Applying Higher-Order Model Transformations to the Semantic Lifting of Model Differences

Timo Kehrer, Manuel Ohrndorf, Udo Kelter and Maik Schmidt

Software Engineering Group
University of Siegen, Germany
{kehrer,mohrndorf,kelter,mschmidt}@informatik.uni-siegen.de

Abstract. Model transformation technologies have reached a high level of maturity. They are nowadays used beyond the traditional purpose of model transformation in model-driven engineering, where transformations are usually designed manually. In some new applications, transformation rules cannot be manually designed for two reasons: The transformation rules become very large, the effort to manually construct them is too high, and the rules must be generated on the fly. One example of such a new type of transformation occurs in the context of model differencing: Model comparison algorithms initially deliver low-level model differences; these low-level differences must be semantically lifted into representations of user-level edit operations. The necessary transformation rules can be automatically derived from rule-based implementations of the corresponding edit operations. This paper describes our approach of generating recognition rules, which are the core of the semantic lifting of low-level differences, from edit rules. We describe experiences with our implementation based on EMF Henshin.

Keywords: higher-order transformation, model comparison, model difference, semantic lifting

1 Introduction

Model transformations play a key role in model-driven engineering (MDE). Their traditional application is to transform abstract models of a system towards executable implementations for a specific runtime platform [16][14]. A lot of research has been investigated into the development of model transformation concepts and tools, which are often based on well-founded theories such as graph transformation.

Nowadays, model transformation technologies have reached a high level of maturity and are used beyond their traditional application in MDE: For example, further applications of model transformations comprise model synchronization [7], model query [17] or model refactoring [6]. They share in common with the traditional application that transformations are usually designed manually. In some new applications, transformation rules cannot be manually designed for two reasons: The transformation rules become very large, the effort to manually construct them is too high, and the rules must be generated on the fly.

One example of such a new type of transformation occurs in the context of model differencing: Model comparison algorithms initially deliver low-level model differences; these low-level differences must be semantically lifted into representations of user-level edit operations in order to be understandable for common tool users. Provided that model differences are represented based on the same technologies that are used to represent models, as for example the Eclipse Modeling Framework (EMF), in-place model transformations can be applied to the semantic lifting model differences, which mainly results in a pattern matching problem on the representation of a difference [11]. However, designing the necessary transformation rules is tedious and prone to errors for two reasons; the rules are getting complex very quickly and usually have to be specified for a large number of edit operations which are available for a given modeling language.

This paper describes our approach of deriving recognition rules from rule-based implementations of the corresponding edit operations. The generation of recognition rules from edit rules is designed as higher-order transformation (HOT) and contributes to the survey of application cases for HOTs given in [20]. We describe experiences with our implementation based on EMF Henshin which can be partly generalized to other applications.

The rest of the paper is structured as follows: Section 2 motivates the problem of low-level differences in the context of model comparison and outlines our rule-based approach to semantically lift these differences. Section 3 analyzes the transformation domain providing the context for the generation of recognition rules and briefly summarizes the technical context of our implementation. Subsequently, a specification of the transformation is given in Section 4, the design of an implementation based on EMF Henshin is illustrated in Section 5 and compared to a direct manipulation approach implemented in Java in Section 6. Section 7 concludes the paper.

2 Semantic Lifting of Model Differences

Model comparison is one of the most basic operations in the context of model versioning. It is commonly agreed that comparing textual representations of models does not produce usable results [2]. Instead, models must rather be compared on the basis of graph-based representations, usually denoted as abstract syntax graphs (ASG). Thus, syntactical comparison algorithms are basically faced with two models which are to be compared based on their ASG representation, e.g. implemented in EMF Ecore [5] or KM3 [9]. The usual processing pipeline consists of two sequential steps [15][4]: Initially, a matching procedure searches for pairs of corresponding model objects which are considered "the same" in both models. Subsequently, a difference is derived: objects and references not involved in a correspondence are considered to be deleted or created.

However, internal model representations contain many platform-specific details. Thus, even basic edit operations, which are available in model editors and which appear seemingly simple from a user's point of view, often lead to many low-level changes in the internal representation. Examples are described in [11][12][13]. Consequently, difference tools which are based on syntactical approaches to model comparison initially produce low-level differences. These are often not understandable for normal tool users which are not familiar with the details of metamodels and their platform-specific implementation.

This problem is intrinsic to all state-based model difference tools; a first approach to overcome it is presented in [11]: Low-level differences are represented based on EMF. Each edit operation being applied to a model leads to a pattern consisting of low-level changes that is characteristic for this edit operation. Thus, model transformation rules can be used to match change patterns and to transform low-level differences into representations of editing operations, which is denoted as *semantic lifting of model differences*. We implemented the approach using the rule-based model transformation framework EMF Henshin [8]. Provided that user-level edit operations are formally specified by edit rules, the recognition rules which are necessary for semantically lifting model differences can be automatically derived from their corresponding edit rules.

3 Transformation Domain

As Henshin provides an explicit transformation metamodel [1], it is well-suited to implement the generation of a recognition rule from its corresponding edit rule as higher-order transformation. The transformation, in the following referred to as *editR2recognR*, is endogenous [3] having the Henshin transformation metamodel as source and target metamodel. Input and output models are Henshin transformation rules[1]. An overview of the transformation is shown in Figure 1.

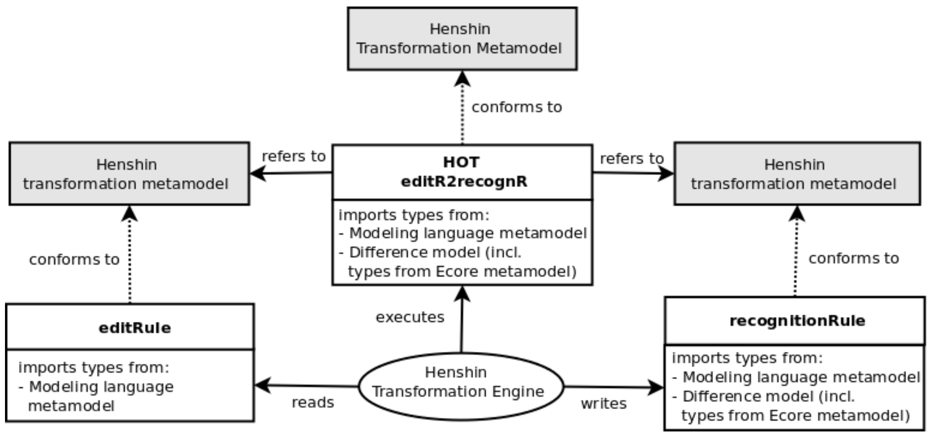

Henshin Transformation Metamodel
conforms to
Henshin transformation metamodel
refers to
HOT editR2recognR
imports types from:
- Modeling language metamodel
- Difference model (incl. types from Ecore metamodel)
refers to
Henshin transformation metamodel
conforms to
conforms to
editRule
imports types from:
- Modeling language metamodel
executes
Henshin Transformation Engine
reads
writes
recognitionRule
imports types from:
- Modeling language metamodel
- Difference model (incl. types from Ecore metamodel)

Fig. 1. Overview: Higher-order transformation *editR2recognR*

[1] To provide a complete configuration for our semantic difference lifter, *editR2recognR* is applied to the set of all edit rules implementing the basic user-level edit operations which are available for a given modeling language.

Figure 2 shows the clipping of the Henshin transformation metamodel which is relevant in the context of our transformation domain: Two graphs describe the left-hand-side (LHS) and the right-hand-side (RHS) patterns of a transformation rule, respectively. Parts of a pattern, i.e. nodes and edges, that are mapped via the respective LHS and RHS nodes are to be found and *preserved* when a rule is applied to an input model. Unmapped parts of an LHS pattern are to be found and *deleted*, unmapped parts of an RHS pattern are to be *created*.

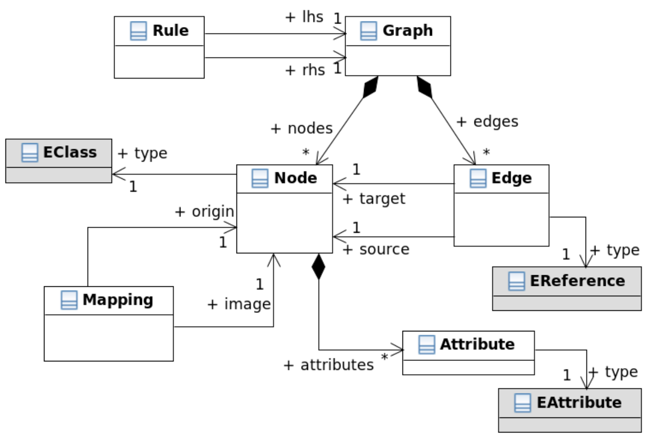


Fig. 2. Henshin transformation metamodel

Nodes, edges and attributes of pattern definitions are further typed over any EMF model. In terms on an edit rule, this is the metamodel related to a certain modeling language, e.g. UML class models, statemachines or any domain-specific language (DSL). A recognition rule imports additional type definitions: The semantic lifting transformation works on our EMF-based representation of model differences. Thus, graph patterns of recognition rules are typed over the *difference model* which is introduced in [11] and which is worth to briefly recall here. The structural parts of the difference model are shown in Figure 3. Two EMF models `modelA` and `modelB` that are being compared are represented by `ResourceSets`. A `Difference` is obtained by the three sequential steps introduced in Section 2:

1. Initially, a matching algorithm populates the correspondences representing the common parts of model A and model B. A `Correspondence` links an `EObject` of model A (`objA`) to an `EObject` of model B (`objB`). Correspondences between object references are given implicitly by their corresponding source and target objects.
2. Subsequently, low-level `Changes` are instantiated during difference derivation. We distinguish the following types of low-level changes:

   - An `AttributeValueChange` represents a value change of an attribute involving a pair of corresponding objects; `objA` is contained by model A, `objB` by model B.
   - A change of type `AddObject` represents the insertion of a new object, i.e. an object that is contained in model B but does not have a corresponding object in model A. Analogously, changes of type `AddReference` represent references that have been inserted.
   - Changes of types `RemoveObject` and `RemoveReference` represent the inverse of changes of types `AddObject` and `AddReference`, respectively.
3. Finally, in terms of semantically lifting a model difference, low-level changes that result from the invocation of a dedicated user-level edit operation are grouped by `SemanticChangeSets`. As each low-level change results from the application of exactly one edit operation, semantic change sets ideally provide a complete partitioning of the set of all low-level changes.

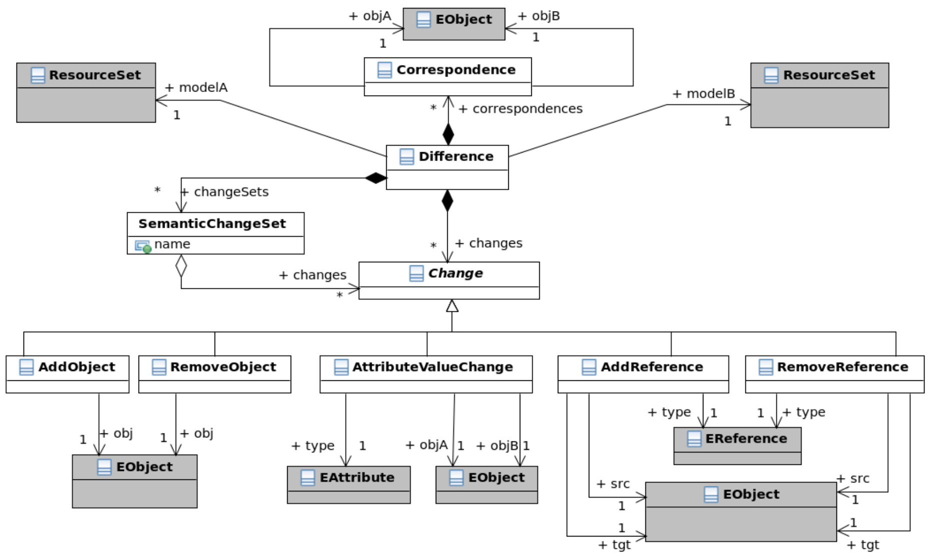


Fig. 3. EMF-based difference model

4 A By-Example Transformation Specification

This section provides a by-example specification of the higher-order transformation *editR2recognR*. We consider the edit operation "create generalization" which is available for UML class models. The example is based on the UML metamodel [18], the part which is relevant for our example is shown in Figure 4.

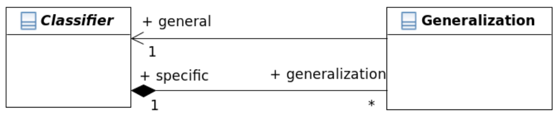


Fig. 4. Generalizations in the UML metamodel

Figure 5 (top) shows the Henshin rule *editR-createGeneralization*, which formally specifies the edit operation "create generalization". The rule is shown in the graphical syntax of the Henshin rule editor integrating LHS and RHS in a unified diagram. Stereotypes `<<create>>`, `<<delete>>` and `<<preserve>>` are utilized to denote creation, deletion and preservation, respectively [10].

According to the UML metamodel, generalization relationships are represented as objects of type `Generalization`. A generalization is owned by the `specific` classifier of a relationship and navigates to the more `general` classifier in the generalization hierarchy. Consequently, *editR-createGeneralization* specifies a `Generalization` object to be created and to be added as child of the specific classifier; by the creation of containment reference `generalization`. The general classifier is referred to via reference `general` which is to be created. The specific and the general classifier are to be preserved by the edit rule. They are provided as input parameters `s` and `g`, respectively.

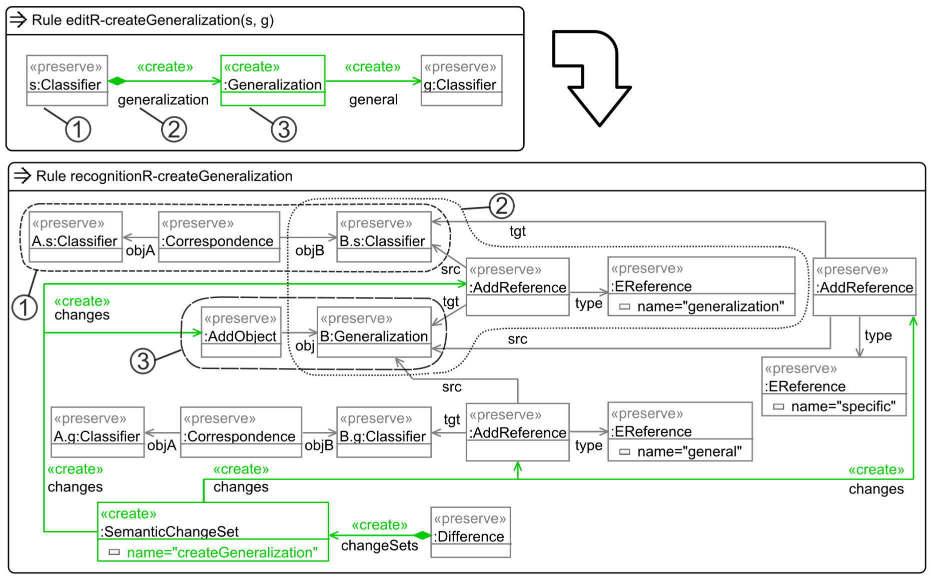


Fig. 5. Transformation of sample edit rule *editR-createGeneralization* to corresponding recognition rule *recognitionR-createGeneralization*

The application of *editR-createGeneralization* to a UML class model leads to a characteristic change pattern on our EMF-based difference representation. This change pattern has to be matched by the corresponding recognition rule *recognitionR-createGeneralization*, which is shown by the bottom part of Figure 5; the preserved parts of the rule specify the change pattern which has to be found. A change pattern to be matched is composed of partly overlapping atomic subpatterns, which can be directly derived from the corresponding edit rule:

Preserve patterns. Preserved parts, i.e. nodes and edges, providing the context of an edit rule lead to two types of preserve patterns; the **correspondence pattern** and the **preserved reference pattern**:

Model objects which are to be preserved by an edit rule are linked by a `Correspondence` in the difference representation. Two instances of the correspondence pattern occur in terms of our example. They represent the preserved classifiers between which the generalization relationship is to be created. The correspondence pattern which results from the preserved specific classifier ① is emphasized in Figure 5.

References which are to be preserved by an edit rule are not explicitly marked in the difference representation, i.e. no direct correspondence links are established between the respective references in model A and model B. However, corresponding references can be identified by their context, i.e. corresponding source and target objects. Thus, a preserved reference pattern usually incorporates two correspondence patterns. An instance of this type of preserve pattern does not occur in terms of our example.

Atomic change patterns. The types of atomic change patterns can be derived from the types of low-level changes that can be specified by an edit rule.

Firstly, we consider objects which are to be created or deleted by an edit rule. In terms of our example, the `Generalization` object which is to be created by the edit rule results in an instance of the **add object pattern**, which is emphasized and annotated with ③ in Figure 5; the change of type `AddObject` references the created generalization object which is contained by model B. The inverse low-level change, i.e. an object that is declared to be deleted by the edit rule, does not occur in terms of our example. However, the **remove object pattern** is structurally equal to the add object pattern.

Secondly, references can be declared to be created or deleted by an edit rule. For example, the reference `generalization`, which is to be created by our sample edit rule *editR-createGeneralization*, results in an instance of the **add reference pattern**, which is emphasized and annotated with ② in Figure 5. As one can see in Figure 5, the add reference pattern ② intersects with the patterns ① and ③. Generally, source and target objects of an add reference pattern are part of a correspondence pattern or an add object pattern and serve as "gluing points" of the respective pattern intersections.

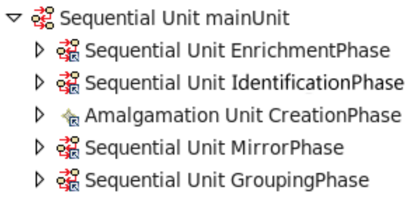
Two further add reference patterns, which are not explicitly marked in Figure 5, occur in terms of our example. The first one represents the reference `general` which is to be explicitly created by *editR-createGeneralization*. The second one represents the reference `specific` which is to be implicitly created as it is declared to be the opposite reference to `generalization` in the EMF-based implementation of the UML metamodel. Reference pairs that are declared as opposite to each other need to be specified in one direction only in terms of edit rules, as the existence of the opposite reference is a general EMF constraint. The **remove reference pattern**, of which no instance occurs in terms of our example, is structurally equal to the add reference pattern. Gluing points are provided by a correspondence pattern or a remove object pattern.

Finally, attribute value changes can be induced by edit rules. These do usually occur for edit rules which implement typical "setter operations". Thus, no occurrence is provided by our example. The object being the owner of a changed attribute is to be preserved. The value to which the attribute is to be set is specified by the RHS of the edit rule. The **attribute value change pattern** to be detected by a corresponding recognition rule consists of an `AttributValueChange` that references the type of the changed attribute and links the owner object of model A with the corresponding attribute owner of model B.

Semantic lifting. The application of a recognition rule semantically lifts all low-level changes of a change pattern by grouping them to a `SemanticChangeSet` which represents the invocation of the respective user-level edit operation. Semantic change sets which are created by our example recognition rule *recognitionR-createGeneralization* consist of four low-level changes. A semantic change set is finally inserted into the overall `Difference`.

5 Rule-based Transformation Design

This section describes the rule-based design of our implementation of *editR2recognR* based on EMF Henshin. The transformation algorithm is structured into five sequential phases which are shown in Figure 6. Each phase produces information which used in the subsequent one. The design of each phase is described during the remainder of this section[2].



Sequential Unit mainUnit
Sequential Unit EnrichmentPhase
Sequential Unit IdentificationPhase
Amalgamation Unit CreationPhase
Sequential Unit MirrorPhase
Sequential Unit GroupingPhase

Fig. 6. Main sequential phases of the transformation algorithm

[2] The implementation of all transformation rules and the complete rule application strategy are provided at: http://pi.informatik.uni-siegen.de/Projekte/sidiff/pipeline/semantic-lifting/hot/index.htm

As Henshin provides support for in-place transformations only, a technical helper structure is needed to integrate edit and recognition rule in a single EMF model. This is the primary purpose of the intermediate structure which is shown in Figure 7 and implemented in Ecore. To setup the transformation, an instance of type `Edit2Recognition` is created, which refers to the given input edit rule and the initially empty output recognition rule, which is also created during transformation setup time.

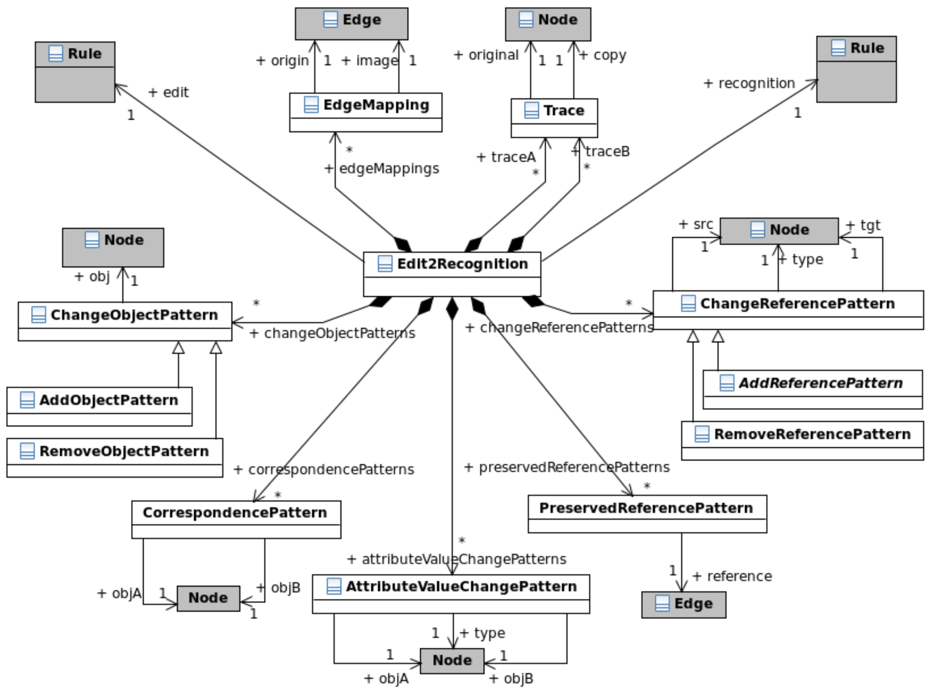
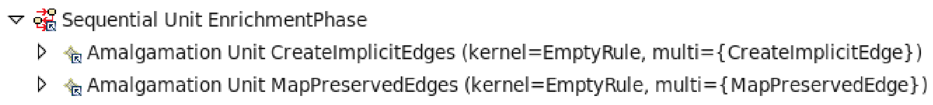


Fig. 7. Intermediate transformation helper structure

5.1 Enrichment Phase

In terms of edit rules, reference pairs that are declared as opposite to each other need to be specified in one direction only (cf. Section 4). In order to implement a consistent derivation of all reference-related patterns in later phases, reference pairs which are only specified partially by the given edit rule are completed. This function is implemented by the rule `CreateImplicitEdge`, which can be applied to all possible matches in parallel, which is scheduled by an amalgamation unit with empty kernel rule (s. Figure 8).

Subsequently, the intermediate structure is further populated with helper information facilitating the handling of preserved references: The information if a reference is to be preserved by an edit rule is not explicitly available. Thus, the rule `MapPreservedEdge` identifies preserved references by means of their context, i.e. source and target nodes which are mapped from LHS to RHS. The information is made explicit by an `EdgeMapping` which maps the `origin` edge of an edit rule LHS to the `image` edge of the edit rule RHS (s. Figure 7). `MapPreservedEdge` can be applied to all possible matches in parallel (s. Figure 8).



```
Sequential Unit EnrichmentPhase
  Amalgamation Unit CreateImplicitEdges (kernel=EmptyRule, multi={CreateImplicitEdge})
  Amalgamation Unit MapPreservedEdges (kernel=EmptyRule, multi={MapPreservedEdge})
```

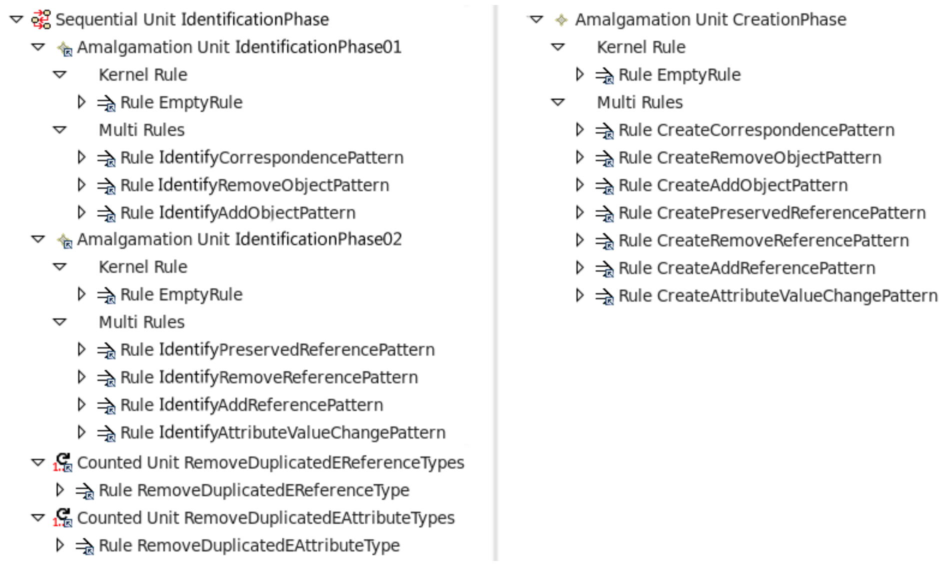
Fig. 8. Rule application strategy for enrichment phase

5.2 Identification Phase

The identification phase analyzes the edit rule and identifies all atomic preserve and change patterns which result from its application. For each change pattern that is identified, a helper object of the respective type of atomic pattern is instantiated and populated with the relevant information (cf. Figure 7).

Usually, several atomic change patterns are partly overlapping to form a complete change pattern which is induced by an edit operation invocation. The "gluing points" are provided by objects that are preserved, deleted or added. Gluing points that are to be matched by a recognition rule are created by the same transformation rules that create the helper objects which are used to identify atomic change patterns in an output rule; `IdentifyCorrespondencePattern`, `IdentifyAddObjectPattern` and `IdentifyRemoveObjectPattern`, respectively (s. "IdentificationPhase01" in Figure 9, left). These rules additionally create `Traces` from nodes of the edit rule to their counterparts in the recognition rule. Tracing information is necessary in order to correctly identify the gluing points for all overlapping atomic change patterns which are identified by the remaining rules of the identification phase, applied to all possible matches in parallel in "IdentificationPhase02".

If modeling language types are referenced multiple times by an edit rule, the schematic creation of change patterns may lead to multiple occurrences of recognition rule nodes representing the types. These have to be finally reduced to a single occurrence for each type due to the following reason: During semantic lifting, each type of the modeling language is represented by a single object of the respective metamodel. As recognition rules are injectively matched into the EMF-based representation of a given difference, multiple occurrences of these type objects have to be removed. This is achieved by a repeated application of the rules `RemoveDuplicatedEReferenceTypes` and `RemoveDuplicatedEAttributeTypes` which terminate as soon as all duplicates are removed (s. Figure 9, left).



```
▽ Sequential Unit IdentificationPhase
  ▽ Amalgamation Unit IdentificationPhase01
    ▽ Kernel Rule
      ▷ Rule EmptyRule
    ▽ Multi Rules
      ▷ Rule IdentifyCorrespondencePattern
      ▷ Rule IdentifyRemoveObjectPattern
      ▷ Rule IdentifyAddObjectPattern
  ▽ Amalgamation Unit IdentificationPhase02
    ▽ Kernel Rule
      ▷ Rule EmptyRule
    ▽ Multi Rules
      ▷ Rule IdentifyPreservedReferencePattern
      ▷ Rule IdentifyRemoveReferencePattern
      ▷ Rule IdentifyAddReferencePattern
      ▷ Rule IdentifyAttributeValueChangePattern
  ▽ Counted Unit RemoveDuplicatedEReferenceTypes
    ▷ Rule RemoveDuplicatedEReferenceType
  ▽ Counted Unit RemoveDuplicatedEAttributeTypes
    ▷ Rule RemoveDuplicatedEAttributeType
```

```
▽ Amalgamation Unit CreationPhase
  ▽ Kernel Rule
    ▷ Rule EmptyRule
  ▽ Multi Rules
    ▷ Rule CreateCorrespondencePattern
    ▷ Rule CreateRemoveObjectPattern
    ▷ Rule CreateAddObjectPattern
    ▷ Rule CreatePreservedReferencePattern
    ▷ Rule CreateRemoveReferencePattern
    ▷ Rule CreateAddReferencePattern
    ▷ Rule CreateAttributeValueChangePattern
```

Fig. 9. Rule application strategy for identification phase (left) and creation phase (right)

5.3 Creation Phase

The creation phase completes the atomic change patterns that have been identified and been partly created by the identification phase: Nodes representing the low-level changes oy type `Correspondence`, `AddObject`, `RemoveObject`, `AddReference` and `RemoveReference` are created and embedded into the change patterns. As these change objects have no references across each other, no sequential dependencies have to be considered. Thus, all rules of the creation phase can be applied in parallel to all possible matches (s. Figure 9, right).

5.4 Mirror Phase

Please note that up to this state of the transformation, only the LHS part of a change pattern to be matched and preserved by the output recognition rule have been created. This keeps transformation rules of the identification and creation phase at a manageable size. The RHS part of a change pattern, which is isomorphic to the RHS part, is created during the mirror phase. The rule `MirrorNodesLHStoRHS` simply copies all nodes of the LHS to the RHS of the recognition rule and instantiates the respective node `Mappings` (cf. Figure 2). Afterwards, attributes and edges can be copied to the LHS by the rules `MirrorAttributeLHStoRHS` and `MirrorEdgesLHStoRHS` in parallel (s. Figure 10).

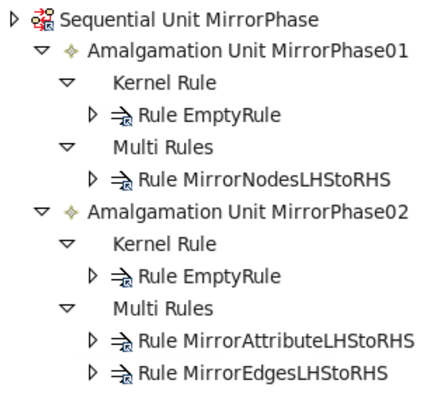

Sequential Unit MirrorPhase
Amalgamation Unit MirrorPhase01
Kernel Rule
Rule EmptyRule
Multi Rules
Rule MirrorNodesLHStoRHS
Amalgamation Unit MirrorPhase02
Kernel Rule
Rule EmptyRule
Multi Rules
Rule MirrorAttributeLHStoRHS
Rule MirrorEdgesLHStoRHS

Fig. 10. Rule application strategy for mirror phase

5.5 Grouping Phase

Finally, the SemanticChangeSet to be created by the recognition rule has to be instantiated and to be connected with all low-level change nodes of the atomic change patterns. The name of the semantic change set is derived from the name of the input edit rule.

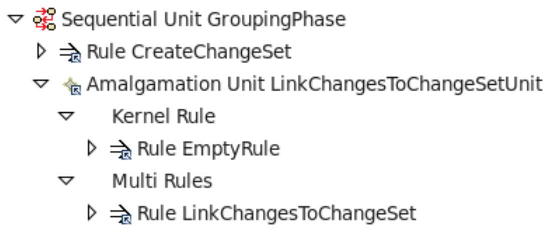

Sequential Unit GroupingPhase
Rule CreateChangeSet
Amalgamation Unit LinkChangesToChangeSetUnit
Kernel Rule
Rule EmptyRule
Multi Rules
Rule LinkChangesToChangeSet

Fig. 11. Rule application strategy for grouping phase

6 Comparison to a Direct Manipulation Approach

A very brief specification sketch of designing *editR2recognR* as direct manipulation approach in imperative logic [3] is given in [11]. We implemented the direct manipulation in Java[3] independently from our implementation in Henshin. A comparison of both implementation design reveals, that almost the same "functions" are used (s. Table 1).

[3] The source code is freely available at: http://pi.informatik.uni-siegen.de/Projekte/sidiff/pipeline/semantic-lifting/hot/index.htm

A design pattern that can be observed for the Henshin-based realization is the sequential enrichment of helper data. This is used to "emulate" procedure calls providing return values of complex data types. For example, the creation of atomic change patterns can be decomposed into two logical functions: Firstly, their cause have to be detected by an analysis of the edit rule before the pattern specification is actually synthesized within the recognition rule. For a concrete type of atomic change pattern, e.g. the correspondence pattern, this is implemented straight-forward in Java: The method `createCorrespondencePatterns()` simply calls the utility method `getLHSIntersectRHSNodes()`, which implements a query that returns the "preserved" nodes of an edit rule. In Henshin, the same utility function is realized by the rule `IdentifyCorrespondencePattern` which creates helper data being later used by the rule `CreateCorrespondencePattern`.

Table 1. Comparison to a direct transformation approach implemented in Java

Phase	**Henshin rule**	**Java method / utility class**
EP:	CreateImplicitEdge	createImplicitEdges()
	MapPreservedEdge	isEdgeMapped()
IP:	MatchCorrespondencePattern	getLHSIntersectRHSNodes()
	IdentifyAddObjectPattern	getRHSMinusLHSNodes()
	⋮	⋮
	RemoveDuplicatedEReferenceTypes	-
	RemoveDuplicatedEAttributeTypes	-
CP:	CreateCorrespondencePattern	createCorrespondencePatterns()
	CreateAddObjectPattern	createAddObjectPatterns()
	⋮	⋮
MP:	MirrorNodesLHStoRHS	-
	MirrorAttributeLHStoRHS	-
	MirrorEdgesLHStoRHS	-
GP:	CreateChangeSet LinkChangesToChangeSet	createChangeSet()
	-	ModelHelper.java
	-	ModelHelperEx.java
	-	NodePair.java

Two Henshin rules that do not have any corresponding Java method implementing the same functionality are `RemoveDuplicatedEReferenceTypes` and `RemoveDuplicatedEAttributeTypes` In terms of the Java implementation, duplicate type nodes are never created because a simple condition checks whether these type nodes already exist when an atomic change pattern is synthesized (cf. lines 258, 340 and 405 in `EditRule2RecognitionRule.java`). In Henshin, the same behavior could be realized using conditional units and implementing a loop processing instead of a parallel creation of atomic change patterns. In this case, `RemoveDuplicatedEReferenceTypes` and `RemoveDuplicatedEAttributeTypes` would be obsolete. However, the total number of rules needed by this transformation design would be significantly higher: For each rule of the identification phase, three rules would be required, i.e. “if rule”, “then rule” and “else rule” of the respective conditional unit.

Further Henshin rules which have no corresponding Java methods implementing the same function are `MirrorNodesLHStoRHS`, `MirrorAttributeLHStoRHS` and `MirrorEdgesLHStoRHS`, which facilitate the output rule synthesis; initially only the LHS of the output recognition rule has to be synthesized which is later mirrored onto the RHS. In Java, the consistent handling of LHS and RHS graphs of a recognition rule is implemented using data abstractions. These data abstractions provide a high-level API to manipulate internal, fine-grained representations of Henshin rules. For example, the utility function `createPreservedNode()` of class `HenshinRuleAnalysisUtilEx` and the wrapper class `NodePair` are provided in order to encapsulate preserved nodes: Creation and management of LHS node, RHS node and the according mapping are handled transparently to a programmer.

7 Conclusion

In this paper, we introduced a new application of higher-order model transformations: Complex recognition rules, which are necessary to semantically lift low-level model differences into representations of user-level edit operations, are generated from their corresponding edit rules, which are much easier to design. In some cases, edit rules are even readily available, e.g. for UML class models [19].

None of the categories of the classification scheme for higher order transformations given in [20] matches perfectly to our application. The most closely related category is transformation type c) as it requires input *and* output models to be model transformations themselves. However, our application does not compose or decompose transformations. Thus, the proposed design patterns *internal* and *external (de-)composition* are not applicable to our use case.

Instead, our rule-based transformation design based on EMF Henshin reveals that patterns that are known form procedural programming can help to structure the transformation design. Utility functions providing structurally complex return types can be realized by the definition of an intermediate helper structure. This pattern can be generalized and applied to all types of rule-based transformations. Two other patterns that are used specifically apply to higher-order transformations: Firstly, providing the explicit information of preserved edges facilitates queries on the rule serving as input of the HOT. Secondly, the mirroring of preserved parts from LHS to RHS facilitates the output rule synthesis. The second pattern is very specific to the transformation domain described in this paper as most parts of a recognition rule are preserved. However, it can be generalized to all HOTs generating transformation rules that are mainly used for pattern matching purposes.

References

1. Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems (MoDELS 2010); Springer, LNCS 6394; 2010
2. Bendix, L.; Emanuelsson, P.: Collaborative Work with Software Models - Industrial Experience and Requirements; Proc. Intl. Conf. Model-Based Systems Engineering, Haifa; 2009
3. Czarnecki, K; Helsen, S.: Feature-based survey of model transformation approaches; p.621-646 in: IBM Systems Journal, Vol. 45, No. 3; 2006
4. EMF Compare; http://www.eclipse.org/emf/compare
5. EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf
6. EMF Refactor; http://www.eclipse.org/modeling/emft/refactor/
7. Giese, H.; Wagner, R.: From model transformation to incremental bidirectional model synchronization p.21-43 in: Software and Systems Modeling, Vol. 8, No. 1; 2009
8. Henshin; http://www.eclipse.org/modeling/emft/henshin/
9. Jouault, F.; Bézivin, J.: KM3: a DSL for Metamodel Specification; p. 171—185 in: Proc. of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems; LNCS; 2006
10. Jurack, S; Tietje J.: Solving the TTC 2011 Reengineering Case with Henshin; p. 181-203 in: Proc. of the 5th Transformation Tool Contest (TTC); EPTCS; 2011
11. Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: Proc. 26th IEEE/ACM Intl. Conf. Automated Software Engineering (ASE 2011); ACM; 2011
12. Kelter, U.: Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen; p.117-128 in: Proc. Software Engineering (SE 2010); GI LNI 159; 2010
13. Kelter, U.; Schmidt, M.: Comparing State Machines; p.1-6 in: Proc. ICSE Workshop on Comparison and Versioning of Software Models (CVSM 2008); ACM; 2008
14. Kleppe, A.;Warmer, J.; Bast, W.: MDA Explained, The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003
15. Kolovos, D.S.; Ruscio, D.D.; Pierantonio, A.; Paige, R.F.: Different Models for Model Matching: An Analysis Of Approaches To Support Model Differencing; p.1-6 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE; 2009
16. MDA Guide Version 1.0.1; OMG, http://www.omg.org; 2003
17. OMG: MOF Query/View/Transformation; http://www.omg.org/spec/QVT/1.0/; 2008
18. OMG: Unified Modeling Language; Superstructure specification version 2.3; http://www.omg.org/spec/UML/2.3/Superstructure/PDF/; 2010
19. Rindt, M.; Kehrer, T.; Kelter, U.; Pietsch, P.: Henshin Rules for Edit Operations in Class Diagrams; Technical Report, QuDiMo Project, University of Siegen, http://pi.informatik.uni-siegen.de/qudimo/download/RiKKP2011.pdf; 2011
20. Tisi, M.; Jouault, F.; Fraternali, P.; Ceri, S.; Bézivin J.: On the Use of Higher-Order Model Transformations; p.18-33 in: Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA); Springer; 2009