Towards a User-Guided Difference-Based Detection of Atomic Changes

Djamel Eddine Khelladi, Reda Bendraou, Marie-Pierre Gervais

Towards a User-Guided Difference-based Detection of Atomic Changes

Djamel Eddine Khelladi
Sorbonne Universités,
UPMC Univ Paris 06, UMR 7606,
F-75005, Paris, France.
Atlanta, Georgia 30332–0250
Email: Djamel.khelladi@lip6.fr

Reda Bendraou
Sorbonne Universités,
UPMC Univ Paris 06, UMR 7606,
F-75005, Paris, France.
Université Paris
Ouest Nanterre La Defense,
F-92001, Nanterre, France.
Email: Reda.Bendraou@lip6.fr

Marie-Pierre Gervais
Sorbonne Universités,
UPMC Univ Paris 06, UMR 7606,
F-75005, Paris, France.
Université Paris
Ouest Nanterre La Defense,
F-92001, Nanterre, France.
Email: Marie-Pierre.Gervais@lip6.fr

***Abstract*—Detecting metamodel atomic changes during evolution is prerequisite for co-evolution of models, constraints, and transformations. They are also essential to detect complex changes over the sequence of atomic ones. However when detecting atomic changes with a difference-based technique, the applied order of the atomic changes is not recovered and some hidden changes are undetected. Thus, the quality of the detected atomic change trace is reduced which could be harmful to both co-evolution and detection of complex changes. This paper proposes to identify potential hidden changes in order to add them to the trace of atomic changes, and also to order the atomic changes with ordering heuristics.**

I. Introduction

Model-Driven Engineering (MDE) has proven to be effective in the development and maintenance of large scale and embedded systems [6]. In *MDE*, metamodels are core components of a modeling language ecosystem [6]. Metamodels naturally evolve throughout their lifespan which may make the model instances, constraints (e.g. OCL[1]), and transformation scripts (e.g. ATL[2] or ETL[3]) inconsistent and invalid.

Over the past years a growing interest has emerged for an automatic *repair*, *maintenance*, and *migration* of the impacted metamodel-based artifacts, with the prerequisite of correctly detecting the metamodel changes.

Two types of metamodel changes are distinguished [8], [9], [7]: a) *Atomic changes* that are additions, removals, and updates of a metamodel element, and b) *Complex changes* that consist of a sequence of atomic changes combined together [5]. If not provided in an embedded editor, complex changes are often detected over the sequence of applied atomic changes (e.g. in [13], [9]).

Therefore, detecting precisely and correctly atomic changes not only helps in better co-evolving the metamodel-based artifacts, but also to correctly detect complex changes that allow increasing and reaching a higher rate of co-evolution in comparison to when atomic changes only are considered during the co-evolution.

[1]http://www.omg.org/spec/OCL/
[2]https://eclipse.org/atl/
[3]http://www.eclipse.org/epsilon/doc/etl/

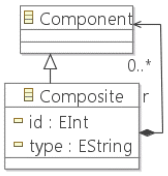
However, existing difference-based approaches (DBAs) neither achieve full recall (i.e. all correct changes are detected) nor full precision (i.e. all detected changes are correct) when detecting atomic changes because of two overlooked issues: 1) *hidden changes* that are missed during the detection, and 2) *non-ordered* sequence of atomic changes.

1) The first issue is that the DBAs cannot detect some changes that are *hidden* by other changes during evolution. Consequently, information might be lost which reduces the quality of the trace of atomic changes. For example, in Figure 1 the move property *type* from class `Composite` to class `Information` is hidden by the change rename property *type* to *kind*. The DBAs cannot detect these last two changes, but identify only two independent operations: deletion of property *type* and addition of property *kind*.

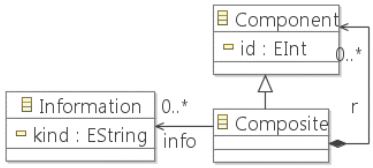
2) The second drawback of the difference-based approach is that the DBAs return an unordered sequence of all the detected atomic changes. However, the chronological order of changes might be relevant during later co-evolution tasks, and can be used during complex change detection for improving precision.

In this paper, we propose to address the issues of hidden changes and changes order when detecting atomic changes with a difference-based technique. We rely on EMF Compare [12], a well-known tool, to detect a first version of the atomic changes. Our approach proposes mechanisms to reintegrate potential hidden changes that may have not been detected as well as ordering heuristics to reach a better ordering of the changes. Our approach is semi-automatic in the sense that it asks for a user confirmation before altering the initial sequence of atomic changes.

The rest of this paper is structured as follows. Section II presents our differencing-based technique to detect atomic changes that is built on top of EMF Compare [12], while considering the hidden change issue as well as the ordering issue of a differencing-based approach in Sections II-B and II-C. Section III illustrates briefly our tool implementation and its capabilities. Section IV presents the related works and Section V concludes this paper and gives future work perspectives.



(a) Original metamodel.



(b) Evolved metamodel.

Fig. 1: An evolution example of a composite pattern. It evolves by deleting the properties *id* and *type* from the class `Composite` and then adding them respectively in `Component` class and in a new class `Information`. The property *type* is then renamed to *kind*.

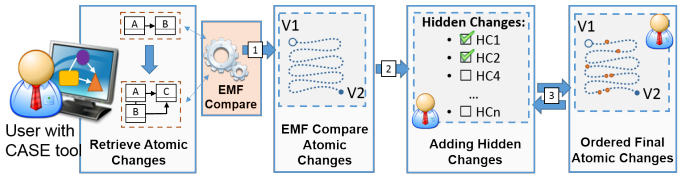


Fig. 2: Overall approach.

II. Overall Approach

This section presents our difference-based approach to detect atomic changes. Figure 2 gives an overview of the approach. We first interface with EMF Compare [12] to retrieve the atomic changes between the original and evolved metamodel versions 1. After that we identify potential hidden changes while allowing user's intervention 2. Finally, ordering heuristics with user interaction allow to sort the atomic changes before to export them into a final trace of atomic changes 3 so that complex changes can be detected and co-evolution can be performed.

A. Extending EMF Compare

We decided to reuse EMF Compare due to its popularity among model-driven community and for three other main reasons: 1) we do not aim to build a difference-base tool from scratch whereas existing tools are well adopted and have been tested in the literature [12], [1], 2) EMF Compare already supports wide range of models such as EMF/Ecore-based models, UML models, the Graphical Modeling Framework and its own extensions, papyrus, ecoretools, etc, and 3) handling the issues related to the difference-based techniques is easier to implement it as an additional top layer than handling them in the core engine of EMF Compare. This also has the benefit to reuse the additional layer on other difference-based tools without altering them.

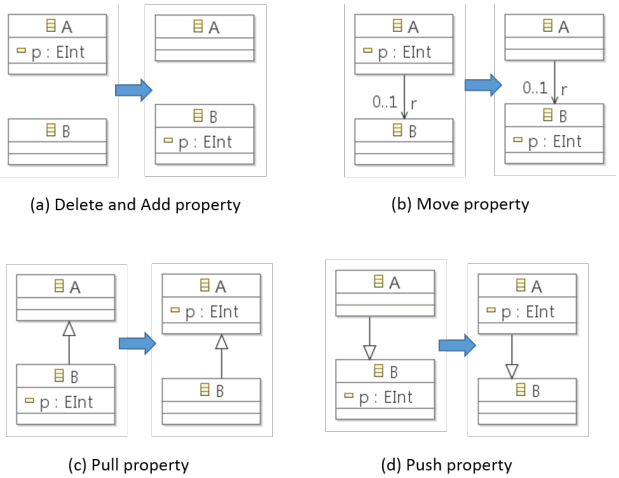


Fig. 3: Examples of move change in EMF Compare.

In EMF Compare, atomic changes (*add*, *delete*, and *update*) are identified as well as a *move* change. The move change is detected when an element is deleted from a container and added to another container. For example, if a property *p* is deleted in Class `A` and added in Class `B`, EMF Compare might detect it as a move *p* from `A` to `B`. Figure 3 shows different situations that are detected as a move change. We observed in our experience with EMF Compare that non-related delete and add changes, or the complex changes *move* property, *pull* property, *push* property can all be grouped as a move change by EMF Compare.

Artifacts such as models or OCL constraints are resolved with different strategies (e.g. [3], [10]) depending on the impacting changes that are detected. Therefore, in the current approach when retrieving the changes, all EMF Compare move changes are divided into delete and add changes so that detection tools can afterward detect the correct applied complex changes.

B. Handling the Hidden Change issue

This section addresses the issue of hidden changes in the retrieved set of atomic changes (step 2). For each case of hidden changes that can be encountered we show how it is resolved in the current approach.

1) False Rename Detection: When deleting an element and adding the same type of element in the same container, it is identified as a rename change. This is due to the heuristics that are used by EMF Compare during the change identification phase, for example by checking type equality. Figure 4 shows an example where a property *email* is deleted and another property *address* is added in the same class `Person`.

In case a rename is detected, the user can decide that it should be listed as a delete and add properties rather than the rename, as shown in Figure 4. Thus, in the current approach for each detected rename, the user confirms whether to keep it or to split it into a delete and an add changes, as shown in Figure 4.

2) Intermediate Atomic Changes: A second case of hidden changes is when an element is moved from several successive

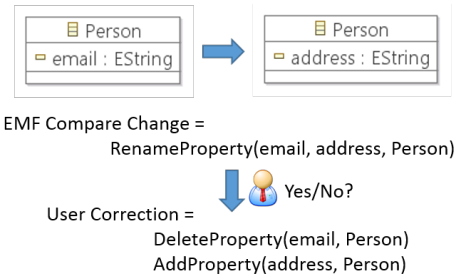


Fig. 4: Examples of a false rename change.

classes that are linked with each other by references and/or inheritances.

Figure 5 shows an example where a property *p* can be moved from a class `A` to a class `B`, before to be pulled up from `B` to a class `C`. EMF Compare detects only the changes delete property *p* in `A` and add property *p* in `C`, whereas the intermediate changes delete property *p* in `B` and add property *p* in `B` are not detected. These unidentified changes are crucial when detecting complex changes. Without the two intermediate changes, these two complex changes move and pull of the property *p* in Figure 5 cannot be correctly detected. Therefore, the impacted metamodel-based artifacts (models, constraints, transformations) cannot be correctly co-evolved.

Let us have the following OCL constraint (1) that is defined on the class `A` and uses the property *p*. Without the intermediate changes, and thus without the detected complex changes, the OCL constraint is deleted by existing approaches [3], [10]. Whereas with the intermediate changes, and thus with the detected complex changes, existing approaches co-evolve the OCL constraint from (1) to (2) by extending the navigation path to access the property *p* in the new location.

context `A` inv: (self.p $>$ 0) and (...) (1)

context `A` inv: (self.**r**.p $>$ 0) and (...) (2)

In the current approach, we propose the identified intermediate changes to the user who can decide whether to include them in the trace of atomic changes or not, i.e. in case the intent was indeed to delete *p* in `A` and to add another property *p* (which happens to have the same name) in `C`. Note that the intermediate changes are identified only if there is a path (represented with reference and/or inheritance) between the source class and target class.

3) User Rename Reconstruction: As shown in Figure 1, when differencing two metamodel versions, some applied renames may not be detected. For example, when a class `A` is renamed to `B`, EMF Compare can detect it as a delete class `A` and add class `B`.

In our approach, we propose to the user the possibility to reconstruct a rename change by selecting a delete and an add changes of the same element type (i.e. delete and add property, delete and add class, etc). Figure 6 shows an example of a class rename reconstruction. We also propose to the user the possibility to select a change and introduce hidden rename after/before it. For example, the add property *kind* in Figure 1 can thus be splitted into an add property *type* and a rename

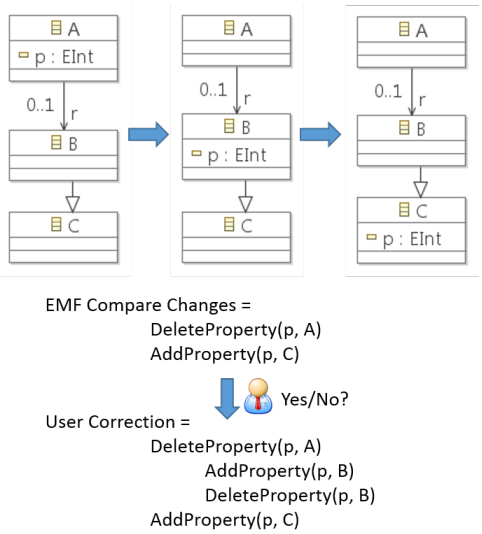


Fig. 5: Examples of intermediate atomic changes.

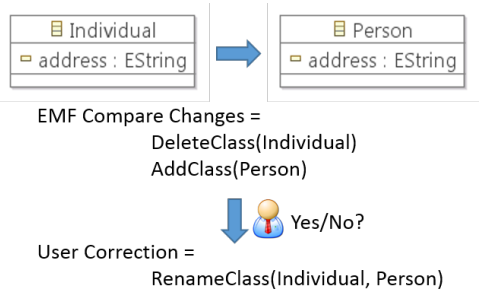


Fig. 6: Examples of a rename change reconstruction.

of *type* to *kind*, where the information *"type"* is specified by the user as input in a dialog box.

C. Handling the Ordering Issue

In this section, we handle the ordering issue when retrieving the atomic changes (step 3 of the tool's workflow).

1) Ordering Heuristics: This section presents three automatic ordering strategies to handle three particular cases of ordering.

The first case is when a container is added, the contained elements must then be added after the addition of the container in the trace. For example, when an operation *op* is added with some parameters, the parameters are added in the trace only after the addition of the operation *op*.

The second case is when a container is deleted, the contained elements must first be deleted in the trace before to delete the actual container. For example, when a class is deleted, its owned properties (attributes, references, operations) are first deleted in the trace.

Finally, the third case that we need to consider when ordering the trace of atomic changes is when the type of an added element is an element that has been added during evolution. For example, an added reference *ref* may have as a type value a new added class `B`, as shown in Figure 7. Thus in the trace of atomic changes, the add class `B` is put before the change add reference *ref* and its type.

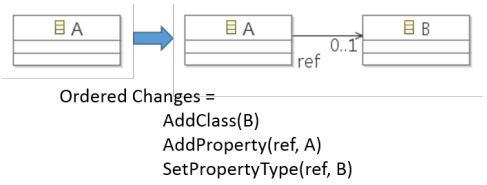


Fig. 7: Examples of ordering changes.

2) User Manual Ordering: The three above strategies apply a principle of a dependency order, i.e. a type of changes must be before/after another type of changes. Before generating the final trace of atomic changes as an output for external tools, the atomic changes are depicted to the user who can change the position of each atomic change if desired by moving it to another position. Thus, reflecting the real order as it was applied by the user.

III. Implementation

We built our tool on top of EMF Compare [12] as an extension to support the detection of potential hidden changes as well as ordering the atomic changes. The processes of identifying hidden changes and the ordering strategies have been implemented as a reusable component for other differencing-based tools. So that it could be reused with UMLDiff [14] or DSMDiff [11]. The core functionalities of this component are implemented and packaged into an Eclipse plug-in.

IV. Related Work

This section discusses the related work with the focus on difference-based detection of atomic change and the issues of order and hidden changes. In our approach, we rely on EMF Compare [12] to detect atomic changes. Other differencing approaches [14], [11] also compute the so-called difference model (DM) between two (meta) model versions.

UMLDiff [14] is non-id-based similarly as EMF Compare and uses a distance relation that considers structure and names. UMLDiff focuses on UML models only, in contrast to EMF Compare that supports EMF/Ecore-based models such as Ecore, UML etc. DSMDiff [11] generalizes the UMLDiff approach to cope with domain-specific modeling languages.

Garces et al. [4] propose to compute the difference model using several heuristics implemented as transformations in the Atlas Transformation Language (ATL) to detect atomic. Cicchetti et al. [2] address the dependency ordering problem of atomic changes. In fact, they focus on reordering atomic changes in such a way that eases the detection of complex changes. Vermolen et al. [13] propose to detect atomic changes with EMF compare. They partially consider the issue of hidden changes by proposing to the user with some atomic changes to add so that the effect of the evolution trace remains the same while user confirms it manually. This corresponds to the case of the intermediate hidden changes in our approach.

Only [2], [13] considers the issue of atomic changes ordering. Only [13] considers partially the issue of hidden changes in a difference model. In this paper, we identified cases of hidden changes to handle while asking for user confirmation and intervention. We also handle the ordering issue with three heuristics and by allowing the user to reorder the atomic changes at the end. To the best of our knowledge, we are the first to consider simultaneously the hidden change issue and the ordering issue when detecting the atomic changes.

V. Conclusions

In this paper we presented a preliminary approach to detect atomic changes via a difference-based technique. The raw atomic changes are first identified with EMF Compare [12]. We then aimed at identifying potential hidden changes as well as ordering the trace of atomic changes. In both activities, the user decides whether to include the detected hidden changes and to change the original order or not. In future work, we plan to investigate how to reduce he user intervention and whether we can automatically find the hidden changes and a better order. Thus, non-expert user can easily adopt our tool. We further plan to integrate other differencing-based tools and to validate our tool with several case studies.

Acknowledgments. We would like to thank Jiangen YU and Hanitriniaina RALIARIMANANA for implementing the eclipse plugin. The research leading to these results has received funding from the industrial innovation Project ANR MoNoGe under grant FUI - AAP no. 15.

References

[1] C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[2] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, pages 35–51. Jan. 2009.

[3] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In A. Moreira, B. Schatz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, number 8107 in Lecture Notes in Computer Science, pages 287–303. Springer Berlin Heidelberg, Jan. 2013.

[4] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *ECMDA-FA*, pages 34–49. Jan. 2009.

[5] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering*, pages 163–182. Jan. 2011.

[6] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *The 33rd ICSE*, pages 471–480. ACM, 2011.

[7] D. E. Khelladi, R. Bendraou, and M.-P. Gervais. Ad-room: a tool for automatic detection of refactorings in object-oriented models. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 617–620. ACM, 2016.

[8] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Detecting complex changes during metamodel evolution. In *The 27th CAISE*, pages 264–278, 2015.

[9] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems*, 2016.

[10] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais. Metamodel and constraints co-evolution: A semi automatic maintenance of ocl constraints. In *International Conference on Software Reuse*, pages 333–349. Springer, 2016.

[11] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: a differentiation tool for domain-specific models. *EJIS*, 16(4):349–361, 2007.

[12] A. Toulmé and I. Inc. Presentation of emf compare utility. In *Eclipse Modeling Symposium*, pages 1–8, 2006.

[13] S. D. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing complex metamodel evolution. In A. Sloane and U. Aßmann, editors, *Software Language Engineering*, pages 201–221. Jan. 2012.

[14] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM ASE*, pages 54–65, 2005.