Volume 86, issue 2, February 2013
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software

The Journal of Systems and Software 86 (2013) 551–566

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

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

A posteriori operation detection in evolving software models

Philip Langer[a,*], Manuel Wimmer[b], Petra Brosch[a], Markus Herrmannsdörfer[c], Martina Seidl[d], Konrad Wieland[a], Gerti Kappel[a]

[a] *Business Informatics Group, Vienna University of Technology, Austria*
[b] *Software Engineering Group, Universidad de Málaga, Spain*
[c] *Institut für Informatik, Technische Universität München, Germany*
[d] *Institute for Formal Models and Verification, Johannes Kepler University Linz, Austria*

ARTICLE INFO

Article history:
Received 20 January 2012
Received in revised form
26 September 2012
Accepted 26 September 2012
Available online 23 October 2012

Keywords:
Model evolution
Model refactoring
Model comparison

ABSTRACT

As every software artifact, also software models are subject to continuous evolution. The operations applied between two successive versions of a model are crucial for understanding its evolution. Generic approaches for detecting operations a posteriori identify *atomic operations*, but neglect *composite operations*, such as refactorings, which leads to cluttered difference reports.

To tackle this limitation, we present an orthogonal extension of existing atomic operation detection approaches for detecting also composite operations. Our approach searches for occurrences of composite operations within a set of detected atomic operations in a post-processing manner. One major benefit is the reuse of specifications available for executing composite operations also for detecting applications of them. We evaluate the accuracy of the approach in a real-world case study and investigate the scalability of our implementation in an experiment.

1. Introduction

As every software artifact, also software models (Bézivin, 2005) are subject to continuous evolution. Knowing the operations applied between two successive versions of a model is not only crucial for helping developers to efficiently understand the model's evolution (Koegel et al., 2010), but it is also a major prerequisite for model management tasks, such as model co-evolution (Herrmannsdoerfer et al., 2009; Mens, 2008) and model versioning (Brosch et al., 2010; Koegel et al., 2010). In general, we may distinguish between two categories of operations. The first category includes *atomic operations*, such as additions, deletions, updates, and moves. The second category comprises *composite operations* (Sunyé et al., 2001) consisting of a set of cohesive atomic operations, which are applied within one transaction to achieve one common goal. The most prominent class of such composite operations are *refactorings* introduced by Opdyke (1992). As reported in Herrmannsdoerfer et al. (2009), Mens (2008), Brosch et al. (2010) and Koegel et al. (2010), the detection of applied refactorings is a crucial prerequisite for automating model management tasks. However, composite operations are not limited to refactorings; they may be used to implement any kind of in-place model transformation for a specific purpose, such as model completion (Sen et al., 2010), refinement (Ruhroth and Wehrheim, 2012), and evolution (Meyers and Vangheluwe, 2011).

Identifying the applied composite operations is a challenging task. One way to acquire the set of applied composite operations is to use *operation recording* (Herrmannsdoerfer and Kögel, 2010; Lippe and Oosterom, 1992); that is, the execution of operations is tracked within the modeling environment while they are performed. Although this leads to a precise operation log (Mens, 2002), there are several drawbacks. Most importantly, such approaches strongly depend on the modeling environment and only those operations are detectable that are supported by the modeling editor. A set of manually applied atomic operations, having together the intent of a composite operation, which is indeed frequently happening in practice (Murphy-Hill et al., 2009), cannot be identified, because no explicit command has been issued in the modeling environment. Finally, in a usual setting, the evolution of models is stored in terms of revisions in version control systems; consequently, the recorded operation logs (Mens, 2002) are not available.

In the absence of an operation log, the applied operations have to be detected *a posteriori* using *state-based model comparison* approaches using either generic model comparison algorithms (Brun and Pierantonio, 2008; Kelter et al., 2005; Lin et al., 2007; Schmidt and Gloetzner, 2008) or language-specific comparison algorithms (Kolovos, 2009; Xing and Stroulia, 2005). Whereas current generic approaches only support detecting atomic operations, some language-specific approaches also allow for detecting composite operations; but only for one specific modeling language. In

* Corresponding author.
E-mail address: langer@big.tuwien.ac.at (P. Langer).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.09.037

addition, the specifications for *executing composite operations* (e.g., Henshin, Arendt et al., 2010 and EMO, Brosch et al., 2009) and the implementation of the algorithms *for detecting* applications of them have to be kept consistent manually.

To tackle these limitations, we propose to *reuse* existing specifications for executing composite operations also for detecting applications of them. In particular, we realize the detection of *composite operations a posteriori* by extending existing model comparison tools with an additional phase. Our detection process takes two inputs. The first input is a *difference model* containing the applied atomic operations obtained from existing comparison tools. The second input is a set of composite operation specifications, which are preprocessed automatically to explicate their *difference pattern* needed for detecting applications of them. In the first step of our three-step detection process, the difference model is scanned for occurrences of composite operations' difference patterns. To consider the semantics of composite operations (Kniesel and Koch, 2004; Cornélio et al., 2010), the second and third step check the considered models regarding the composite operations' pre- and postconditions, respectively. The final output is a difference model enriched with composite operations' applications that aggregate the atomic operations they consist of.

The benefits of our approach are the following. Our approach does not rely on any editor-based operation tracking; thus, it is independent of the used modeling environment. As a further consequence, our approach is also capable of detecting applications of composite operations, even if they have been performed manually by applying their comprised atomic operations. Our approach is designed to be metamodel-agnostic. As the implementation of our approach is based on the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008), it is applicable to any Ecore-based modeling language, such as UML, any domain-specific modeling language (Gray et al., 2007), and even to Ecore itself which allows to apply the approach not only to models but also to metamodels. Our approach does not require for additional detection rules. On the contrary, the set of detectable composite operations is derived automatically from specifications for executing composite operations. Thus, already existing executable operation specifications can be reused directly from previous work (Brosch et al., 2009) or new composite operations may be defined easily using existing operation specification tools (e.g., Arendt et al., 2010; Brosch et al., 2009) without having to keep the detection rules consistent with the specifications for their execution. Finally, any kind of composite operations are supported, being either used for refactoring or for any other purpose.

This paper is organized as follows. In Section 2, we provide an overview on the prerequisites of our approach and discuss the present gap between specifications for executing composite operations and for detecting applications of them as well as how this gap can be bridged. Our approach for detecting applications of composite operations is presented in Section 3. In Section 4, we evaluate the correctness and completeness of our implementation and investigate the scalability and performance of our implementation in Section 5. In Section 6, we survey related work and in Section 7, we conclude with a short summary and possible extensions of the presented work.

2. From atomic operations to composite operations

In this section, we first introduce the prerequisites for our detection approach by outlining briefly the metamodeling stack and illustrating the power of metamodeling frameworks, which constitute the basis for processing models using generic, language-independent algorithms.

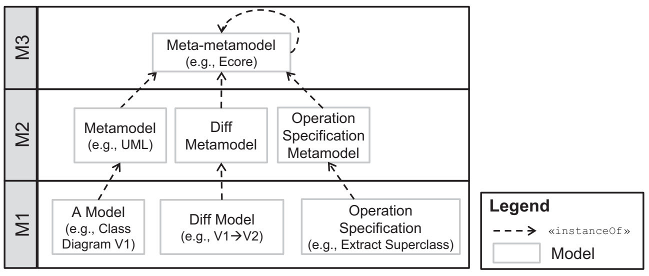

M3
M2
M1
Meta-metamodel (e.g., Ecore)
Metamodel (e.g., UML)
Diff Metamodel
Operation Specification Metamodel
A Model (e.g., Class Diagram V1)
Diff Model (e.g., V1→V2)
Operation Specification (e.g., Extract Superclass)
Legend
«instanceOf»
Model

Fig. 1. Metamodeling stack.
Adopted from Bézivin and Gerbé (2001).

2.1. Prerequisite: metamodel-agnostic processing of models

In an endeavor to establish a commonly accepted conceptual framework for the rapidly growing number of domain-specific modeling environments, the Object Management Group (OMG) released the specification for Model Driven Architecture (MDA) (Object Management Group, 2005), standardizing the definition and usage of (meta-)metamodels resulting in a metamodeling stack as depicted in Fig. 1. The meta-metamodel level M3 manifests the role of a common meta-modeling language. The metamodel level M2 contains any metamodel defined with the meta-metamodel to represent a modeling language's abstract syntax in terms of metaclasses and their structural features. The model level M1 contains models being instances of metamodels, such as a concrete UML class diagram or business process model.

A very useful property of this metamodeling stack is that every model is a direct instance of its respective metamodel and an indirect instance of the meta-metamodel. This property is leveraged by current metamodeling frameworks, such as EMF, for providing the means for processing all models uniformly by applying model processing mechanisms based on the M3 level. In particular, by using a reflection mechanism, similar to the reflection mechanism of programming languages, the metamodeling frameworks allow to process models generically without considering the M2 level. For instance, in EMF, every model element is an instance of EObject, which provides methods for accessing the object's feature values, its metaclass, etc.

We base our approach on two types of such metamodel-agnostic tools. First, generic model comparison tools use the reflection power of the underlying metamodeling framework for comparing models, irrespectively of their metamodel. The detected differences are represented in terms of Diff Models (M1) conforming to a Diff Metamodel (M2). Second, we consider tools for specifying and executing composite operations. Usually, such tools introduce their own Operation Specification Metamodel (M2) to define composite operations (M1). To support the specification of operations for any modeling language, these tools have to be independent from the modeling languages' metamodels. However, as concrete composite operations are specific to a modeling language, the operation specification metamodel usually introduces a template concept: thereby, templates in operation specifications refer to the metaclasses of the specific modeling language's metamodel to restrict the applicability of a composite operation to specific model element types. Thus, the tools themselves are metamodel-independent, whereas concrete operation specifications are designed for a specific modeling language.

2.2. Gap between difference models and operation specifications

Current model comparison tools apply a two-phase process: (i) correspondences between model elements are computed by *model matching* algorithms (Kolovos et al., 2009), and (ii) a *model diffing* phase computes the differences between two models from the

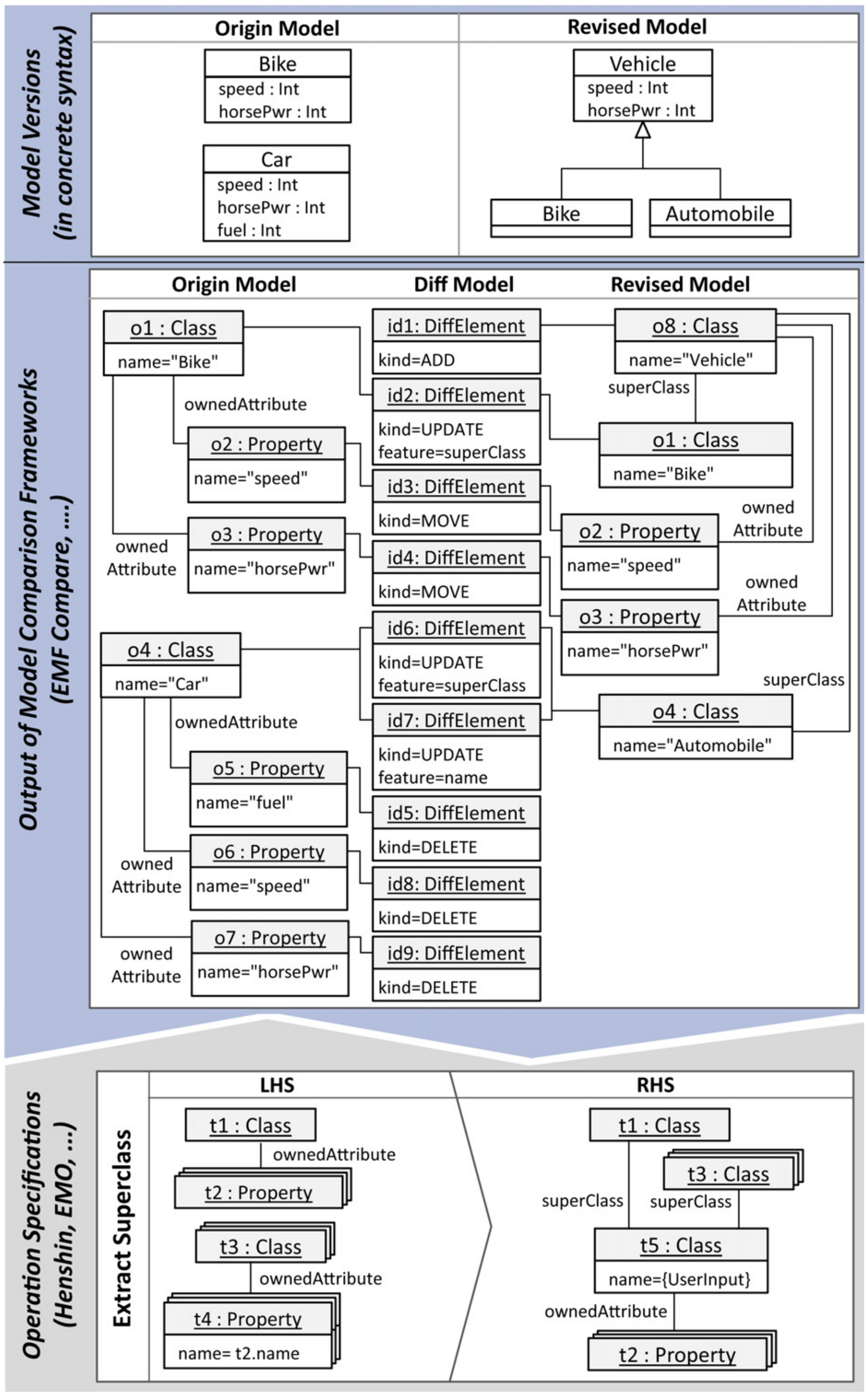


Fig. 2. Gap between diff models and composite operation specifications.

established correspondences. For instance, EMF Compare (Brun and Pierantonio, 2008) – a prominent representative of model comparison tools in the Eclipse ecosystem – is capable of detecting the following types of atomic operations:

- **Add**: A model element only exists in the revised version.
- **Delete**: A model element only exists in the origin version.
- **Update**: A feature of a model element has a different value in the revised version than in the origin version.
- **Move**: A model element has a different container in the revised version than in the origin version.

An example for a generic diff model comprising solely atomic operations between two versions of a UML class diagram is depicted in the upper half of Fig. 2. More specifically, this figure shows an origin model and a revised model in the concrete syntax, as well as in the abstract syntax in terms of a UML object diagram. Between the origin model and the revised model, the refactoring *Extract Superclass* has been applied among other atomic operations. In the course of the applied refactoring, the new superclass Vehicle is introduced for the two existing classes and all common properties contained by the existing classes are pulled up to the new superclass.

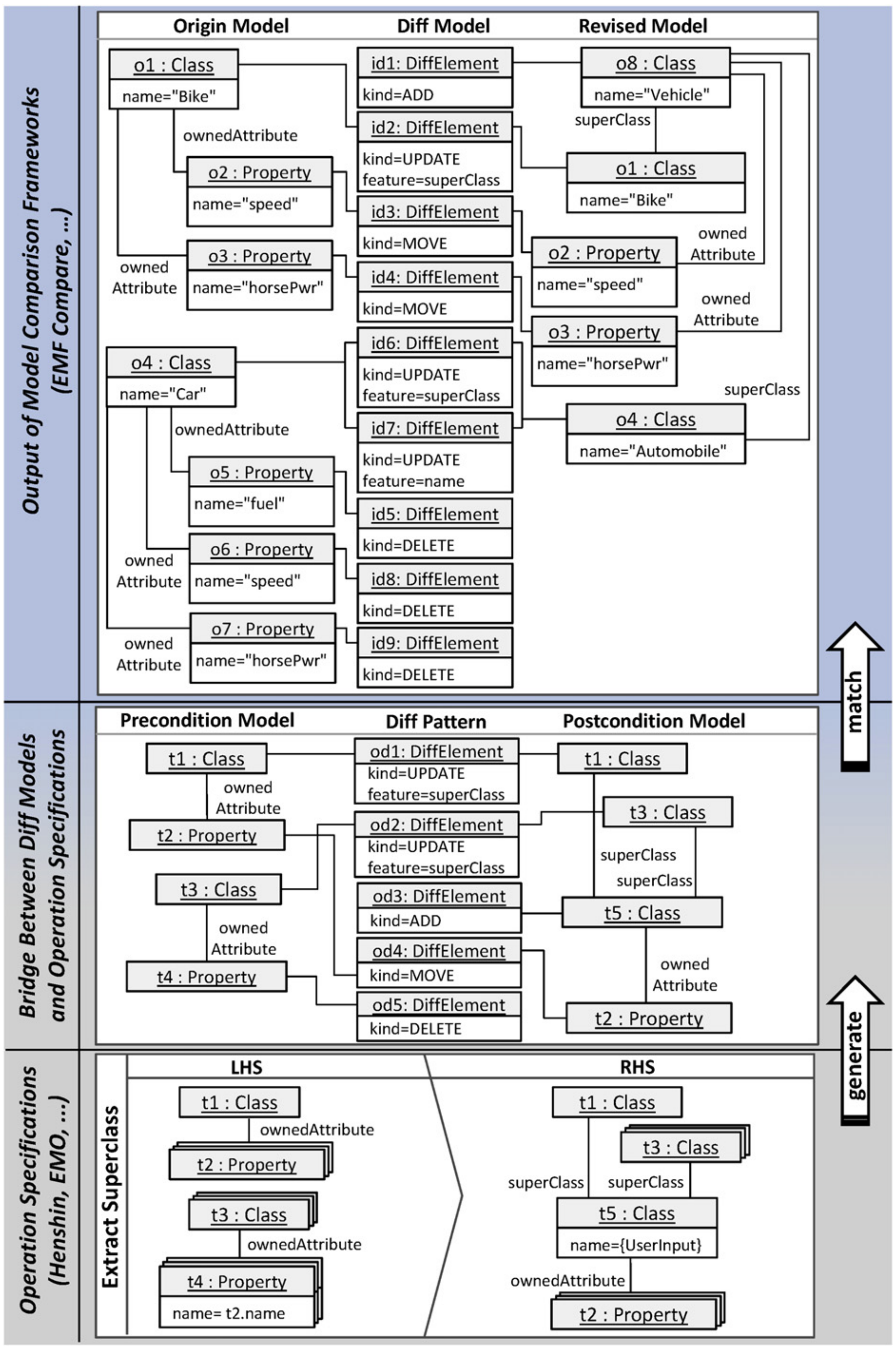


Fig. 3. Bridging diff models and operation specifications.

From these two model versions, all applied *atomic operations* can be derived using a state-based model comparison. The obtained diff model is depicted in the middle of Fig. 2, which comprises a set of diff elements representing one addition of the new class, two updates for setting the new class as the superclass of both existing classes, two moves of the original properties of class Bike, and the deletion of the pulled-up properties of class Car. Besides the refactoring, other atomic operations have been performed in this example: the class Car has been renamed to Automobile and the property fuel has been deleted. In the absence of a recorded operation log, the only way for humans to reconstruct the applied composite operations, e.g., the refactoring *Extract Superclass*, is to reason about the obtained atomic operations in combination with the origin model and revised model which is a cognitive challenge even for small examples. As reported in Brosch et al. (2010), Herrmannsdoerfer et al. (2009), Koegel et al. (2010) and Mens (2008), working only on this level does not scale.

To address this issue, a more concise view of model differences is required that aggregates the atomic operations into composite operation applications such that the intent of the change is becoming explicit. Existing solutions (Hartung et al., 2010; Küster et al., 2008; Xing and Stroulia, 2006) only provide language-specific operation detection algorithms. However, due to the plethora of existing modeling languages, this is an unfavorable solution. The situation could be improved significantly when *reusing existing specifications of composite operations*, which presently only allow for their automatic execution, also for the *detection of their applications* by using a *generic detection approach*.

Model transformations (cf. Czarnecki and Helsen, 2006 for an overview) are the current technique of choice for specifying

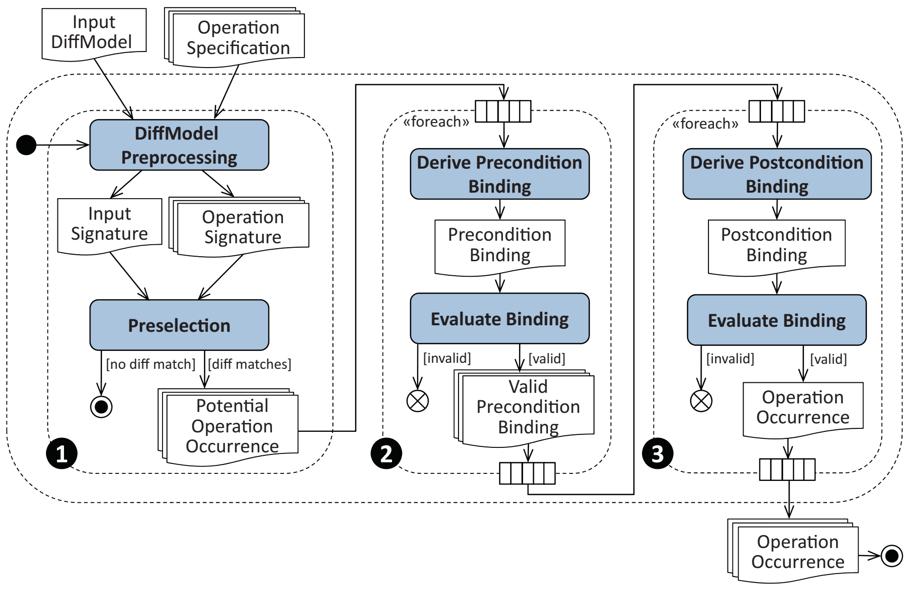


Fig. 4. Composite operation detection process: (1) Diff pattern matching, (2) precondition matching and (3) postcondition matching.

executable composite operations. In particular, composite operations are specified by rules stating the operation's preconditions, postconditions, and actions that have to be executed for applying the operation. An example operation specification is depicted using graph transformation syntax (Heckel, 2006) in the lower part of Fig. 2 for the refactoring *Extract Superclass*. The left-hand side (LHS) represents the precondition of the operation and the postcondition is specified in the right-hand side (RHS). Please note that graph transformations are a declarative specification approach; thus, the operation's actions are implicitly defined by the LHS and RHS. The precondition of the example operation states that only equally named properties may be pulled up to the new superclass. Additionally, the operation should be applicable for a set of properties and it should enable to extract the superclass for a set of classes. Therefore, so-called *iterations* (also referred to as *multi-objects* in the literature) are used. Iterations indicate that more than one object may be bound to one element in the preconditions (e.g., t4) such that all matching objects are transformed equally when executing the composite operation. Thanks to the iteration concept, the transformation rule is capable of pulling up multiple properties matching t2 and deleting all equally named properties matching t4 contained by classes matching t3.

Current execution engines for model transformations provide means for executing composite operation specifications, but they do not support detecting occurrences of them. Additionally, operation specifications are not designed to be matched directly with diff models produced by current model comparison frameworks. Thus, there is a gap between these two worlds.

2.3. Bridging difference models and operation specifications

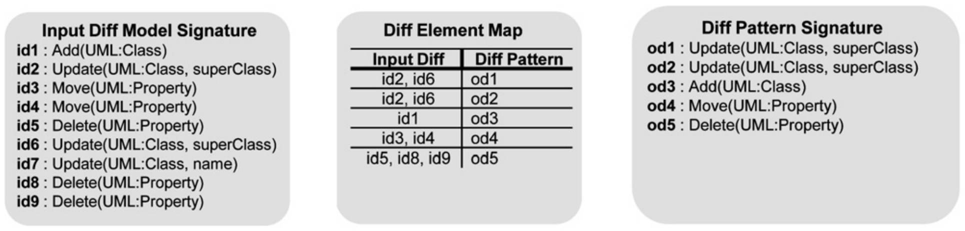
To bridge the mentioned gap, we propose to automatically generate an intermediate structure from operation specifications as shown in Fig. 3. This intermediate structure extends composite operation specifications by making explicit their comprised atomic operations. These atomic operations form the operation's *diff pattern*, which can be matched with a diff model obtained from model comparison tools. If a match of a diff pattern is found in a diff model, we proceed with evaluating the pre- and postconditions on the origin and revised models, respectively, and if these conditions are fulfilled, an occurrence of the respective composite operation is reported.

For making the diff pattern explicit, we compute a diff model by diffing the LHS and RHS of the transformation rules. Advanced constructs of graph transformation rules, such as iterations, positive and negative application conditions, are not considered as they are checked in the subsequent evaluation of the pre- and postconditions. Producing a diff pattern using model comparison requires that the LHS and the RHS of transformation rules be represented by "pure" models (i.e., direct instances of the modeling language's metamodel) and not as models conforming to a transformation language metamodel as is often the case (cf. Section 2.1). Therefore, we apply a dedicated transformation to translate the LHS and RHS of the transformation rules to pure models. This transformation is the inverse of the transformation used in Kühne et al. (2009) for generating a language-specific transformation language out of a modeling language. More precisely, the LHS and the RHS of a transformation rule comprise *templates*, which possess a *type*, a *variable*, *links* to other templates, *conditions* on attribute values in the LHS, *actions* for setting attribute values in the RHS, as well as a flag, which defines whether the template represents an iteration or not. As shown in Fig. 3, these templates are easily transformable to pure objects. Moreover, an ID that corresponds to the template variable (e.g., t1) is assigned to each object for preserving the trace between LHS and RHS objects, which is important for the comparison process. Having generated pure models, existing model comparison tools can be used to derive an explicit diff pattern representing the minimal set of atomic operations the composite operation consists of. The computed diff patterns now close the gap between operation specifications and diff models.

3. A posteriori composite operation detection

In this section, we discuss our approach for detecting composite operations by first giving an overview of the entire process and, subsequently, present each phase of the process in detail.

The composite operation detection process is depicted in Fig. 4 in terms of a UML activity diagram. This process has two inputs and consists of three phases. The first input is the diff model called input diff model containing the atomic operations that have been applied. Secondly, our process takes as input an arbitrary number of operation specifications comprising also their explicit diff patterns. In the first phase of the process, the operation specifications' diff patterns are exploited for efficiently preselecting all composite



Input Diff Model Signature

id1 : Add(UML:Class)
id2 : Update(UML:Class, superClass)
id3 : Move(UML:Property)
id4 : Move(UML:Property)
id5 : Delete(UML:Property)
id6 : Update(UML:Class, superClass)
id7 : Update(UML:Class, name)
id8 : Delete(UML:Property)
id9 : Delete(UML:Property)

Diff Element Map

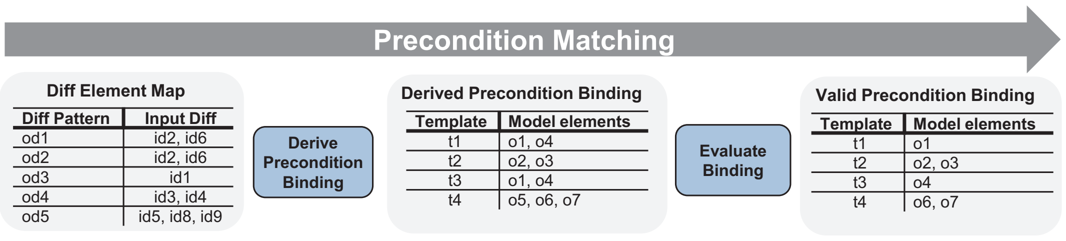
Input Diff	Diff Pattern
id2, id6	od1
id2, id6	od2
id1	od3
id3, id4	od4
id5, id8, id9	od5

Diff Pattern Signature

od1 : Update(UML:Class, superClass)
od2 : Update(UML:Class, superClass)
od3 : Add(UML:Class)
od4 : Move(UML:Property)
od5 : Delete(UML:Property)

(a) Input Diff Model (b) Diff Element Map (c) Diff Pattern of *Extract Superclass*

Fig. 5. Example for preselection.



Diff Element Map

Diff Pattern	Input Diff
od1	id2, id6
od2	id2, id6
od3	id1
od4	id3, id4
od5	id5, id8, id9

Derived Precondition Binding

Template	Model elements
t1	o1, o4
t2	o2, o3
t3	o1, o4
t4	o5, o6, o7

Valid Precondition Binding

Template	Model elements
t1	o1
t2	o2, o3
t3	o4
t4	o6, o7

Fig. 6. Example for precondition matching and evaluation.

operations that potentially have been applied between the two versions of a model. Therefore, the input diff model is searched for occurrences of the diff patterns. As diff patterns abstract from details of the transformation specifications, such as pre- and post-conditions, these parts have to be considered in the subsequent phases. Thus, in the second and third phase, for each potential composite operation occurrence, the pre- and postconditions of the composite operation are evaluated, respectively. If both are valid, an application of a composite operation is at hand and added to the output list of operation occurrences. In the following, the three phases are described in detail by using the running example introduced in Section 2.

3.1. Phase 1: diff pattern matching

This phase checks whether the diff patterns of the given operation specifications are contained by the input diff model.

Diff model preprocessing. In a first step, the input diff model and the diff patterns of all operation specifications are translated into so-called signatures, which are referred to as *input signature* for the diff model and *operation signatures* for the diff patterns. These signatures contain the relevant information of the diff elements in an easily processable format. In particular, the signature represents, similar to method signatures in programming languages, the operation kind and the metamodel types it affects.

Example: If a UML class has been added, the corresponding signature has the form of `Add(UML:Class)` (cf. Fig. 5a). Thus, this signature encodes the type of the atomic operation and the used metamodel type. Please note that in the implementation of our approach, additional information, such as the type of the new parent and sibling diff elements, is encoded in the signature to increase the matching precision. In this paper, we omit this additional information for the sake of readability.

Preselection. The preselection of potentially applied composite operations is accomplished based on the input signature and the operation signatures. The applied procedure for realizing the preselection is depicted in Algorithm 1. This algorithm adds all operation signatures to the preselection (cf. line 1 in Algorithm 1). Subsequently, it iterates through all operation signatures and checks whether each of its elements is contained in the input signature (cf. lines 3 and 4 in Algorithm 1). If at least one element is missing, the respective operation has obviously not been applied entirely and, hence, is removed from the preselection. All operation signatures that remain in the preselection after the algorithm terminates constitute *potential operation occurrences*. After the preselection, a so-called *diff element map* is created for storing the correspondences between the elements of the input diff model and the diff patterns. The preselection together with the diff element map is handed over to the next phase.

Algorithm 1. Preselection based on diff patterns

```
   Input: inputSignature, operationSignatures
   Output: preselection
   // Initially add all operation signatures to the preselection
 1 preselection.addAll(operationSignatures)
 2 for operationSignature ∈ operationSignatures do
 3     for operationSignatureElem ∈ operationSignature do
 4         if ¬ inputSignature.contains(operationSignatureElem) then
               // No complete operationSignature occurrence in inputSignature
 5             preselection.remove(operationSignature)
 6             break
 7         end
 8     end
 9 end
10 return preselection
```

Example: The input diff model and the diff pattern from the operation specification are preprocessed and translated into the signature format (cf. Fig. 5a and c). The preselection (cf. Algorithm 1) is applied to these signatures, reporting one *potential operation occurrence*, and the diff element map is built (cf. Fig. 5b). For instance, the first two diff elements in the diff pattern representing the update of the superclass feature (cf. od1 and od2) match with two diff elements (cf. id2 and id6) in the diff model.

3.2. Phase 2: precondition matching

For each potential operation occurrence, the preconditions are checked.

Derive precondition binding. The evaluation of conditions is costly. Thus, we check the conditions, which are organized in terms of *templates* (cf. Section 2), only for those model elements that have been modified according to the diff pattern. Therefore, we first compute a so-called *derived template binding*. A template binding maps the respective model elements to their corresponding templates. The derived template binding is created based on the diff element map and by exploiting the reference from diff elements to the affected model elements. Therefore, we collect for each diff element in the diff pattern the corresponding precondition template, and for each input diff element the affected element of the origin model. Based on this information, the bindings from elements in the origin model to templates used in the transformation rules are established.

Example: The derived precondition binding in Fig. 6 is computed as follows. The diff element od1 in the diff pattern affects the template t1 (cf. Fig. 3). According to the diff element map, od1 is mapped to the diff elements id2 and id6 from the input diff model. These two diff elements in turn affect the model elements o1:Bike and o4:Car in the origin model, respectively. Thus, a binding of template t1 to these model elements is created. The same procedure is repeated, until all diff elements in the diff pattern have been processed, except diff pattern elements representing additions (e.g., od3). These are only of interest for the postcondition templates as we see later.

Evaluate precondition binding. The derived precondition binding is evaluated using a condition evaluation engine that has to fulfill the following requirements. First, it must be capable of *detecting all valid bindings* among a set of candidates contained in the derived precondition binding, whereas multiple bindings to one template are only allowed if iterations are attached to the respective template. Second, since one operation specification may have been applied multiple times, the evaluation engine must *unfold* the derived precondition binding into separate valid bindings for each single potential application. Third, the engine must be capable of *completing* bindings for templates which have no candidate assigned yet. This is necessary for templates acting as preconditions only, i.e., they do not perform actions in the transformation.

During the evaluation, certain model elements may be rejected as they do not fulfill the conditions and the diff element map is updated accordingly. Mapped diff elements that affect rejected model elements are removed such that the map ultimately contains only diff elements that are relevant in the potential operation occurrence currently under consideration.

Example: The derived precondition binding depicted in Fig. 6 has to be evaluated. At this moment, the classes o1:Car and o4:Bike are bound to templates t1 and t3. During the condition evaluation, however, these multiple bindings can be resolved because the properties o2:speed and o3:horsePwr that are bound to template t2 must be contained by the object bound to template t1; this is only fulfilled by class o1:Car. Thus, o4:Bike is rejected for template t1. The same applies to the multiple bindings of template t3. Moreover, the property o5:fuel is bound to the template t4, as it is a valid model element according to the diff element map. However, the condition evaluation engine rejects this object, because the precondition `t4.name = t2.name` fails as there is no property object for template t2 having the name fuel. The result of this phase is the valid precondition template binding depicted at the right side of Fig. 6. Although there are two templates (t2 and t4) having more than one bound model element, these bindings are valid, because these two templates represent iterations. Finally, the diff element map is cleaned according to the rejected objects leading to the cleaned version depicted in Fig. 7.

3.3. Phase 3: postcondition matching

For each valid precondition binding, we now evaluate the postconditions.

Derive postcondition binding. Apart from the fact that the postcondition binding links postcondition templates and elements in the revised model, the derivation works analogously to the precondition binding derivation. Note that this derivation is based on the *cleaned* diff element map; thus, only diff elements are considered that affect model elements fulfilling the preconditions. As we now deal with postconditions, we have to consider templates representing additions that have been ignored in the previous phase.

Example: Based on the cleaned diff element map, we directly obtain unique bindings in the *derived postcondition binding* for all templates that already existed in the precondition binding (cf. Fig. 7). However, we further have to derive the bindings for the additional template t5, which represents the addition of a model element. Thus, all model elements of the type `Class` that have been added in the revised model are selected. According to the diff model of our example, this applies only to o8:Vehicle.

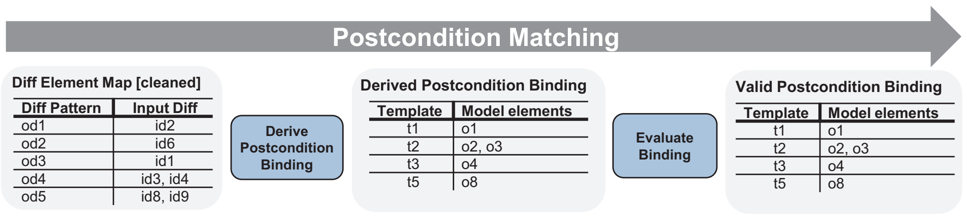
Evaluate postcondition binding. The derived postcondition bindings are evaluated using the condition evaluation engine as described before for the preconditions. If a valid binding is found, a *composite operation occurrence* is added to the diff model.

Example: The shown derived postcondition binding can be validated successfully; thus, an occurrence of *Extract Superclass* is reported.

3.4. Iterative composite operation detection

In several scenarios, multiple composite operations are applied sequentially to overlapping parts of the model which may result in dependencies between the operations themselves. For instance, consider the example depicted in Fig. 8. One developer first applies the composite operation *Specialize Superclass* by changing the superclass of C to B. Subsequently, the same developer performs the composite operation *Pull Attribute* by moving the attribute att from class C to its new superclass B. When considering only the origin model V_o and the revised model V_r, our approach is only capable of detecting the first composite operation *Specialize Superclass* because the preconditions of the second operation *Pull Attribute* are not fulfilled as in V_o the class B is not a superclass of C.

To address this limitation, we apply an iterative composite operation detection. Therefore, we leverage the fact that the operation specifications for detecting applications of composite operations are the same as the specifications for executing them automatically. Consequently, we *apply* all detected composite operations (which are detected between V_o and V_r) to V_o leading to a new origin model denoted with V'_o in Fig. 8 and re-start the operation detection process again to the new scenario $V'_o \rightarrow V_r$. Thereby, we first apply the model differencing algorithm for computing the atomic operations to V'_o and V_r, and subsequently, search again for occurrences of composite operations in the resulting diff model as presented above. This iterative process is repeated ($V''_o \rightarrow V_r$, $V'''_o \rightarrow V_r$, ...) until a fixpoint is reached.



Diff Element Map [cleaned]

Diff Pattern	Input Diff
od1	id2
od2	id6
od3	id1
od4	id3, id4
od5	id8, id9

Derived Postcondition Binding

Template	Model elements
t1	o1
t2	o2, o3
t3	o4
t5	o8

Valid Postcondition Binding

Template	Model elements
t1	o1
t2	o2, o3
t3	o4
t5	o8

Fig. 7. Example for postcondition matching and evaluation.

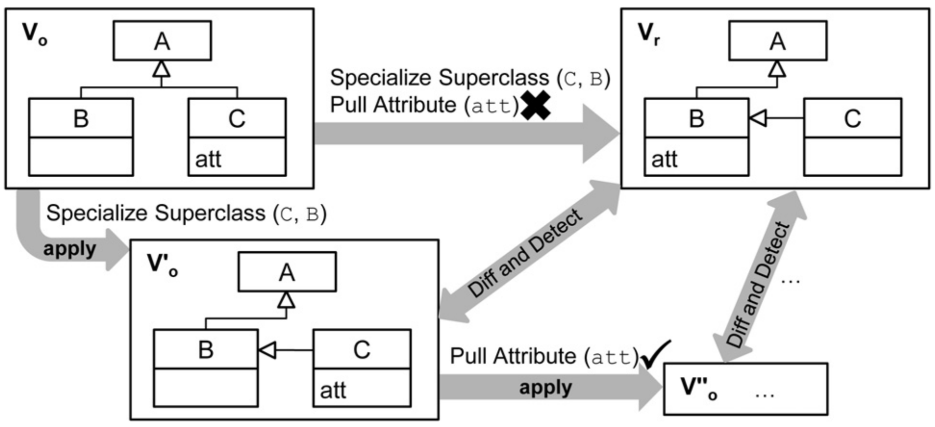


Fig. 8. Iterative composite operation detection.

Coming back to the example shown in Fig. 8, this iterative procedure now enables also the detection of *Pull Attribute* because the preconditions of this composite operation, which restrict class B being a superclass of C, are fulfilled in the new origin model V_o'.

3.5. *Implementation*

We implemented the presented approach based on EMF. Thus, models conforming to any modeling languages defined in Ecore are supported. The state-based model comparison for deriving the atomic operations is realized by an extension of EMF Compare Brun and Pierantonio (2008). In particular, we replaced EMF Compare's heuristic matching component by an own component, which makes use of universally unique identifiers (UUIDs). Based on the UUIDs, also significantly modified and moved model elements can be matched uniquely, which finally leads to a more precise set of detected atomic operations. For the specification and execution of composite operations, we integrated our prototype with our in-place transformation tool EMF Modeling Operations[1] (EMO) (Brosch et al., 2009), which basically implements the concepts of graph transformations. In EMO, the pre- and postconditions are expressed using the Object Constraint Language Object Management Group (2010) (OCL). For finding valid template bindings, an adaptation of Ullmann's graph pattern matching algorithm (Ullmann, 1976) is used. However, in contrast to Ullmann's algorithm, we do not enumerate all potential binding combinations in a tree in advance; we rather apply a recursive backtracking algorithm, which dynamically selects the next model elements to be evaluated. For evaluating whether model elements fulfill the respective conditions, we integrated EMF's implementation of OCL.[2]

In EMO, it is straightforward to generate a diff pattern from an operation specification, because each specification comprises an explicit LHS and RHS model, for which the diff pattern is directly computable using EMF Compare. This is the main reason why we have chosen EMO for our prototypical implementation of the operation detection approach.

4. Case study

We performed a positivist case study (Lee, 1989) based on real-world models and their evolution to evaluate the accuracy of our approach. In particular, following the guidelines for conducting empirical explanatory case studies Runeson and Höst (2009), we applied our approach to detect composite operations that have been performed in the course of the evolution of models coming from a public open source project.

4.1. *Research questions*

The study was performed to quantitatively assess the completeness and correctness of our approach when applied to a real-world scenario. More specifically, we aimed at answering the following research questions:

1. *Operation specifications versus detection rules*. Can operation specifications for executing the operation, in general, also be reused for detecting applications of the respective operation or is any information missing for properly detecting them?
2. *Correctness*. Are the detected operation applications correct in the sense that all reported applications have really been applied? If our approach reports incorrect applications of composite operations, what is the reason for these failures?
3. *Completeness*. Are the detected operation applications complete in the sense that all actually applied composite applications are correctly detected; or does our approach miss to detect applications? If the set of detected operations applications is incomplete, what is the reason for missed applications?

[1] http://www.modelversioning.org/emf-modeling-operations.
[2] http://www.eclipse.org/modeling/mdt/?project=ocl.

4.2. Case study design

Requirements. As an appropriate input to this case study, we require EMF-based models that already have an extensive evolution history. Besides, we do not only need to be equipped with all intermediate versions of these models, we further require the information on the actual composite operations that have been applied in the course of the models' evolution; otherwise, we would not be able to compare the results obtained by our approach with the actual correct set of the composite operation applications. To accomplish an appropriate coverage of different detection scenarios, the evolution of these models should comprise scenarios having a small set of applied atomic operations but also scenarios having a large number of *atomic* operations applied at a time. Moreover, there should be scenarios that comprise only a few applications of *composite* operations, as well as scenarios comprising a higher number composite operations that have been performed at once. Finally, the evolution should comprise a large number of different types of composite operations to avoid distorting the results.

Setup. We chose to analyze the extensive evolution of Ecore metamodels coming from the Graphical Modeling Framework[3] (GMF), an open source project for developing graphical modeling editors. As Ecore metamodels are EMF-based models themselves, they serve as a valid input to assess our approach. In our case study, we considered the evolution from GMF's release 1.0 over 2.0 to release 2.1 covering a period of two years. For achieving a broader data basis, we analyzed the evolution of *three metamodels*, namely the Graphical Definition Metamodel (GMF Graph for short), the Generator Metamodel (GMF Gen for short), and the Mappings Metamodel (GMF Map for short). The respective metamodel versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actually applied composite operations between successive metamodel versions. These two steps have already been done in the course of a case study for evaluating COPE (Herrmannsdoerfer et al., 2009). As a consequence, the manually determined set of composite operations is unbiased in relation to our case study, because the information on applied operations has been independently collected. Moreover, metamodel/model co-evolution, which was actually the purpose of gathering the data in the first place for evaluating COPE, is in fact one major application field of our operation detection approach. Thus, comparing the operation applications obtained by our approach with the gathered data of the original evaluation of COPE constitutes a perfect base for assessing the accuracy for one of the intended use cases.

Additionally, we had to manually specify all composite operations that have been applied across all metamodel versions using EMO. In total, 32 different types of composite operations have been applied; however, we had to create 48 operation specifications, because EMO does not support to specify generic operations using abstract metaclasses, as they are foreseen in COPE. For instance, the operation *Push Feature* has been realized by two distinct operation specifications; one for pushing *attributes* and one for pushing *references*. Having created the operation specifications, we developed a program that automatically performs the operation detection with all revisions of the models and compares the results with the expected results represented in the operation history from the COPE case study. The input data and the results are available online at our project website.[4]

Characteristics of the input data. The evolution of three different models provides a relatively large set of revisions, atomic operation applications, and composite operation applications. In total, the evolution of the considered models comprises 45 revisions that involved at least one composite operation; we omitted revisions, to which only atomic operations have been applied. Overall, in the course of the models' evolution, 141 composite operations and 342 atomic operations have been applied, whereas one transition between two revisions (called *commit* hereafter) contains on average five atomic operations and two composite operations. Thereby, we encountered several different types of commits. As depicted in Fig. 9, most of the commits comprise between 1 and 14 atomic operations and between one and eight composite operations. Nevertheless, the evolution also includes commits having between 15 and 35 atomic operations and one commit, which comprises even 52 atomic and 26 composite operations. Please note that the model elements in the investigated models exhibit universally unique identifiers (UUIDs). In the model comparison phase for obtaining the atomic operations, these UUIDs help significantly to find corresponding model elements more precisely across the successive model versions, in comparison to the use of similarity-based matching heuristics. We decided to use UUIDs in this case study to avoid affecting the results of this study by the selection of specific similarity-based matching heuristics (Kolovos et al., 2009).

Measures. To assess the accuracy of our approach, we compute the measures *precision* and *recall* (Olson and Delen, 2008) originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, precision denotes the fraction of *correctly detected* composite operations among the set of *all detected* operations (i.e., how many detected operations are in fact correct). Recall indicates the fraction of *correctly detected* composite operations among the set of *all actually applied* composite operations (i.e., how many operations have not been missed). These two measures may be thought of as probabilities: the precision denotes the probability that a detected operation is correct and the recall is the probability that an actually applied operation is detected. Thus, both values may range from 0 to 1, whereas a higher value is better than a lower one. The precision and recall may be further combined into the so-called *f-measure* in terms of a harmonic mean ($F = 2 \cdot (\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$).

4.3. Results

The results of our case study are depicted in Table 1. In the upper area, we show the results grouped by the three considered models. In the lower part, the results are grouped by type of composite operation. Overall, using our iterative operation detection approach, we were able to correctly detect 99 composite operations among all 141 composite operations (i.e., around 70%), whereas only two composite operations have been incorrectly detected, which leads to a precision of around 0.98. It is worth noting that the evolution history of these three models is very different. GMF Graph was extensively modified within only *one large revision* comprising 52 atomic operations and 26 composite operations, which lead to a quite low recall of 0.5 (i.e., only 13 of the 26 composite operations could be detected). On the contrary, GMF Gen was subjected to 40 revisions, some comprised a large number of atomic operations and some only a low number. Thus, the evolution of this model is a very representative mixture of different scenarios for the detection of composite operations leading to a precision of 0.98 and a recall of 0.73. The evolution of the third model under consideration, GMF Map, contained four revisions and in the course of each revision at maximum three composite operations have been applied. Using our approach, we could identify *all* applied composite operations correctly.

Looking at the results grouped by the type of composite operation, we can see that the two most occurring operation types, *Rename* and *Delete Feature*, have largely been detected correctly.

[3] http://www.eclipse.org/modeling/gmf.

[4] http://www.modelversioning.org/index.php?option=com_content&view=article&id=64.

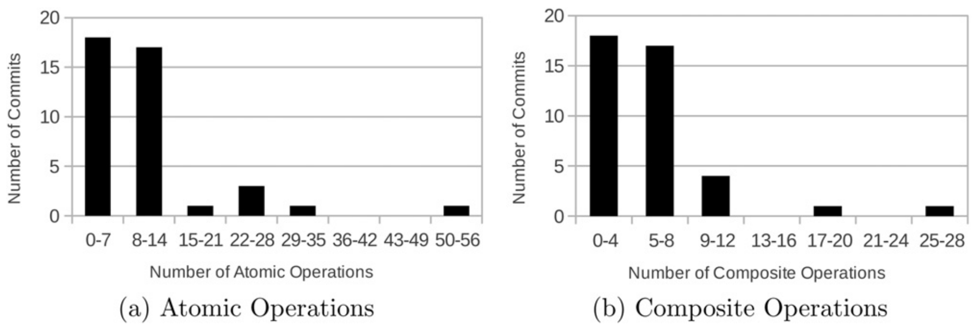


Fig. 9. Distribution of operations per commit.

These composite operation types are, however, comparatively small in terms of comprised atomic operations; thus, the detection is easier. Nevertheless, also for the larger composite operation types such as *Extract Superclass* or *Specialize Supertype*, we achieved good results. However, there are several composite operation types, whether they are small or large, which our approach could not detect at all (e.g., *Specialize Reference Type* and *Pull Feature*).

4.4. Interpretation of the results

Research question 1. The overall *f*-measure of 0.82 across all considered models and commits is very promising. As the operation specifications used in this study have been created using EMO just as we would create them for *executing* them, we may already answer the first research question and conclude that, in general, it is possible to reuse the same operation specifications also for *detecting* them *a posteriori*.

Research question 2. Especially, the *precision* obtained by our approach is de facto perfect. Nearly all detected composite operation applications are correct. The reason for the lost 2% in precision in GMF Gen is actually not because the two indicated occurrences of the operation Delete Feature are incorrect. In fact, the reason is that the composite operation Flatten Hierarchy has not been detected and in the course of this operation, two features (containment references) have been deleted. Thus, not detecting the larger composite

Table 1
Results of the case study.

	# expected	# correct	# wrong	Precision	Recall	*f*-measure
Case study						
GMF Graph	26	13	0	1.00	0.50	0.67
GMF Gen	107	78	2	0.98	0.73	0.84
GMF Map	8	8	0	1.00	1.00	1.00
Overall	141	99	2	0.98	0.70	0.82
Composite operation						
Collect Feature	4	0	0	0.00	0.00	0.00
Combine Feature	1	0	0	0.00	0.00	0.00
Delete Feature	18	17	2	0.89	0.94	0.91
Drop Opposite	1	0	0	0.00	0.00	0.00
Extract and Group Attribute	1	0	0	0.00	0.00	0.00
Extract Subclass	1	1	0	1.00	1.00	1.00
Extract Superclass	9	9	0	1.00	1.00	1.00
Flatten Hierarchy	1	0	0	0.00	0.00	0.00
Generalize Attribute	1	0	0	0.00	0.00	0.00
Generalize Reference	6	4	0	1.00	0.67	0.80
Imitate Supertype	1	0	0	0.00	0.00	0.00
Inline Superclass	3	2	0	1.00	0.67	0.80
Make Abstract	1	0	0	0.00	0.00	0.00
Make Containment	1	1	0	1.00	1.00	1.00
Make Feature Volatile	6	6	0	1.00	1.00	1.00
Move Feature	3	0	0	0.00	0.00	0.00
New Opposite Reference	14	9	0	1.00	0.64	0.78
Operation to Volatile	3	2	0	1.00	0.67	0.80
Propagate Feature	1	1	0	1.00	1.00	1.00
Pull Feature	3	0	0	0.00	0.00	0.00
Pull Operation	3	2	0	1.00	0.67	0.80
Push Feature	7	2	0	1.00	0.29	0.45
Push Operation	1	0	0	0.00	0.00	0.00
Remove Supertype	1	1	0	1.00	1.00	1.00
Rename	29	27	0	1.00	0.93	0.96
Replace Class	2	2	0	1.00	1.00	1.00
Replace Enum	4	4	0	1.00	1.00	1.00
Replace Inheritance	3	2	0	1.00	0.67	0.80
Replace Literal	1	1	0	1.00	1.00	1.00
Specialize Reference Type	4	0	0	0.00	0.00	0.00
Specialize Supertype	6	5	0	1.00	0.83	0.91
Volatile To Opposite	1	1	0	1.00	1.00	1.00
Overall	141	99	2	0.98	0.70	0.82

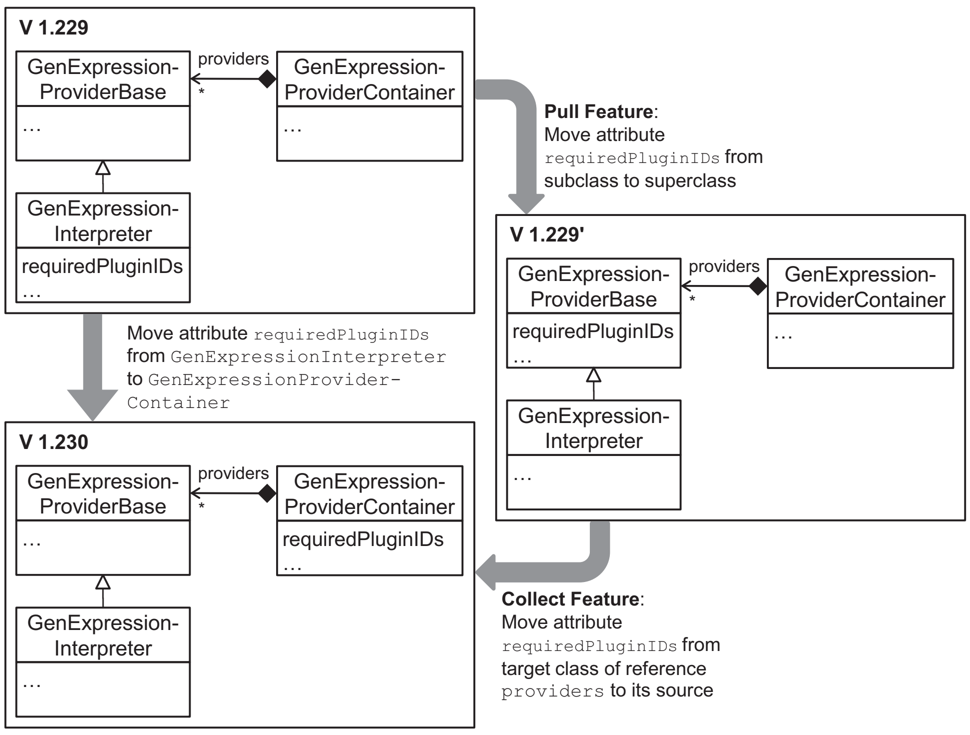

V 1.229
GenExpression-ProviderBase
providers
*
GenExpression-ProviderContainer
…
GenExpression-Interpreter
requiredPluginIDs
Pull Feature: Move attribute requiredPluginIDs from subclass to superclass
V 1.229'
Move attribute requiredPluginIDs from GenExpressionInterpreter to GenExpressionProvider-Container
V 1.230
Collect Feature: Move attribute requiredPluginIDs from target class of reference providers to its source

Fig. 10. Overlapping sequence of composite operations (from GMF Gen).

operation caused the "incorrect" detection of two smaller operations that are part of the missed larger operation.

Research question 3. Although the precision is very satisfying, the recall values are rather low for some commits and operation types. For investigating the causes for these low recall values, we analyzed the missed operation applications in more detail. Our first guess that the low recall values are caused by the complexity and size of the respective composite operation type could not be verified by analyzing the resulting data. Several large composite operations having complex pre- and postconditions could be detected without any issues and the data representing the size of the operation types does not seem to correlate with the recall values of their detection. Admittedly, the sample size is quite low for arguing the statistical independence of these two variables though because there are some types of operations for which we only have one or two expected but no detected applications.

After a more detailed analysis of the specific cases, in which expected applications could not be detected, we identified the actual cause for the low recall values: *overlapping sequences* of composite operation applications. More specifically, the impact of overlapping sequences of composite operations on the operation detection is twofold. First, one *preceding operation* may *enable* the applicability of a *succeeding operation* by manipulating the model such that the *preconditions* of the succeeding operation are fulfilled which was not the case before applying the preceding operation. Second, one succeeding operation may *hide* a preceding operation by invalidating the *postconditions* of the latter. The first case is addressed using our iterative detection approach. This works, however, only if at least the preceding composite operation can be detected between the original and the revised version; otherwise, the preceding operation is unknown and, thus, no intermediate model state can be computed. Unfortunately, in many scenarios it is not possible to detect the preceding application, because of the second case.

Consider an example from the GMF Gen model evolution, a subset of the operations applied between revision 1.229 and 1.230 (cf. Fig. 10). In this revision, the developer first applied a Pull Feature by shifting the attribute requiredPluginIDs from GenExpressionInterpreter to its superclass GenExpressionProviderBase. The resulting intermediate state is shown in V 1.229' in Fig. 10. Subsequently, the developer applied the composite operation Collect Feature by again shifting the same attribute over the reference provides from this reference's target to its source class GenExpressionProviderContainer. However, when only considering the state of the model depicted in V 1.229 and V 1.230, as done by our approach, only one atomic operation can be detected, which is the move of attribute requiredPluginIDs from GenExpressionInterpreter to GenExpressionProviderContainer. As a consequence, neither the preconditions of Collect Feature nor the postconditions of Pull Feature match the origin state in V 1.229 and the revised state in V 1.230, respectively.

The correlation between the number of applied (i.e., expected) composite operations and the recall value can also be statistically shown based on the data gathered in our study. More precisely, we computed the *relative number of composite operation applications* of each commit; that is, the number of expected composite operations in one commit divided by the number of model elements in the respective model, and compared it to the achieved recall values for the corresponding commit. Although the sample size is relatively small, we obtained a Pearson's correlation (Rodgers and Nicewander, 1988) of around −0.67 between these two variables. Our interpretation of this correlation is as follows: the more composite operations have been applied within one commit, the more likely it is that composite operations are sequentially overlapping with the consequence that the overlapping composite operations cannot be detected. This, as a result, leads to a lower recall value. The relationship between the number of applied composite operations to the recall value is depicted in Fig. 11. Please note that, for the sake of readability, we grouped equal numbers of composite

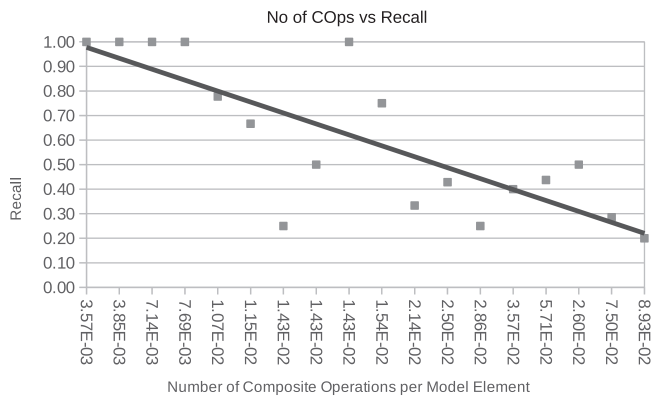


Fig. 11. Recall versus number of applied composite operations.

operations per model element and averaged the respective recall values within one group in this graph. The black solid line depicts the linear regression (i.e., the estimated trend), whereas the coefficient of determination R^2 is around 0.62.

4.5. Threats to validity

As we investigated real-world models and their evolutions, considered a quite large number different types of composite operations, and compared the achieved results of our algorithm with the expected results that have been obtained manually for a different purpose, we believe that the computed precision and recall values constitute a valid basis for drawing the above conclusions. One potential threat is, however, that we considered only one modeling language (Ecore). The precision and recall might be affected when applying the presented algorithm to models that conform to different modeling languages, although we do not see any specific reason for that, because our algorithm makes no assumptions about the modeling language as it processes models and evolutions generically.

5. Performance evaluation

We also explored the scalability and performance of our approach. In particular, we investigated the effects of an increasing model size and an increasing number of concurrently applied atomic operations on the runtime of our algorithm. Therefore, we conducted an experiment based on purposive synthetic scenarios, which have been created using our tool called Ecore Mutator.[5] Please note that we separated the performance evaluation from the case study presented in Section 4 to have more control about the characteristics of the investigated scenarios to isolate the effects of these characteristics on the required runtime.

5.1. Setup

For assessing the scalability of our approach, we measured the required runtime to detect composite operations in four detection scenarios, which contain (i) five, (ii) two, (iii) one, and (iv) zero composite operation occurrences. For these scenarios, we measured the steady state performance[6] of our implementation while, on the one hand, stepwise increasing the size of the evolving models and, on the other hand, stepwise increasing the number of concurrently applied atomic operations. Thereby, we isolate the effects on the runtime when the size of the model increases or the number of concurrently applied atomic operations increases. The experiment was conducted using an Intel® Core™2 Duo with 2.53 GHz running Ubuntu 11.04. The input data and the results are available online at our project website.[7]

5.2. Results

Increasing model size. The results of our experiment for increasing the model size are illustrated in the left-hand side graph in Fig. 12. In this graph, we depicted the overall runtime required for the four detection scenarios, as well as the runtime required for comparing the models (dashed gray line). When stepwise increasing the model size for all four detection scenarios from 127 to 2117 model elements, whereas keeping the number of concurrently applied atomic operations constantly at around 60, the increase in runtime is largely equal to the increase of the runtime needed for the model comparison. Especially in the scenario, in which *no composite operation* has been applied, the overall runtime for every evaluated model size is constantly around 100 milliseconds (ms) higher than the time required for only obtaining the atomic operations. However, for detecting *five applications* of composite operations, the runtime of our approach grew from 543 ms for the smallest model to 745 ms for the largest model.

Increasing number of atomic operations. The results of our experiment for increasing the number of concurrently applied atomic operations are depicted in the right graph in Fig. 12. The time required for comparing the models is represented by the dashed gray line as a reference. In this experiment, we used a rather large model consisting of 2117 model elements as a basis. The range of concurrently performed atomic operations has been increased stepwise from 118 to 459 applied operations. To determine correctly that *no composite operation* has been applied among 118 atomic operations, the algorithm needed only 550 ms. More precisely, the composite operation detection took additional 320 ms to the time needed for the detecting of atomic operations. However, when 459 atomic operations have been concurrently applied, the runtime was 2688 ms, which are additional 1411 ms in comparison to the time required for obtaining the atomic operations. A similar increase of runtime was measured for the other scenarios. The required runtime increased to 1269 ms for finding *five composite operations* among 118 atomic operations and 4101 ms for detecting the same composite operations among 459 atomic operations.

We also measured the share of the runtime that each phase accounts for. The model comparison causes on average 34.54% of the runtime, whereas the preselection accounts for 41.69% and the condition evaluation had a share of 23.77%. However, please note that these shares vary strongly because of the different characteristics of the respective detection scenario. For instance, in scenarios having a low number of atomic and composite operations, the share of the preselection time is much lower.

5.3. Interpretation

What we can learn from this experiment is that the runtime of our approach only slightly depends on the model size but grows overproportionally with increasing numbers of atomic operations.

The growth of the overall runtime with increasing model size is mainly caused by the model comparison, which certainly depends on the model size. The additional time required for detecting composite operations is because of the condition evaluation potentially has to evaluate more model elements, if the considered model contains more elements. Nevertheless, thanks to the preselection

[5] http://code.google.com/a/eclipselabs.org/p/ecore-mutator.
[6] A program is run repeatedly until the execution time of each run stabilizes.
[7] http://www.modelversioning.org/index.php?option=com_content&view=article&id=64.

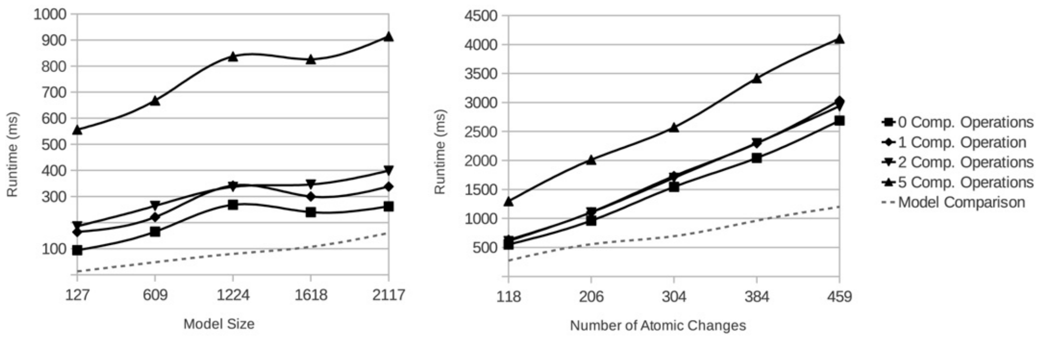


Fig. 12. Operation detection runtime.

phase based, the model elements to be evaluated in the condition evaluation are kept at a minimum: only those model elements are checked that have been affected by atomic operations. The positive effect of the preselection phase can be observed when comparing the scenario depicted in the left graph of Fig. 12 at 1224 model elements on the *x*-axis with the other scenarios in this experiment (127 elements, 609 elements, 1618 elements, and 2117 elements). We applied atomic operations randomly alongside the composite operations in all scenarios. In the scenario with 1224 model elements, however, we "accidentally" applied a diff pattern that is similar to a composite operation specification. Thus, the preselection phase reported a potential occurrence and the algorithm proceeded with the evaluation of the preconditions before the potential occurrence could be finally discarded based on the failing preconditions. As a result, we observe an additional increase of the runtime with 1224 model elements in comparison to the other scenarios.

Our approach significantly depends on the number of concurrently applied atomic operations. The more atomic operations have been applied, the larger is the search space to be examined for finding diff patterns. Furthermore, having a large number of atomic operations, it is more likely to encounter a matching diff pattern, which forces the algorithm to perform the evaluation of the pre- and postconditions.

In summary, we argue that the runtime of our approach is satisfying. When considering the potential fields of applications of our approach, which are, among others, mining of model repositories and model versioning, we face diverging performance requirements. Mining of model repositories may pose a very large number of atomic and composite operations potentially causing longer runtimes. For instance, the time required for processing the 45 revisions and 342 atomic changes in the GMF case study (cf. Section 4) was around 2 min. Luckily, runtime is usually not a crucial criterion in such scenarios. On the contrary, in model versioning, a fast execution time has high priority, as it would cause developers to wait while they check in their models. Fortunately, the number of operations applied between two successively modified revisions of one model is rather small: on average, one commit had five atomic and two composite operations in the GMF case study.

5.4. Threats to validity

In this experiment, we investigated the impact of the models' size and of the number of applied changes on the performance of our algorithm. As we used generated synthetic models and random atomic operations, the validity of the experiment's results might be affected by the characteristics of the synthetic models and the random selection of atomic operations, as well as how they relate to the five composite operations that have been applied. In particular, if we had used composite operations that only transform model elements of certain types that are not instantiated often, the algorithm would perform much better. The same is true for the type of atomic operations: if the random atomic operations are not involved in the used composite operations, the chance for finding diff patterns is very low. To counteract this threat, we generated models exhibiting a constant ratio of model element types that are involved in at least one composite operation to those that are not involved in a composite operation. Throughout all models in this experiment, this ratio is constantly around 5:2, which means that two out of five model element could potentially be involved in a composite operation application. Moreover, we ensured that the randomly applied atomic operation types are normally distributed across all applied atomic operations.

6. Related work

Several approaches recently emerged to detect composite operation applications in different technical spaces. Most of them are designed for detecting refactorings in object-oriented (OO) programs, but there are also some dedicated approaches for high-level specifications such as models and ontologies.

OO programming. The easiest way to capture applied refactorings is to track their execution in the development environment. Such approaches are often referred to as *operation-based* versioning/merging (Lippe and Oosterom, 1992). Refactoring tracking is realized by Dig et al. (2008), Ekman and Asklund (2004) and Robbes (2007). All these approaches highly depend on the used development environment. Furthermore, manually performed refactorings are not detectable and refactorings which have been made obsolete by successive changes might be wrongly indicated. *State-based* refactoring detection mechanisms aim to reveal refactorings a posteriori. For instance, Dig et al. (2006) propose an approach to detect applied refactorings in Java code. They first perform a fast syntactic analysis followed by a semantical analysis. A similar approach is followed by Weissgerber and Diehl (2006). After preprocessing and syntactical analysis have been conducted, conditions indicating the application of a refactoring are evaluated. A heuristic-based approach is presented in Demeyer et al. (2000) in which a combination of various software measures as indicator for a certain refactoring is used. For instance, a decrease in a method's size, among other measures, is used to indicate that the refactoring *Split Method* has been applied.

Refactoring detection in code artifacts is in general more challenging than in model artifacts. In models, relationships between the model elements are usually explicitly available in form of direct references represented by an address or an ID. In code, elements usually have no persistent address or ID and, therefore, have to be matched using name- and content-similarity measures. On the other hand, when detecting model refactorings, we face a multitude of different modeling languages. Consequently, there is a plethora of different refactorings and diverging implementations of the same refactoring. Hence, hard-coded detection strategies for a pre-defined set of refactorings are not a desirable solution.

Model engineering. To the best of our knowledge, three approaches exist for detecting composite operations in evolving software models a posteriori.

First, there is the approach by Xing and Stroulia (2006) for detecting refactorings in evolving software models which is integrated in *UMLDiff*. Refactorings are expressed by change pattern queries used to search a diff model obtained by a state-based model comparison. Although, the general goal of UMLDiff is comparable to ours, there are several major differences. First, UMLDiff is tailored to a fixed modeling language, namely a subset of structural UML diagrams while our approach is applicable for any modeling language. Second, to add further composite operations, users of UMLDiff have to develop new queries which is an additional burden compared to simply reusing existing specifications available for executing the composite operations. Here a major benefit of our approach is also the guaranteed consistency between executing composite operations and reporting their applications, since only one specification exists which is used for both purposes. Third, UMLDiff only allows to query the diff model, but no complex pre- and post-conditions for the original and revised models are regarded. Finally, UMLDiff lacks support for iterative composite operation detection due to the fact that only detection rules are used, but no execution rules are available to produce the intermediate model versions.

Second, the approach by Vermolen et al. (2011) copes with the detection of complex evolution steps between different versions of a metamodel. They use a diff model comprising primitive operations as input and calculate on these basis complex operations. The approach is tailored to the core of current metamodeling languages, but follows a similar methodology as UMLDiff. However, a specific feature is the detection of masked operations, i.e., operations hidden by other operations, by defining additional detection rules. Nevertheless, the approach is again dedicated to one single modeling language and does not allow to reuse the operation specifications used for execution for the detection process.

Third, Küster et al. (2008) calculate hierarchical change logs for process models. The authors apply the concept of Single-Entry–Single-Exit fragments to calculate the hierarchical change logs after computing the correspondences between two process models. Thereby, several atomic changes are hidden behind one compound change. The difference to our work is twofold. First, we consider the detection of composite operations comprising changes cross-cutting the whole model, i.e., we are not restricted to reason only about one hierarchy branch, and second, our approach is language independent, thus we are not restricted to process models.

Ontology engineering. There is widely related work in the field of ontology engineering. Hartung et al. (2010) present an approach for generating so called semantically enriched evolution mappings between two versions of an ontology. Evolution mappings can be seen as diff models which comprise atomic as well as composite operations. Their goal is to produce a minimal diff model by using a rule-based system for minimizing the atomic operations by aggregating them to composite operations. The approach is tailored to an ontology language as well as to a small set of composite operations such as moving, splitting, and merging concepts by providing specific detection rules. Finally, they apply aggregation functions to further shrink down the size of the diff model by combining composite operations, which is in our approach supported by using iterations in the transformation rules.

In summary, the presented approach of this paper is the first generic solution that allows the reuse of specifications for execution composite operations also for detecting applications of them. All other approaches are either based on operation tracking or they are restricted to a dedicated language and pre-defined composite operations, which have to be re-formulated as detection rules.

7. Conclusion

In this paper, we introduced a third phase for model comparison approaches for aggregating atomic operations to composite operations. Our approach is language independent and allows to reuse existing composite operation specifications used for execution purposes. We support an iterative detection process to find a form of overlapping operation sequences. The feasibility of our approach has been demonstrated in terms of an EMF-based implementation, which supports Ecore-based modeling languages and composite operation specifications developed with EMO. To evaluate the accuracy of our implementation, we conducted a real-world case study. Thereby, we showed that all detected composite operation applications are correct and that approximately 70% of all expected applications could be found.

Possible extensions. Although, the general performance of the approach showed a high potential, some of the investigated scenarios indicated potential limitations. In the following, we discuss these limitations and sketch some possible extensions to address them.

Fuzzy diff pattern matching. When state-based model comparison is applied, only the *effective atomic operations* are obtained. For instance, if a model element has been updated and subsequently deleted, only the deletion of the model element is detected. However, there might exist scenarios, in which a composite operation has been performed, although succeeding atomic operations hide essential elements of a composite operation's diff pattern. To enable our approach to support such scenarios, we plan to consider atomic operation kinds that potentially hide other operations as "joker". For instance, a deletion of a model element can be considered to be an update and a move of the same model element in the diff pattern checking phase. If such a strategy is applied, the recall is increased by lowering the precision.

Partial condition evaluation. In our approach, a composite operation occurrence is reported if, and only if, *all pre- and postconditions are valid*. With this strategy, a very high recall is obtained as can be seen in the case study. However, in some scenarios, it might happen that atomic operations have to be performed first to obtain valid preconditions before a composite operation can be applied. Analogously, it is possible that succeeding operations lead to failing postconditions of preceding composite operations. Our condition evaluation engine supports loosening the strictness of conditions in a way that a *partial* condition validity (to a certain threshold) can be proven. Of course, this strategy would increase the number of detected composite operations, but also leads to imprecision.

Composite composite operations. Currently, our approach only uses one level for aggregating operations by combining atomic changes into composite ones. However, this may be extended to allow for aggregating composite operations into larger composite operations and so on. One promising way seems to be the precalculation of potentially combinable composite operation specifications on the basis of their pre- and postconditions. For instance, if the preconditions of composite operation A fit to postconditions of composite operation *B*, *A* potentially might be executed after *B*. If a valid combination is revealed, both operation specifications can be automatically merged to create a new "composite composite operation". For limiting the search space of combinable composite operations, we may use the critical pair analysis comparable to how it has been done in Mens (2006).

Acknowledgements

This work has been partially funded by the Austrian Science Fund (FWF) under grant J 3159-N23 and by the Vienna Science and Technology Fund (WWTF) through project ICT10-018.

References

Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.,2010. Henshin: advanced concepts and tools for in-place EMF model transformations. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'10), vol. 6395 of LNCS. Springer, pp. 121–135.

Bézivin, J., Gerbé, O.,2001. Towards a precise definition of the OMG/MDA framework. In: Proceedings of the International Conference on Automated Software Engineering (ASE'01). IEEE, pp. 273–280.

Bézivin, J., 2005. On the unification power of models. Software and Systems Modeling 4 (2), 171–188.

Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.,2009. An example is worth a thousand words: composite operation modeling by-example. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'09), vol. 5795 of LNCS. Springer, pp. 271–285.

Brosch, P., Kappel, G., Seidl, M., Wieland, K., Wimmer, M., Kargl, H., Langer, P., 2010. Adaptable model versioning in action. In: Tagungsband der Modellierung 2010, vol. 161 of LNI, GI, pp. 221–236.

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

Cornélio, M., Cavalcanti, A., Sampaio, A., 2010. Sound refactorings. Science of Computer Programming 75 (3), 106–133.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Systems Journal 45 (3), 621–646.

Demeyer, S., Ducasse, S., Nierstrasz, O., 2000. Finding refactorings via change metrics. In: Proceedings of the Conference on Object-oriented Programming, Systems, Languages and Applications (OOPSLA'00), ACM, pp. 166–177.

Dig, D., Comertoglu, C., Marinov, D., Johnson, R.,2006. Automated detection of refactorings in evolving components. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06), vol. 4067 of LNCS. Springer, pp. 404–428.

Dig, D., Manzoor, K., Johnson, R.E., Nguyen, T.N., 2008. Effective software merging in the presence of object-oriented refactorings. IEEE Transactions on Software Engineering 34 (3), 321–335.

Ekman, T., Asklund, U., 2004. Refactoring-aware versioning in eclipse. Electronic Notes in Theoretical Computer Science 107, 57–69.

Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J., 2007. Domain-specific Modeling. In: Handbook of Dynamic System Modeling. CRC Press.

Hartung, M., Gross, A., Rahm, E., 2010. Rule-based generation of diff evolution mappings between ontology versions. Computing Research Repository (CoRR) abs/1010.0122.

Heckel, R., 2006. Graph transformation in a nutshell. Electronic Notes in Theoretical Computer Science 148 (1), 187–198.

Herrmannsdoerfer, M., Kögel, M., 2010. Towards a generic operation recorder for model evolution. In: Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS'10, ACM.

Herrmannsdoerfer, M., Benz, S., Juergens, E.,2009. COPE – automating coupled evolution of metamodels and models. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'09), vol. 5653 of LNCS. Springer, pp. 52–76.

Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.,2009. Language evolution in practice: the history of GMF. In: Proceedings of the International Conference on Software Language Engineering (SLE'09), vol. 5969 of LNCS. Springer.

Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.,2009. Explicit transformation modeling. In: Models in Software Engineering, vol. 6002 of LNCS. Springer, pp. 240–255.

Küster, J.M., Gerth, C., Förster, A., Engels, G.,2008. Detecting and resolving process model differences in the absence of a change log. In: Proceedings of the International Conference on Business Process Management (BPM'08). LNCS, Springer, pp. 244–260.

Kelter, U., Wehren, J., Niere, J.,2005. A generic difference algorithm for UML models. In: Software Engineering. LNI, GI, pp. 105–116.

Kniesel, G., Koch, H., 2004. Static composition of refactorings. Science of Computer Programming 52, 9–51.

Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J., Bruegge, B., 2010. Merging model refactorings – an empirical study. In: Proceedings of the Workshop on Model Evolution @ MoDELS'10.

Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., Joern, D.,2010. Comparing state- and operation-based change tracking on models. In: Proceedings of the Enterprise Distributed Object Computing Conference (EDOC'10). IEEE Computer Society, pp. 163–172.

Kolovos, D., Di Ruscio, D., Pierantonio, A., Paige, R., 2009. Different models for model matching: an analysis of approaches to support model differencing. In: Proceedings of the International Workshop on Comparison and Versioning of Software Models @ ICSE'09, IEEE.

Kolovos, D.,2009. Establishing correspondences between models with the epsilon comparison language. In: Proceedings of the International Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'09), vol. 5562 of LNCS. Springer, pp. 146–157.

Lee, A., 1989. A scientific methodology for MIS case studies. MIS Quarterly, 33–50.

Lin, Y., Gray, J., Jouault, F., 2007. DSMDiff: a differentiation tool for domain-specific models. European Journal of Information Systems 16 (4), 349–361.

Lippe, E., Oosterom, N.V.,1992. Operation-based merging. In: Proceedings of the 5th Symposium on Software Development Environments. ACM, pp. 78–87.

Mens, T., 2002. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering 28 (5), 449–462.

Mens, T.,2006. On the use of graph transformations for model refactoring. In: Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05), vol. 4143 of LNCS. Springer, pp. 219–257.

Mens, T., 2008. Introduction and roadmap: history and challenges of software Evolution. In: Software Evolution. Springer, Berlin Heidelberg, pp. 1–11.

Meyers, B., Vangheluwe, H., 2011. A framework for evolution of modelling languages. Science of Computer Programming 76 (12), 1223–1246.

Murphy-Hill, E., Parnin, C., Black, A.,2009. How we refactor and how we know it. In: Proceedings of the International Conference on Software Engineering (ICSE'09). IEEE, pp. 287–297.

Object Management Group, Model-driven Architecture (MDA), http://www.omg.org/mda/specs.htm).

Object Management Group (OMG), 2010. Object Constraint Language (OCL), Version 2.2, http://www.omg.org/spec/OCL/2.2

Olson, D., Delen, D., 2008. Advanced Data Mining Techniques. Springer.

Opdyke, W.F., 1992. Refactoring object-oriented frameworks. Ph.D. thesis. University of Illinois at Urbana-Champaign.

Robbes, R., 2007. Mining a change-based software repository. In: in: Proceedings of the Workshop on Mining Software Repositories (MSR'07), IEEE Computer Society, pp. 15–23.

Rodgers, J., Nicewander, W., 1988. Thirteen ways to look at the correlation coefficient. The American Statistician 42 (1), 59–66.

Ruhroth, T., Wehrheim, H., 2012. Model evolution and refinement. Science of Computer Programming 77 (3), 270–289.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14 (2), 131–164.

Schmidt, M., Gloetzner, T.,2008. Constructing Difference tools for models using the SiDiff framework. In: Companion of the International Conference on Software Engineering. ACM, pp. 947–948.

Sen, S., Baudry, B., Vangheluwe, H., 2010. Towards domain-specific model editors with automatic model completion. Simulation 86 (2), 109–126.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2008. Eclipse Modeling Framework 2.0. Addison-Wesley Professional, Boston, Massachusetts.

Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.,2001. Refactoring UML models. In: Proceedings of the International Conference on the Unified Modeling Language (UML'01), vol. 2185 of LNCS. Springer, pp. 134–148.

Ullmann, J., 1976. An algorithm for subgraph isomorphism. Journal of the ACM 23 (1), 31–42.

Vermolen, S., Wachsmuth, G., Visser, E., 2011. Reconstructing complex metamodel evolution. Tech. Rep. TUD-SERG-2011-026, Delft University of Technology.

Weissgerber, P., Diehl, S., 2006. Identifying refactorings from source-code changes. In: Proceedings of the International Conference on Automated Software Engineering (ASE'06), IEEE, pp. 231–240.

Xing, Z., Stroulia, E.,2005. UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the International Conference on Automated Software Engineering (ASE'05). ACM, pp. 54–65.

Xing, Z., Stroulia, E.,2006. Refactoring detection based on UMLDiff change-facts queries. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06). IEEE, pp. 263–274.

Philip Langer is a postdoctoral researcher in the Business Informatics Group at the Vienna University of Technology. His research is focused on model evolution, model transformations, and model execution in the context of model-driven engineering. For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/planger or contact him at langer@big.tuwien.ac.at.

Manuel Wimmer is a postdoctoral researcher in the Business Informatics Group at the Vienna University of Technology. Currently, he is on leave as research associate at the University of Málaga. His research interests comprise Web engineering, model-driven engineering, and model management. For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/mwimmer or contact him at wimmer@big.tuwien.ac.at.

Petra Brosch is a postdoctoral researcher at the Business Informatics Group of the Vienna University of Technology. Her research interests include various topics in the area of model-driven engineering, especially model evolution, model versioning, and model transformation. In her PhD thesis, she worked on conflict resolution in model versioning with special emphasis on enabling merge support directly in the concrete syntax of models and recommending conflict resolution patterns.

Markus Herrmannsdoerfer works as a research assistant at the Software & Systems Engineering group of Prof. M. Broy at Technische Universität München. His academic interests include model-driven engineering, language engineering and language evolution. He received a PhD degree from Technische Universität München under the supervision of Prof. M. Broy, working on evolutionary metamodeling.

Martina Seidl holds a PhD in computer science and works at the Business Informatics Group of the Vienna University of Technology and the Institute for Formal Models and Verification of the Johannes Kepler University Linz. Her research interests include various topics from the area of model evolution and model versioning,

automated reasoning with special focus on the evaluation of quantified Boolean formulas as well as software verification.

Konrad Wieland is technical consultant at LieberLieber GmbH. He studied Business Informatics at the Vienna University of Technology from 2003 to 2009. From 2009 to 2011, he worked as research assistant at the Business Informatics Group (Vienna University of Technology) in the area of model versioning and collaborative conflict resolution. He received the PhD degree from the Vienna University of Technology in 2011.

Gerti Kappel is a full professor at the Institute of Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Until 2001, she was a full professor of computer science and head of the Department of Information Systems at the Johannes Kepler University of Linz. She received the Ms and PhD degrees in computer science and business informatics from the University of Vienna and the Vienna University of Technology in 1984 and 1987, respectively. Her current research interests include model engineering, Web engineering, as well as process engineering.