Detecting Complex Changes During Metamodel Evolution

Djamel Eddine Khelladi[1(✉)], Regina Hebig[1], Reda Bendraou[1], Jacques Robin[1], and Marie-Pierre Gervais[1,2]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005 Paris, France
djamel.khelladi@lip6.fr
[2] Université Paris Ouest Nanterre La Defense, F-92001 Nanterre, France

Abstract. Evolution of metamodels can be represented at the finest grain by the trace of atomic changes: add, delete, and update elements. For many applications, like automatic correction of models when the metamodel evolves, a higher grained trace must be inferred, composed of complex changes, each one aggregating several atomic changes. Complex change detection is a challenging task since multiple sequences of atomic changes may define a single user intention and complex changes may overlap over the atomic change trace. In this paper, we propose a detection engine of complex changes that simultaneously addresses these two challenges of variability and overlap. We introduce three ranking heuristics to help users to decide which overlapping complex changes are likely to be correct. We describe an evaluation of our approach that allow reaching full recall. The precision is improved by our heuristics from 63% and 71% up to 91% and 100% in some cases.

Keywords: Metamodel · Evolution · Complex change · Detection

1 Introduction

In the process of building a domain-specific modeling language (DSML) multiple versions are developed, tried out, and adapted until a stable version is reached. As by one of our industrial partners in the automotive domain, such intermediate versions of the DSML are used in product development, where often further needs are identified. A challenge hereby is that each time the metamodel of the DSML is changed to a next version, already developed models need to be co-evolved too. This is not only the case for DSMLs, but also for more generic metamodels, e.g. the UML officially evolved in the past every two to three years.

To cope with this evolution of metamodels, mechanisms are developed to co-evolve artifacts, such as models and transformations that may become invalid. A challenging task herein is to detect all the changes that lead a metamodel from a version *n* to a version *n+1*, called Evolution Trace (ET). Automatically detecting it, not only helps developers to automatically keep track of the metamodels' evolution, but also to trigger and/or to apply automatic actions based on these changes. For instance, models and transformations that are defined based on the metamodel are automatically co-evolved i.e. corrected based on the detected

J. Zdravkovic et al. (Eds.): CAiSE 2015, LNCS 9097, pp. 263–278, 2015.
DOI: 10.1007/978-3-319-19069-3_17

ET (e.g. [7,8]). Here the rate of automatically co-evolved metamodel changes depends significantly on the precision and accuracy of the ET.

In such a context, it becomes crucial to provide correct and *precise* detection of changes. Two types of changes are distinguished: a) *Atomic changes* that are additions, removals, and updates of a metamodel element. b) *Complex changes* that consist in a sequence of atomic changes combined together. In comparison to the atomic changes alone, complex changes include additional knowledge on the interrelation of these atomic changes. For example, move property[1] is a complex change, where a property is moved from one class to another via a reference. This is composed of two atomic changes: delete property and add property. During co-evolution of models, the move property provides the valuable information that instance values of the deleted property are not to be deleted, but moved to instances of the added property. Many further complex changes [10] are used in literature to improve co-evolution rate of models and transformations.

Therefore, the detection of complex changes is essential for automating co-evolution. One approach towards that are operator-based approaches. By directly applying complex changes in form of operators, the user traces complex changes himself. However, more than 60 different complex changes are known to occur in practice [10]. Modelers might not be willing to learn and remember such a high number of operators, increasing the likelihood of workarounds with atomic changes. Thus, operator based approaches cannot provide a guarantee that all complex changes are recorded.

Vision. Consequently, a detection of complex changes needs to work on the basis of atomic changes. This task has one inherent difficulty that one needs to be aware of: a guarantee that all identified complex changes are correct is hard to reach. Existing approaches [4,6,7,11,17] neither reach a 100% recall, nor 100% precision. This is due to the fact that recovering the user's intent during an evolution is never certain. For example, when a property *id* is removed in one class and another property *id* is added to another class. This might be detected as move property, although it is just a coincidence.

Thus, final decisions can only be made by the user. Two options exist after an initial list of complex changes has been detected: the user might correct the list by a) removing incorrectly detected changes (such as [17]) and/or b) manually forming further complex changes based on found atomic changes (such as [11]). The later step, however, implies much higher effort for the user than just picking correct and incorrect complex changes from a complete list.

Therefore, we think that a detection approach should aim at 100% recall, meaning that all potential complex changes should be detected. To further increase precision and support the user in making the selection, identified changes should be prioritized concerning their probability with the help of heuristics.

Problem Statement. Automatically detecting complex changes is a difficult task, mainly because of two reasons: *overlap* that is ignored so far and *indefinite length*.

i) *Overlap*. Different complex changes might be composed based on overlapping sets of atomic changes. Figure 1 shows an example, two complex changes

[1] For sake of readability we refer to metaclass and metaproperty as class and property.

(a) Original metamodel (b) Evolved metamodel

Fig. 1. An evolution example of a composite pattern

pull property (via the generalization) and *move property* (via the relation r) might be formed based on the same set of atomic changes: deleting property *id* from the `Composite` class and adding *id* to the `Component` class. Only one of the changes was intended by the user. However, since we cannot know which complex change is the correct one, both must be detected. This phenomenon is reinforced, when a lot of changes are performed on closely related or even the same metamodel elements.

ii) *Indefinite length.* Complex change types have variable numbers of involved atomic changes. For example, in Figure 1 property *id* is pulled from one subclass `Composite`, yet a pull might also be applied when multiple subclasses contain the same property. Thus, the number of property deletions varies with the number of involved subclasses. Both issues reduce the recall that can be reached with existing approaches.

iii) A further issue arises due to the fact that all existing approaches [4,6,7,11,17] base the detection of complex changes on a set of atomic changes that has been computed as the difference between the old and the new version of the metamodel, the so-called difference model (DM). However, relying on the DM suffers from two main drawbacks:

(1) The first is that the DM cannot detect some changes that are *hidden* by other changes during evolution (called masked changes in [17]). Consequently, information might be lost, which impacts both recall and precision of the detection approaches. 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 DM cannot detect these last two changes, but sees only two independent operations: deletion of property *type* and addition of property *kind* as summarized in Table 1.

(2) The second drawback of the difference-based approach is that the DM returns an unordered sequence of all the detected 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.

Contributions. We address these challenges by four contributions:

– First, we propose to record at run-time the trace of atomic changes, by listening and logging modeler's editing actions within the modeling tool (editor). This way drawbacks of the difference-based approaches can be tackled.

Table 1. Recorded trace VS difference trace: An example of hidden changes

Applied Changes	Metamodel Difference Changes
1. *addClass(Information)*	1. *addClass(Information)*
2. *addProperty(info, Composite, Information)*	2. *addProperty(info, Composite, Information)*
3. *deleteProperty(id, Composite)*	3. *addProperty(id, Component, int)*
4. *deleteProperty(type, Composite)*	4. *addProperty(kind, Information, String)*
5. *addProperty(id, Component, int)*	5. *deleteProperty(id, Composite)*
6. *addProperty(type, Information, String)*	6. *deleteProperty(type, Composite)*
7. *renameProperty(type, kind, Information)*	

- Second, we propose a definition for complex changes that respects their variable character. Thus, all variants of a complex change can be detected.
- Third, we introduce a generic detection algorithm that consumes such variable complex change definitions as input together with the ET of atomic changes. In contrast to existing approaches [4,6,7,11,17], the algorithm systematically detects all possible candidates, even in case of *overlapping changes*, and thus reaching 100% recall. Furthermore, the approach can be easily extended to new complex changes, by just providing their definitions as input to the algorithm. We implemented the algorithm as a Complex Change Detection Engine (CCDE).
- Fourth, we propose to optimize precision by defining three heuristics that weight the detected overlapping complex changes. Especially, when many changes have been applied, these heuristics rank them in order of likelihood of correctness to the user, who can then pick and confirm the correct choice.

In this work, we apply our approach to detect seven complex changes: *move property, pull property, push property, extract super class, flatten hierarchy, extract class, and inline class* [10]. Our evaluation on two real case studies, including the evolution of GMF and UML Class Diagram metamodels, shows promising results by always reaching 100% recall of detection, and the precision is improved by our heuristics from 63% and 77% up to 95% and 100% in some cases.

In *Model-Driven Engineering* models are used in order to capture the different aspects of a system. This covers the system's architecture, its data structure or its design and GUI classses. While we focus in this paper on the evolution of metamodels, our approach for detecting complex changes theoretically applies on object-oriented models in general.

The rest of the paper is structured as follows. Section 2 illustrates our approach for detecting Complex changes. Sections 3 and 4 present the implementation and the evaluation of our approach. Sections 5 and 6 present the related work, discussion and conclude this paper.

2 An Approach for Complex Change Detection

This section presents an extensible approach to detect complex changes. We first describe how we obtain atomic changes, and then we introduce how a complex

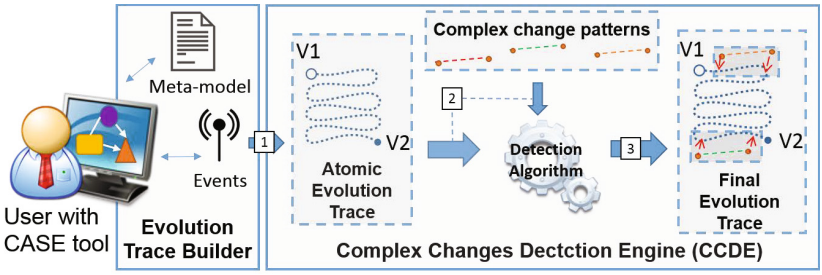


Fig. 2. Overall approach

change is defined, and what should be considered for its detection. After that, we present our detection algorithm, before applying it for seven complex changes.

Figure 2 depicts our overall approach. The atomic change trace is first recorded. Complex change patterns are then matched to it in order to generate a complex change trace. Based on the discussion of challenges in the introduction of this paper, we formulate four requirements for our approach:

- *R1. No changes must be hidden from the detection to not decrease the recall.*
- *R2. Detection must be able to cope with the variability to cover all possible variants of a complex change.*
- *R3. Detect all potential complex changes, i.e. high recall (100%). It means that no complex change is missed during the detection.*
- *R4. Prioritizing between overlapping complex changes to support the user in choosing those changes that conform to her intention.*

2.1 Atomic Change Detection

We propose a tracking approach that records at run-time all changes applied by users within a modeling tool without changing its interface. Thus, no changes are *hidden* or lost in the ET that serves for the detection of complex changes. This answers to the requirement R1, in contrast to the difference-based approaches. In order to implement the tracking mechanism we reuse an existing tool, Praxis [3] developed by our team. Praxis tool interfaces with a modeling editor to record all the changes that occur during an evolution. Existing works based on Praxis already provided good performances and scalability results [1–3,5].

Definition 1. We consider the following set of atomic changes that can be used during a metamodel evolution: *add, delete, and update* metametamodel elements. An *update*, changes the value of a property of an element, such as type, name, upper/lower bounds properties etc. The list of metametamodel elements that are considered in this work is: *package, class, attribute, reference, operation, parameter, and generalization*. Those elements represent the core feature of a metamodel, as in EMF/Ecore [15], MOF [13] metamodels.

2.2 Complex Change Definition

As mentioned previously, complex changes can be defined as a sequence of atomic changes [7,11,17]. However, this definition is not sufficient, mainly because: 1) The variability of a complex change is not part of the definition. 2) Some conditions have to be checked on the sequence of atomic changes that compose a complex change before to be considered as valid. For instance, for a move property p from `Source` to `Target`, a reference must exist from `Source` to `Target`.

Definition 2. We define a complex change as a pattern, each one comprising:

1. A set of atomic change types ($SOACT$) allowed to appear in the pattern, each with its multiplicity constraint. The multiplicity is a range between a minimum and a maximum [Min..Max]. For undefined Max value, a star is put instead, e.g. [1..*].
2. Conditions relating pairs of change type elements that additionally have to be satisfied for the pattern to match. Four types of conditions are used in our current approach:
   (a) Name equality between two named elements e_1 and e_2: $e_1.name == e_2.name$
   (b) Type equality between two typed elements e_1 and e_2: $e_1.type == e_2.type$
   (c) Equality between two typed elements e_1 and e_2: $e_1 == e_2 \Leftrightarrow e_1.name == e_2.name \wedge e_1.type == e_2.type$
   (d) Presence of a generalization relationship (Inheritance) between two classes c_1 and c_2: $c_1.inheritance.from == c_2$
   (e) Presence of a Reference relationship between two classes c_1 and c_2: $c_1.reference.type == c_2$

The above definition of a complex change answers to the requirement R2 of variability by explicitly specifying a multiplicity for each change.

2.3 Detection Algorithm of Complex Changes

The detection Algorithm 1 takes as input the pattern definitions of the complex changes that have to be detected and search for all their occurrences. In particular, our algorithm works in two passes. The first pass, (lines 1-11) generates all complex changes candidates, i.e. collects sets of atomic changes that might together form a complex change based on type and multiplicity of the pattern only. At each iteration, the algorithm browses through the evolution trace of n atomic changes, and if the current atomic change is part of a definition of a certain complex change, then a *candidate* is created with the current atomic change. After that, for all already existing candidates that might include this atomic change, we add a candidate instance that includes the current atomic change. The second pass (lines 12-13) scans the candidate set and only keeps those that satisfy the pattern, i.e. whether enough atomic changes could be identified and the conditions are fulfilled.

The main advantage of Algorithm 1 is its time complexity: it runs *one* time through the n atomic changes, and not k times for each complex change. The algorithm is designed to be extensible to detect other complex changes by defining additional definitions, the core detection remains unchanged. Its main drawback is the memory complexity, since it may create k candidates of complex

Algorithm 1. The Algorithm of Detection for Complex Changes.

Input: ET: The recorded evolution trace of atomic changes
LDef: List of definitions of the complex changes
Output: L: List of detected complex changes

1: $CCC : CandidateComplexChanges \leftarrow \{\}$; ▷ List of candidates of complex changes.
2: **while** $Not\ end\ of\ ET$ **do**
$current \leftarrow ET.current$;
3: **for all** $c \in CCC$ **do**
4: **if** $c.isItPossibleToAdd(current)$ **then** $c.add(current)$;
5: **for all** $d \in LDef$ **do**
6: **if** $current \in d$ **then** $x \leftarrow d.createCandiate()$;
$x.add(current)$; $CCC.add(x)$; ▷ add the atomic change to the created candidate and then to the list of candidates
7: **for all** $c \in CCC$ **do**
$List < ComplexChanges >= c.validate()$; ▷ validate the candidate complex changes to confirm and return only the valid ones

changes at each iteration, in the worst case at the end, $n * k$ candidates need to be validates. We evaluate the practical occurrence of this worst case in section 4.

Algorithm 1 answers to the requirement R3 by systematically creating in pass 1 all complex changes candidates that match the type and multiplicity of the pattern. Thus, the algorithm achieves full recall by construction. It is guaranteed to always return all complex changes, However, it may return false positives by returning multiple overlapping complex changes reusing the same atomic changes occurrences. Thus, we propose heuristics to rank the overlapping complex changes to help users to decide which ones are correct. These heuristics are discussed in section 2.5.

2.4 Application to Concrete Complex Changes

In the literature, over sixty complex changes are proposed [10]. We apply the detection algorithm for a list of seven complex changes: *move property, pull property, push property, extract super class, flatten hierarchy, extract class, and inline class* [10]. A study of the evolution of GMF[2] in practice, showed that these seven changes constitute 72% of all the complex changes used during the evolution of GMF [9,11]. Table 2 lists the seven complex changes and their definitions as a set of atomic changes following the *Definition 2*.

2.5 Prioritizing Between Overlapping Complex Changes.

We define three optional heuristics to rank overlapping changes and help the user to quickly choose which ones to keep. The input of each heuristic is just the list of overlapping complex changes. The output is the same list of changes but prioritized from most probable correct complex change to the least probable.

[2] Graphical Modeling Framework http://www.eclipse.org/modeling/gmf.

Table 2. Definitions of seven complex changes

Complex Changes	Set of Atomic Changes
Move Property	$SOACT$ = {delete property p[1..1], add property p'[1..1]} *Conditions:* $(p == p')\wedge\ (\exists\ reference \in p.class : reference.type = p'.class)$
Pull Property	$SOACT$ = {delete property p [1..*], add property p' [1..1]} *Conditions:* $(\forall\ p : p == p')\wedge$ $(\forall\ p : \exists\ inheritance \in p.class : inheritance.from == p'.class)$
Push Property	$SOACT$ = {add property p [1..*], delete property p' [1..1]} *Conditions:* $(\forall\ p : p == p')\wedge$ $(\forall\ p : \exists\ inheritance \in p.class : inheritance.from == p'.class)$
Extract Class	$SOACT$ = {add class c [1..1], add property p [1..*], delete property p' [1..*]} *Conditions:* $(\exists! p, \exists! p' : p == p')\wedge\ (\forall p, p.class == c)\wedge$ $(\forall p', \exists\ reference \in p'.class : reference.type == c)$
Inline Class	$SOACT$ = {add property p [1..*], delete property p' [1..*], delete class c [1..1]} *Conditions:* $(\exists! p, \exists! p' : p == p')\wedge,\ (\forall p', p'.class == c)\wedge$ $(\forall p, \exists\ reference \in p.class, reference.type == c)$
Extract Superclass	$SOACT$ = {add class c [1..1], delete property p [1..*], add property p' [1..*]} *Conditions:* $(\forall\ p : \exists! p' : p == p')\wedge\ (\forall\ p' : p'.class == c)\wedge$ $(\forall\ p : \exists\ inheritance \in p.class : inheritance.from == c)\wedge$ $(\forall\ p'_1, p'_2 \in p', \forall\ p_1 \in p : p'_1 == p_1, \exists\ p_2 \in p : p'_2 == p_2\ \wedge\ p_1.class == p_2.class)$ $\wedge(\forall\ p'_1, p'_2 \in p', \forall\ p : (p == p'_1 \wedge p == p'_2) \Rightarrow p'_1 == p'_2)$
Flatten Hierarchy	$SOACT$ = {add property p [1..*], delete property p' [1..*], delete class c [1..1]} *Conditions:* $(\forall\ p : \exists! p' : p == p')\wedge\ (\forall\ p' : p'.class == c)\wedge$ $(\forall\ p : \exists\ inheritance \in p.class : inheritance.from == c)\wedge$ $(\forall\ p'_1, p'_2 \in p', \forall\ p_1 \in p : p'_1 == p_1, \exists\ p_2 \in p : p'_2 == p_2\ \wedge\ p_1.class == p_2.class)$ $\wedge(\forall\ p'_1, p'_2 \in p', \forall\ p : (p == p'_1 \wedge p == p'_2) \Rightarrow p'_1 == p'_2)$

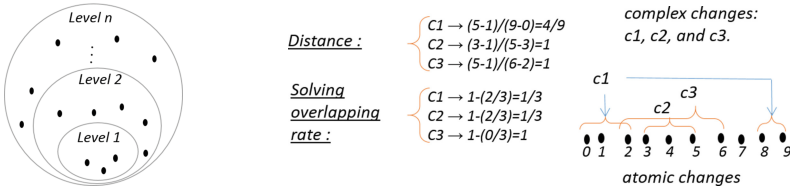
The first case of overlap between complex changes is when the first one is fully contained into the second one. The following heuristic handles this case.

Containment Level (h1). This heuristic assigns a containment level to each member of an overlapping complex change set. A complex change of higher containment level is ranked higher than one of lower containment level. For example, an extract class of one property p from class `Source` to class `Target`, contains the complex change move property p from `Source` to `Target` that is also detected. The former gets a higher priority than the latter. Figure 3a shows an example of containment between n complex changes.

The second case of overlap, is when several complex changes share only part of their atomic changes. The following two heuristics handle this case.

Distance of a complex change (h2). The atomic changes making up a complex change can be contiguous or not within the atomic change trace. A user who pulls a property p from half of the sub classes, then performs other actions, before coming back to pull p from the rest of the sub classes is an example of complex change composed of non-contiguous atomic changes. Heuristic 2: $Distance = \frac{S_{CC}-1}{E_P-SP}$, is the *Size of the Complex Change* divided by the *difference between the End Position and the Start Position* of the complex change in the ET. The distance is between 1 and 0. The higher is the distance value, the likelier the complex change to be the intended one among overlapping candidates.

Solving Overlapping Rate (h3). Our third heuristic ranks higher complex changes which removal from the candidate list minimizes the number of overlapping changes in this list. Users can rely on this heuristic to remove the least



(a) Containment heuristic h1 (b) Distance and Solving rate heuristics h2 and h3

Fig. 3. Three prioritizing heuristics

possible complex changes. Heuristic 3: $SolvingOverlappingRate = 1 - \frac{N_{LOCC}}{N_{OCC}}$, where N_{LOCC} is the *Number of Left Overlapping Complex Changes* and N_{OCC} is the *Number of Overlapping Complex Changes*. The fraction represents the rate of the remaining overlapping complex changes when the current one is removed.

Examples of heuristics 2 and 3 are presented in Figure 3b. The three above heuristics answer the requirement R4.

3 Implementation

We extended the Praxis prototype [3] to support the detection of complex changes out of the recorded atomic changes. The algorithm and heuristics presented in Section 3 have been implemented as the Complex Change Detection Engine (CCDE) component and integrated within Praxis. It detects in the trace of atomic changes the seven most used complex changes we addressed in this paper based on their definitions in Table 2 (Other complex changes can easily be considered). The core functionalities of this component is implemented with Java (4946 LoC) and are packaged into an Eclipse plug-in that interfaces with the existing Praxis plug-ins.

Figure 4 displays a screenshot of this integration. Window (1) shows a metamodel drawn with EMF Ecore tool editor. Praxis builds the evolution trace of atomic changes while the user is evolving the metamodel as shown in Window (2). In Window (3) the CCDE detects complex changes over the atomic changes evolution trace and our heuristics can be used as a support for users. The final evolution trace contains both atomic and complex changes.

4 Evaluation

This section presents the evaluation of our approach. We first describe an experiment in which we use our tool to detect complex changes. After that we evaluate the quality of the approach based on the quality metrics in [14]: *precision, recall, and f-score*. Time performance and memory consumption are evaluated as well.

4.1 Experiment Set/Scenario

In our evaluation, we have chosen two real case studies. We first evaluate a real case study: UML Class Diagram (CD) evolution, in particular from version 1.5

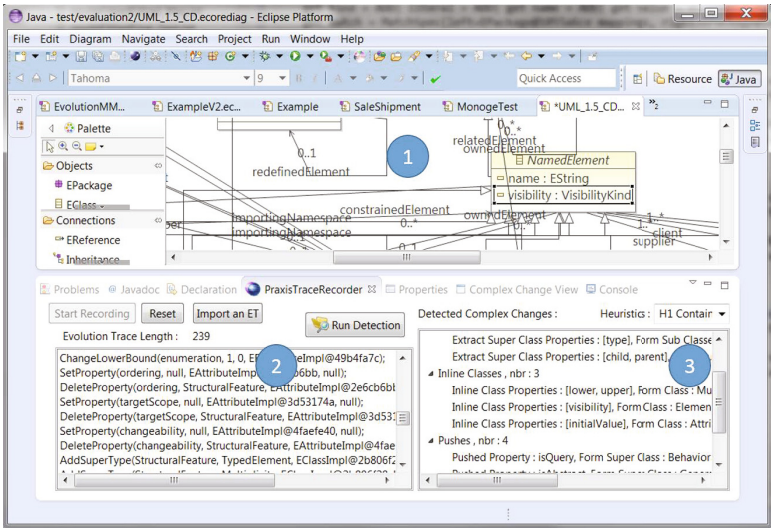


Fig. 4. Screenshot of the Tool

to 2.0, which contains significant changes. We manually analyzed beforehand the 238 atomic changes that leads UML CD 1.5 to 2.0 in order to know what are the expected complex changes. Without doing this analysis, we cannot assess the quality of detection of our tool, i.e. comparing what is expected against what is detected. In the UML CD case study, we expect 10 complex changes: 2 pulls properties, 4 pushes properties, 2 inlines class, and 2 extracts super class. The size of the UML CD metamodel 1.5 is of 143 elements.

For the second real case study we take the GMF graphical metamodel, GMF mapping metamodel, and GMF generator metamodels, all three making up the GMF metamodel. For the sake of simplicity, we will denote these three metamodels as "The GMF metamodel" in the rest of this paper. We thus consider the evolution of GMF metamodel from version 1.29 to version 1.74 that is investigated in [9,11]. After we manually analyzed the 220 atomic changes that leads GMF 1.29 to 1.74, we expect in the GMF case study 10 complex changes: 1 move property, 2 pull properties, 3 push properties, 3 extracts super class, and 1 flatten hierarchy. The size of the GMF metamodel 1.29 is of 2339 elements.

To run the experiment, we take the UML CD 1.5 and the GMF 1.29 metamodels and we manually evolve them until versions UML CD 2.0 and GMF 1.74 are reached, while our tool records the modeling actions.

In this experiment, we measure the accuracy of our tool by using the three metrics [14] $precision = \frac{CorrectFoundChanges}{TotalFoundChanges}$, $recall = \frac{CorrectFoundChanges}{TotalCorrectChanges}$, $f - score = \frac{2*Recall*Precision}{Recall+Precision}$. The values of these metrics are between 0% and 100%. The higher the precision value, the smaller is the set of wrong detections (i.e. false positives). The higher the recall value, the smaller is the set of the complex changes that have not been detected (i.e. false negatives). f-score combines precision and recall into a single evaluation measure of the best trade-off between

minimizing false positives and negatives. It is a useful metric since in most cases, methods manage to reach high precision with low recall and vice-versa. The higher the f-score the better the overall quality of our detection algorithm.

We measure the three quality metrics on the detected complex changes for the following cases: 1) without using the ranking heuristics defined in section 2.5, 2) using the heuristics separately. In our approach we do not remove complex changes from the overlapping list, but we only prioritize them with our heuristics, so that the user still has the chance to indicate that lower prioritized changes are correct instead of higher prioritized changes. While this prioritization supports the user, it does not impact the recall and precision. However, to nonetheless measure the quality of the heuristics, we simulate in this evaluation the situation that the user decides to keep only the highest prioritized changes. For the resulting list of complex changes we recalculate precision and recall. The whole process is performed once per heuristic.

Finally, we also measure the overall time of detection and memory consumption. We ran these experiments on a PC VAIO with i7 1.80 GHz Processor and 8GB of RAM with Windows 7 as OS.

4.2 Results

A. Without using heuristics. Figure 5 shows the results of the evaluation performed by our tool. Figure 5a shows the quality metrics on the raw detected complex changes without using heuristics. It shows 100% recall for both case studies UML CD and GMF metamodels evolutions without using any heuristics. This confirms the ability of our detection algorithm to reach our goal of a full recall that is essential in this paper.

In the UML CD case study, we detected 14 complex changes whereas we expected only 10. Three additional detected complex changes are due to the full overlapping issue when a change is contained in another one. In fact, each extract class, extract super class and flatten hierarchy respectively contain move, pull and push properties. Thus, in the UML CD case study, for each of the two applied extract super class, one of one reference and one of two references, we also detect three pulls of the same references that are incorrect.

One case of partly overlapping complex changes occurred in the UML CD case study, between one pull property *visibility* from sub classes `Feature` and `AssociationEnd` to super class `ModelElement`, and one unexpected inline class from `ElementOwnership` to `ModelElement` that contains the property *visibility*. They share only the add property *visibility* to the same class `ModelElement`. In this case, only the pull property *visibility* is correct and not the inline class. Thus, the precision is 10/14 that represents 71%.

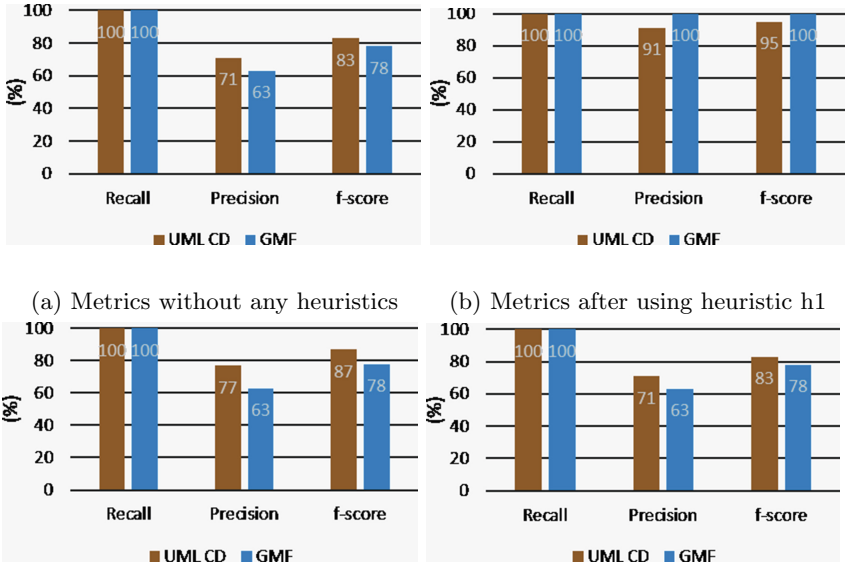
In the GMF case study, we detected 16 complex changes whereas we expected only 10. All the six additional complex changes are due to the full overlapping issue. In the GMF evolution, we applied one extract super class of one reference that explains one additional pull reference. We also applied two extracts super class of two properties each that explain the four additional pull properties. For the one flatten hierarchy of one reference, we detect an additional push reference. Thus, the precision is 10/16, i.e. 63% as shown in Figure 5a. The overall f-score

for the GMF and the UML CD case studies respectively reaches 78% and 87%.
B. Using only heuristic h1. Figure 5b shows the quality metrics after using the heuristic h1. For the GMF case study, h1 allows us to reach 100% of precision. This is possible since the case study contains only complex changes that overlap completely. The heuristic h1 ranks the additional push and pulls properties with a lower priority than the flatten hierarchy and the extracts super class, which are in our case study indeed the expected complex changes.

For the UML CD case study, h1 allows to reach 91% of precision (10/11), since the three additional pulls of a reference get a lower priority from h1 than the two expected extracts super class of the same references. Since the expected complex changes are ranked with the highest priority by h1, the recall thus stays 100% for both case studies. The f-score is improved to 95% and to 100% respectively for the UML CD and the GMF case studies.
C. Using only heuristic h2. Figure 5c shows the quality metrics after using the heuristic h2. For the GMF case study, h2 does not change the precision since there was no case of partly overlapping changes. However, for the UML CD case study, h2 allows to reach 77% of precision (10/13) by giving the highest priority to the pull property *visibility* that occurred at once in contrast to the unexpected inline class. Again, the recall stays unchanged for both case studies. The f-score reaches 87% for UML CD and stays unchanged for the GMF case study.
D. Using only heuristic h3. Figure 5d shows the results of the heuristic h3. It gives similar results as those when no heuristic is used, because h3 is useful when more than two complex changes partly overlap over different atomic changes, which is a situation that did not occur in our case studies. Note that for the UML CD case study, h3 fails to improve the precision by giving the same priority 1 that represents the rate of solving the overlap issue, to the two partly overlapping changes: the pull property *visibility* and the unexpected inline class.
E. Discussion. The results of the overall evaluation show that the recall is always 100% that denotes the completeness of our detection. The results also show that the heuristics h1 and h2 in some cases allow to improve the precision and thus the f-score. In the UML CD case study, we noticed five cases on *hidden* changes each by a rename action. Three properties in the two inlines class and two properties in the two extracts super class were all renamed afterward. One *hidden* change was noticed in the GMF case study regarding the one move property. Thus results of our detection engine would have been distorted if we have retrieved the atomic changes from the difference model.

In both case studies, the situation when a complex change partly overlap with other complex changes over different atomic changes, did not occur. Even though, it seems to be seldom in practice, we cannot exclude it. Heuristic h3 still could be useful in other case studies.

The evaluation experiments runs returned instantly and used insignificant memory. They detected all complex changes in less than 470 milliseconds over a trace of 220 atomic changes, while consuming less than 7.6 megabytes.



(a) Metrics without any heuristics (b) Metrics after using heuristic h1

(c) Metrics after using heuristic h2 (d) Metrics after using heuristic h3

Fig. 5. Evaluation Results

4.3 Threats to Validity

Internal Validity: We had to apply the evolution of the metamodels ourselves to retrieve the ETs. We applied each complex change at once, i.e all its atomic changes were applied in sequence without being interrupted by other atomic changes. In consequence, the ET does not include cases where the actual expected change has a lower probability than 1 due to the distance between the involved atomic changes, which might lead to an over-estimation of the efficiency of the results for heuristic h2. However, we assume that only in seldom cases such a separate application at different timestamps happens. Therefore, this threat to validity is acceptable here. Yet, more experiments are needed to further evaluate the ranking heuristics. Furthermore, to be able to evaluate the benefit of the heuristics, we simulated the user choice by keeping only highest prioritized changes. Indeed a user might also decide differently. However, since we can assume that the user knows his intent, these deviating user decision can in practice only improve the precision compared to this evaluation.

External validity: The quality of the presented approach depends on the quality of the recorded trace, i.e. the logging mechanism of the underlying framework. This concerns correctness and granularity of the provided atomic changes. Thus, it is difficult to generalize the measured precisions, recalls, and f-scores for other modeling frameworks. However, since Ecore is one of the most used modeling frameworks, we think that this limitation is acceptable for now. In future work, we plan to evaluate on other modeling frameworks as well.

Conclusion validity: Finally, from a statistical point of view it would surely be better to evaluate more case studies in order to gain a more precise measure of the actual precision, recall and f-score. However, results gained herein are sufficient to show the strength of our approach concerning recall and to get an idea of the potential impact of our heuristics on the precision. For a more detailed measures and also comparisons of performances to approaches from related work, we plan to identify and use more case studies on real metamodel evolution.

5 Related Work

Concerning related work, surprisingly, and to the best of our knowledge, we found only delta i.e. differencing-based approaches in the literature.

Differencing approaches[12,16,18] compute the so-called difference model (DM) between two (meta) model versions.The DM contains atomic changes add, delete and updates element, and one complex change move property only.

Cicchetti et al. [4] address the dependency ordering problem of atomic changes in order to ease the detection of complex changes. In contrast, Garces et al. [6] propose to compute the difference model using several heuristics implemented as transformations in the Atlas Transformation Language (ATL) to detect atomic and also complex changes. Langer et al. [11] use graph-based transformation to define a complex change with the left-hand side (LHS) and the right-hand side (RHS). Garcia et al. [7] detect complex changes with predicates that check occurrences of atomic change class instances. The predicates are implemented as ATL transformation scripts for each complex change. However, neither [12] nor [17] address the complex changes of variable length. Vermolen et al. [17] propose to detect complex changes over a manually ordered sequence of atomic changes returned from a difference model. However, in contrast to [4,6,7,11], [17] consider the issue of variability inside a complex change.

All the previous approaches [4,6,7,11,17] are based on differencing approaches such as [12,16,18] or implement themselves a differencing approach as in [4,6]. Thus, they suffer from the drawbacks of non ordered, and potentially hidden changes. Only [4,17] considers the issue of atomic changes order in a difference model, by defining strategies to reorder the atomic changes. Only [17] considers the issue of hidden changes in a difference model by proposing to the user with some changes to add so that the effect of the evolution trace remains the same. In this paper, we overcome those two issues by relying on Praxis and recording the evolution trace at run-time.

No related work addresses the issue of overlapping changes. They thus cannot reach full recall in the general case. We tackle the overlap issue by proposing prioritization heuristics. To the best of our knowledge, we are the first to consider the overlap issue, to address it by proposing three ranking heuristics, and to cope simultaneously with the four above issues.

6 Conclusion and Future Works

In this paper, we addressed the topic of complex change detection when a metamodel evolves. We detect precisely complex changes by relying on the real evo-

lution trace that is recorded by a user editing action observer. This approach has the advantage to preserve the evolution i.e. no changes are hidden from the detection. It thus supports full complex change recall, as shown in our evaluation. Relying on our tool, (meta) modelers are able to increase and to optimize the co-evolution percentage of artifact that relates to a metamodel, such as models. As mentioned previously, our approach can be applicable for object-oriented models in general, which can represent (1) the architecture of the system, (2) the design patterns on which it is based and (3) its data structures.

In a future work, we first aim to further improve the precision by optimizing the prioritizing heuristics and to propose new ones. In this paper, we applied the heuristics separately. Thus, we will assess how the precision is impacted by the different combinations of the heuristics.

Moreover, when recording the ET, there is a risk to record changes that cancel previous recorded changes. When there are ctrl-z events that undo the last changes, we can remove the last added changes from our ET. In case users perform a manual ctrl-z, we cannot deal with it. Yet, this does not impact the recall that remains 100%, but may lead to false positives. To cope with this issue, we will process the ET before the detection, searching for opposite changes that cancel each other and removing them from the ET.

Acknowledgments. The research leading to these results has received funding from the ANR French Project MoNoGe under grant FUI - AAP no. 15.

References

1. Bendraou, R., da Silva, M.A.A., Gervais, M.-P., Blanc, X.: Support for deviation detections in the context of multi-viewpoint-based development processes. In: CAiSE, pp. 23–31 (2012)
2. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 32–46. Springer, Heidelberg (2009)
3. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE 2008, pp. 511–520 (2008)
4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009)
5. da Silva, M.A.A., Blanc, X., Bendraou, R.: Deviation management during process execution. In: 26th IEEE/ACM ASE, pp. 528–531 (2011)
6. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009)
7. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 144–163. Springer, Heidelberg (2013)
8. García, J., Dìaz, O., Cabot, J.: An adapter-based approach to co-evolve generated sql in model-to-text transformations. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 518–532. Springer, Heidelberg (2014)

9. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: the history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
10. Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 163–182. Springer, Heidelberg (2011)
11. Langer, P., Wimmer, M., Brosch, P., Herrmannsdorfer, M., Seidl, M., Wieland, K., Kappel, G.: A posteriori operation detection in evolving software models. Journal of Systems and Software **86**(2), 551–566 (2013)
12. Lin, Y., Gray, J., Jouault, F.: Dsmdiff: a differentiation tool for domain-specific models. European Journal of Information Systems **16**(4), 349–361 (2007)
13. OMG. Meta object facility (mof) (2011). http://www.omg.org/spec/MOF/
14. Rijsbergen, C.: Information retrieval. Butterworths (1979)
15. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
16. Toulmé, A.: Intalio Inc Presentation of emf compare utility. In: Eclipse Modeling Symposium, pp. 1–8 (2006)
17. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 201–221. Springer, Heidelberg (2012)
18. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: 20th IEEE/ACM ASE, pp. 54–65 (2005)