Delft University of Technology
Software Engineering Research Group
Technical Report Series

Reconstructing Complex Metamodel Evolution

Sander D. Vermolen, Guido Wachsmuth, Eelco Visser

Report TUD-SERG-2011-026

TUD-SERG-2011-026

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 4th International Conference on Software Language Engineering (SLE 2011), Lecture Notes in Computer Science, Springer, 2011.

Reconstructing Complex Metamodel Evolution

Sander D. Vermolen, Guido Wachsmuth, Eelco Visser

Software Engineering Research Group, Delft University of Technology, The Netherlands,
{s.d.vermolen, g.h.wachsmuth, e.visser}@tudelft.nl

Abstract. Metamodel evolution requires model migration. To correctly migrate models, evolution needs to be made explicit. Manually describing evolution is error-prone and redundant. Metamodel matching offers a solution by automatically detecting evolution, but is only capable of detecting primitive evolution steps. In practice, primitive evolution steps are jointly applied to form a complex evolution step, which has the same effect on a metamodel as the sum of its parts, yet generally has a different effect in migration. Detection of complex evolution is therefore needed. In this paper, we present an approach to reconstruct complex evolution between two metamodel versions, using a matching result as input. It supports operator dependencies and mixed, overlapping, and incorrectly ordered complex operator components. It also supports interference between operators, where the effect of one operator is partially or completely hidden from the target metamodel by other operators.

1 Introduction

Changing requirements and technological progress require metamodels to evolve [8]. Preventing metamodel evolution by downwards-compatible changes is often insufficient, as it reduces the quality of the metamodel [2]. Metamodel evolution may break conformance of existing models and thus requires model migration [22]. To correctly migrate models, the evolution – implicitly applied by developers – needs to become explicit. Metamodel evolution can be specified manually by developers, yet this is error-prone, redundant, and hard in larger projects. Instead, evolution needs to be detected automatically from the original and evolved metamodel versions.

The most-used solution for detecting evolution is matching [24]. Metamodel matching attempts to link elements from the original metamodel to elements from the target metamodel based on similarity. The result is a set of atomic differences highlighting what was created, what was deleted and what was changed. In practice, groups of atomic differences may be applied together to form complex evolution steps such as pulling features up an inheritance chain or extracting super classes [13]. In model migration, a complex operator is different from its atomic changes. For example, pulling up a feature preserves information, whereas deleting and recreating it loses information. To correctly describe evolution, we therefore need to detect complex evolution steps. There are three major problems in reconstructing complex evolution steps:

Dependency. While metamodel changes are unordered, evolution steps are generally applied sequentially and may depend on one another [4]. These dependencies need to be respected by a mapping from metamodel changes to evolution steps.

Detection. To detect a complex evolution step, we must find several steps which make up this complex step. But these steps are likely to be separated, incorrectly ordered, and mixed with parts of other complex evolution steps.

Interference. An evolution step can hide, change, or partially undo the effect of another step. Multiple steps can completely mask a step. As such, some or all steps forming a more complex step may be missing, which impedes its detection.

Example. The upper part of Figure 1 shows two metamodel versions for a tag-based issue tracker. In the original metamodel on the left-hand side, each issue has a reporter, a title, and some descriptive text. Projects are formed by a group of users and have a name and a set of issues. Users can comment on issues and tag issues. Additions and removals of tags are recorded, such that they can be reverted.

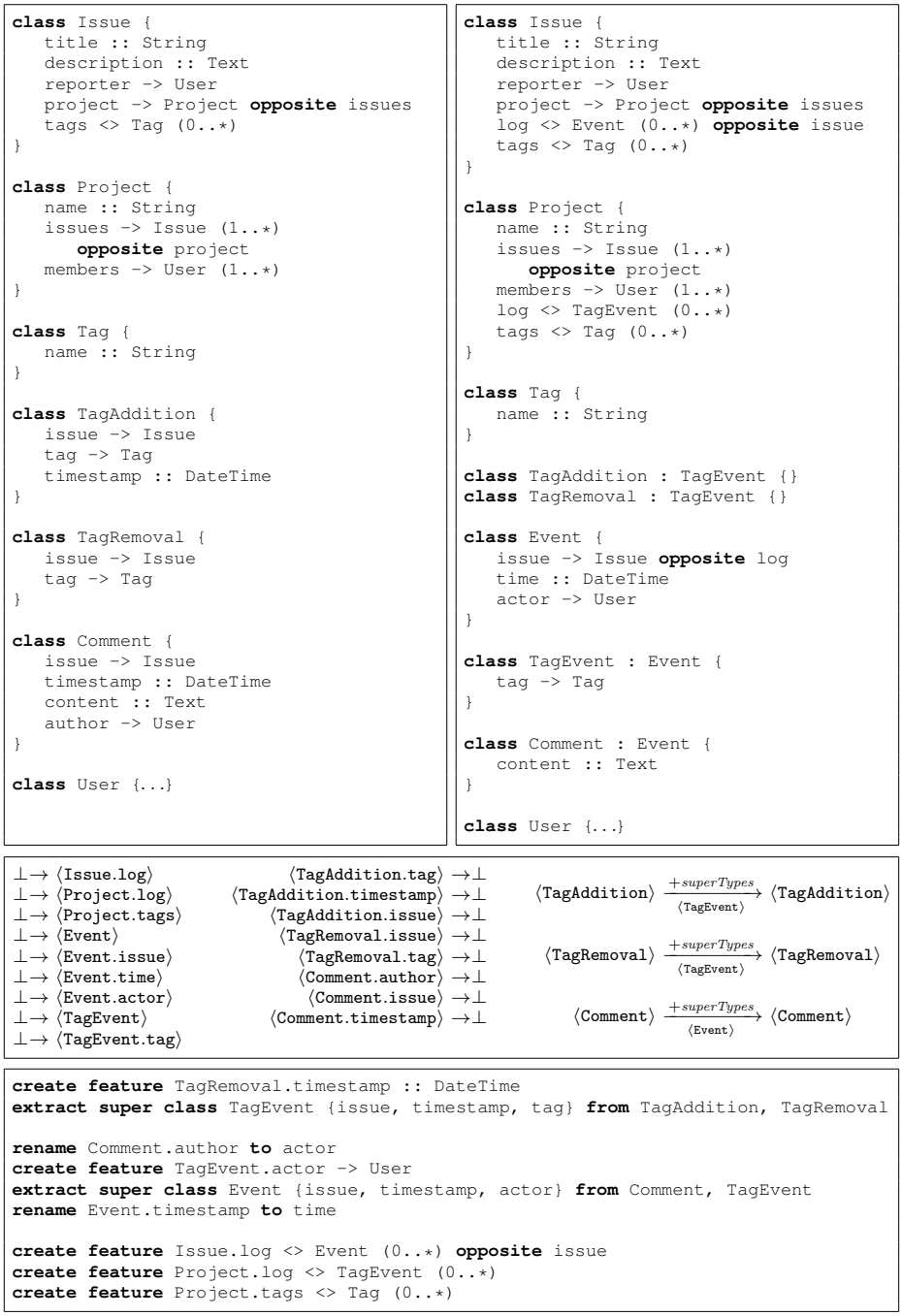
While evolving the issue tracker, tagging became the primary approach for organization. As such, it became apparent, that not only issues, but also projects should be taggable. Additionally, the metamodel structure had to be improved to allow users to more easily subscribe to events, as to send them email updates. The resulting metamodel is shown at the upper right of Figure 1. An Event entity was introduced, which comprises comments as well as tag events (tag additions and removals). Furthermore, projects obtained room for storing tags and events on these tags.

Matching the original and evolved metamodel yields the difference model presented in the middle part of Figure 1. Two classes and seven features were added to the evolved metamodel (left column), eight features were subtracted (middle column), and three classes have an additional super type in the evolved metamodel (right column). We will use this difference model as a starting point to detect the complex evolution steps involved in the evolution of the original metamodel.

The evolution of the metamodel can also be captured in an evolution trace as shown in the bottom part of Figure 1. At the metamodel level, the trace specifies the creation of five new features, the renaming of two other features, and the extraction of two new classes. At the model level, it specifies a corresponding migration. From the properties of the involved operators, we can conclude that the evolution is constructive and that we can safely migrate existing models without losing information.

In detecting the example evolution trace from the difference model, we face all three major problems in trace reconstruction several times. For example, the second step depends on the first step as it can only be applied if `TagRemoval` has a `timestamp`. Furthermore, the second step comprises several of the presented differences. And finally, the first step interferes with the second, since its effect is completely hidden from the difference model. The step needs to be reconstructed during detection.

Contribution. In this paper, we provide an approach to reconstruct complex evolution traces from difference models automatically. It is based on the formalization of the core concepts involved, namely metamodels, difference models, and evolution traces (Section 2). First, we provide a mapping from changes in a difference model to primitive operators in an evolution trace. We solve the dependency problem by defining preconditions for all primitive operators. Based on these preconditions, we define a dependency relation between operators which allows us to order operators on dependency and to



```
class Issue {
   title :: String
   description :: Text
   reporter -> User
   project -> Project opposite issues
   tags <> Tag (0..*)
}

class Project {
   name :: String
   issues -> Issue (1..*)
      opposite project
   members -> User (1..*)
}

class Tag {
   name :: String
}

class TagAddition {
   issue -> Issue
   tag -> Tag
   timestamp :: DateTime
}

class TagRemoval {
   issue -> Issue
   tag -> Tag
}

class Comment {
   issue -> Issue
   timestamp :: DateTime
   content :: Text
   author -> User
}

class User {...}
```

```
class Issue {
   title :: String
   description :: Text
   reporter -> User
   project -> Project opposite issues
   log <> Event (0..*) opposite issue
   tags <> Tag (0..*)
}

class Project {
   name :: String
   issues -> Issue (1..*)
      opposite project
   members -> User (1..*)
   log <> TagEvent (0..*)
   tags <> Tag (0..*)
}

class Tag {
   name :: String
}

class TagAddition : TagEvent {}
class TagRemoval : TagEvent {}

class Event {
   issue -> Issue opposite log
   time :: DateTime
   actor -> User
}

class TagEvent : Event {
   tag -> Tag
}

class Comment : Event {
   content :: Text
}

class User {...}
```

$\bot \rightarrow \langle \texttt{Issue.log} \rangle$
$\bot \rightarrow \langle \texttt{Project.log} \rangle$
$\bot \rightarrow \langle \texttt{Project.tags} \rangle$
$\bot \rightarrow \langle \texttt{Event} \rangle$
$\bot \rightarrow \langle \texttt{Event.issue} \rangle$
$\bot \rightarrow \langle \texttt{Event.time} \rangle$
$\bot \rightarrow \langle \texttt{Event.actor} \rangle$
$\bot \rightarrow \langle \texttt{TagEvent} \rangle$
$\bot \rightarrow \langle \texttt{TagEvent.tag} \rangle$
$\langle \texttt{TagAddition.tag} \rangle \rightarrow \bot$
$\langle \texttt{TagAddition.timestamp} \rangle \rightarrow \bot$
$\langle \texttt{TagAddition.issue} \rangle \rightarrow \bot$
$\langle \texttt{TagRemoval.issue} \rangle \rightarrow \bot$
$\langle \texttt{TagRemoval.tag} \rangle \rightarrow \bot$
$\langle \texttt{Comment.author} \rangle \rightarrow \bot$
$\langle \texttt{Comment.issue} \rangle \rightarrow \bot$
$\langle \texttt{Comment.timestamp} \rangle \rightarrow \bot$

$\langle \texttt{TagAddition} \rangle \xrightarrow[\langle \texttt{TagEvent} \rangle]{+superTypes} \langle \texttt{TagAddition} \rangle$

$\langle \texttt{TagRemoval} \rangle \xrightarrow[\langle \texttt{TagEvent} \rangle]{+superTypes} \langle \texttt{TagRemoval} \rangle$

$\langle \texttt{Comment} \rangle \xrightarrow[\langle \texttt{Event} \rangle]{+superTypes} \langle \texttt{Comment} \rangle$

```
create feature TagRemoval.timestamp :: DateTime
extract super class TagEvent {issue, timestamp, tag} from TagAddition, TagRemoval

rename Comment.author to actor
create feature TagEvent.actor -> User
extract super class Event {issue, timestamp, actor} from Comment, TagEvent
rename Event.timestamp to time

create feature Issue.log <> Event (0..*) opposite issue
create feature Project.log <> TagEvent (0..*)
create feature Project.tags <> Tag (0..*)
```

Fig. 1. Original and evolved metamodel, difference model, and evolution trace

construct valid primitive evolution traces from a difference model (Section 3). Second, we show how to reorder primitive traces without breaking their validity and provide patterns for mapping sequences of primitive operators to complex operators. We solve the detection problem by reordering primitive traces to different normal forms in which the patterns can be detected easily (Section 4). Finally, we extend our method to detect also partial patterns in order to solve the interference problem (Section 5).

2 Modeling Metamodel Evolution

Metamodeling Formalism. Metamodels can be expressed in various metamodeling formalisms. In this paper, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out packages, enumerations, annotations, derived features, and operations.

Figure 2 gives a textual definition of the metamodeling formalism used in this paper. A metamodel defines a number of classes which consist of a number of features. Classes can have super types to inherit features and might be abstract. A feature has a multiplicity (lower and upper bound) and is either an attribute or a reference. An attribute is a feature with a primitive type, whereas a reference is a feature with a class type. We only support predefined primitive types like Boolean, Integer, and String. An attribute can serve as an identifier for objects of a class. A reference may be composite and two references can be combined to form a bidirectional association by making them opposite of each other. In the textual notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the kind of a feature. We use `::` for attributes, `->` for ordinary references, and `<>` for composite references.

If we want to reason about properties of metamodels and their evolution, a textual representation is often not sufficient. Thus, we provide in Figure 3 a more formal representation of metamodels in terms of sets, functions, and predicates. In the upper left, we define instance sets for the metaclasses from Figure 2. In the upper right, we formalize most metafeatures from Figure 2 in terms of functions and predicates. Since super types and features of a class c form subsets of instance sets, we formalize them accordingly.

```
class MetaModel {
  classes     <> Class (0..*)
}

abstract class NamedElement {
  name        :: String (1..1)
}

abstract class Type : NamedElement {}

class Class : Type {
  isAbstract  :: Boolean (1..1)
  superTypes  -> Class (0..*)
  features    <> Feature (0..*)
}
```

```
class DataType : Type {}

abstract class Feature : NamedElement {
  lowerBound  :: Integer (1..1)
  upperBound  :: Integer (1..1)
  type        -> Type (1..1)
}

class Attribute : Feature {
  isId        :: Boolean (1..1)
}

class Reference : Feature {
  isComposite :: Boolean (1..1)
  opposite    -> Reference
}
```

Fig. 2. Metamodeling formalism providing core metamodeling concepts

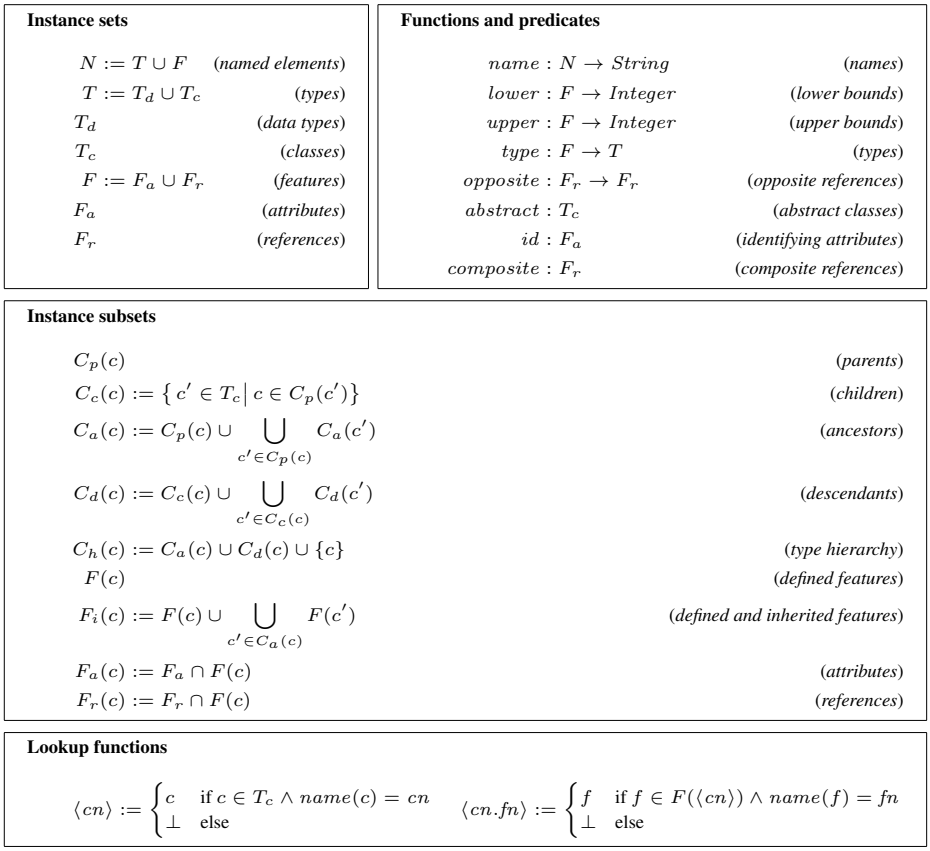


Fig. 3. Formal representation of metamodels in terms of sets, functions, and predicates

In terms of these subsets, we define other interesting subsets, e.g., children, ancestors and descendants of c in the middle part. Typically, we refer to a class c by its name cn and to a feature f of class c by $cn.fn$ where cn and fn are the names of c and f, respectively. To access classes and features referred by name, we define lookup functions in the last box. The formalization so far also captures invalid metamodels, such as metamodels with duplicate class names, or cycles in an inheritance hierarchy. Therefore, we define metamodel validity by a number of invariants in Figure 4.

Difference Models. Difference-based approaches to coupled evolution use a declarative evolution specification, generally referred to as the difference model [3, 9]. This difference model can be mapped automatically onto a model migration. With an automated detection of the difference model, the process can be completely automated. Matching algorithms provide such a detection [17, 7, 5, 15, 30, 1].

In this paper, we do not rely on a particular matching algorithm and abstract over concrete representations of difference models. We model the difference between an original metamodel m_o and an evolved version m_e as a set $\Delta(m_o, m_e)$. The elements

Metamodel validity $\vdash m$

$\forall c, c' \in T_c : name(c) = name(c') \Rightarrow c = c'$	(*unique class names*)
$\forall c \in T_c : \forall f, f' \in F_i(c) : name(f) = name(f') \Rightarrow f = f'$	(*unique feature names*)
$\forall c \in T_c : c \notin C_a(c)$	(*non-cyclic inheritance*)
$\forall f \in F : lower(f) \leq_b upper(f) \wedge upper(f) >_b 0$	(*correct bounds*)
$\forall f \in F_a : type(f) \in T_d$	(*well-typed attributes*)
$\forall f \in F_r : type(f) \in T_c$	(*well-typed references*)
$\forall f, f' \in F_r : opposite(f) = f' \Leftrightarrow opposite(f') = f$	(*inverse reflectivity*)

Difference model validity $\vdash \Delta(m_o, m_e)$

$\vdash m_o \wedge \vdash m_e$	(*source and target validity*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : t(\delta) = t(\delta') \neq \perp \Rightarrow s(\delta) = s(\delta')$	(*unique sources*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) = s(\delta') \neq \perp \Rightarrow t(\delta) = t(\delta')$	(*unique targets*)
$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) \in F(s(\delta')) \wedge t(\delta) \neq \perp \Rightarrow t(\delta) \in F(t(\delta'))$	(*non-moving features*)

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$\vdash m_o$	(*source validity*)
$\forall i \in 1, \dots, n : \vdash O_1 \circ \dots \circ O_i(m_o)$	(*valid applications*)
$O_1 \circ \dots \circ O_n(m_o) = m_e$	(*target validity*)

Fig. 4. Validity of metamodels, difference models, and evolution traces

of this set are three different kinds of changes [26, 3]: *Additive changes* $\perp \rightarrow e$, where the evolved metamodel contains an element e which was not present in the original metamodel. *Subtractive changes* $e \rightarrow \perp$, where the evolved metamodel misses an element e which was present in the original metamodel. *Updative changes*, where the evolved metamodel contains an element e' which corresponds to an element e in the original metamodel and the value of a metafeature of e' is different from the value in e. We distinguish three kinds of updates: *Additions* $e \xrightarrow[v]{+mf} e'$, where the multi-valued metafeature mf of e' has an additional value v which was not present in e. *Removals* $e \xrightarrow[v]{-mf} e'$, where the multi-valued metafeature mf of e' is missing a value v which was present in e. *Substitutions* $e \xrightarrow{mf} e'$, where the single-valued metafeature mf of e' has a new value which is different from the value in e. A complete list of possible metamodel changes with respect to our metamodeling formalism is given in the left column of Figure 5.

For validity of difference models, we have three requirements: First, the original and evolved metamodel need to be valid. Second, two changes should not link the same source element with different target elements or the same target element with different source elements. Element merges and splits are represented as separate additions and removals and will be reconstructed during detection. Third, we expect changing features not to move between classes, i.e., the class containing a changed feature should be the same or a changed version of the class containing the original feature. We define these requirements formally in Figure 4. Note that $s(\delta)$ yields the source element of a change (left-hand side of an arrow) while $t(\delta)$ gives the target element (right-hand side).

Evolution Traces. Operator-based approaches to coupled evolution provide a rich set of coupled operators which work at the metamodel level as well as at the model level [29, 11]. At the metamodel level, a *coupled operator* defines a metamodel transformation capturing a common evolution step. At the model level, it defines a model transformation capturing the corresponding migration. Following the terminology from [13], we differentiate between primitive and complex operators. *Primitive operators* perform an atomic metamodel evolution step that can not be further subdivided. A list of primitive operators which is complete with respect to our metamodeling formalism is given in the left column of Figure 7. *Complex operators* can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but typically not at the model level. For example, a feature pull-up can be decomposed into feature deletions in the subclasses followed by a feature creation in the parent class. At the model level, the feature deletions cause the deletion of values in instances of the subclasses while the feature creation requires the introduction of default values in instances of the parent class. Thus, values for the feature in instances of the subclasses are replaced by default values. This is not an appropriate migration for a feature pull-up which instead requires the preservation of values in instances of the subclasses. We will define only a few complex operators in this paper. For an extensive catalog of operators, see [13].

Each operator has a number of formal parameters like class and feature names. Instantiating these parameters with actual arguments results in an *operator instance* O. This notation hides the actual arguments but is sufficient for this paper. We can now model the evolution of a metamodel as a sequence of such operator instances $O_1 \dots O_n$. We call this sequence an *evolution trace*. We distinguish *primitive traces* of only primitive operator instances from *complex traces*. There are three requirements for the validity of an evolution trace with respect to the original and the evolved metamodel. First, we require the original metamodel to be valid. Second, each operator instance should be applicable to the result of its predecessors and should yield a valid metamodel. Third, applying the complete trace should result in the evolved metamodel. Again, we capture these requirements formally in Figure 4.

3 Reconstructing Primitive Evolution Traces

This section shows how to reconstruct a correctly ordered, valid evolution trace from a difference model. First, we provide a mapping from metamodel changes to sequences of primitive operator instances. Second, we define a dependency relation between operator instances based on preconditions of these instances. This allows us to order primitive evolution traces on dependency resulting in valid primitive evolution traces.

Mapping. The mapping of changes onto sequences of operator instances is presented in Figure 5. The left column shows the different metamodel changes. The right column shows the corresponding operator instances. The middle column shows conditions to select the right mapping and to instantiate parameters correctly. Note that we omit conditions of the form $xn = name(x)$. We assume such conditions implicitly whenever there is a pair of variables x and xn. This way, cn refers to the name of a class c, fn to

the name of a feature f, and tn to the name of a type t. Figure 6 (left) shows the result of the mapping applied to the example difference model from Figure 1.

Dependencies between Operator Instances. Despite the atomicity of primitive operators, not all primitive evolution traces can be completely executed. Reconsider the left trace in Figure 6. Step 5 creates a reference to `TagEvent` at a point where no class `TagEvent` exists. Similarly, step 8 references a non-existent class `Tag` and step 24 attempts to create an inheritance chain with duplicate feature names. Operator instances cannot be applied to all metamodels: Features can only be created in classes that exist, classes can only be created if no equivalently named class is present and a class can only be dropped if it is not in use anywhere else. These restrictions either come directly from the meta-metamodel or from the invariants for valid metamodels. We can translate these restrictions into preconditions. An operator precondition $O_{pre}(m)$ ensures that an operator instance O can be applied to a metamodel m and that the application on a valid m yields again a valid metamodel. Figures 7 and 8 give a complete overview of the preconditions for primitive operators.

One condition for the validity of a trace of operators is the validity of each intermediate metamodel. Since succeeding operator preconditions ensure this validity, we can redefine trace validity in terms of preconditions:

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$$O_{1,pre}(m_o) \wedge \forall_{i \in 2..n} : O_{i,pre}((O_1 \circ \dots \circ O_{i-1})(m)) \qquad \textit{(valid applications)}$$

Applying operator instances enables or disables other operator instances. For example, the creation of a class c can enable the creation of a feature $c.f$. The class creation operator validates parts of the precondition of the feature creation operator. To model the effect of an operator instance on conditions, we use a backward transformation description as introduced by Kniesel and Koch [14]. A backward description O_{bd} is a function that, given a condition C to be checked after applying an operator instance O, computes a semantically equivalent condition that can be checked before applying O: $O_{bd}(C)(m) \Leftrightarrow C(O(m))$. We define backward description functions for the primitive operators based on the postconditions specified in Figures 7 and 8: A backward description rewrites any clause in a condition C with $true$, when it is implied by the operator postcondition. Using these backward description functions, we can define enabling and disabling operator instances as dependencies: Operator instance O_2 depends on operator instance O_1, if the backward description of operator O_1 changes the precondition of O_2. Typically, operator instances are dependent if they affect or target the same metamodel element. Examples are creation and deletion of the same class, creation of a class and addition of a feature to this class, and creation of a class and of a reference to this class.

Dependency Ordering. To ensure trace validity, we need to ensure that the preconditions of all operator instances are enabled and thus all dependencies are satisfied. The dependency relation between operator instances is a partial order on these instances. To establish validity, we apply the partial dependency order to the trace and

Metamodel Difference	Conditions	Primitive Operator Instances
$\bot \rightarrow c$	$c \in T_c$ $abstract(c)$ $C_p(c) = \{sc_1, \ldots, sc_k\}$	**create class** cn [**make** cn **abstract**] [**add super** scn_1 **to** cn $\vdots$ **add super** scn_k **to** cn]
$c \rightarrow \bot$	$c \in T_c$	**drop class** cn
$e \xrightarrow{name} e'$	$e \in T_c$	**rename** en **to** en'
	$e \in F(c)$	**rename** $cn.en$ **to** en'
$c \xrightarrow{isAbstract} c'$	$\neg abstract(c)$	**make** cn **abstract**
	$abstract(c)$	**drop** cn **abstract**
$c \xrightarrow[sc]{+superTypes} c'$		**add super** scn **to** cn
$c \xrightarrow[sc]{-superTypes} c'$		**drop super** scn **from** cn
$\bot \rightarrow f$	$f \in F_a(c) \wedge t = type(f)$ $l = lower(f) \wedge l >_b 0$ $u = upper(f) \wedge u >_b 1$ $id(f)$	**create feature** $cn.fn$ :: tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **identifier**]
	$f \in F_r(c) \wedge t = type(f)$ $l = lower(f) \wedge l >_b 0$ $u = upper(f) \wedge u >_b 1$ $composite(f)$ $f' = opposite(f)$	**create feature** $cn.fn$ -> tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **composite**] [**make** $cn.fn$ **inverse** fn']
$f \rightarrow \bot$	$f \in F(c)$	**drop feature** $cn.fn$
$f \xrightarrow{lowerBound} f'$	$l = lower(f') \wedge l <_b lower(f)$	**generalize lower** $cn.fn$ **to** l
	$l = lower(f') \wedge l >_b lower(f)$	**specialize lower** $cn.fn$ **to** l
$f \xrightarrow{upperBound} f'$	$u = upper(f') \wedge u >_b upper(f)$	**generalize upper** $cn.fn$ **to** u
	$u = upper(f') \wedge u <_b upper(f)$	**specialize upper** $cn.fn$ **to** u
$f \xrightarrow{type} f'$	$f \in F(c)$ $f' \in F_a(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $id(f')$	**drop feature** $cn.fn$ **create feature** $cn'.fn'$ :: tn [**specialize lower** $cn'.fn'$ **to** l] [**generalize upper** $cn'.fn'$ **to** u] [**make** $cn'.fn'$ **identifier**]
	$f \in F(c)$ $f' \in F_r(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $composite(f')$ $f'' = opposite(f')$	**drop feature** $cn.fn$ **create feature** $cn'.fn'$ -> tn [**specialize lower** $cn'.fn'$ **to** l] [**generalize upper** $cn'.fn'$ **to** u] [**make** $cn'.fn'$ **composite**] [**make** $cn'.fn'$ **inverse** fn'']
$f \xrightarrow{isId} f'$	$\neg id(f)$	**make** $cn.fn$ **identifier**
	$id(f)$	**drop** $cn.fn$ **identifier**
$f \xrightarrow{isComposite} f'$	$\neg composite(f)$	**make** $cn.fn$ **composite**
	$composite(f)$	**drop** $cn.fn$ **composite**
$f \xrightarrow{opposite} f'$	$f' \in F_r(c) \wedge f'' = opposite(f') \neq \bot$	**make** $cn.fn'$ **inverse** fn''
	$f' \in F_r(c) \wedge opposite(f') = \bot$	**drop** $cn.fn'$ **inverse**

Fig. 5. Possible metamodel changes and corresponding sequences of primitive operator instances

```
1  create feature Issue.log <> Event
2  generalize upper Issue.log to -1
3  make Issue.log composite
4  make Issue.log inverse Event.issue
5  create feature Project.log
       <> TagEvent
6  generalize upper Project.log to -1
7  make Project.log composite
8  create feature Project.tags <> Tag
9  generalize upper Project.tags to -1
10 make Project.tags composite
11 add super TagEvent to TagAddition
12 drop feature TagAddition.issue
13 drop feature TagAddition.tag
14 drop feature TagAddition.timestamp
15 add super TagEvent to TagRemoval
16 drop feature TagRemoval.issue
17 drop feature TagRemoval.tag
18 create class Event
19 create feature Event.issue -> Issue
20 create feature Event.time
       :: DateTime
21 create feature Event.actor -> User
22 create class TagEvent : Event
23 create feature TagEvent.tag -> Tag
24 add super Event to Comment
25 drop feature Comment.issue
26 drop feature Comment.timestamp
27 drop feature Comment.author
```

```
create feature Project.tags <> Tag
generalize upper Project.tags to -1
make Project.tags composite
drop feature TagAddition.issue
drop feature TagAddition.tag
drop feature TagAddition.timestamp
drop feature TagRemoval.issue
drop feature TagRemoval.tag
create class Event
create feature Issue.log <> Event
generalize upper Issue.log to -1
make Issue.log composite
create feature Event.issue -> Issue
make Issue.log inverse Event.issue
create feature Event.time
   :: DateTime
create feature Event.actor -> User
create class TagEvent : Event
create feature Project.log
   <> TagEvent
generalize upper Project.log to -1
make Project.log composite
add super TagEvent to TagAddition
add super TagEvent to TagRemoval
create feature TagEvent.tag -> Tag
drop feature Comment.issue
drop feature Comment.timestamp
drop feature Comment.author
add super Event to Comment
```

Fig. 6. Unordered and dependency-ordered primitives mapped from the difference model

Primitive Operator	Preconditions	Postconditions
create class cn	$\langle cn \rangle = \bot$	$\langle cn \rangle \neq \bot \wedge F(\langle cn \rangle) = \emptyset$ $\neg targeted(\langle cn \rangle) \wedge \neg abstract(\langle cn \rangle)$
drop class cn	$\langle cn \rangle \neq \bot \wedge F(\langle cn \rangle) = \emptyset$ $\neg targeted(\langle cn \rangle)$	$\langle cn \rangle \neq \bot$
create feature $cn.fn$ **::** tn	$\langle cn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) : \forall f' \in F(c') :$ $name(f') \neq fn$	$\langle cn.fn \rangle \neq \bot$ $\langle cn.fn \rangle \in F_a$
create feature $cn.fn$ **->** tn	$\langle cn \rangle, \langle tn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) : \forall f' \in F(c') :$ $name(f') \neq fn$	$\langle cn.fn \rangle \neq \bot$ $\langle cn.fn \rangle \in F_r \wedge type(\langle cn.fn \rangle) = \langle tn \rangle$ $\nexists f' : opposite(\langle cn.fn \rangle) = f')$ $\neg composite(\langle cn.fn \rangle) \wedge \neg id(\langle cn.fn \rangle)$
drop feature $cn.fn$	$\langle cn.fn \rangle \neq \bot$	$\langle cn.fn \rangle = \bot$

Fig. 7. Pre- and postconditions for structural primitive operators

make the ordering complete by arbitrarily ordering independent operator instances. Figure 6 (right) shows the dependency-ordered trace of primitive operators for the running example.

4 Reconstructing Complex Evolution Traces

This section shows how to reconstruct valid complex evolution traces from valid primitive traces. First, we provide patterns for mapping sequences of primitive operator in-

Primitive Operator	Preconditions	Postconditions
rename class cn **to** cn'	$\langle cn \rangle \neq \bot \wedge \langle cn' \rangle = \bot$	$\langle cn \rangle = \bot \wedge \langle cn' \rangle \neq \bot$
rename feature $cn.fn$ **to** fn'	$\langle cn.fn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) : \forall f' \in F(c') :$ $name(f') \neq fn'$	$\langle cn.fn \rangle = \bot$ $\langle cn.fn' \rangle \neq \bot$
make cn **abstract**	$\langle cn \rangle \neq \bot \wedge \neg abstract(\langle cn \rangle)$	$abstract(\langle cn \rangle)$
drop cn **abstract**	$\langle cn \rangle \neq \bot \wedge abstract(\langle cn \rangle)$	$\neg abstract(\langle cn \rangle)$
add super cn_{sup} **to** cn_{sub}	$\langle cn_{sup} \rangle, \langle cn_{sub} \rangle \neq \bot$ $\langle cn_{sup} \rangle \notin C_h(\langle cn_{sub} \rangle)$ $\forall c \in C_h(\langle cn_{sub} \rangle) : \forall f \in F(c) :$ $\langle cn_{sup}.name(f) \rangle = \bot$	$\langle cn_{sup} \rangle \in C_p(\langle cn_{sub} \rangle)$
drop super cn_{sup} **from** cn_{sub}	$\langle cn_{sub} \rangle, \langle cn_{sup} \rangle \neq \bot$ $\langle cn_{sup} \rangle \in C_p(\langle cn_{sub} \rangle)$	$\langle cn_{sup} \rangle \notin C_p(\langle cn_{sub} \rangle)$
generalize type $cn.fn$ **to** cn'	$\langle cn.fn \rangle \neq \bot \wedge \langle cn' \rangle \neq \bot$ $\langle cn' \rangle \in C_a(\langle cn \rangle)$	$type(\langle cn.fn \rangle) = \langle cn' \rangle.$
specialize type $cn.fn$ **to** cn'	$\langle cn.fn \rangle \neq \bot \wedge \langle cn' \rangle \neq \bot$ $\langle cn' \rangle \in C_d(\langle cn \rangle)$	$type(\langle cn.fn \rangle) = \langle cn' \rangle.$
generalize upper $cn.fn$ **to** u	$\langle cn.fn \rangle \neq \bot$ $u >_B upper(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
generalize lower $cn.fn$ **to** l	$\langle cn.fn \rangle \neq \bot$ $l < lower(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$
specialize upper $cn.fn$ **to** u	$\langle cn.fn \rangle \neq \bot$ $u <_B upper(\langle cn.fn \rangle)$ $u \geq_B lower(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
specialize lower $cn.fn$ **to** l	$\langle cn.fn \rangle \neq \bot$ $l > lower(\langle cn.fn \rangle)$ $l \leq upper(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$
make $cn.fn$ **inverse** $cn'.fn'$	$\langle cn.fn \rangle, \langle cn'.fn' \rangle \neq \bot$ $\nexists f : opposite(\langle cn.fn \rangle) = f$ $\vee opposite(\langle cn'.fn' \rangle) = f$	$opposite(\langle cn.fn \rangle) = \langle cn'.fn' \rangle$
drop $cn.fn$ **inverse**	$\langle cn.fn \rangle \neq \bot$ $\exists f' : opposite(\langle cn.fn \rangle) = f'$	$\nexists f' : opposite(\langle cn.fn \rangle) = f'$
make $cn.fn$ **identifier**	$\langle cn.fn \rangle \neq \bot \wedge \neg id(\langle cn.fn \rangle)$	$id(\langle cn.fn \rangle)$
drop $cn.fn$ **identifier**	$\langle cn.fn \rangle \neq \bot \wedge id(\langle cn.fn \rangle)$	$\neg id(\langle cn.fn \rangle)$
make $cn.fn$ **composite**	$\langle cn.fn \rangle \neq \bot$ $\neg composite(\langle cn.fn \rangle)$	$composite(\langle cn.fn \rangle)$
drop $cn.fn$ **composite**	$\langle cn.fn \rangle \neq \bot$ $composite(\langle cn.fn \rangle)$	$\neg composite(\langle cn.fn \rangle)$

Fig. 8. Pre- and postconditions for non-structural primitive operators

stances to complex operator instances. Second, we discuss how to reorder evolution traces without breaking their validity. This allows us to reorder traces into different normal forms in which the patterns can be detected easily and be replaced by complex operator instances.

Patterns. A complex operator instance comprises a sequence of (less-complex) operator instances. We can use patterns on these sequences to detect complex operator instances. Figure 9 lists the decompositions and conditions for two complex operators

Complex Operator	Conditions	Equivalent Trace
pull up feature $cn.fn$	$C_c(\langle cn \rangle) = \{c_1, \ldots, c_k\}$ $\langle cn_1.fn \rangle \equiv_F \cdots \equiv_F \langle cn_k.fn \rangle$ $t = type(\langle cn_1.fn \rangle) \wedge t \in T_d$ $l = lower(\langle cn_1.fn \rangle) \wedge l >_b 0$ $u = upper(\langle cn_1.fn \rangle) \wedge u >_b 1$ $id(\langle cn_1.fn \rangle)$	**drop feature** $cn_1.fn$ $\ldots$ **drop feature** $cn_k.fn$ **create feature** $cn.fn$ **::** tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **identifier**]
	$C_c(\langle cn \rangle) = \{c_1, \ldots, c_k\}$ $\langle cn_1.fn \rangle \equiv_F \cdots \equiv_F \langle cn_k.fn \rangle$ $t = type(\langle cn_1.fn \rangle) \wedge t \in T_c$ $l = lower(\langle cn_1.fn \rangle) \wedge l >_b 0$ $u = upper(\langle cn_1.fn \rangle) \wedge u >_b 1$ $composite(\langle cn_1.fn \rangle)$ $f' = opposite(\langle cn_1.fn \rangle)$	**drop feature** $cn_1.fn$ $\ldots$ **drop feature** $cn_k.fn$ **create feature** $cn.fn$ **->** tn [**specialize lower** $cn.fn$ **to** l] [**generalize upper** $cn.fn$ **to** u] [**make** $cn.fn$ **composite**] [**make** $cn.fn$ **inverse** fn']
extract super class cn $\{fn_1, \ldots, fn_j\}$ **from** $cn_1, \ldots, cn_k$	$true$	**create class** cn **add super** cn **to** cn_1 $\ldots$ **add super** cn **to** cn_k **pull up feature** $cn.fn_1$ $\ldots$ **pull up feature** $cn.fn_j$
fold super class cn **from** cn'	$F_i(\langle cn \rangle) = \{f_1, \ldots, f_k\}$ $\forall i = 1 \ldots k : \langle cn'.fn_i \rangle \equiv_F f_i$	**drop feature** $cn'.fn_1$ $\ldots$ **drop feature** $cn'.fn_k$ **add super** cn **to** cn'

Fig. 9. (De-)Composition patterns for complex operators

working across inheritance. When read from left to right, it shows how to decompose a complex operator instance, when read from right to left, it defines its detection pattern. Given a source metamodel m, we can recursively decompose an operator instance O into a sequence of primitive operator instances $\lfloor O \rfloor_m = P_1 \ldots P_n$. As a precondition, a complex operator instance needs to fulfill the backward descriptions of the preconditions of these primitives. But typically this is not enough and an operator instance requires additional preconditions.We highlight these additional preconditions with a box in Figure 9.

Reordering traces. Figure 10 shows an excerpt of Figure 6. It displays the extraction of super class `Event` from class `Comment`. Operator ordering is still determined by the dependency ordering from the previous section. To simplify the example, we changed the operator on `Comment.author` to work on `Comment.actor`. We will look at `author` and the complete trace in the next section. Consider applying the patterns from Figure 9. There is no consecutive sequence of operator instances satisfying any of the patterns. We could detect pulling up feature `timestamp` in instances 2 and 4, yet where do we put the detected complex operator: at position 2 or at position 4?

Detection patterns typically cannot be applied directly. Instead, traces need to be reordered to find consecutive instances of a pattern. Dependency ordering is partial and therefore leaves room for swapping independent operators. In the example, we can

```
1 create class Event
2 create feature Event.timestamp
                    :: DateTime
3 create feature Event.actor -> User
4 drop feature Comment.timestamp
5 drop feature Comment.actor
6 add super Event to Comment
```

```
1 create class Event
6 add super Event to Comment
5 drop feature Comment.actor
3 create feature Event.actor -> User
4 drop feature Comment.timestamp
2 create feature Event.timestamp
                    :: DateTime
```

Fig. 10. Excerpt of dependency-ordered operators in Figure 6

swap 2 and 3 as they work on different features; 2 and 4 as they work on different types; 2 and 5 which also work on different types; 4 and 5 which work on different features; 3 and 5, which work on different types; and finally, we can repeatedly swap 6 to follow operator 1, as all features that are created are dropped from the inheritance chain first. The reordered trace is shown at the right of Figure 10. We can now apply the patterns for pulling up `timestamp` and `actor`. Subsequently, we see the pattern for class extraction emerge, which yields a super class extraction of `Event {timestamp, actor}` from `Comment` and `TagEvent`.

Normal forms. In the example, we carefully swapped operators. Not only did we avoid swapping dependent operators (as to preserve trace validity), we also chose swaps, which gave us a detectable pattern. In particular, we focused on obtaining a consecutive feature creation and drop, of features that only differ in position in the inheritance chain. A set of swap rules can bring an evolution trace into a format most suitable for detecting a pattern. In general, these rules obey the dependency relation. However, some dependent instances can still be swapped by adjusting their parameters. For example, **`rename class`** `A` **`to`** `B` and **`create feature`** `B.f...` can be swapped to: **`create feature`** `A.f...` and **`rename class`** `A` **`to`** `B`.

Repeated application of a set of swap rules will result in a normal form defined by this set. Each normal form targets to bring potential components of a pattern together and to satisfy the operator precondition. For example, to detect a feature pull up, we rely on feature similarity: Class creations and super additions get precedence over other operators. Feature creations, changes, and drops are sorted on feature name, type, and modifiers. Class drops and destructive updates on the inheritance chain go last. Different patterns need different trace characteristics and thus different normal forms. But operators with similar kinds of patterns can share normal forms.

5 Reconstructing Masked Operator Instances

In this section, we extend the detection to deal not only with complete but also partial patterns. First, we revisit the problem of operator interference and study its effects on detection. Second, we show how to complete partial patterns by the additions of operator instances in a validity preserving fashion. This allows us to detect operator instances which patterns are partially or even completely hidden by other instances.

Masked Operators. We reconsider the running example from Figure 1. During evolution, several features of the classes `TagAddition` and `TagRemoval` were extracted

into a new super class `TagEvent`. In order to extract the feature `timestamp` it needs to be present in both `TagAddition` and `TagRemoval`. Yet, it is not. As a human, we deduce that `timestamp` must have been added in the process of extracting `TagEvent`. There is, however, no explicit record of such feature creation. Detection will therefore fail. Later in the evolution, when extracting the class `Event`, we seek to pull up a feature `actor`. The class `Comment`, which we are extracting from, only offers a feature `author`. Again as a human, we assume that `author` must have been renamed to `actor` (like we did in the previous section), yet this operation is not present in the original evolution trace. Similarly, we have to create the feature `actor` in `TagEvent` before extracting `Event` and rename the feature `timestamp` to `time` after extracting `Event` to yield the target metamodel. Each of these operations has no record in the difference set obtained from the matching algorithm.

When evolutions become more complex, individual evolution steps no longer need to have an explicit effect on the target metamodel and are therefore not explicit in the matching result. An operator instance can hide or even undo parts of the effect of another instance. This is a strong variant of dependency, which we call masking. A primitive operator P_1 masks another primitive operator P_2 when composition of the two can be captured in a third primitive operator P_3. More generally, we define masking for arbitrary operator instances as the presence of a mask in decompositions:

$$P_1\ masks_m\ P_2 \Leftrightarrow \exists P_3 : (P_1 \circ P_2)(m) = P_3(m)$$
$$O_1\ masks_m\ O_2 \Leftrightarrow \exists P_1 \in \lfloor O_1 \rfloor_m : \exists P_2 \in \lfloor O_2 \rfloor_m : P_1\ masks\ P_2$$

Most operators can be masked by renaming. All operators are masked by their inverses, in which case P_3 is the identity operator. Extraction of class `TagEvent` in the running example masks extraction of class `Event`. Note that a trace obtained from a valid difference model will only contain masks that involve complex operators.

Masked Detection Rules. Detection of masked operator instances follows a trace rewriting approach similar to the original detection of complex operator instances: We try to rewrite a sequence of operator instances into another sequence which has the same effect on the metamodel. Instead of checking the operator precondition in a pattern, like we did in the previous section, we now *ensure* the precondition by deducing a suitable sequence to rewrite to. We now discuss how to derive a detection rule for a masked complex operator instance, e.g., for pulling up an attribute $cn_{sup}.fn$. Its decomposition is the following:

drop feature $cn_{sub}1.fn$
...
drop feature $cn_{sub}i.fn$
create feature $cn_{sup}.fn$

[**specialize lower** $cn_{sup}.fn$ **to** l]
[**generalize upper** $cn_{sup}.fn$ **to** u]
[**make** $cn.fn$ **identifier**]

From the decomposition we choose a trigger, which tells us that there may have been a feature pull up. We choose one of the feature drops (number `x`). We use the trigger as a pattern on the left-hand side of a rewrite rule and assume on the right-hand side that there must have been a feature pull up:

drop feature $cn_{sub}x.fn$ -> ... **pull up feature** $cn_{sup}.fn$ ...

When the dots are left blank, application of the left-hand side to a metamodel does not have an equivalent effect as application of the right-hand side. Instead, we fill the dots, to establish equivalence. The left set of dots ensures that the pull up feature operator can be applied, i.e., its precondition is satisfied. The right set of dots ensures that application of the trace is equivalent to application of the left-hand side of the rewrite rule. Both sets of dots are filled in using inverses of the operators found in the pattern. The left set of dots is replaced by inverses of each of the primitive operators whose precondition is not already satisfied. For pull up feature, we create features in all sibling classes if they do not exist yet and remove the target feature if it already exists. The right set of dots is replaced by inverses that neutralize the effect of the complex operator and bring the metamodel back to its original state. For pull up feature, we need to create all sibling features, which were present beforehand, as these were deleted during pull up and we need to drop the target feature if it was not present beforehand. The rewrite rule for detecting a masked feature pull up is (leaving out the operations on feature modifiers, for simplicity):

		create feature $cn_{sib}n1.fn$
		...
		create feature $cn_{sib}nj.fn$
		[**drop feature** $cn_{sup}.fn$]
drop feature $cn_{sub}x.fn$	->	**pull up feature** $cn_{sup}.fn$
		create feature $cn_{sib}e1.fn$
		...
		create feature $cn_{sib}ek.fn$
		[**drop feature** $cn_{sup}.fn$]

In which cn_{sup} is chosen arbitrarily from $C_p(\langle cn_{sub}x \rangle)$, $cn_{sib}n$ is the set of all sibling classes which do not have a feature named fn and thus need to obtain the feature to pull it up. $cn_{sib}e$ is the set of all sibling classes which do have a feature named fn and thus need to be reequipped with fn to neutralize the effect of pulling it up. The feature drops are conditional. The first drop should be present if $\langle cn_{sup}.fn \rangle \neq \bot$ and the latter should be present if $\langle cn_{sup}.fn \rangle = \bot$. In addition to the pattern on the left-hand side of a rewrite rule for a masked complex operator O, a rewrite rule is also conditioned by the operator's precondition O_{cpre}. It is checked in addition to the trigger. For feature pull up, the operator precondition O_{cpre} ensures presence of an inheritance chain between cn_{sub} and cn_{sup}. The metamodel invariants ensure feature names uniqueness across inheritance. The precondition of the trigger ensures fn exists in cn_{sub}. Therefore, fn cannot exist in cn_{sup}. The rewrite rule for feature pull up can thus be simplified by removing the top drop feature and always using the bottom drop feature.

Using the presented approach, we can derive masked detection rules for any complex operator. By definition, such rules expand the trace. To find a suitable evolution, we need to compact the trace again. First, we can rewrite any pair of inverse operators to the identity function, as their effect on the metamodel is canceled out and they are unlikely to have been part of the original evolution. Second, we combine a creation and deletion of two features, which only differ by name into a feature rename. This allows us to detect complex operators, which are masked by a rename, such as a pull up of feature f, followed by a rename of f to f'. Combining rules for inverses requires a normal form grouping on operator category and the renaming rule requires a normal form on feature similarity.

```
1  drop feature TagAddition.issue      ->  pull up feature TagEvent.issue
                                           drop feature TagEvent.issue
                                           create feature TagRemoval.issue

2  create feature TagRemoval.issue     ->  identity
   drop feature TagRemoval.issue

3  create feature TagEvent.tag -> Tag  ->  pull up feature TagEvent.tag
   drop feature TagAddition.tag            create feature TagRemoval.timestamp
   drop feature TagAddition.timestamp      pull up feature TagEvent.timestamp
   drop feature TagRemoval.tag             drop feature TagEvent.timestamp

4  pull up feature TagEvent.issue      ->  drop class TagEvent
                                           drop super TagEvent from TagAddition
                                           drop super TagEvent from TagRemoval
                                           extract super TagEvent
                                              {issue, tag, timestamp}
                                              from {TagAddition, TagRemoval}
                                           push down feature TagEvent.tag
                                           push down feature TagEvent.timestamp
```

Fig. 11. Masked detection applied to running example

Applying Masked Detection Rules. We apply masked detection rules to the running example. Figure 11 shows the intermediate steps. Step 1 applies feature pull up detection to `TagAddition.issue`. After normalizing the trace, we apply an inverse pattern to creation and drop of `TagRemoval.issue` and reduce the trace (step 2). `TagEvent.issue` is not reduced yet. It will be used later as a component of extracting class `Event`. Next, we repeat steps 1 and 2 by pulling up `tag` and `timestamp` (step 3). Subsequently, the pull up of `TagEvent.issue` triggers detection of super class extraction of `TagEvent` in step 4. The drop class, both super drops, and both feature push downs are subsequently neutralized by a class creation, super additions, and pull ups respectively. We then repeat detection of super class extraction for `Event`, using the rename pattern to neutralize create and drops of `timestamp` and `time` as well as `author` and `actor`. Finally, we get the result shown in Figure 1 (bottom).

All regular rewrite rules, which we defined in the previous section, reduced the number of operators in the trace. Furthermore, we did not consider overlapping (interfering) complex operators. These two assumptions enabled fast detection. The rules for detecting masked operators, on the other hand, can increase the size of the trace. For example, the feature pull up pattern increases the trace by the number of occurrences of this feature in sibling classes plus one (for dropping the pulled up feature). Furthermore, for each trace, several rules may be applicable at different positions in the trace. To find a solution, we therefore use a backtracking approach. Each backtracking step tries to apply each of the rules to a trace, yielding zero or more new traces, to which rule application is applied recursively.

6 Related Work

Research on difference detection is found in differencing textual documents, matching structured artifacts, and detection of complex evolution. Text differencing is ignorant of structure or semantics. We discuss related work on matching and complex detection.

Matching. A matching algorithm detects evolution between two artifacts by linking elements of one artifact to elements of the other. Links are either established based on similarity, or using an origin tracking technique such as persistent identifiers. Links are concerned with one element in each artifact. Consequently, matching approaches detect atomic changes. They do not offer support for detecting complex changes. Nevertheless, we discuss them as potential input to our approach. Matching has received attention in the domains of UML, source code reorganization, database schemas and metamodels.

In the domain of UML, Ohst et al. first proposed a solution to compare two UML documents [19]. They compare XML files and use persistent ids for matching. Later work by Xing and Stroulia presents UMLDiff, a matching tool set using similarity metrics instead of persistent ids to establish links [30]. Lin et al. propose a generalization of the work of Xing and Stroulia, which is not restricted to UML models, but uses domain specific models as input instead [16].

In the domain of source code reorganization, Demeyer et al. proposes to find refactorings using change metrics [6]. Later work by Tu and Godfrey uses statistical data and metrics to match evolved software architectures, a process referred to as origin analysis [25].The work on evolving architectures is extended by Godfrey and Zou, by adding detection of merged and split source code entities [10]. In schema matching, a body of work exists, which generally offers a basis for the other works presented in this section. Rahm et al. and later Shvaiko et al. present surveys on schema matching [20, 21]. Sun and Rose present a study of schema matching techniques [24].

Lopes et al. consider schema matching applied in the context of model-driven engineering, but propose a new matching algorithm for models [17]. Instead, Falleri et al. take the existing similarity flooding algorithm from the field of schema matching and apply it to metamodels [7]. Work by DelFabro et al. [5] and by Kolovos et al. [15] propose new matching algorithms to the modeling domain. Finally, EMFCompare offers metamodel independent model comparison in the Eclipse Modeling Framework [1]. It relies on heuristic-based matching and differencing, which are both pluggable.

Complex Detection. Detection of complex operators has received significantly less attention in research than matching. Cicchetti et al. discuss an approach for model migration along complex metamodel evolution [3]. They obtain the complex evolution from an arbitrary matching algorithm, but do not offer such an algorithm on their own. Instead, they emphasize the need for a matching algorithm able to detect complex evolution. Our approach fulfills this need. Later work of Cicchetti addresses the problem of dependencies between evolution steps [4]. Since their work focuses only on dependency ordering but not on complex operator detection, they specify operator dependency only statically in terms of the metamodeling formalism. This is too restrictive for the detection of complex operators since it limits possible reorderings dramatically. By defining dependency only in the context of an actual metamodel, our approach enables reordering into various normal forms which allow for the detection of complex operators.

Garcès et al. present an approach to automatically derive a model migration from metamodel differences [9]. The difference computation uses heuristics to detect also complex changes. Each heuristic refines the matching model, and is implemented by a model transformation in ATL. The transformation rules for detecting complex changes

are similar to the patterns presented in Section 4. Yet, the approach does not cover operator dependencies, was not able to detect complex changes in a Java case study, and does not address operator masking.

7 Discussion

Metamodeling Formalism. In this paper, we focus only on core metamodeling constructs that are most interesting for coupled evolution of metamodels and models. Concrete metamodeling formalisms like Ecore [23] or MOF [18] provide additional metamodeling constructs like packages, interfaces, operations, derived features, volatile features, or annotations. Since our approach allows for extension, we can add support for these constructs. Therefore, we need to provide additional primitive operators, define their preconditions, extend existing preconditions with respect to new invariants, derive additional complex operators, and define detection patterns for them.

Implementation. We implemented our approach prototypically in *Acoda*[1], a data model evolution tool for WebDSL [28], which is a DSL for web applications. Acoda offers an Eclipse plugin to seamlessly integrate into regular development. The plugin provides editor support for evolution traces (such as syntax highlighting, instant error marking and content completion), generation of SQL migration code, application of migrations to a database, and the evolution detection presented in this paper. The implementation uses an existing data model matching algorithm. We relied on rewrite rules in Stratego [27] to specify each step of the reconstruction algorithm, i.e., mapping data model changes to primitive operators, dependency ordering, normal form rewriting, complex operator detection, and masked operator detection. Acoda presents different evolution traces to the user, who can select and potentially modify the best match.

Trace Selection. Involving the user in the selection process prevents complete automation, but with a rich set of supported coupled operators, detection is likely to yield several suitable traces. Only the user can decide which migration is correct. We can assist this decision by presenting migrations of example models. Conversely, the user can assist the detection by giving examples for original and migrated models. The detection can then drop all traces which cannot reproduce the examples. Additionally, the user may choose to only consider information-preserving traces, thereby narrowing down the set of suitable traces.

Completeness. The set of primitive operators guarantees completeness at the metamodel level as it allows us to evolve any source metamodel to any target metamodel. Completeness at the model level is not feasible since it would imply that we can detect any model transformation between the instances of two arbitrary metamodels. Though, we can add more complex coupled operators to our detection. This increases the search space for both the user and for the detection. As for the user, we have a tradeoff between

[1] http://swerl.tudelft.nl/bin/view/Acoda

completeness and usability. There will be many similar operators with minor differences in their migration. Understanding and distinguishing operators becomes harder. In a number of real-life case studies, we identified the most common operators [13]. We propose to support only the detection of these operators and to leave rare cases to the user. As for the detection, supporting more complex operators increases the search space and we have a tradeoff between completeness and performance.

Performance. Besides the number of supported complex operators, detection performance is influenced by evolution size and mask depth, but not by metamodel size, which only affects the matching process. The GMF case study [12] showed us that a larger distance between original and evolved metamodel reduces the matching algorithm precision, making it more unlikely to still detect a good evolution trace. On the other hand, we found that an evolution between two commits to the repository could mostly be captured by 20 evolution steps. A preliminary case study of Acoda on part of the evolution of Researchr[2], a publication management system, showed the applicability of detection. Traces in Researchr between subsequent repository commits are short, hence we applied the detection to steps of ten subsequent commits, which yields traces up to 52 steps in length. A detection run generally takes several seconds and is significantly shortened when reducing the number of commits considered in a single detection run.

Acknowledgments This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. C. Brun and A. Pierantonio. Model differences in the eclipse modelling framework. *UPGRADE, The European Journal for the Informatics Professional*, 2008.
2. E. Casais. *Managing class evolution in object-oriented systems*, chapter 8, pages 201–244. Prentice Hall International (UK) Ltd., 1995.
3. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, EDOC*. IEEE, 2008.
4. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In *ICMT2009 - International Conference on Model Transformation*, LNCS, pages 35–51. Springer, 2009.
5. M. D. Del Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 963–970. ACM, 2007.
6. S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177. ACM, 2000.
7. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *MODELS 08: Model Driven Engineering Languages and Systems*, LNCS, pages 326–340. Springer, 2008.

[2] http://researchr.org

8. J.-M. Favre. Languages evolve too! changing the software time scale. In *IWPSE 05: Eighth International Workshop on Principles of Software Evolution*, pages 33–42. IEEE, 2005.
9. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *LNCS*, pages 34–49. Springer, 2009.
10. M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, pages 166–181, 2005.
11. M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *ECOOP 2009 - Object-Oriented Programming*. Springer, 2009.
12. M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of GMF. In *SLE*, volume 5969 of *LNCS*, pages 3–22. Springer, 2010.
13. M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE '10: Software Language Engineering, Third International Conference*, LNCS. Springer, 2010.
14. G. Kniesel and H. Koch. Static composition of refactorings. *SCP*, 52(1-3):9–51, 2004.
15. D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, pages 1–6, May 2009.
16. Y. Lin, J. Gray, and F. Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007.
17. D. Lopes, S. Hammoudi, and Z. Abdelouahab. Schema matching in the context of model driven engineering: From theory to practice. In *Advances in Systems, Computing Sciences and Software Engineering*, pages 219–227. Springer, 2006.
18. Object Management Group. Meta Object Facility (MOF) core specification version 2.0. http://www.omg.org/spec/MOF/2.0/, 2006.
19. D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *Proc. of the 9th European software engineering conference*, ESEC/FSE, pages 227–236. ACM, 2003.
20. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.
21. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, volume 3730 of *LNCS*, pages 146–171. Springer, 2005.
22. J. M. Sprinkle. *Metamodel driven model migration*. PhD thesis, Vanderbilt University, 2003.
23. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley, 2009.
24. X. L. Sun and E. Rose. Automated schema matching techniques: An exploratory study. *Research Letters in the Information and Mathematical Science*, 4:113–136, 2003.
25. Q. Tu and M. Godfrey. An integrated approach for studying architectural evolution. In *Program Comprehension, 2002. 10th International Workshop on*, pages 127–136, 2002.
26. S. D. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *MODELS 08: Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008.
27. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Springer, June 2004.
28. E. Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering*, LNCS. Springer, 2008.
29. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.
30. Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65. ACM, 2005.

TUD-SERG-2011-026
ISSN 1872-5392