dom(P'(i))}$
deleted slots	$c_d =$	$P(i) \downarrow_{dom(P'(i))} / c_u$
changed slots	$c_c =$	$P'(i)/(c_u \cup c_a)$

Clearly, all of these sets can be computed trivially and efficiently. Observe that the result of applying these operators are not necessarily consistent models. For instance, if model P contains only a class A, and model P' contains also a class B and an association between A and B, then $P' \downarrow_{dom(P)} / U$ (the set of added elements) contains class B and the association, but not class A, that is, the association contains a dead link. Take Fig. 2 below as an illustration, where A is class Insurance and B is class Product, so that the whole model from Fig. 2 as P' and P is the submodel that contains only class Insurance.

Algorithm 1 Compute difference between two models OLD and NEW

function DIFF(OLD, NEW)
  compute the sets A, D, and C, from OLD and NEW as defined
  for all $c \in C$ compute c_a, c_c, and c_d
  tag all elements in these sets with their respective change type
  return $A \cup D \cup \bigcup_{c \in C}(c_a \cup c_c \cup c_d)$
end function

Consider the example in Fig. 1. It shows two subsequent versions of a class model in the insurance domain. Obviously, this is only a small toy example with a modest number of straightforward changes, finding all of them can be difficult. But even a complete and explicit list of the changes may be difficult to read if the number of changes grows too large. Even in the small example from Fig. 1, there are 36 changes between the two models. Clearly, this level of detail is not helpful when trying to understand a model difference report.

However, we have observed patterns in these low-level change reports: usually, groups of changes occur together as the effect of a single modeling action. Reconstructing these high-level changes from the low level observations will improve the understanding of model changes.

3.2 Difference Interpretation

The overall idea is to provide rules to explain sets of changes in a more abstract way. For instance, if a class C is deleted, its transitive parts and attached associations are typically also deleted. So, computing the difference results in a large number of detail changes that might confuse the modeler. This set of detail changes can be replaced by a single modeler-level explanation, e.g., something like 'deleted C and parts (cascading)'. Also, some changes are more important than others, and some may be reported summarily. For instance, changing the name of a class is often more important than changing its visibility. So we defined the following rules for abstracting low-level changes, based on observations on change protocols of difference computations of a set of modeling case studies (see [8] for more details on these case studies).

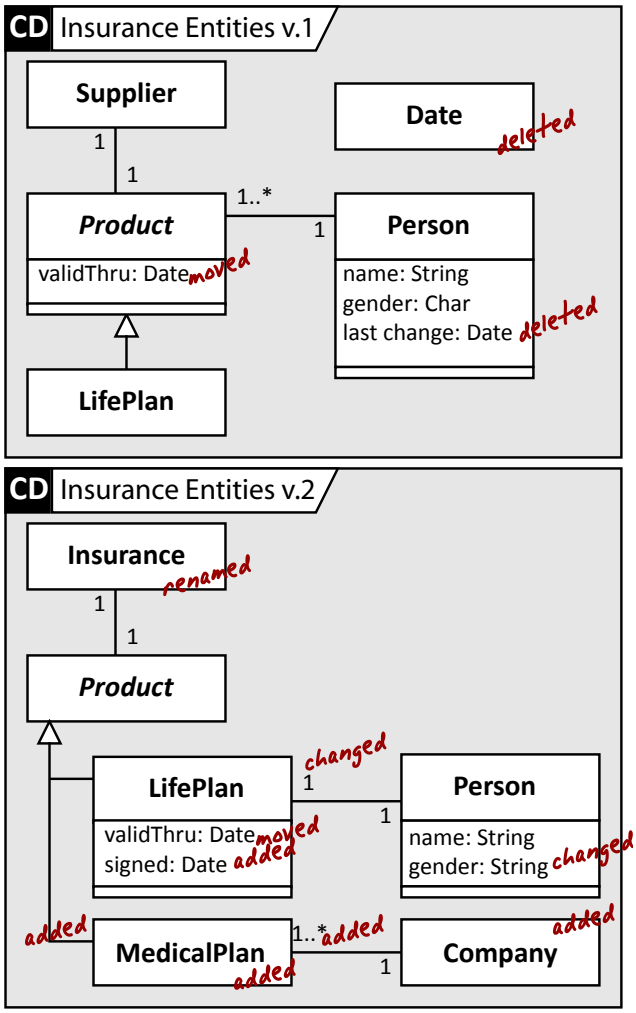


Figure 1: Running example for versioning models: original version (top); and subsequent version (bottom), with changes annotated in red. The blue annotations are internal identifiers for model element which we will use below.

Rename Replacing a change *"update attribute 'name' of class 'Supplier' to 'Insurance' "* into *"rename 'Supplier' to 'Insurance'* obviously does not reduce the number of changes, but makes it easier to understand the change.

Move Moving an element from one container to another changes the list of contained elements of both containers, so a pair of low-level changes can be accounted for with a single high-level change, and the operation "move" is easier to make sense of than a pair of addition and deletion, in particular if these are not presented together.

Delete Deleting an element also deletes its parts, removes a link to it from the part-list of its container, and any references from other elements to it. Typically model elements have several parts, which might again have parts so that deleting a single element may cascade a number of times, and a substantial number low-level changes can be abstracted this way.

Add Similarly, adding an element again also changes the part-list of its container, often adds parts, and possibly references from other elements to it. Furthermore, similar additions could be grouped, such as when adding several properties to a class: a single change can summarize such a change set.

Associate/Dissociate Adding or removing an association between some elements amounts to adding or removing the element itself, its (transitive) parts and properties, and references to these parts. For instance, an association is a connection between properties of the associated elements, and it is them who own the properties rather than the association.

Re-associate Exchanging one participant of an association by another replaces a pair of overlapping associate and dissociate-changes.

Tool specific Some changes refer to tool specific elements such as extension elements, internal libraries and so on. These should be suppressed in a high-level view.

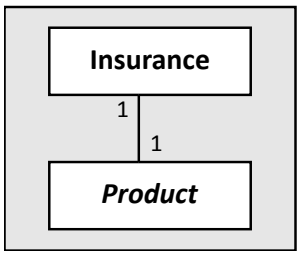
Implementing these rules must satisfy two major design goals. Firstly, the number of the reported changes should be substantially reduced and their understandability increased (**Goal 1:** reduce number of changes). Secondly, it should be easy to inspect a high-level change and find out what low-level changes it actually accounts for, so that both the results and the procedure are transparent (**Goal 2:** account for abstractions). Thirdly, the rules should be independent of each other such that it is easy to add or change them (**Goal 3:** Independence).

In a first attempt, we tried to provide a set of explanation rules that would consume the computed changes of a model difference. This way, each high-level change could be made to contain the low-level changes it accounts for which would satisfy the design goals 1 (change reduction) and 2 (accountability). Such an algorithm works fine when considering only renaming, moving, and additions/deletions of classes, properties, and similar entities.

However, there are many cases where the same low-level change can be explained by different explanations. For instance, consider the following three high-level changes possible to deduce from our above rules.

- adding class 'Company',
- associating class 'Company' to class 'MedicalPlan', and
- re-associating another class with 'Company'

All of these high-level changes would account for the low-level changes *"add anonymous property"* and *"update ownedMember of 'Company' "*. Of course, creating a high-level change report should explain as many low-level changes as possible, and so each rule is "greedy". This would imply that each rule must have a great number of case distinctions that encode deep knowledge about all other rules and what low-level changes may or may not have already been accounted for by another rule. Clearly, this contradicts the third design




```
me(      class-md1621, [name-'Insurance', ownedMember-ids([md1726]),
                        visibility-public]).
me(      class-md1641, [name-'Product', ownedMember-ids([md1727]),
                        visibility-public, isAbstract-true]).
me(association-md1625, [visibility-public, memberEnd-ids([md1727,md1726])]).
me(   property-md1727, [visibility-private, association-id(md1625)]).
me(   property-md1726, [visibility-private, association-id(md1625)]).
```

Figure 2: The internal representation of a fragment of the example model shown in Fig. 1 (left).

goal (independence of rules). So, the first design of simply consuming low-level changes is inadequate.

Instead we chose a different approach. If a high-level explanation E is found to account for a set of changes $C = \{c_1, \ldots, c_n\}$, then the elements of C are marked as accounted for, and linked to their high-level explanation E, while E is being added to the list of computed changes. Clearly, this approach allows to satisfy design goals 2 (accountability) and 3 (independence), but not 1 (reduce number of changes), since the number of changes steadily increases. However, the number of changes not accounted for *does* decrease, and so a slight variant of design goal 1 is indeed satisfied. This solution has the added benefit that there may now also be rules to abstract from previous abstractions, as happens in the rule "Re-associate".

Since the number of low- and high-level changes never reaches zero and we cannot know in advance how far their number may be reduced, we use a fixed-point algorithm to implement this solution. In other words, the rules are applied until the set of changes does not change any further, which includes both the overall size of the set and the number of accounted-for changes. Conflicts between different explanation rules need to be resolved by the programmer at compile-time; different orderings might result in different interpretations. The final algorithm is shown as Algorithm 2. Clearly, the algorithm terminates, if there is at least one explanation for each change. We achieve this by adding the default explanation, where a change is explained by itself. This rule applies only as a fall-back, i.e., if no other explanation applies earlier.

Observe that as an added benefit, this approach is independent of the ordering in which rules for interpreting low-level changes are applied. While the ordering of model changes has no influence on the interpretation, some sequences of operations are currently not detected. For instance, deleting an element and creating another element that is equal up to identity in another place might be understood as a movement by a user, but will not be recognized as such by the tool.

Algorithm 2 Interpret difference D between two models OLD and NEW

function EXPLAIN($DIFF$)
  pick some $d \in DIFF$ that is marked as 'fresh'
  if there is an explanation E for change d **then**
    find the largest $D \subseteq DIFF$ that is explained by E
    add E to $DIFF$
    link D to E for tracing
    mark E as 'fresh' in $DIFF$
    mark all D as 'stale' in $DIFF$
  end if
  return $DIFF$
end function

function INTERPRET($DIFF$)
  mark all $D \in DIFF$ as 'fresh'
  $DIFF' \leftarrow DIFF$
  repeat
    $DIFF \leftarrow DIFF'$
    $DIFF' \leftarrow explain(DIFF)$
  until $DIFF = DIFF'$
  remove all 'stale' elements in $DIFF'$
  return $DIFF'$
end function

4. IMPLEMENTATION

As in previous work on other model management operations (see [9, 8]), we equate domain models with domain knowledge bases and represent models as collections of Prolog facts. Therefore, we have implemented our algorithm in Prolog, too. There is a straightforward bijective mapping between more conventional model formats such as XMI and our Prolog representation (see Fig. 2 for an example). Contrary to common prejudice, using a high-level language like Prolog does not necessarily provide development efficiency at the expense of execution efficiency. In fact, we have consistently observed that our approach outperforms conventional Eclipse-based tools, often by an order of magnitude or more.

Our tool offers several options for the result presentation to the modeler, in particular, a tabular and a textual representation, and some statistics about the model size, number of changes, and change rate.

5. EVALUATION

In this section, we evaluate some aspects of our approach by comparing it to EMF Compare. Using the running example from Fig. 1, we yield a choice of difference reports as shown in Fig. 3: the output of Eclipse Compare (1), the low-level output from our approach (2), and our high level output (3), presented in tabular (a) and textual (b) format.

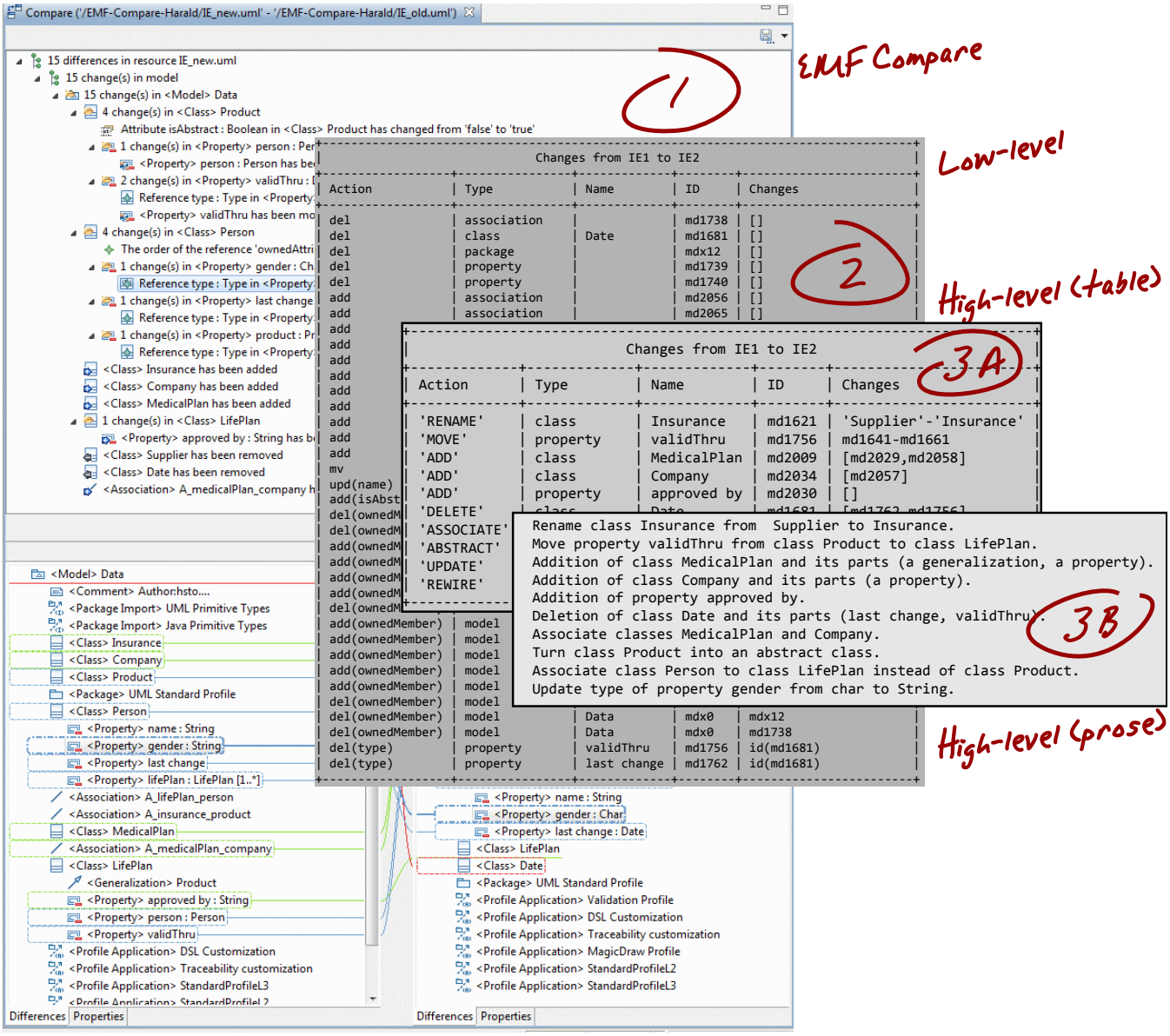


Figure 3: Comparing model difference presentations: EMF Compare (1), low level changes (2), and two different formattings for high level changes (3 A/B, respectively).

Observe that the table and the prose text contain identical information; they just present them in different formats. Clearly, EMF Compare uses by far the most screen real estate to present changes, which will be detrimental in maintaining an overview over large differences. Also, the difference presentation of EMF Compare is much more verbose without offering more information, while at the same time demanding heavy interaction by the modeler inspecting the differences when unfolding the difference tree. This will likely increase the effort of the modeler in trying to understand a model difference.

A quantitative comparison of the number of changes is shown in Fig. 4. It is obvious that the number of high-level changes is lower than the number of changes reported by EMF Compare, which in turn is markedly smaller than the number of low-level changes (36 vs. 15). Observe, however, that the difference in numbers of changes reported by EMF Compare and the low-level changes is almost entirely explained by changes to the container attributes ("ownedMember" et al.), and EMF Compare does not report most of these changes.[1] Simply dropping this type of information from the low-level change report will result in almost identical numbers as compared to EMF Compare.

The number of high-level changes, on the other hand, is notably smaller than the number of changes reported by EMF Compare (10 vs. 15). What is more important, however, is the way the changes are presented in our approach, which we believe is much more understandable. In order to investi-

[1]The exception being the order of the entries in one such container attribute, which, conversely, our approach does not consider.

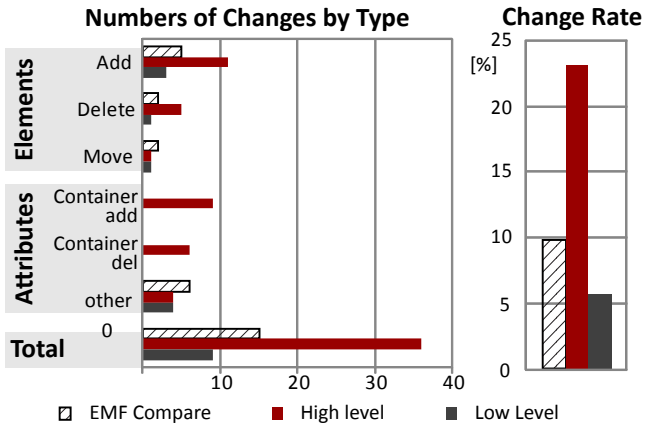


Figure 4: Quantitative comparison of approaches: numbers of changes (left); and reported changes relative to model size (right).

gate this hypothesis, we conducted a little explorative study into the relative understandability of the different change reports.

We presented printouts of the sample models shown in Fig. 1 and the different kinds of model changes to four graduate and undergraduate CS students. We presented the subjects with both the EMF Compare report and either the tabular or the prose representation of the changes and asked them to validate the correctness of the change computations at their own pace. We would observe their activity and record how often and how long they peruse the different represnetations. On completion, we would ask them which representation they like best, and why. Invariably, subjects use the EMF Compare change report only a little, and spent most of their time using the other representations. Post-task interviews clearly indicated a strong dislike of the EMF Compare change report, while no clear preference for the tabular and prose representation was evident. While obviously lacking the validity of a full-scale human factors study, this explorative study allows us to formulate suitable hypotheses for further studies, and equips us with the confidence to proceed with this line of research.

6. CONCLUSION

Building on previous work (see [7, 6]) we propose a novel approach to computing differences of UML class models which looks at models as knowledge bases providing an abstract view into some application domain. We use notions, techniques, and tools known from knowledge engineering and show how they can be used for efficient computation and effective presentation of model differences.

In this paper, we focus on the effective communication of found model difference by improved means of presentation, notably abstracting from low-level changes. We have implemented the approach, and hypothesize that it is more effective than EMF Compare, which represents the state of the art. We have conducted an explorative user study to test our hypothesis; preliminary results support it, though, obviously, more substantial research is needed.

We currently pursue two paths of extending the work reported here. First, we acknowledge the importance of evaluating our approach in a realistic scenario. Therefore, we have integrated it into the MagicDraw UML tool, but struggle with technical difficulties rooted in technological weaknesses and faults of the JPL Java-to-Prolog connector.

On the other hand, we plan to extend our abstraction facility by adding rules to discover more complex model changes, such as factoring out a property found in sibling subclasses into their common superclass. Another class of high-level model changes arises from modeling methodologies which might defined standardized model structures, specific submodels/views, or links between them. Changes motivated by these pragmatic rules could also be detected and provide a very high-level account of a great number of low-level changes. For instance, adding a UseCase "make payment" and refining it with a corresponding Activity might be accounted for as *"Addition of business process 'make payment'."* or similar. This way, a single high-level change could replace in the order of 50 low-level changes.

7. REFERENCES

[1] J. Bertin. *Graphics and Graphic Information-Processing*. Verlag Walther de Gruyter, 1981.

[2] Martin Girschick. Difference detection and visualization in UML class diagrams. Technical Report TUD-CS-2006-5, TU Darmstadt, 2006.

[3] Daniel Ohst, M. Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proc. 3rd Eur. Software Engineering Conf. 2003 (ESEC'03)*, pages 227–236, September 2003.

[4] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. In *14th IEEE Intl. Conf. Engineering of Complex Computer Systems*, pages 335–340. IEEE, 2009.

[5] SE group at the University of Siegen, Germany. Bibliography on Comparison and Versioning of Software Models. `http://pi.informatik.uni-siegen.de/CVSM/cvsm_bibliography.html`, last visited September 19th, 2012.

[6] Harald Störrle. A formal approach to the cross-language version management of models. In Ludwik Kuzniarz, Miroslaw Staron, Tarja Systä, and Mia Persson, editors, *Proc. 5th Nordic Ws. Model Driven Engineering (NW-MODE'07)*, pages 83–97. Blekkinge Tekniska Högskolan, August 2007.

[7] Harald Störrle. An approach to cross-language model versioning. In Udo Kelter, editor, *Proc. Ws. Versionierung und Vergleich von UML Modellen (VVUU'07)*. Gesellschaft für Informatik, May 2007. appeared in Softwaretechnik-Trends 2(27)2007.

[8] Harald Störrle. Towards Clone Detection in UML Domain Models. *J. Software and Systems Modeling*, 2011. (in print).

[9] Harald Störrle. VMQL: A Visual Language for Ad-Hoc Model Querying. *J. Visual Languages and Computing*, 22(1), February 2011.

[10] S. Wenzel. Scalable visualization of model differences. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 41–46. ACM, 2008.