Information Systems 62 (2016) 220–241

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Detecting complex changes and refactorings during (Meta) model evolution

Djamel Eddine Khelladi [a,*], Regina Hebig [b], Reda Bendraou [a], Jacques Robin [a], Marie-Pierre Gervais [a,c]

[a] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005 Paris, France
[b] Chalmers and University of Technology Gothenburg, Sweden
[c] Université Paris Ouest Nanterre La Defense, F-92001 Nanterre, France

ARTICLE INFO

Available online 31 May 2016

Keywords:
Metamodel
Evolution
Complex change
Refactoring
Detection

ABSTRACT

Evolution of metamodels can be represented at the finest grain by the trace of atomic changes such as add, delete, and update of 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.

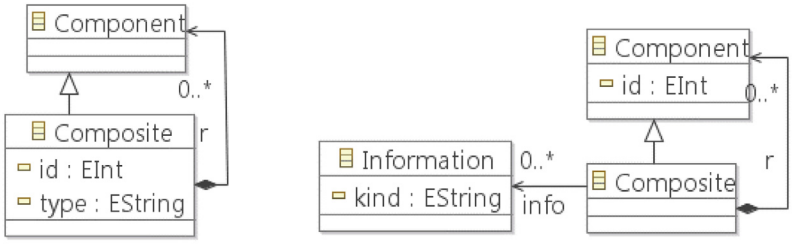
In this paper, we propose a detection engine of complex changes that simultaneously addresses these two challenges of variability and overlap. We introduce three ranking heuristics to help users to decide which overlapping complex changes are likely to be correct. In our approach, we record the trace of atomic changes rather than computing them with the difference between the original and evolved metamodel. Thus, we have a complete and an ordered sequence of atomic changes without hidden changes. Furthermore, we consider the issue of undo operations (i.e. change canceling actions) while recording the sequence of atomic changes, and we illustrate how we cope with it. We validate our approach on 8 real case studies demonstrating its feasibility and its applicability. We observe that a full recall is always reached in all case studies and an average precision of 70.75%. The precision is improved by the heuristics up to 91% and 100% in some cases.

1. Introduction

In the process of building a domain-specific modeling language (DSML) multiple versions are developed, tested, and adapted until a stable version is reached. Evolution of DSMLs and their metamodels is inevitable and necessary. 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 evolved to a new version, already developed models need to be co-evolved too. This is not only the case for DSMLs, but also for more generic metamodels such as the Unified Modeling Language (UML), which officially evolved in the past every two to three years. Besides models, also about 750 OCL [34] constraints associated to the UML metamodel need to be rechecked and co-evolved.

* Corresponding author.
E-mail addresses: Djamel.Khelladi@lip6.fr (D.E. Khelladi), hebig@chalmers.se, Regina.Hebig@lip6.fr (R. Hebig), Reda.Bendraou@lip6.fr. (R. Bendraou)Jacques.Robin@lip6.fr (J. Robin), Marie-Pierre.Gervais@lip6.fr (M.-P. Gervais).

http://dx.doi.org/10.1016/j.is.2016.05.002
0306-4379/



(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 than adding them respectively in `Component` class and in a new class `Information`. The property *type* is then renamed to *kind*. (a) Original metamodel and (b) Evolved metamodel.

To cope with this evolution of metamodels, mechanisms are developed to co-evolve artifacts, such as models, OCL constraints, 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 does not only help 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 ET [14,13,40,17,16,4,10,26]. The rate of automatically co-evolved impacted artifacts 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 of 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. Refactorings are complex changes that are used to improve the structure of a metamodel. 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 are used in the literature to improve co-evolution of models and transformations with metamodel changes, such as pull property (i.e the property is deleted from the subclass and added to the superclass) etc. [19].

Therefore, the detection of complex changes is essential for automating co-evolution. One way 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 [19]. Modelers might not be willing to learn and remember such a high number of operators [16], increasing the likelihood of workarounds with atomic changes. Thus, operator based approaches cannot provide a guarantee that all complex changes are recorded.

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 [22,39,13,5,11,45,30] neither reach a 100% recall (i.e. all correct changes are detected) [36], nor 100% precision (i.e. all detected changes are correct) [36]. 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 [39]) and/or (b) manually forming further complex changes based on found atomic changes (such as [22,16]). 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. Thus, existing co-evolution approaches that does not focus on detecting the ET [14,13,40,16,4] can be chained with our detection approach to be taken as a solid basis for their automatic actions.

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. Fig. 1 shows an example, two complex changes *pull property* (via the generalization) and *move property* (via the relation r)

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

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 Fig. 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 [22,39,13,5,11,45,30] 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 [39]). Consequently, information might be lost, which impacts both recall and precision of the detection approaches. For example, in Fig. 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 five 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.
- Second, we propose a formal 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 [22,39,13,5,11,45,30], 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 quickly pick and confirm the correct choice.
- Fifth, we handle undo operations (i.e. canceling actions) that are applied by the user during evolution either manually of automatically with a ctrl-z. Indeed, when recording at run-time the trace of atomic changes we may encounter undo operations. Detecting them allows co-evolution approaches to avoid applying unwanted resolutions in response of the undone complex changes. In particular, to avoid loss of information in the existing artifacts.

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* [19]. Our validation on 8 real case studies shows satisfying results by always reaching 100% recall of detection and an average precision of 70.75%. The precision is improved by our heuristics up to 91% and 100% in some cases. The detection of the applied complex changes in each case study was performed in less than a half a second.

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 classes. While we focus in this paper on the evolution of metamodels, the current approach for detecting complex changes theoretically applies on object-oriented models in general.

The rest of the paper is structured as follows. Section 2 establish foundations and motivates the detection of complex changes. Section 3 illustrates our approach for detecting Complex changes. Sections 4 and 5 present the implementation and the validation with a discussion of the results. Sections 6 and 7 present the related work and conclude this paper.

2. Foundation and motivation

In this section, we first define what is a metamodel evolution as it occurs in practice. After that, we illustrate the impact that a metamodel evolution can have on its related artifacts. We show the benefit of detecting complex changes and their role in the maintenance of the impacted artifacts.

2.1. What is a metamodel evolution in practice?

In our approach we consider an evolution of a metamodel to be equivalent to one or several releases of the modeling language. Indeed, an evolution of a metamodel from version n to $n+m$ can be equivalent to one or several releases of the modeling language from the version n to $n+m$ where $m \geq 1$. For example in the validation section,

the evolution of UML Class Diagram metamodel from 1.5 to 2.0 corresponds to one release of the UML from 1.5 to 2.0. Whereas, the evolution of Graphical Modeling Framework (GMF) metamodel from 1.29 ro 1.74 corresponds to several releases that are investigated in [18,22].

For the detection of complex changes, whether a metamodel's evolution corresponds to one or several releases does not impact the detection mechanism. A complex change is usually applied in one release and not throughout several releases, i.e. started in a version n (with some of its composing atomic changes) and completed in version $n+m$ where $m > 1$ (with the rest of its composing atomic changes). This situation is confirmed in our validation.

2.2. Impact on metamodel's related artifacts

As mentioned previously, each time a metamodel evolves already developed metamodel-based artifacts (e.g. model instances, transformation scripts, OCL constraints etc.) of the modeling language need to be co-evolved as well. In the literature, several works aim at co-evolving model instances [4,17,16,40,10], transformation scripts [12–14], and OCL constraints [7,8,25,26] that are all based on the metamodel changes. During co-evolution of metamodel-based artifacts, complex changes provides additional and valuable knowledge that is used to preserve information in the artifacts from being lost. In what follows we illustrate with an example the benefit of considering complex changes and how they are used to co-evolve impacted artifacts.

Table 1
Recorded trace VS difference trace: An example of hidden changes. The first column is the ET obtained by recording it, and the second column is the ET obtained by computing the difference between the original and the evolved metamodel of Fig. 1.

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*)	

Fig. 2 shows an example of a metamodel's evolution where two complex changes are applied. The property *id* is pulled from class `Composite` to the superclass `Component`. Also the property *type* is moved from the class `Composite` to the class `Information`.

Figs. 3 and 4 give an example of a model instance and an OCL constraint checking that the property *type* is not null and it is not empty. Both the model instance and the OCL constraint become invalid due to the delete of properties *id* and *type* in class `Composite`. Furthermore, Figs. 3 and 4 also illustrate the co-evolution of both artifacts in two different scenarios: (1) when considering only atomic changes, (2) when considering complex changes, too.

Models co-evolution. Step 1 of Fig. 3 shows the co-evolution of the model instance when considering the atomic changes only. A common resolution among models' co-evolution approaches [4,40] is to delete the property's instances in the model when the property is deleted in the metamodel. For a newly added property a default value or a null value is added in the models. Therefore, the co-evolved model instance looses the previous values of the properties *id* and *type*, which are respectively 1 and 'AAA', as depicted in Fig. 3.

Moreover, step 2 of Fig. 3 shows the co-evolution of the model instance when considering the complex changes too. On the one hand, the move property *type* brings the additional knowledge that the property values of *type* must be moved to instances of the class Information. On the other hand, the pull property *id* brings the additional knowledge that the property values of *id* are to be maintained since the property *id* is now in the superclass `Component`. Hence, property *id* stays in `Composite` too by inheritance.

OCL constraint co-evolution. Similarly, for the OCL constraint of Fig. 4, when considering only the atomic changes in step 1, existing approaches propose to delete the constraint since the property *type* is deleted in the class `Composite`. Thus, the OCL constraint is lost.

In contrast to step 1, in step 2 a common resolution [26] is to co-evolve the OCL constraint by extending the navigation path of the property *type* by the reference "*info*" from "*self.type*" to "*self.info.type*". Therefore, the values of the property *type* are accessed in the new location, i.e. in the instances of the class `Information` where the property *type* is moved to.

(a) Original metamodel. (b) Evolved metamodel.

Fig. 2. Example of a metamodel evolution. (a) Original metamodel and (b) evolved metamodel.

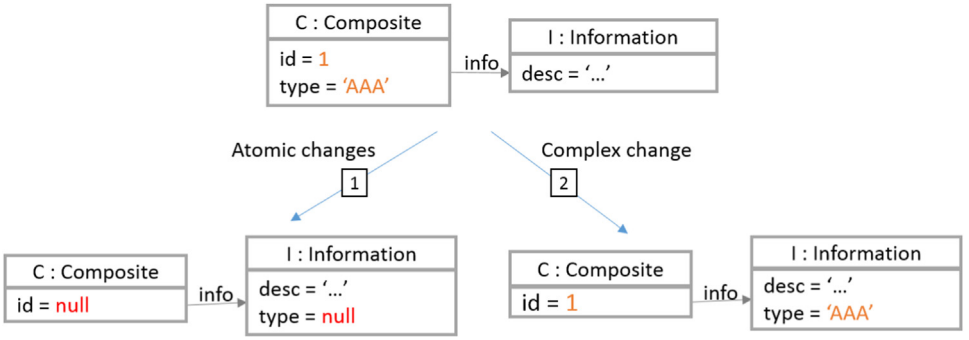


Fig. 3. Co-evolved model instance when considering only atomic changes (1) and when considering only complex changes (2).

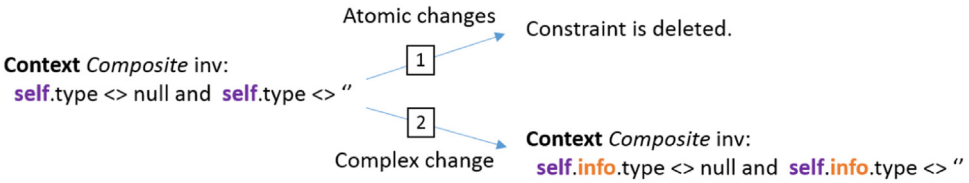


Fig. 4. Co-evolved OCL constraint when considering only atomic changes (1) and when considering only complex changes (2).

Clearly, considering atomic changes only during the co-evolution process will lead to a loss of information. Whereas, considering atomic changes as well as complex changes will not only improve the rate of co-evolution but will also prevent loss of information.

Unfortunately, most co-evolution approaches fail at detecting complex changes. Some of the existing approaches [26,40,4] assume the metamodel changes to be present as input, which makes their approach unusable in practice in industry. This is one of the reasons why industrials are still reluctant to use *Model-Driven Engineering* techniques despite the promises of reducing cost, time to market, and improvement of quality. In this paper, we aim at detecting complex changes and providing them as an output for co-evolution approaches. Note that the co-evolution of model instances and OCL constraints (presented above) is out of the scope in the current work.

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

Fig. 5 depicts the 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.*
- *R5. Detect all potential undo operations similarly as we aim with complex changes.*

3.1. Atomic change detection

We propose a tracking approach that records at runtime 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. It has already been used in the context of artifacts described in different languages like EMF, UML, XML and Java[2] and integrated with Obeo Designer, Papyrus, and MagicDraw tools. 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 [1–3,6].

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*, *property* (*i.e. attribute*, *reference*,

[2] http://code.google.com/p/harmony.

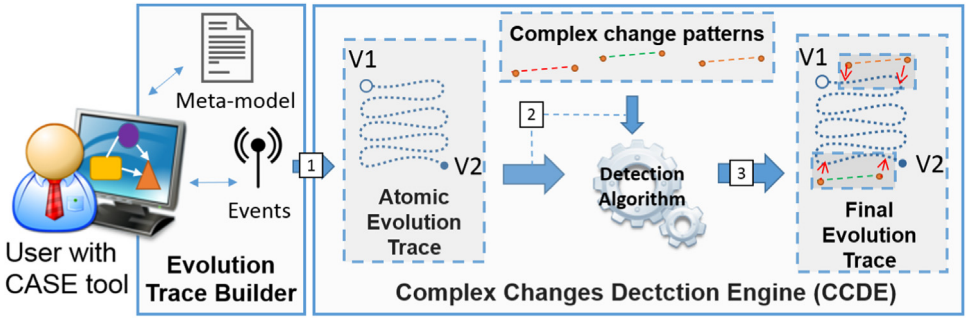


Fig. 5. Overall approach.

operation), *parameter*, *and generalization*. Those elements represent the core feature of a metamodel, as in EMF/Ecore [37], MOF [33] metamodels.

3.2. Complex change definition

As mentioned previously, complex changes can be defined as a sequence of atomic changes [39,22,13]. 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. Therefore, 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 the 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: $c_1 \cdot reference \cdot type == c_2$

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

3.3. 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 1–10) 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 (line 11) scans the candidate set and only keeps those that satisfy the pattern, i.e. enough atomic changes are 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 changes at each iteration. In the worst case at the end, $n * k$ candidates need to be validated. We evaluate the practical occurrence of this worst case in Section 5.

Note that we designed the algorithm to detect a complex change whatever the order of its composing atomic changes. A logical order of the atomic changes could exist such as a class is added and then a property is added to it, not the inverse. In theory, considering these situations would reduce a little bit the number of created candidates in our algorithm. However, it risks to reduce the recall that is undesired herein at all. Scalable performances of Algorithm 1 are further evaluated in Section 5.

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. Therefore, we propose heuristics to rank the overlapping complex changes to help users to decide which ones are correct. These heuristics are discussed later in Section 3.5.

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..∗]$\}$ $Conditions$: $(\forall p: \exists! p': p == p') \wedge (\forall p': p'.class == c) \wedge$ $(\forall p: \exists inheritance \in p.class: inheritance.from == c)$ $\wedge (\forall p'_1, p'_2 \in p', \forall p_1 \in p: p'_1 == p_1, \exists p_2 \in p: p'_2 == p_2 \wedge p_1.class == p_2.class) \wedge$
Flatten hierarchy	$SOACT = \{$add property p [1..∗], delete property p′ [1..∗], delete class c [1..1]$\}$ $Conditions$: $(\forall p: \exists! p': p == p') \wedge (\forall p': p'.class == c) \wedge$ $(\forall p: \exists inheritance \in p.class: inheritance.from == c) \wedge$ $(\forall p'_1, p'_2 \in p', \forall p_1 \in p: p'_1 == p_1, \exists p_2 \in p: p'_2 == p_2 \wedge p_1.class == p_2.class) \wedge$ $(\forall p'_1, p'_2 \in p', \forall p: (p == p'_1 \wedge p == p'_2) \Rightarrow p'_1 == p'_2)$

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

3.4. Application to concrete complex changes

In the literature, over sixty complex changes are proposed [19]. 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*. A study of the evolution of GMF[3] in practice showed that these seven changes constitute 72% of all the complex changes used during the evolution of GMF [18,22]. Those seven complex changes are defined as follows:

- Move property: it consists of moving a property p from a class A to a class B referenced from A.
- Pull property: a property p is added in a super class A and is deleted from sub classes B1,… Bn.
- Push property: a property p is deleted from a super class A and is added in sub classes B1,… Bn.
- Extract super class: a new class A is added as a super class to classes B1,… Bn. Then, a set of properties p1,… pn are deleted from the sub classes B1,… Bn and added in the super class A.
- Flatten hierarchy: a set of properties p1,… pn in a super class A are deleted, and added in the sub classes B1,… Bn. Then the super class A is deleted.
- Extract class: a new class B is added, and a reference is added from a class A to class B. Then a set of properties p1, … pn are deleted from the class A and added to the class B.

[3] Graphical Modeling Framework http://www.eclipse.org/modeling/gmp/.

- Inline class: it is the opposite of Extract class, a set of properties p1,… pn are deleted from a class B and added to a class A. Then the class B is deleted.

Table 2 lists the seven complex changes and their definitions as a set of atomic changes following the Definition 2.

3.5. Prioritizing between overlapping complex changes

As mentioned earlier, by aiming at 100% recall some incorrect complex changes can be detected i.e. false positive. In the following 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.

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.

3.5.1. 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. Fig. 6(a) 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.

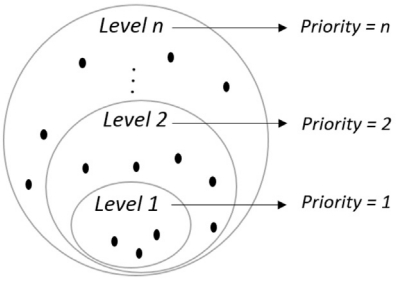
3.5.2. Distance of a complex change (h2)

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 determined by 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.

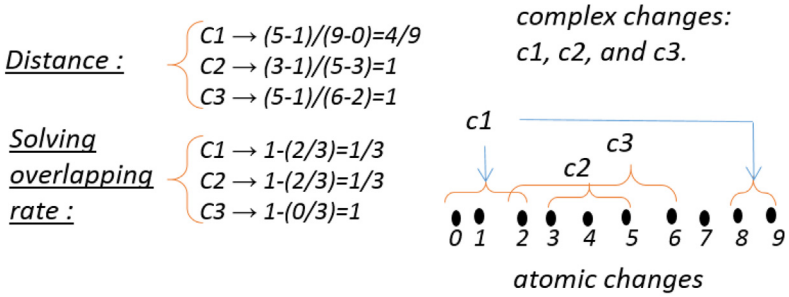
3.5.3. Solving overlapping rate (h3)

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 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 Fig. 6 (b). The three above heuristics answer the requirement R4.



(a) Containment heuristic h1.



(b) Distance and Solving rate heuristics h2 and h3.

Fig. 6. Three prioritizing heuristics. (a) Containment heuristic h1 and (b) distance and Solving rate heuristics h2 and h3.

3.6. Handling undo operations

This section discusses the issue of undo operations during the metamodel's evolution an their consequences on the quality of detection that can impact negatively the co-evolution tasks.

During the metamodel's evolution, the user can undo her actions either manually or automatically by using the ctrl-z functionality of the modeling editor.

However, when using a recording mechanism of the evolution, the canceled changes as well as the canceling actions are still added to the trace of atomic changes.

As mentioned previously, atomic and complex changes are used in co-evolution approaches to resolve impacted metamodel-based artifacts. In consequence, unwanted resolutions can be applied whereas they must not due the user undo actions. For example, a property *p* can be deleted (accidentally or not) and then added again in the same class, which undoes the deleted action. Considering both atomic changes can be disastrous for co-evolution approaches. Indeed, to co-evolve model instances w.r.t. the change delete property *p*, instance values of *p* are deleted. Further, either default or null values are added in the model instance as a result of its co-evolution w.r.t. the change add property *p*. Therefore, unintended loss of information can occur due to the unhandled issue of undo operations.

Fig. 7 shows an example of an application of a complex change before being canceled as well as the recorded trace of atomic changes. Step 1 shows the application of two complex changes "pull property *x*" and "move property *y*" as well as the recorded trace at time-stamp t_1. Step 2 shows the undo of the "move property *y*" by deleting the property *y* from the class `B` and adding it to the class `A`, which are the traces 5 and 6 in the recored trace at time-stamp t_2.

When running our algorithm on the recorded trace at t_1, both complex changes are detected which are indeed applied at t_1. At t_2 only the "pull property *x*" is effectively applied. However, the algorithm detects the "pull property *x*" as well as a "move property *y*", since the changes n^o 3 and n^o 4 composing the move property are still present in the recorded trace. Unwanted and unnecessary resolutions can be applied as a response to the change "move property *y*". Therefore, is it important to guarantee a consistent evolution trace w.r.t. the user actions. In what follows, we discuss what Pitfalls an approach must avoid, and then we present our solution.

3.6.1. What must be avoided

Having identified the issue of undo operations, one can think of removing both an atomic change and its inverse (i.e. the atomic change canceling it) from the evolution trace, as proposed in Alix et al. [31]. For instance, the inverse change of an *add class* `C` is the change *delete class* `C'`, where `C == C'`. Both changes can be removed when encountered in the evolution trace [31]. On Fig. 7 at trace t_2, the tuple of changes (no 3, no 6) and (no. 4, no 5) would be removed by [31]. However, proceeding this way can impact the performances and the quality of a detection approach. In particular, our main goal to detect all applied complex changes by reaching 100% recall can be compromised by working on an incomplete evolution trace.

Fig. 8 shows an example, where the user intent is to first delete the property *id* in the class `Information` and then move the property *id* from the class `Element` to `Information`. Here the intention is to move the property values of *id* to instances of `Information`. It is a coincidence that trace t_2 contains both a delete and add of the same property name *id*. However, both atomic changes concern two different properties with the same name *id*. If the change n^o 1 and its inverse n^o 3 are removed from the evolution trace t_2, we loose the user intent. Therefore, the move property will not be detected on the remaining change n^o 2 of the trace t_2.

Removing a correct change and its wrongly detected inverse would lead to the phenomenon of *hidden changes* as we know them from the difference-based approaches. Thus the recall can decrease as a consequence.

3.6.2. The adopted solution

Similarly as we proceed with the detection of complex changes, we aim at detecting all applied undo operations on the evolution trace, and only the user must confirm which undo operations were indeed applied.

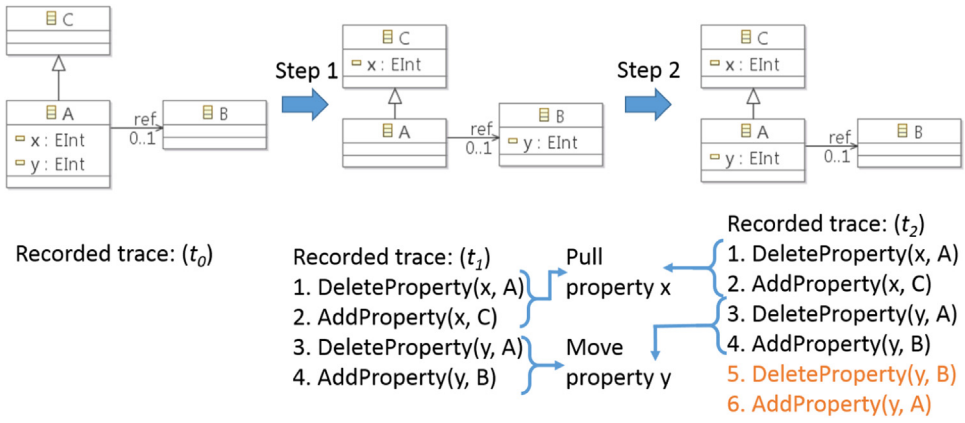


Fig. 7. Example of an undo during evolution.

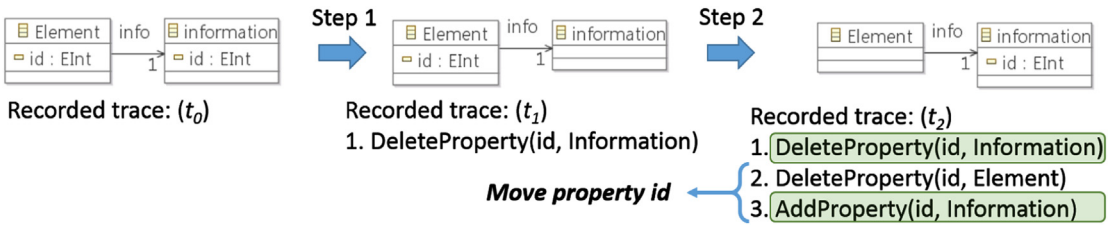


Fig. 8. Example of an undo during evolution.

Table 3
Definitions of seven undo complex changes.

Complex changes	**Definition of the undo operations**
Undo move property	Change C: $SOACT = \{$delete property $p[1..1]$, add property p' $[1..1]\}$ *Conditions*: $(p == p') \wedge (\exists reference \in p.class: reference.type = p'.class)$ Inverse C^{-1}: $SOACT = \{$delete property $p_1[1..1]$, add property $p'_1[1..1]\}$ *Conditions*: $(p_1 == p'_1) \wedge (\exists reference \in p'_1.class: reference.type = p_1.class) \wedge (p == p'_1) \wedge (p' == p_1) \wedge (p.class == p'_1.class) \wedge (p'.class == p_1.class)$
Undo (Push/Pull) property	$SOACT = \{$push property p $[1..1]$ from A to $A_1, \ldots, A_n$, pull property p' $[1..1]$ from $B_1, \ldots, B_n$ to $B\}$ *Conditions*: $(p: p == p') \wedge (A == B) \wedge (\forall i \in [1..n] \exists j \in [1..n]: A_i == B_j) \wedge (\forall i \in [1..n] \exists j \in [1..n]: B_i == A_j)$
Undo (Inline/Extract) class	$SOACT = \{$Inline class$[p_1, \ldots, p_n][1..1]$ from A to B, Extract class $[p'_1, \ldots, p'_n][1..1]$ from A′ to B′$\}$ *Conditions*: $(\forall i \in [1..n] \exists j \in [1..n]: p_i == p'_j) \wedge (\forall i \in [1..n] \exists j \in [1..n]: p'_i == p_j) \wedge (A == B') \wedge (A' == B)$
Undo (Flatten hierarchy/extract super class)	$SOACT = \{$*Flattenhierarchy* $[p_1, \ldots, p_n][1..1]$ from A to $A_1, \ldots, A_m$, Extract super class $[p'_1, \ldots, p'_n][1..1]$ from $B_1, \ldots, B_m$ to B$\}$ *Conditions*: $(\forall i \in [1..n] \exists j \in [1..n]: p_i == p'_j) \wedge (\forall i \in [1..n] \exists j \in [1..n]: p'_i == p_j) \wedge (\forall i \in [1..m] \exists j \in [1..m]: A_i == B_j) \wedge (\forall i \in [1..m] \exists j \in [1..m]: B_i == A_j) \wedge (A == B)$
Atomic changes	**Definition of the Undo Operations**
Undo (Add/delete) class	$SOACT = \{$Add class A $[1..1]$, Delete class A′ $[1..1]\}$ *Conditions*: $(A == A') \wedge (A.package == A'.package)$
Undo (Add/delete) property	$SOACT = \{$Add property p $[1..1]$, Delete property p′ $[1..1]\}$ *Conditions*: $(p == p') \wedge (p.class == p'.class)$
Undo rename property	$SOACT = \{$Rename property p_1 to p'_1 $[1..1]$, Rename property p_2 to p'_2 $[1..1]\}$ *Conditions*: $(p_1 == p'_2) \wedge (p'_1 == p_2) \wedge (p_1.class == p'_2.class) \wedge (p'_1.class == p_2.class)$

In the current approach, we base the detection of undo operations on our Algorithm 1 by defining for each metamodel (atomic/complex) change its undo operation as a complex change pattern following the Definition 2. Thus, an undo operation is a complex change that must contain both: (1) the atomic changes composing a change C, (2) the atomic changes composing the inverse C^{-1} that cancels the change C.

For instance, for the atomic change add property p in a class C, its undo operation that we must detect, is a complex change composed of: (1) an add property p in a class C, (2) a delete property p in a class C. Another example, for the change pull property, its undo operation contains: (1) the pull property p from $B_1, \ldots, B_n$ to A, (2) the push property p from A to $B_1, \ldots, B_n$.

Table 3 presents the definition of the undo operations for the complex changes defined in Table 2. Table 3 further presents the definition of the undo operations for some atomic changes as an example.

We define the undo operations in terms of atomic changes for the complex change move property only. For

sake of readability, for the rest of complex changes we define their undo operations in terms of complex changes.

Note that the definition of the undo of a pull property is the same definition of the undo of a push property, since both contains the changes pull property *p* and push property *p* that cancel each other. Similarly, the definitions of the undo of an inline class and a flatten hierarchy are respectively equal to the definitions of the undo of an extact class and an extract super class. Alike, for the atomic changes, the definition of the undo of an add element is the same definition of the undo of a delete element, too.

Consequently, by definition the complex change of an undo operation will overlap with the undone change C, by containing its atomic changes. Therefore, we can use our heuristics, in particular *h1*, to prioritize the undo operation over the undone change C to support the user in her decision. Furthermore, prioritizing the complex change of the undo operation notifies co-evolution approaches that a change had been undone during the evolution. Thus, to those undo operations no automatic resolution should be applied during the co-evolution process preventing eventually loss of information.

Let us further illustrate on the example of Fig. 8 how the heuristics help the user in her decision. Following the above described approach, we detect at the trace t_2 the complex changes "move property *id*" and the undo operation "undo delete property *id*", as shown in Fig. 9. As expected, the undo delete property *id* overlaps with the atomic change delete property *id* (n^o 1 in t_2). Using the heuristic *h1* will then prioritize the undo delete property *id* over the delete property *id*. Both changes gets respectively the priorities 2 and 1. Furthermore, the detected undo delete property *id*, overlaps partially with the move property *id* on the atomic change add property *id* (n^o 3 in t_2). Using the heuristic *h2* herein prioritizes the move property *id* over the undo delete property *id*. Both changes gets respectively the priorities 1 and 1/2. Indeed, in the example of Fig. 8 the undo delete property was not intended by the user during the evolution, in contrast to the move property. The heuristics *h2* in this case shows to be efficient in supporting the user in the correct decision, i.e. selecting the correct changes.

Using Algorithm 1 to detect undo operations also answers to the requirement R5 similarly as with requirement R4 for complex changes.

4. Implementation of the approach

4.1. Tool

In [3], the Praxis prototype is presented that is integrated to Eclipse EMF Framework. It traces the atomic model changes actions applied by the user while creating or evolving a (meta) model.

This prototype has been extended in order 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. The core functionalities of this component is implemented with Java (6898 LoC) and are packaged into an Eclipse plug-in that interfaces with the existing Praxis plug-ins [20,3].

Fig. 10 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. Fig. 11 shows the tool part of the heuristics presenting the priorities for the overlapping complex changes that helps the user in selecting the correct changes. The final evolution trace contains both atomic and complex changes.

4.2. Extensibility

Fig. 12 shows the extensible part of our tool's architecture. In particular, the `DetectionEngine` implements our detection algorithm that requires the complex change definitions as well as the evolution trace of the atomic changes. It first detects and creates all candidates of complex changes and then validates which candidates are complex changes. It is worth noting that the implementation allows for extensibility. In fact, if a user wants to include a new complex change, the following process is to be pursued:

1. A new complex change class is created inheriting the class `ComplexChanage`

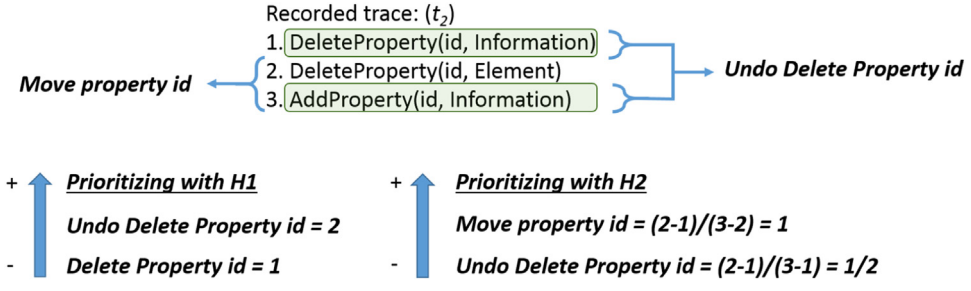


Fig. 9. Ranking detected changes and undo operations of Fig. 8.

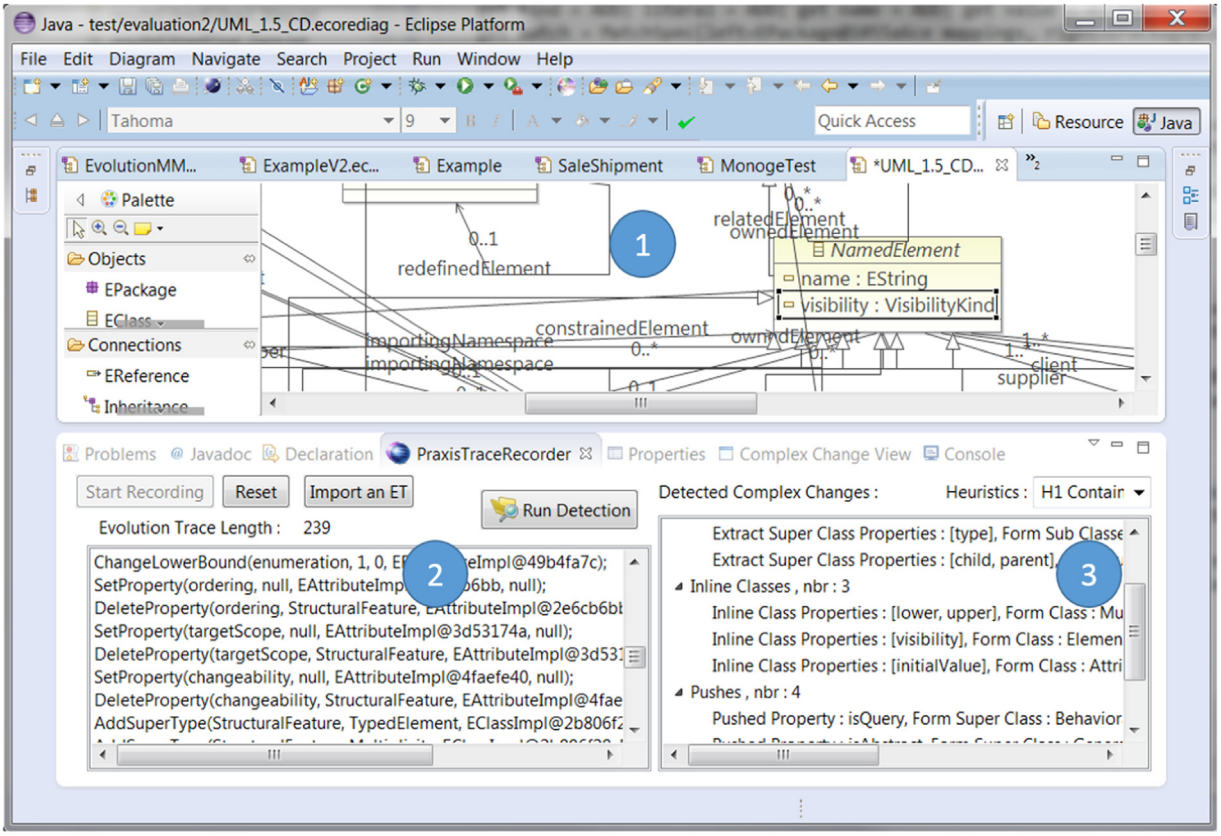


Fig. 10. Screenshot of the tool.

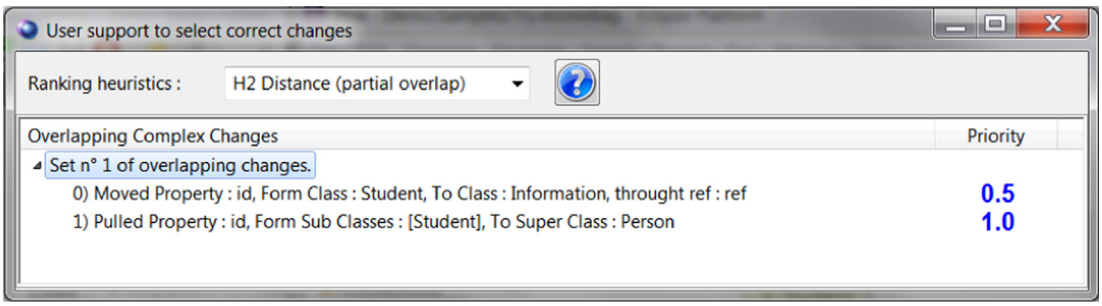


Fig. 11. Screenshot of the heuristics part of the tool.

2. A definition of the complex change is added inheriting the class `ComplexChanageDefinition` while implementing the two existing methods. The first method checks whether a given atomic change can be part of the complex change, and the second method creates a candidate for this complex change.
3. Finally, a candidate for the complex change is also added inheriting from the class `ComplexChanageCandidate`. Three methods must be implemented: (1) the first one checks if it is possible to add the current atomic change to the candidate while respecting the multiplicities. (2) the second one adds the current atomic change to the ones composing the candidate. (3) the third one checks the different conditions on the list of added atomic changes to validate (i.e. create) the corresponding complex change or not.

No changes are made in the Detection engine since we made it work on a generic level of the definitions and the candidates. Note that we separated the structure of the complex change, its definition, and its candidates for sake of simplicity and extensibility.

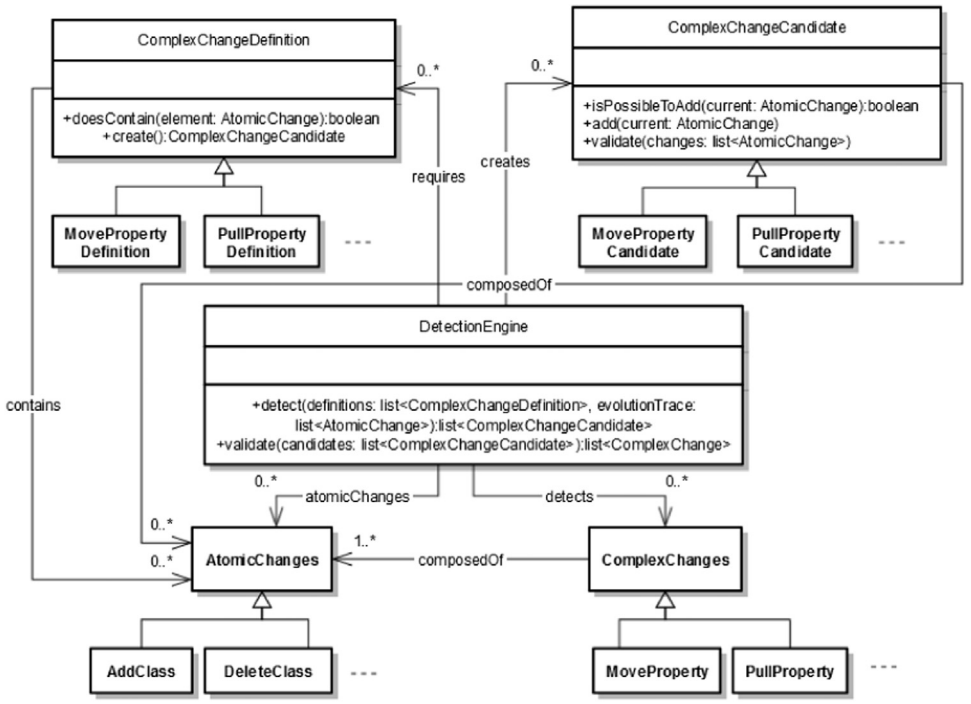


Fig. 12. Tool's architecture: extensible part.

5. Validation

This section presents the validation of the current 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 [36]: *precision, recall, and f-score*. Time performance and memory consumption are evaluated as well.

5.1. DataSet

At this stage of our research, we aim at validating empirically on 8 real case studies namely: Unified Modeling Language Class Diagram (UML CD) [35], Graphical Modeling Framework (GMF)[4] [15], OCL Pivot [28], BaseCST [28], CompleteOCLCST [28], EssentialOCLCST [28], ExtendedTypes [29], Benchmark [27].

In this validation, we have studied variety of metamodels of modeling languages from the Object Management Group (OMG) standards [32] to Private initiatives and concrete implementations for eclipse Model Development Tools (MDT)[5] such as Papyrus [29], Modisco [27], OCL [28] etc. We were interested in finding occurrences in practice of complex changes to show the capability of the current approach to detect them. We manually analyzed beforehand the atomic changes that lead a metamodel from a version n to a version $n+1$ in order to know what are the expected complex changes. To ensure that no complex changes are missed in the manual analysis, we also carefully studied the specifications and documentations that came with the metamodels. Without doing this analysis, we cannot assess the quality of detection of our approach, i.e. comparing what is expected in practice against what is detected by our tool.

Table 4 shows the selected case studies, their sources, versions, and the number of occurred atomic and complex changes in practice between the two versions. Table 5 gives the list of expected complex changes for each case study.

5.2. Process of the validation

To run the experiment, we take the first version n of each metamodel and we manually evolve it until version $n+1$ of each metamodel is reached while recording the modeling actions. For instance, by manually evolving the UML CD 1.5 and the GMF 1.29 metamodels to reach the UML CD 2.0 and the GMF 1.74 metamodels. In the meantime, our tool records the trace of atomic changes that serves for the detection of complex changes.

In this experiment, we measure the accuracy of our tool by using the three metrics *precision*, *recall*, *and f-score* [36] defined as follows:

- $precision = \frac{CorrectFoundChanges}{TotalFoundChanges}$
- $recall = \frac{CorrectFoundChanges}{TotalCorrectChanges}$

[4] We considered 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, similarly as investigated in [18,22].

[5] http://www.eclipse.org/modeling/mdt/

Table 4
Case studies.

Case studies	Source	Versions	Atomic changes	Complex changes
UML CD	OMG [32]	1.5 to 2.0	238	10
GMF	GMP [15]	1.29 to 1.74	220	10
OCL Pivot	OCL [28]	3.2.2 to 3.4.4	265	3
BaseCST	OCL [28]	3.1.0 to 5.0.4	104	6
CompleteOCLCST	OCL [28]	3.1.0 to 5.0.4	42	1
EssentialOCLCST	OCL [28]	3.1.0 to 5.0.4	43	1
ExtendedTypes	Papyrus [29]	0.9.0 to 1.1.0	42	4
Benchmark	Modisco [27]	0.9.0 to 0.13.0	70	10

Table 5
Complex changes of each case study.

Case studies	Expected complex changes						
	Move property	Pull property	Push property	Extract class	Inline class	Extract super class	Flatten hierarchy
UML CD	n/a	2	4	n/a	2	2	n/a
GMF	1	2	3	n/a	n/a	3	1
OCL Pivot	n/a	1	2	n/a	n/a	n/a	n/a
BaseCST	1	n/a	1	2	n/a	2	n/a
CompleteOCLCST	n/a	n/a	1	n/a	n/a	n/a	n/a
EssentialOCLCST	n/a	n/a	n/a	n/a	n/a	1	n/a
ExtendedTypes	n/a	2	1	n/a	n/a	1	n/a
Benchmark	3	5	1	n/a	1	n/a	n/a

- $f-score = \frac{2*Recall*Precision}{Recall+Precision}$

The values of these metrics are between 0 and 1, i.e. 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 (also called f-measure) 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 the detection algorithm.

Furthermore, we measure the three quality metrics on the detected complex changes for the following cases: (1) without using the ranking heuristics defined in Section 3.5, (2) when using the heuristics separately.

Note that 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.

Nonetheless, to measure the quality of the heuristics, we simulate in this evaluation the situation that the user decides to keep only the highest prioritized changes, i.e. we remove the lower prioritized changes automatically from the whole list of identified overlapping 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 8 GB of RAM with Windows 7 as OS.

5.3. Results

Figs. 13–17 show the results of the validation performed by our approach. In the following, we first present the results of the detection without using any heuristics. After that we present the observed results when using our heuristics.

5.3.1. Without using heuristics

Table 6 shows the number of detect complex changes opposed to the number of expected complex changes. For most case studies, we detected the expected complex changes as well as other incorrect complex changes.

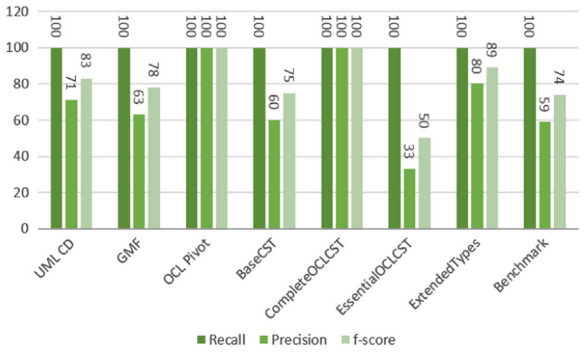


Fig. 13. Metrics without any heuristics.

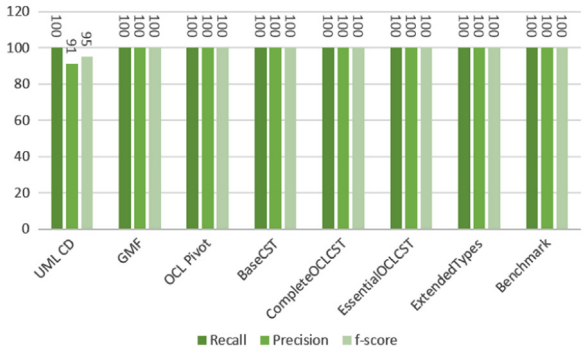


Fig. 14. Metrics with heuristic h1.

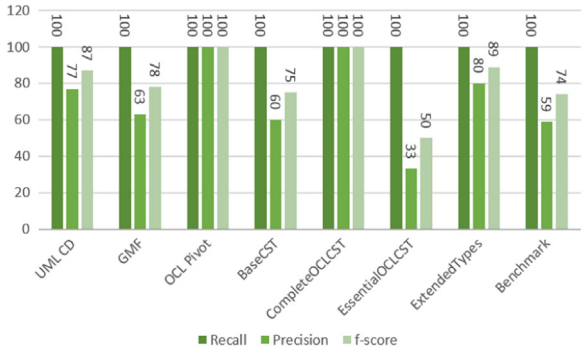


Fig. 15. Metrics with heuristic h2.

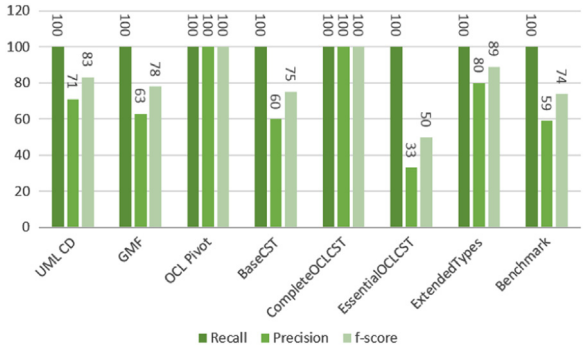


Fig. 16. Metrics with heuristic h3.

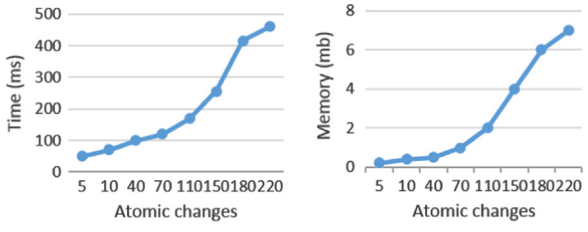


Fig. 17. Time and memory consumption.

Fig. 13 shows the quality metrics on the raw detected complex changes. It shows 100% recall for all case studies, i.e. metamodels evolutions, without using any heuristics.

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, respectively of one reference *type* and of two references *child and parent*, we also detect three pull properties 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 *featureSeqInitializer* that explains one additional pull property. We also applied one extract super class of *one operation getFeatureQualifiedPackageInterfaceName and one reference featureSeqInitializer*, and further one extract super class of two attributes *default and qualifiedToolName*, which explain the four additional pull properties. For the one flatten hierarchy of one reference *metaFeature*, we detect an additional push property. Thus, the precision is 10/16, i.e. 63% as shown in Fig. 13.

In the OCL Pivot and the CompleteOCLCST case studies, we respectively detected 3 complex changes and 1 complex change that were all expected. Indeed, we applied one pull property of *ownedAnnotation* and two push properties of *ownedRule and ownedPrecedence* for the OCL Pivot, and one push property of *parameters* for the CompleteOCLCST that were all correctly detected. Thus, the precision is 100% for both case studies.

In the Base CST case study, we detected 10 complex changes whereas we expected only 6, due to the full overlapping issue. For the first extract super class of one reference *ownedNestedPackage* and the second extract super class of one attribute *pivot* we detected two additional pulls of the same properties. In addition, for the each of the two extract class of one attribute *uri* we also detected an additional move of the same property. Thus, the precision is 6/10, i.e. 60% as shown in Fig. 13.

In the EssentialOCLCST case study, we detected 3 complex changes whereas we expected only 1. Besides the one extract super class of two operations *getNamedElement and getSimpleNamedExp* we detected two pull properties which are the two additional complex changes. Thus, the precision is 1/3, i.e. 33%.

In the ExtendedTypes case study, we detected 5 complex changes whereas we expected only 4. For the one extract super class of an attribute *id* we detect an additional pull of the same property making the precision 4/5, i.e. 80%.

Table 6
Detected complex changes for each case study.

	UML CD	GMF	OCL Pivot	Base-CST	Complete-OCLCST	Essential-OCLCST	Extended-types	Benchmark
Expected complex changes	10	10	3	6	1	1	4	10
Detected complex changes	14	16	3	10	1	3	5	17

Finally, in the Benchmark case study, we detected 17 complex changes whereas we expected only 10. For the one Extract class of five attributes *discoveryDate, maxUsedMemoryInBytes, saveTimeInSeconds, discoveryError, and totalExecutionTimeInSeconds* we detect five additional move properties of the same attributes. Similarly, for the one extract super class of two attributes *name and totalSizeInBytes* we detected two additional pull properties of the same attributes. Thus, the precision is 10/17, i.e. 59% as shown in Fig. 13.

Having a 100% recall for all case studies, the overall f-score for the UML CD, the GMF, the OCL Pivot, the BaseCST, the CompleteOCLCST, the EssentialOCLCST, the ExtendedTypes, and the Benchmark case studies respectively reaches 83%, 78%, 100%, 74%, 100%, 50%, 89%, and 74%.

5.3.2. Using only heuristic h1

Fig. 14 shows the quality metrics after using the heuristic h1. For the GMF, the BaseCST, the EssentialOCLCST, the ExtendedTypes, and the Benchmark case studies, h1 allows us to reach a precision of 100%. This is possible since these case studies contain only complex changes that overlap completely. The heuristic h1 ranks respectively the additional move, pull and push properties with a lower priority than the extract class, the extracts super class, and the flatten hierarchy, which are in our case study indeed the expected complex changes. Fig. 20 shows an example of an extract super class of two references *child and parent* from the UML CD case study. Along with this complex change, we also detect two pull properties of *child and parent* which are contained by the extract super class. The heuristic h1 gives the highest priority to the extract super class, i.e. 2, and a lower priority to both pull properties, i.e. 1.

For the UML CD case study, h1 allows to reach a precision of 91% (10/11), since the additional pull properties get a lower priority from h1 than the two expected extracts super class.

Furthermore, since the expected complex changes are ranked with the highest priority by h1, the recall thus stays 100% for all case studies. Therefore, the f-score is improved to 95% and to 100% respectively for the UML CD and the rest of the case studies.

5.3.3. Using only heuristic h2

Fig. 15 shows the quality metrics after using the heuristic h2. For the GMF, the BaseCST, the EssentialOCLCST, the ExtendedTypes, and the Benchmark case studies, h2 does not change the precision since there was no case of partly overlapping changes. However, the UML CD case study contains one case of partial overlap that is illustrated in Fig. 21. Along with the pull property *visibility*, an inline class of the same property is detected too. The heuristic h2 gives the highest priority to the pull property that occurred at once in contrast to the unexpected inline class. Thus, for the UML CD case study h2 allows to reach a precision of (10/13), i.e. 77%. Again, the recall stays unchanged for both case studies. Therefore, the f-score reaches 87% for UML CD and stays unchanged for the rest of the case studies.

5.3.4. Using only heuristic h3

Fig. 16 shows the quality metrics after using 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 does not improve the precision by giving the same priority 1 (i.e. 1–(0/2)) to the two partly overlapping changes: the pull property *visibility* and the unexpected inline class of the property *visibility*.

5.4. Performances

To measure the time perfomance and the memory consumption, we have respectively used the stopwatch technique with the System.currentTimeMillis() Java method, and the Runtime.totalMemory() and Runtime.freeMemory() Java methods.[6]

The validation experiments runs returned instantly and used insignificant memory as depicted Fig. 17. We detected the complex changes in less than 470 milliseconds (ms) over a trace of 220 atomic changes while consuming less than 7.6 megabytes (mb).

To further evaluate the performances on a bigger evolution trace, i.e. the scalability of our tool, we randomly generated large evolution traces *t*. In parallel, we also generated random complex changes and we introduced their composing atomic changes into the traces *t*. We ran several executions to measure the average time performance as well as the average memory consumption. Figs. 18 and 19 show the measured results when respectively 10, 20, 50, and 100 complex changes are introduced to the evolution traces *t*. We observe that up to 1000 atomic changes the execution time is less than 18 s. The maximum execution time is less than 5 min when 100 complex changes are present in 10,000 atomic changes. Similarly, memory consumption increased proportionally to the number of atomic changes to reach a maximum of

[6] Execution time = end Time − start Time. Used memory = end used memory − start used memory = (end total memory − end free memory) − (start total memory - start free memory)

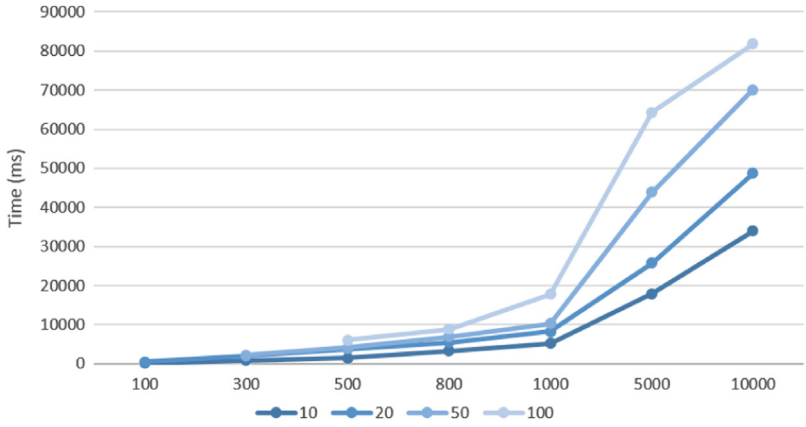


Fig. 18. Time performance scalability measures.

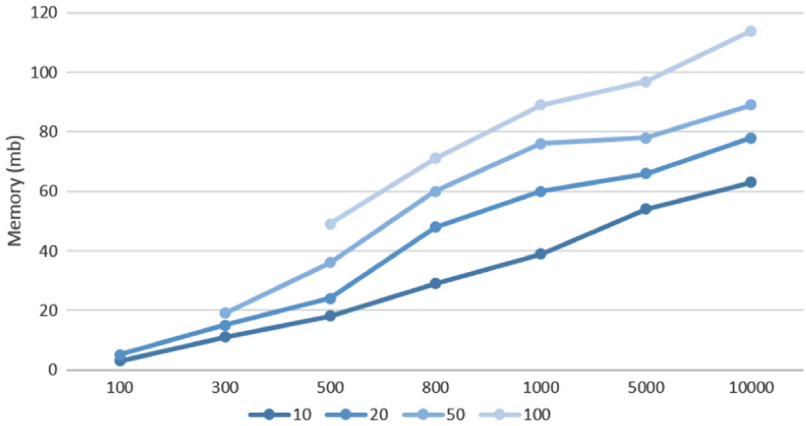


Fig. 19. Memory consumption scalability measures.

114 MB when 100 complex changes are present in 10,000 atomic changes.[7]

5.5. Discussion

In this paper, we proposed an approach for detecting complex changes during metamodel's evolution. Throughout this paper we showed that ambiguous situations could occur during evolution and thus some complex changes can overlap with each other. Our vision is therefore to aim at detecting all eventual complex changes, in particular the ones that overlap. The results show that we always reach a 100% recall as expected since we designed our detection algorithm to reach this goal that denotes the completeness of our detection. Reaching a full recall is essential in the current work since it guarantees that no applied complex change is missed during the detection phase. Moreover, reaching a 100% recall is possible since we do not rely on the difference between two metamodel's versions, but we rather record the evolution trace at run-time avoiding the *hidden changes* issue. It's worth noting that all complex changes defined in Table 2 are covered in our validation. Each complex change is at least covered by one case study.

Moreover, we observed that in most of our case studies the precision never reaches 100% due to the overlap issue. The precision interval is from 33% to 100% without using any ranking heuristics, making an average precision of 70.75%. This is acceptable in particular since all correct complex changes are detected. In our approach at the end the user confirms her intent by selecting the correct complex changes. To further support the user in this phase, we evaluated the benefit of our ranking heuristics. The results show that the heuristics h1 and h2 allow to improve the precision and thus the f-score. Therefore, theses heuristics can help the user in gaining time during the selection of the correct changes. However, h3 failed at improving the precision metric because the circumstances where h3 can be of help were not met. Indeed, in all case studies the situation when a complex change partly overlaps with more than two other complex changes over different atomic changes, did not occur. Although it seems

[7] Note that, when introducing 50 and 100 complex changes it already gives a trace of atomic changes larger than respectively 100 and 300. Thus we start measuring time and memory respectively from a trace of 300 and 500 atomic changes.

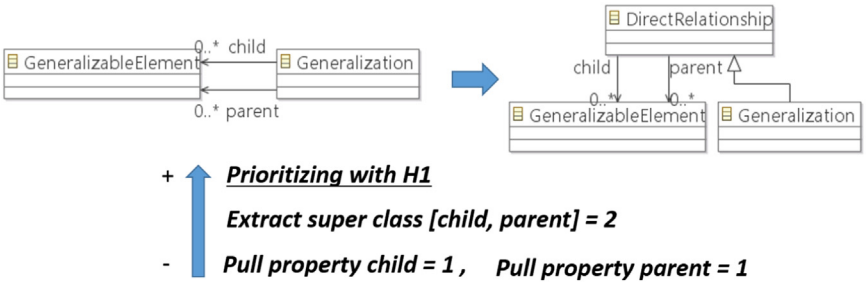


Fig. 20. Example of full overlap and the raking of h1 from the UML CD case study.

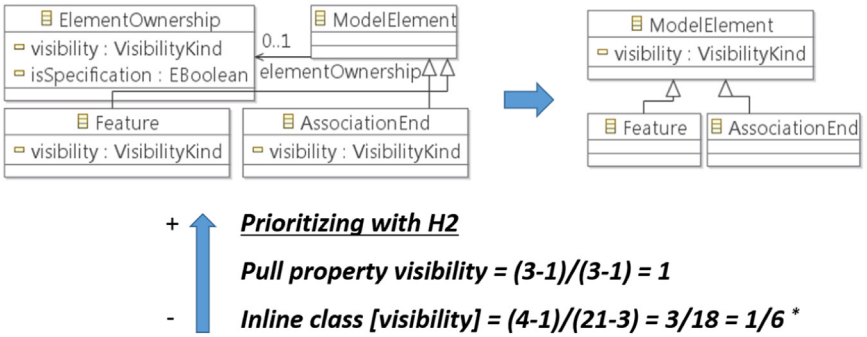


Fig. 21. Example of partial overlap and the raking of h2 from the UML CD case study. *: the add property visibility occurred here at the trace 3, and the last atomic change composing the inline class occurred at the trace 21.

Table 7
Occurred hidden changes for each case study.

	UML CD	GMF	OCL Pivot	Base-CST	Complete-OCLCST	Essential-OCLCST	Extended-types	Benchmark
Occurred hidden changes	5	1	n/a	1	n/a	1	1	5

to be seldom in practice, we cannot exclude it. Heuristic h3 still could be useful in other case studies.

In our case studies the issue of hidden changes did not occur since we used a tracing mechanism of the atomic changes. Nonetheless, we were interested whether this issue would have occurred in a difference-based approach. Thus, we analyzed the metamodels' evolutions in order to identify occurrences of hidden changes. In particular we analyzed the list of atomic changes that would be computed by the difference between the original and evolved metamodels. We identified several cases of hidden changes; Table 7 gives the number of hidden changes for each case study.

In five of our case studies, the hidden change issue occurred each time by a rename action. Meaning that after applying a complex change, one or several renames are applied on the involved elements. In the UML CD case study, we noticed five cases of *hidden* changes. Three properties in the two inlines class and two properties in the two extracts super class are all renamed afterward. One *hidden* change was noticed in the GMF case study regarding the one move property. One property in the extract super class of both the EssentialOCLCST and the ExtendedTypes case studies is renamed afterward. Finally, one property in the extract class and another one in a pull property are all renamed afterward in the Benchmark case study.

Furthermore, we observed two interesting cases of *hidden* changes that we call herein *successive complex changes*. Indeed, we had cases of complex changes that hide each other in the sens that the first complex change is hidden by the second successive one which is in turn hidden by the third successive one etc. In our validation we had one case in the BaseCST case study and another case in the Benchmark case study. As shown in Fig. 22, in the BaseCST case study the property *uri* is extracted first to the class `PathNameCS` and then a second time to the class `PathElementCS` before being pushed to the class `PathElementWithURICS`. Similarly, in the Benchmark case study the properties *saveTimeStandardDeviation, executionTimeStandardDeviation, and averageSaveTimeInSeconds* are first moved to the class `ProjectDiscovery` and then pulled to the class `Discovery`, as shown in Fig. 23. Some of those successive complex changes were also renamed afterwards.

While in our validation we detected these complex changes, it would not have been possible if we have retrieved the atomic changes from the difference model. The results of our detection engine would have been distorted reducing the recall and the precision measures. Fortunately, we record the evolution trace and therefore we hereby overcame the *hidden* change issue. We must

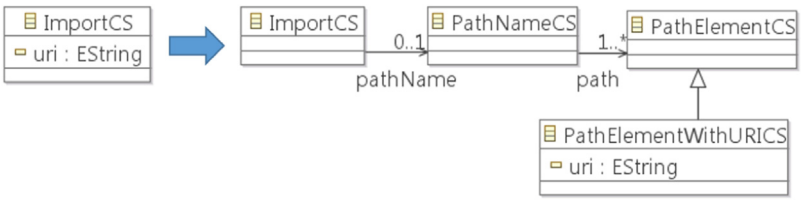


Fig. 22. Part of BaseCST metamodel showing an example of successive complex changes.

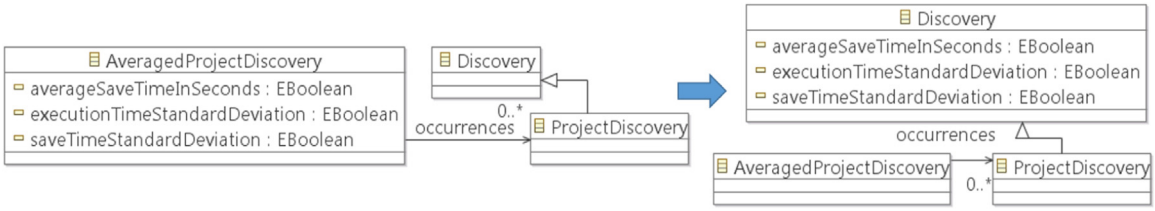


Fig. 23. Part of Benchmark metamodel showing an example of successive complex changes.

emphasize that this issue of *hidden* change is still a future work challenge for difference-based approaches to solve. Therefore, any research that is based on a difference-based approach must consider this issue carefully.

Furthermore, the scalability of our tool seems acceptable on both time and memory consumption. For a faster detection and a lower memory consumption when many of complex changes ($>10^2$) exist on large traces ($>10^4$), an improvement of the detection algorithm may be needed. Nonetheless, the performances are satisfying to cope with real case studies.

In summary, the validation shows that we are able to detect all applied complex changes and in case of ambiguity we can support the user in deciding for the correct complex changes.

5.6. Threats to validity

In this section we discuss internal, external, and conclusion validity after Wohlin et al. [43].

5.7. Internal validity

We had to manually apply the evolution of the metamodels ourselves in the modeling editor 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 specifically the ranking heuristics when external engineers apply the evolution. Moreover, 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 her intent, these deviating user decisions can in practice only improve the precision compared to this evaluation.

Furthermore, we did not evaluate the undo awareness mechanism (Section 3.6). However, in practice without the undo awareness mechanism the precision with real traces would be smaller than in our experiment. This is due to the side effects of the undo operations, if applied, making our approach to detect undone complex changes and thus lowering the precision. Nonetheless, the undo operations are detected similarly as the complex changes with the same algorithm. Thus, the results for the undo operations detection would be similar to the obtained results for the complex change detection.

5.8. 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 apply the evaluation on other modeling frameworks as well.

5.9. Conclusion validity

Finally, the validation on 8 real case studies already shows the strength of our approach in detecting all applied complex changes reaching a 100% recall. From a statistical point of view it would surely be better to evaluate more case studies in order to gain a more precise measures of the actual precision and to what extent our ranking heuristics are of help. However, results gained herein are sufficient to get an idea of the potential impact of the heuristics on the precision, in particular for h1 and h2. For a more detailed measures especially on the heuristics we

plan to identify and use more case studies/experiments on real metamodels' evolutions.

6. 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 [38,44,24] 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.

EMF Compare [38] is an Eclipse plug-in which is able to compare, to match, and to merge (two and three way merging) EMF/Ecore-based models. It does not rely on unique identifiers but rather uses a distance relation to match the same elements. UMLDiff [44] it is also non-id-based 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 [24] generalizes the UMLDiff approach to cope with domain-specific modeling languages.

Up to now, all existing approaches in the literature that detect complex changes rely on difference-based approaches to compute the atomic changes.

Williams et al. [41], they aim at computing a (near) optimal history model with a search-based technique. Thus, the history model will contain multiple alternatives of changes that lead a model from a version to another one. The authors defined a fitness function that select the most probable evolution. Note that there approach does not detect complex changes directly, but they rather rely on an operator-based tool namely the EAdapt tool.[8] It gives a list of operators which can be atomic and/or complex changes. Thus, they focus more on finding the correct sequence of operators that best describe the evolution.

Di Ruscio et al. [9], describe a language that users can use to manually specify the changes that occurred during evolution. Otherwise, EMF Compare [38] can be used to compute their list of changes.

Levendovszky et al. [23], introduced the Model Change Language (MCL) to be able to specify a metamodel evolution as well as co-evolution strategies. They defined metamodel atomic and complex changes that have to be detected as *idioms*. An *idiom* is a matching rule that search for a left-hand side (LHS) in the original metamodel and a the right-hand side (RHS) in the evolved version. If both are found, a change is then detect.

However, none of [41,9,23] considered the issues we tackled in this paper.

Cicchetti et al. [5] address the dependency ordering problem of atomic changes in order to ease the detection of complex changes but not directly the detection of complex changes. In fact, they focus on reordering atomic changes in such a way that eases the detection of complex changes. Thus, their work is helpful only if the atomic changes are retrieved by a difference-based approach, which is not the case of our current research.

Garces et al. [11] 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. However, Garces et al. [11] remains unclear about which complex changes and how their detection is performed.

Langer et al. [22] proposed to detect complex changes based on EMF Compare [38]. They use graph-based transformation to define a complex change with the left-hand side (LHS) and the right-hand side (RHS). A complex change is identified when LHS is present in the original version and the RHS in the evolved version. The LHS and the RHS can be expressed with two snapshots of an original metamodel and its evolved version. Thus, the variability in a complex change is not addressed. Langer et al. [22] measured the precision and the recall reaching respectively an average of 98% and 70%. Their goal is to ensure that their detection is correct despite missing some correct changes. In contrast to [22], we ensure the detection of all correct changes which may lead to the detection of some incorrect ones, and we then efficiently support the user in her final decision.

Garcia et al. [13] use EMF Compare [38] to detect atomic changes in a first step. They then 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. They mention to detect the most containing complex change without the contained ones, which addresses partially the overlap issue while potentially decreasing the recall. However, they do not consider variable complex changes.

Xing et al. [45] and Moghadam et al. [30] both propose to detect complex changes on UML CD based on the difference model computed with UMLDiff [44]. Both [45,30] represent the atomic changes in a database and they further define a complex changes as an SQL query that search for occurrences of atomic changes in the database.

Vermolen et al. [39] propose to detect complex changes over a manually ordered sequence of atomic changes returned from a difference model, which is computed by EMF compare [38]. They considered data models (i.e. schema) which do not reflect all metamodel constructions such as packages, operations ect. However, to the best of our knowledge they have the merit to be the first to consider the issue of hidden changes. In contrast to [22,13,5,11,45,39,30] also consider the issue of variability inside a complex change.

Table 8 gives a comparison of our approach and the related work. Note that we considered so far 7 most of applied changes out of 61 complex changes identified in [19]. It is difficult to compare with existing works since the considered complex changes in each existing approach may be different from ours. Thus, we only compare w.r.t. the following criteria: (1) the way of detecting atomic changes, (2) is the order issue of atomic changes handled? (3) is the hidden changes issue deal with? (4) is the overlap issue tackled? and (5) is the indefinite length issue handled?

As shown in Table 8 all the previous approaches [22,39,13,5,11,45,30] are based on differencing approaches

[8] http://www.eclipse.org/edapt

Table 8
Comparison of the different approaches that aim at detecting complex changes. Legend [✓: handled, ×: not handled, *Partially*: partially handled].

Approaches	Way to detect atomic changes (diff, recording)	Order of atomic changes	Hidden changes	Overlap	Indefinite length
Williams et al. [41]	Operator-based: EAdapt	✓	×	×	×
Di Ruscio et al. [9]	Diff or Manual	×	×	×	×
Levendovszky et al. [23],	matching LHS/RHS	×	×	×	×
Cicchetti et al. [5]	Diff	✓	×	×	×
Garces et al. [11]	Diff	×	×	×	×
Langer et al. [22]	Diff	×	×	×	×
Garcia et al. [13]	Diff	×	×	*Partially*	×
Xing et al. [45]	Diff	×	×	×	×
Moghadam et al. [30]	Diff	×	×	×	✓
Vermolen et al. [39]	Diff	✓	*Partially*	×	✓
Current approach	Recording	✓	✓	✓	✓

such as [38,44,24] or implement themselves a differencing approach as in [5,11]. Thus, they suffer from the drawbacks of non ordered, and potentially hidden changes.

Only [5,39] considers the issue of atomic changes order in a difference model, by defining strategies to reorder the atomic changes. For example, by putting an update change of an element *e* after its addition and not before.

Only [39] considers the issue of hidden changes in a difference model 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. However, no evidence is provided to show that the handling process of the hidden changes issue is correct. In this paper, we overcome those two issues by relying on Praxis [3] 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. In addition to detect overlapping changes, we further tackle the overlap issue by proposing ranking heuristics to better support the user. 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.

7. 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 evolution 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 the validation. Moreover, using the same detection algorithm we detect undo operations and we further use our heuristics to prioritize them in case of overlap. Thus, co-evolution approaches relying on our approach can avoid applying unwanted resolutions in response to an undone atomic/complex change while being aware of its undone operation.

We validated our approach on 8 real case studies from different domains showing its ability to detect all complex changes reaching a 100% recall and to supporting the user in selecting the correct changes. The validation demonstrates the applicability and the substantial benefit of our approach to the co-evolution approaches. Performance tests showed fair scalability results for both time and memory, yet still improvable. In the validation, we discussed the identified occurrences of hidden changes among our case studies. Vermolen et al. [39] considered the issue of *hidden* changes in theory, but to the best of our knowledge we are the first to encounter them in practice. Indeed, 14 hidden changes occurred in 6 out of the 8 case studies. This issue of *hidden* changes remains a challenge for difference-based approaches to solve in future works. Finally, our tool [20,21] is currently integrated with EMF/Ecore editor and UML CD/Papyrus editor.

For future work, we first aim at extending our approach to detect more complex changes from the list of already identified ones in [19]. Furthermore, we plan to do further research on improving the precision of our tool. In particular, we plan to introduce a new ranking heuristic that relies on machine learning techniques [42]. The idea is to learn from the user behavior and her decisions of the correct changes from a set of overlapping changes. Therefore, a ranking of the overlapping changes can be computed based on the established knowledge. Finally, to broaden the use of our approach to already evolved metamodels, we plan to offer the possibility to compute the atomic changes from an original and its evolved metamodels while addressing the issue of hidden changes.

Acknowledgments

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

References

[1] R. Bendraou, M.A.A. da Silva, M.-P. Gervais, X. Blanc, Support for deviation detections in the context of multi-viewpoint-based development processes, in: CAiSE, 2012, pp. 23–31.
[2] X. Blanc, A. Mougenot, I. Mounier, T. Mens, Incremental detection of model inconsistencies based on model operations, in: CAiSE, Springer, 2009, pp. 32–46.
[3] X. Blanc, I. Mounier, A. Mougenot, T. Mens, Detecting model inconsistency through operation-based model construction, in: ICSE'08, 2008, pp. 511–520.

[4] A. Cicchetti, D.D. Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: EDOC '08, 2008, pp. 222–231.
[5] A. Cicchetti, D.D. Ruscio, A. Pierantonio, Managing dependent changes in coupled evolution, in: R.F. Paige (Ed.), Theory and Practice of Model Transformations, 2009, pp. 35–51.
[6] M.A.A. da Silva, X. Blanc, R. Bendraou, Deviation management during process execution, in: Proceedings 26th IEEE/ACM ASE, 2011, pp. 528–531.
[7] A. Demuth, R. Lopez-Herrejon, A. Egyed. Automatically generating and adapting model constraints to support co-evolution of design models, in: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2012, pp. 302–305.
[8] A. Demuth, R.E. Lopez-Herrejon, A. Egyed, Supporting the co-evolution of metamodels and constraints through incremental constraint management, in: A. Moreira, B. Schatz, J. Gray, A. Vallecillo, P. Clarke (Eds.), Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science, 8107, Springer, Berlin Heidelberg, 2013, pp. 287–303.
[9] D. Di Ruscio, L. Iovino, A. Pierantonio, What is needed for managing co-evolution in mde?, in: Proceedings of the 2nd International Workshop on Model Comparison in Practice. ACM, 2011, pp. 30–38.
[10] K. Garcés, F. Jouault, P. Cointe, J. Bézivin, Adaptation of models to evolving metamodels, 2008.
[11] K. Garcés, F. Jouault, P. Cointe, J. Bézivin, Managing model adaptation by precise detection of metamodel changes, in: R.F. Paige, A. Hartman, A. Rensink (Eds.), ECMDA-FA, 2009, pp. 34–49.
[12] K. Garcés, J.M. Vara, F. Jouault, E. Marcos, Adapting transformations to metamodel changes via external transformation composition, Softw. Syst. Model. 13 (2) (2014) 789–806.
[13] J. Garcia, O. Diaz, M. Azanza, Model transformation co-evolution: a semi-automatic approach, in: K. Czarnecki, G. Hedin (Eds.), Software Language Engineering, 2012, pp. 144–163.
[14] J. Garcia, O. Dìaz, J. Cabot, An adapter-based approach to co-evolve generated sql in model-to-text transformations, in: Jarke, Mylopoulos, Quix, Rolland, Manolopoulos, Mouratidis, Horkoff (Eds.), CAiSE, 2014, pp. 518–532.
[15] GMP, Graphical modeling project, graphical modeling framework (gmf). ⟨http://www.eclipse.org/modeling/gmp/⟩, 2015.
[16] M. Herrmannsdoerfer, COPE a workbench for the coupled evolution of metamodels and models, in: B. Malloy, S. Staab, M.v.d. Brand (Eds.), Software Language Engineering, 2011, pp. 286–295.
[17] M. Herrmannsdoerfer, S. Benz, E. Juergens, COPE – automating coupled evolution of metamodels and models, in: S. Drossopoulou (Ed.), ECOOP 2009 Object-Oriented Programming, Lecture Notes in Computer Science, 5653, Springer, Berlin Heidelberg, 2009, pp. 52–76.
[18] M. Herrmannsdoerfer, D. Ratiu, G. Wachsmuth, Language evolution in practice: the history of GMF, in: M.V.D. Brand, D. Gasevic, J. Gray (Eds.), Software Language Engineering, 2010, pp. 3–22.
[19] M. Herrmannsdoerfer, S.D. Vermolen, G. Wachsmuth, An extensive catalog of operators for the coupled evolution of metamodels and models, in: Malloy, Staab, Brand (Eds.), Software Language Engineering, 2011, pp. 163–182.
[20] D.E. Khelladi, R. Bendraou, 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, ACM, 2016, pp. 617–620.
[21] D.E. Khelladi, R. Hebig, R. Bendraou, J. Robin, M.-P. Gervais, Detecting complex changes during metamodel evolution, in: The 27th CAISE, Springer, 2015, pp. 264–278.
[22] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdorfer, M. Seidl, K. Wieland, G. Kappel, A posteriori operation detection in evolving software models, J. Syst. Softw. 86 (2) (2013) 551–566.
[23] T. Levendovszky, D. Balasubramanian, A. Narayanan, F. Shi, C. van Buskirk, G. Karsai, A semi-formal description of migrating domain-specific models with evolving domains, Softw. Syst. Model. 13 (2) (2014) 807–823.
[24] Y. Lin, J. Gray, F. Jouault, Dsmdiff: a differentiation tool for domain-specific models, Eur. J. Inf. Syst. 16 (4) (2007) 349–361.
[25] S. Markovic, T. Baar, Refactoring OCL annotated UML class diagrams, in: L.BriandC. Williams (Ed.), Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, 3713, Springer, Berlin Heidelberg, 2005, pp. 280–294.
[26] S. Markovic, T. Baar, Refactoring OCL annotated UML class diagrams, Softw Syst Model 7 (1) (2008) 25–47.
[27] MDT, Model development tools, modisco, ⟨http://www.eclipse.org/modeling/mdt/?project=modisco⟩, 2015.
[28] MDT, Model development tools, object constraints language (ocl). ⟨http://www.eclipse.org/modeling/mdt/?project=ocl⟩, 2015.
[29] MDT, Model development tools, papyrus, ⟨http://www.eclipse.org/modeling/mdt/?project=papyrus⟩, 2015.
[30] I.H. Moghadam, M.O. Cinneide, Automated refactoring using design differencing, in: 16th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2012, pp. 43–52.
[31] A. Mougenot, Praxis: Detection of inconsistency within distributed models, Ph.D. thesis, Paris 6, 2010.
[32] OMG, Object management group, ⟨http://www.omg.org/spec/⟩, 2015.
[33] OMG, Object management group, meta object facility (mof), ⟨http://www.omg.org/spec/MOF/⟩, 2015.
[34] OMG, Object management group, object constraints language (ocl), ⟨http://www.omg.org/spec/OCL/⟩, 2015.
[35] OMG, Object management group, unified modeling language (uml), ⟨http://www.omg.org/spec/UML/⟩, 2015.
[36] C. Rijsbergen, Information retrieval, Butterworths, 1979.
[37] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[38] A. Toulmé, I. Inc, Presentation of emf compare utility, in: Eclipse Modeling Symposium, 2006, pp. 1–8.
[39] S.D. Vermolen, G. Wachsmuth, E. Visser, Reconstructing complex metamodel evolution, in: A. Sloane, U. Aßmann (Eds.), Software Language Engineering, 2012, pp. 201–221.
[40] G. Wachsmuth, Metamodel adaptation and model co-adaptation, in: E. Ernst (Ed.), ECOOP 2007 Object-Oriented Programming, 2007, pp. 600–624.
[41] J.R. Williams, R.F. Paige, F.A. Polack, Searching for model migration strategies, in: Proceedings of the 6th International Workshop on Models and Evolution, ACM, 2012, pp. 39–44.
[42] I.H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2005.
[43] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science Business Media, 2012.
[44] Z. Xing, E. Stroulia, Umldiff: an algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM ASE, 2005, pp. 54–65.
[45] Z. Xing, E. Stroulia, Refactoring detection based on umldiff change-facts queries, in: Proceedings of the 13th Working Conference on Reverse Engineering, WCRE'06, IEEE, 2006, pp. 263–274.