Towards Refactoring of Graph Transformation Systems – Long Version

Gabriele Taentzer[1], Claudia Ermel[2], Reiko Heckel[3],Thorsten Arendt[1]

[1] Philipps-Universität Marburg, Germany
{taentzer, arendt} @informatik.uni-marburg.de
[2] Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de
[3] University of Leicester, UK
reiko@mcs.le.ac.uk

Abstract. The more graph transformations are applied in various application domains, the more questions about the quality of these applications arise. In this paper, we present a first approach towards improving the quality of graph transformation systems based on refactorings. After discussing possible quality aims for graph transformation systems, a first selection of refactorings is presented showing a variety of potential improvements of graph transformation systems. Each refactoring is presented in a systematic way including an explanation how the quality is improved, a description of its pre- and post-conditions, a possible refactoring strategy, and finally an example. Moreover, we discuss how the presented refactorings preserve semantics. All discussed refactorings can be directly implemented in Henshin, a model transformation engine based on graph transformation concepts, using Henshin in combination with the Eclipse plug-in EMF Refactor.

1 Introduction

Graph transformation is being applied to various domains of computer science such as domain-specific language engineering, model and program transformation, concurrent system design, service-oriented and self-adaptive systems. Moreover, it has also been applied to areas like logistics, biology, multimedia, etc. (For an overview see e.g. [8,21,11].)

According to that wide variety of application areas, the group of graph transformation system developers grows continuously. Especially, the group of newcomers not coming from established sites of the graph transformation community is growing. Thus, experiences on the development of graph transformation systems cannot easily be forwarded to new developers. Therefore, it is worthwhile to make expert knowledge on how to write graph transformation systems explicit. We start this task in this paper by using a well-known technique: refactoring. Originally, refactoring means to improve program structures without changing their behavior [12]. Meanwhile, this technique has also been used to improve other kinds of software artifacts such as models. Here, we consider the refactoring of graph transformation systems and present a first collection.

Since graph transformation systems can be considered as some kind of software and system models, we aim to adapt well-known quality assurance techniques to this kind of models. In [3] we present a two-stage quality assurance process where first quality aspects are determined and domain- and project-specific quality assurance techniques based on metrics, smells, and refactorings are specified. Thereafter, they are applied to specific models as long as their quality is not good enough. Adapting this approach to graph transformation systems, we need quality aspects for graph transformation systems and have to think about well-suited metrics, smells, and refactorings that make existing knowledge about how to write graph transformation systems explicit. We implemented supporting tools for this quality assurance process helping us to automate its specification and execution to a large extent[4]. They will be reused in the context of graph transformation systems improvements.

Our selection of refactorings is guided by mainly two aspects: First of all, we concentrate on what we consider the kernel features of graph transformation systems which are typed, attributed graphs allowing node type inheritance, and rules with left- and right-hand sides as well as positive and negative application conditions. Furthermore, refactorings have been selected according to selected quality aspects. We concentrate on conciseness and changeability of graph transformation systems as well as on the simplicity of used transformation features. Quality improvements are indicated by a variety of metrics.

The main contribution of this paper is a first collection of useful refactorings for graph transformation systems, described in a systematic way. To integrate these refactorings into a systematic quality assurance process, we define main quality aspects for graph transformation systems and smells (indicators of low quality) based on metrics. Each refactoring description is presented by means of a short description, an explanation in which ways the quality is improved, a description of its pre- and post-conditions, a possible refactoring strategy, and an example. The presented collection shows a variety of refactorings serving different quality aims.

While the collection of refactorings is presented pretty independently of a specific graph transformation approach (although influenced by the algebraic approach [7] as presented in AGG), we discuss their implementation on the basis of Henshin, a model transformation engine for the Eclipse Modeling Framework (EMF) [22] based on graph transformation concepts. Henshin transformation systems can be refactored in a straightforward way, since the Henshin transformation model is an EMF model and we developed a tool for EMF model refactoring. The implementation of Henshin refactorings is model-driven meaning that the specification of a Henshin refactoring can be done by Henshin again and is translated to Java code thereafter.

Structure of the paper: In Section 2, the kernel features of algebraic graph transformation systems are recalled. Valuable quality aims for graph transformation systems are motivated in Section 3 . Section 4 presents our refactoring

[4] See http://www.mathematik.uni-marburg.de/~arendt/scico/ for more information on the quality assurance process and to download the tools.

collection, and different forms of semantics preservation are discussed in Section 5. In Section 6 we present the implementation of a sample refactoring using the tool HENSHIN. Finally, we consider related work and conclude the paper.

2 Kernel Features of Graph Transformation Systems

In object-oriented modeling, graph transformation has proven to be a suitable formal framework for a controlled manipulation and evolution of models [20,8]. As inheritance is an important and widely spread concept for the elegant expression of hierarchy [6,17], *typed (attributed) graphs with inheritance* have been introduced enabling a formal description of hierarchy[16,7]. For transformations of typed (attributed) graphs with inheritance, called abstract transformations, abstract nodes in transformation rules can be refined to concrete ones in the model.

In this section, we introduce the notion of typed (attributed) graphs and define algebraic graph transformation with NACs [7].

2.1 Typed Graphs

Definition 1 (Graph). *A graph $G = (G_N, G_E, s_G, t_G)$ consists of a set G_N of nodes, a set G_E of edges, as well as source and target functions $s_G, t_G : G_E \to G_N$.*

Remark 1. The main idea of an *attributed graph* is that nodes and edges of a graph may carry attribute values. These are defined in an underlying data structure, given by an algebra, where only distinguished attribute value sorts are used for attribution. For the formal definition, the attributes are represented by edges into the corresponding data domain, which is given by a node set called *attribute nodes*. For the most of the presented refactorings of graph transformations in this paper, the formalization of attributes plays a minor role and is neglected in the following due to space limitations. But all concepts and definitions have also been extended to the case of typed attributed graphs and graph transformations (see [7]).

Definition 2 (Type graph with inheritance). *A* type graph with inheritance *$TG = (T, I, A)$ consists of a graph $T = (T_N, T_E, s_T, t_T)$, called* type graph*, a graph $I = (I_N, I_E, s_I, t_I)$, called* inheritance graph *and a set $A \subseteq T_N$ of* abstract nodes*. We require that $I_N = T_N$ and $I_E \cap T_E = \emptyset$. Moreover, I has to be a forest, i.e. acyclic and for all $e, e' \in I_E$ we have that $s_I(e) = s_I(e')$ implies $e = e'$.*

For each node n in T_N, the inheritance clan *is defined by $clan_I(n) = \{m \mid (m, n) \in I\}$. We write $m < n$ for $m \in clan_I(n)$ and say that m* inherits from *n.*

Definition 3 (TG-typed graph). *Given a type graph with inheritance $TG = (T, I, A)$, a tuple $G^T = (G, type)$ of a graph G together with a typing morphism*

$type_G$: $G \to TG$ is called a TG-typed graph *if the following condition holds:*

(correct typing) *The typing morphism $type_G$: $G \to TG$ consists of a pair of functions $(type_{G_N} : G_N \to T_N, type_{G_E} : G_E \to T_E)$ with*

$$type_{G_N} \circ s_G(e) < s_T \circ type_{G_E}(e) \text{ and } type_{G_N} \circ t_G(e) < t_T \circ type_{G_E}(e).$$

The typing morphism $type_G$ is called concrete *if $(type_{G_N}(n) \notin A \quad \forall n \in G_N$.*

Example 1 (TG-typed graph). Consider type graph TG of a Phone model in the upper leftmost screen shot in Figure 1. Type *Phone* has two sub-types *MobilePhone* and *FixedPhone* inheriting from *Phone*. Both sub-type nodes have a Boolean attribute called *isIdle*. All four graphs shown to the right of TG are TG-typed instance graphs. They contain one node each. In the two left instance graphs, the node is typed by *MobilePhone*, and in the two right instance graphs, the node is typed over node type *FixedPhone*. The respective attribute values of attribute *isIdle*, which are *true* or *false* in the instance graphs, are of data type *Boolean*.

2.2 Typed Graph Transformation

In order to define graph transformation rules for typed graphs, we first define typed graph morphisms, i.e. graph morphisms that are structure and type compatible. Typed graph morphisms are needed to define graph rules and their application to typed graphs.

Definition 4 (TG-typed graph morphism).
Given a type graph TG and two TG-typed graphs G, H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a valid TG-typed graph morphism $f : G \to H$ if it has the following properties:

(1) (structure compatibility):
$f_N \circ s_G(e) = s_H \circ f_E(e)$, $f_N \circ t_G(e) = t_H \circ f_E(e)$, and

(2) (type compatibility):
$type_{H_N}(f_N(n)) < type_{G_N}(n)$ for all $n \in G_N$ and $type_{H_E} \circ f_E = type_{G_E}$. (We say that $type_H \circ f$ is finer *than $type_G$.)*

If f_N and f_E are inclusions, G is called a subgraph of H, denoted by $G \subseteq H$.

Now we can define transformation rules with negative application conditions (NACs) modeling graph transformation steps over typed graphs. A rule consists of a left-hand side graph L (the precondition for the transformation, defining the pattern to be found in the graph), a right-hand side graph R (the postcondition, defining the actions to be performed on the pattern), and an intersection graph with morphisms relating L to R. A NAC further restricts the applicability of a rule. It consists of an extension of L by a structure which is prohibited to occur in the graph.

The conditions in Definition 5 are due to the use of abstract types for rule elements and express that *(1)* retyping of elements is not allowed, *(2)* newly created object nodes must not be typed over abstract node types, and *(3)* nodes in negative application conditions of a rule may be typed finer than in the rule's left-hand side.

Definition 5 (Transformation rule). *A transformation rule typed over a type graph* $TG = (T, I, A)$ *is given by* $p = (L \supseteq K \subseteq R,\ type,\ NAC)$*, where* L, K *and* R *are* TG*-typed graphs called left-hand side (L), intersection (K), and right-hand side (R),* $type$ *is a triple of typing morphisms* $type = (type_L\colon L \to TG,\ type_K\colon K \to TG,\ type_R\colon R \to TG)$*, and* NAC *is a set of pairs* $nac_i = (N_i, type_{N_i}), i \in \mathbb{N}$ *with* $L \subseteq N_i$*, and* $type_{N_i}\colon N_i \to TG$ *a typing morphism, such that the conditions* (1)-(3) *below hold. We denote the sets of deleted / newly created elements, resp., by* $L'_X := L_X - K_X$ *and* $R'_X := R_X - K_X$ *(with* $X = N, E$*).*

(1) (no retyping of elements in rules): $type_L \supseteq type_K \subseteq type_R$

(2) (created nodes are concretely typed): $type_{R_N}(R'_N) \cap A = \emptyset$

(3) (NACs are finer typed than LHS): $\forall (N_i, type_{N_i}) \in NAC :$
$\forall n \in L_N, e \in L_E : type_{N_i}(n) < type_L(n)$ *and* $type_{N_i}(e) = type_L(e)$

Note that in the transformation rules introduced in the following, we omit the intersection graph K. Instead, we show a mapping from elements in L to elements in R by (the elements that are preserved by the rule). Graph K then can be derived from the mapping of L to R since K contains all elements occurring in both L and R.

Another (implicit) application condition for a graph transformation rule is the so-called *dangling condition* which allows the application of a rule only if adjacent edges of nodes to be deleted occur in the L, thus are also scheduled for deletion.

Applying a graph transformation rule to a graph G, a match (typed graph morphism) from L to G has to be found, i.e. the LHS has to be mapped such that the dangling condition, the typing constraints and all NACs are satisfied. The rule application yields a unique result graph H that is constructed by taking the original graph G, deleting all items in the match of L but not in R, and then adding all R-items not being in L, to G disjointly.

In Definition 6, we state that a transformation rule may be applied to a graph if *(1)* nodes are deleted only if no dangling edges remain, *(2)* different rule items may be identified, i.e. matched to one and the same graph item only if they are both preserved, and *(3)* all NACs are fulfilled.

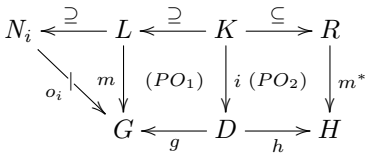
Definition 6 (Matching and application of transformation rules). *Let* $p = (L \supseteq K \subseteq R,\ type,\ NAC)$ *be a transformation rule as defined in Definition 5,* G *a* TG*-typed graph with* $type_G\colon G \to TG$ *being a concrete type morphism, and* $m\colon L \to G$ *a* TG*-typed graph morphism. Then,* m *is a match with respect to* p *and* G *if the following conditions hold:*

(1) (dangling condition): $\forall n \in L'_N : \nexists e \in G_E - m_E(L_E)$ *with* $s_G(e) = m_N(n) \vee t_G(e) = m_N(n)$

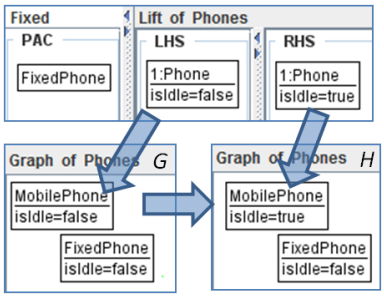
(2) (identification condition): $\forall x_1, x_2 \in L_N$ *with* $m_N(x_1) = m_N(x_2) : x_1, x_2 \in K_N$ *(analogously for edges)*

(3) (m satisfies NAC): for each $nac_i = (N_i, type_{N_i}) \in NAC, i \in J$ *there does not exist a TG-typed graph morphism* $o_i \colon N_i \to G$ *such that* $o_i|_L = m$, $o_i|_{N_i - L}$ *is injective and* $type_G \circ o_i$ *is finer than* $type_{N_i}$.

By $G \overset{p,m}{\Longrightarrow} H$ *we denote the direct transformation where rule p is applied to G at match m leading to the result graph H. The formal construction of a direct transformation is a double-pushout (DPO) which is shown in the diagram below with pushouts* (PO_1) *and* (PO_2) *in the category of typed graphs. Graph D is the intermediate graph after removing* $m(L)$*, and H is constructed as gluing of D and R along K. Typing morphisms are not shown. The transformation definition is unique up to isomorphism [7].*



Example 2 (Direct transformation).
The diagram to the right shows an example, where rule *Lift* is applied to graph G containing one *MobilePhone* node and one *FixedPhone* node. The match maps the abstract *Phone* node to the concrete *MobilePhone* node. The PAC is satisfied since there is an additional *FixedPhone* node in G. The rule application results in the direct graph transformation $G \Longrightarrow H$, where the attribute value of the *MobilePhone* node is set to *true* in H. Note that due to the PAC rule *Lift* cannot be mapped to the *FixedPhone* in G.



3 Quality Aspects and Smells for Graph Transformation Systems

As with software models in general, correctness of a graph transformation system can be considered at a syntactic level, w.r.t. a language definition, as well as semantically, w.r.t. the requirements of the system to be developed and the application domain. Refactorings should neither affect syntactic nor semantic correctness because they have to preserve both well-formedness and semantics of models.

3.1 Quality aspects

As for other software artifacts, the *correctness* of a graph transformation system is defined w.r.t. the transformation language used and the understanding of the

domain. While language correctness is considered syntactical the understanding forms the system's semantics. Refactorings are supposed to produce results that are again syntactically correct and preserve the system semantics, i.e. the correctness is preserved.

Conciseness is concerned with the compactness of systems which should be presented on the right abstraction level. To measure conciseness we can consider the numbers of node and edge types, rules, rule elements, pre-conditions, etc. The smaller these numbers the more concise is the system.

By *Simplicity*, we mean the simplicity of the transformation approach and not of individual transformation systems (which we consider by conciseness). We are mainly interested in the number and selection of transformation features used to specify a graph transformation system.

A graph transformation system is *changeable*, if it can be evolved rapidly and continuously. Conciseness and moreover, low redundancy and low coupling of modules, seem to be necessary prerequisites for changeability.

A graph transformation system is *comprehensible* if it is understandable by the intended users. *Comprehensibility* is increased if a system is simple, concise, and structured enough to grasp its design. Moreover, comprehensibility is also influenced by the quality of used graph layouts, however, we do not consider this quality aspect throughout this paper.

3.2 Selected smells

In the following, we present a small set of selected smells for graph transformation systems that are all based on some metrics. Smells report on suspicious system parts which should be inspected closer. Since we are mainly interested in the conciseness of graph transformation systems and the simplicity of the used approach, we investigate size and redundancy on the one hand, and the kind of used features on the other hand.

Smell "Large Rule"

Description: A rule specifies a specific graph pattern and replaces it. Thus, it should handle a single aspect of the behavior. A rule having too many elements (nodes or edges) seem to care about too many different concerns.
Metric: This smell can be easily detected by counting the number of elements in a given rule. For simplicity, we do not count attributes.
Usable refactorings: Extract pre-condition, Loop edges to Boolean attribute, Split rule into set of action rules;
Affected quality aspects: Large rules do not represent a good modular design and can contain redundant information. Conciseness and comprehensibility might be affected.

Smell "Redundant Attributes and Rules"

Description: Several node types have equal attributes (with equal names and types). Furthermore, there might be several rules which differ in used node types only.

Metric: This smell can be detected by comparing the number of all attributes and the number of attributes with equal names and types.
Usable refactorings: Pull up attribute;
Affected quality aspects: Redundant information blows up the type graph and potentially also the rule set. It affects the conciseness, comprehensibility, and changeability of graph transformation systems.

Smell "Loop Edges"

Description: Loop edges are often used to encode Boolean information about nodes.
Metric: This smell can be detected by counting the number of loops of an edge type.
Usable refactorings: Loop edges to Boolean attribute;
Affected quality aspects: Modeling Boolean information about nodes by loop edges blows up graphs more than Boolean attributes. Thus, loop edges affect the conciseness of graph rules and maybe also their comprehensibility.

Smell "Advanced Features"

Description: Advanced features like pre-conditions, attributes, and control constructs are used for specification.
Metric: This smell can be detected by looking for advanced features, i.e. counting the number of pre-conditions, attribute types, control constructs, etc.
Usable refactorings: Inline pre-condition, Boolean attribute to loop edges;
Affected quality aspects: Advanced features decrease the simplicity of the transformation approach.

4 Selected Refactorings

In this section, we present a collection of refactorings for graph transformation systems, each described in a systematic way. The collection mirrors our experiences in the application of graph transformation to various domains. It is by far not complete (in fact, it can never be so), but it shows a pretty representative selection of refactorings serving different quality aims. Refactoring "Pull up Attribute" reduces the amount of redundancy wrt. to attribute definitions and potentially also reduces the number of rules, while "Loop Edges to Boolean Attribute" and "Extract Pre-Condition" reduce the number of rule elements and thus improve the conciseness. "Split Rule into Set of Action Rules" influences the concurrency of modeled actions.

Here, we do not present a refactoring which is probably most useful, i.e. the renaming of graphs, rules, types, etc., since it is obvious. Furthermore, most of the refactorings presented below come along with an inverse one that can take back the original refactoring effect. E.g. the inverse refactoring of "Pull Up Attribute" is "Push Down Attribute" which might be useful to prepare a variation of attribute definitions in subtypes. The inverse of "Extract Pre-Condition",

"Inline Pre-Condition", is able to simplify the approach by getting rid of pre-conditions, and refactoring "Boolean Attribute to Loop Edges" can also simplify the approach by reducing the number of attributes or even getting rid of them. In this article, inverse refactorings are not presented in detail, due space limitations.

4.1 Refactoring "Pull Up Attribute"

An attribute is pulled up to a more abstract type. It is optional to compact the rule set.

Input parameter: Name of the attribute and name of the node type it shall be pulled up to;

Example: Phones are refined into fixed and mobile phones. Both are attributed by Boolean attribute *isIdle*. Two rules describe the lifting of fixed resp. mobile phones (see the upper part of Figure 1). After the refactoring, attribute *isIdle* is pulled up to node type *Phone*. In the lower part of Figure 1, a lift rule for phones in general is shown which is merged from the two original lift rules.

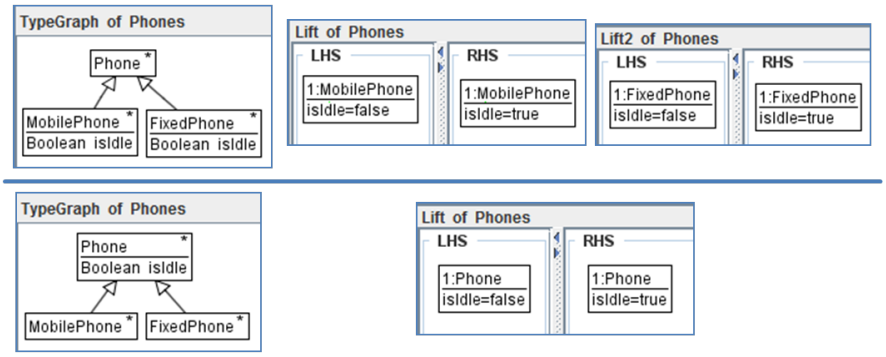


Fig. 1: Before and after refactoring "Pull Up Attribute"

Pre-condition: The attribute is contained in all subtypes of the selected node type and it has always the same type. Moreover, there is not an attribute with the same name and type in the selected node type and any of its super types.

Strategy:

1. Delete the attribute in all subtypes and create it in their super type.
2. Optional: Condense the rule set by looking for rules which check or set the attribute and differ in subtypes only; such rules can be merged to one rule using their super type and performing the same actions on the attribute.

Post-condition: The named attribute is pulled up. Optionally, the rule set is condensed such that there does not occur two rules which perform the same actions and differ in subtypes only.

Quality improvement: After the refactoring the number of redundant attribute definitions is reduced. Moreover, it can happen that the number of rules is reduced. The refactoring improves the conciseness and changeability of the system.

Semantics: The semantics is preserved, since the same transformation sequences are induced (see Section 5).

4.2 Refactoring "Extract Pre-condition"

This refactoring reduces the preserved part of a rule and extracts it as positive application condition.

Input parameter: name of the rule
Example: A customer takes an item out of the shelf. The rule mainly consists of context which has to be determined. We extract this context into a positive application condition which makes the rule considerably smaller (see Figure 2).

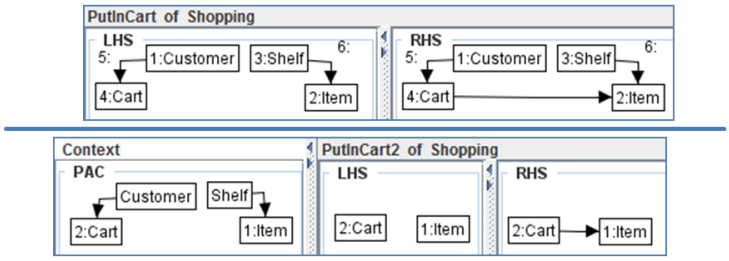


Fig. 2: Before and after refactoring "Extract Pre-Condition"

Pre-condition: none
Strategy:
1. Determine the preserved part of the input rule.
2. Create a new PAC and put the preserved part into it.
3. Reduce the rule's preserved part to the boundary nodes needed for inserting new edges.

Post-condition: The preserved part of the rule is minimal.
Quality improvement: The rule is smaller because of reduced redundancy, thus it is more concise. It might be better to comprehend, since the pre-condition is expressed more explicitly.
Semantics: The semantics is preserved, since the same transformation sequences are induced (see Section 5).

4.3 Refactoring "Loop Edges To Boolean Attribute"

It is quite common to use loop edges for modeling flags. A set flag is modeled by a loop edge. It is unset by deleting the loop edge. The flag is refactored by a Boolean attribute.

Input parameter: name of the loop edge type;
Example: Before the refactoring, the on-status of a machine is modelled by an loop edge named *on*. Rules *start* and *stop* add this loop to a machine to start and delete it to stop (see upper part of Figure 3) After the refactoring, this loop is replaced by a Boolean attribute called *isOn*. Rules *start* and *stop* are adapted. (see bottom part of Figure 3).

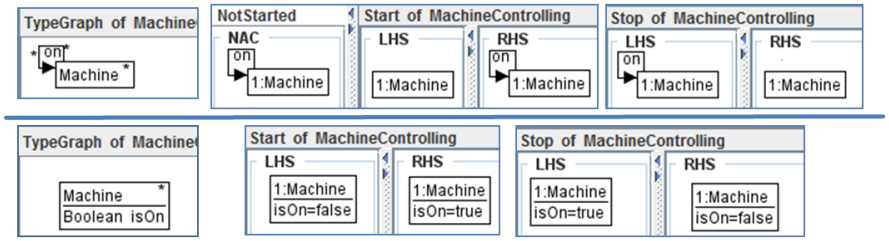


Fig. 3: Before and after refactoring "Loop Edges To Boolean Attribute"

Pre-condition: Edges of the selected type occur in instance graphs and rules as loop edges only.
Strategy:

1. Change type graph by erasing the loop edge type and inserting a new Boolean attribute in the node type which has been used as source and target node type for the erased edge type. The name of the new attribute is constructed by prefixing the original loop edge type name by "is".
2. Change instance rule graphs by replacing all loops of the indicated type by a corresponding attribute with value "True" in the same node which functions as source and target of the loop.
3. Replace each loop occurring in a NAC by a Boolean attribute with value "False" in the LHS. If a NAC does not have forbidden elements anymore, it is erased.

Post-condition: The selected loop edge type and all its instances in graphs and rules are erased. The corresponding attribute is inserted instead and equipped with value "True" if a loop edge existed at that node before. If it is equipped with value "False", the corresponding node was not equipped with a loop edge.

Quality improvement: In general, the number of rule elements and the number pre-conditions decrease. If all pre-conditions are concerned with loop edges to be replaced, the whole transformation system might get rid of pre-conditions and thus, the used approach might become simpler (improving conciseness). However, this is not the case, if attributes are introduced instead.

Semantics: A correspondence relation can be established between graphs with loops and corresponding attributes. It has been shown that a transformation sequence on graphs with loops can be bijectively translated into a transformation sequence on graphs with corresponding attributes, and vice versa [9].

4.4 Refactoring "Split Rule into Set Of Action Rules"

This refactoring transforms a rule into a set of *action rules* such that each one models one of the original rule's actions. An action is defined by one of the following atomic modifications: (1) insert a node, optionally together with adjacent edges, (2) insert an edge, (3) delete a node, together with all adjacent edges, (4) delete / move an edge, and (5) update / set / unset an attribute value. This refactoring is useful for modelling e.g. system errors that may occur at any time during a system's normal behavior [10]. In this case, more complex rules modeling normal behavior should be split into action rules allowing for more interleaving with rules modeling exceptions at any possible system state.

Input parameter: name of the rule to be split

Example: In a workflow scenario, two processes execute a task and move to the next task in the workflow. The task that has been executed before is deleted from the graph. The rule for this scenario is shown in the upper part of Figure 4. Two NACs ensure that this rule is applied only if there are not already todo edges linking the processes to the second task. We identify the actions *Move edge* (which occurs twice, for each todo edge) and *Delete node together with adjacent edges* which is applied to the left Task node. Hence, we split the original rule in two action rules, shown in the bottom part of Figure 4.

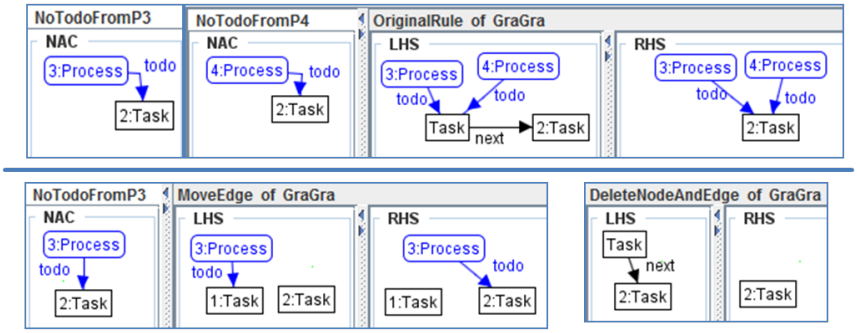


Fig. 4: Before and after refactoring "Split Rule into Set Of Action Rules"

Applying the rule sequence (MoveEdge, MoveEdge, DeleteNodeAndEdge) together with a suitable control structure defining an object flow, we get the same effect as applying the original rule. This object flow maps 1:Task to 1:Task or Task in the LHS of the next rule, and 2:Task to 2:Task in the LHS of the next rule.

Pre-condition: The ACs (NACs or PACs) of the original rule $r_o = (LHS_o, RHS_o)$ must be *splittable*, i.e. each AC is either embedded completely in the RHS of an action rule, or the origin of the AC morphism from LHS_o into the AC graph consists of one node only.

Strategy:

1. Identify the actions of original rule.
2. Split these actions into separate action rules $r_i = (LHS_i, RHS_i), i = 1, \ldots, n$ such that each action rule r_i models one of the actions defined above.
3. For each action rule, copy only those PACs and NACs from r_o to r_i, where the origin of the PAC (NAC) morphism from LHS_o to the PAC (NAC) graph is contained completely in LHS_i.
4. Delete action rules that occur more than once.

Post-condition: If rule r_o is applicable to graph G, then there is a sequence of action rules that is applicable at essentially the same matches as r_o such that the result after applying r_o to G is equal to the result of applying the action rule sequence to G.

Quality improvement: The resulting rule sequence allows for more action interleaving with other rules than the original rule. The action rules are smaller (improving conciseness and comprehensibility) than the original rule.

Semantics: One concurrent rule constructed from a suitable sequence of the action rules $r_1, \ldots, r_n$ yields the original rule r_o. The overlappings of rules in this sequence should best be defined by an object flow, mapping nodes and attribute values from the RHSs of earlier rules in the sequence to LHSs of later (not necessarily directly subsequent) rules in the sequence. Without this object flow, there may be more transformation sequences where the resulting action rule applications are interleaved with other rule applications. Furthermore, there may be transformation sequences containing action rules that do *not* yield the same result as any transformation by the original GTS. Hence, the semantics is preserved only in the sense that each graph pair in the original input/output relation of the GTS before the refactoring is still present in the relation of the refactored GTS, but there may be additional pairs after the refactoring (see Section 5). Here, an advanced modeling feature like control structures for rule sequences with object flow is necessary to ensure full semantics preservation.

4.5 Refactoring "Merge Rules Differing in Types Only"

If there are two rules which differ in node types only and these node types are sub-types of the same super-type, they can be merged to one rule. This refactoring is a sub-refactoring of "Pull Up Attribute".

Input parameter: Names of the two rules to be merged
Example: Since this refactoring is part of "Pull Up Attribute", we can reuse the example of that refactoring. Phones are specialized to fixed and mobile phones. Two rules describe the lifting of fixed resp. mobile phones. See the upper part of Figure 1. After the refactoring, a lift rule for phones in general is shown which is merged from the two original lift rules. (See the lower part of Figure 1.)

Pre-condition: Indicated rules differ in node types only. The set of node types found contains all sub-types of a common super-type.

Strategy:

1. Identify all node types with common super type.
2. Construct a new rule by taking one original rule and replacing identified sub-types by identified super-type.
3. Delete all original rules.

Post-condition: All original rules are replaced by one new rule using identified super types.
Quality improvement: The number of rules becomes smaller.
Semantics: The semantics is preserved, since the same transformation sequences are induced (see Section 5).

4.6 Refactoring "Erase Non-injective Matching"

A rule which may allow for non-injective matching is replaced by a rule set which allows for injective matching only.

Input parameter: Name of the rule
Example: Using graph transformation for the generation of finite automata, there is rule "createTransion" which creates transitions between states Figure 5. The non-injective matching of this rule is needed to create transition loops. After the refactoring, a set of two rules is created which consists of the original one (now restricted to injective matching) and a new one, called "creatLoop", for the creation of transition loops, i.e. nodes of type *State* are merged here.

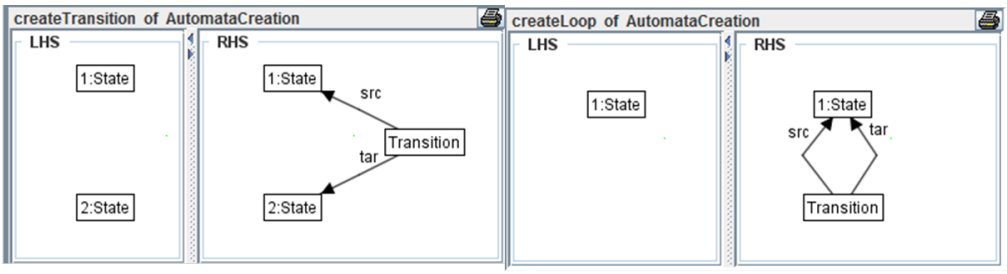


Fig. 5: Example rules for refactoring "Erase Non-injective Matching"

Pre-condition: None
Strategy:

1. Identify node sets with common node type.
2. Identify edge sets with common edge type and common source and target node types.
3. Create a rule set containing rules for each possible merging of nodes and edges in these sets.

Post-condition: The resulting rule set contains rule for each possible merging of nodes and edges of common types.

Quality improvement: Non-injective matching is not needed anymore.

Semantics: The semantics is preserved in the sense that the same graphs are created and the same effects are performed (see Section 5).

4.7 Refactoring "Move Versus Delete and Create"

Rule elements which are deleted and created in the original rule are moved afterwards.

Input parameter: Name of the rule

Example: Taking up the *Phone* example again, we consider a rule which replace a fixed phone at one location by another phone at another location, i.e. the fixed phone at the original location is deleted and a new one is created at the new location. After the refactoring, the rule specifies the movement of a fixed phone from one location to another one Figure 6.

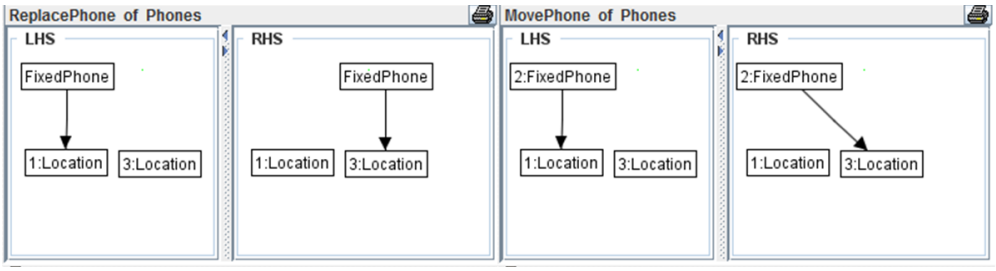


Fig. 6: Refactoring of deletion and creation of a fixed phone to the movement of that phone

Pre-condition: None

Strategy:

1. Identify nodes and edges which are deleted and created afterwards. If these nodes and edges are attributed, they are identified only if the attribute values of the created elements are the same as of the deleted ones. Nodes are identified only if their adjacent edges are created in the same way they existed before.
2. Preserved identified elements instead of deleting and creating them.

Post-condition: The rule does not contain any element that is deleted and created in the same way and context.

Quality improvement: The resulting rule allows for more concurrency.

Semantics: The semantics is preserved in the sense that the same graphs are created, however, the number of transformation effects when applying the refactored rule is reduced (see Section 5).

4.8 Refactoring "Unify Rules with Same Actions"

Given a set of rules which show a subset of same actions. This subset is encapsulated in a new rule to be applied first. The original rules are reduced to their remaining actions each.

Input parameter: set of rule names
Example: For registering a new phone, it is enough for mobile phones to give the person who will own it. For fixed phones, their location has to be registered in addition. These two cases are specified in rules "RegisterMobilePhone" and "RegisterFixedPhone". However, the owner registration is common to both rules. Thus, we have a kernel for phone registration afterwards containing the owner registration only (plus flag creation). The remainder rules of the original ones do not care about the owner registration anymore, i.e. "RegisterFixedPhoneRemainder" specifies the location registration only and for "RegisterMobilePhoneRemainder" there is not any original action left. However, both remainder rules have to delete the flag again.

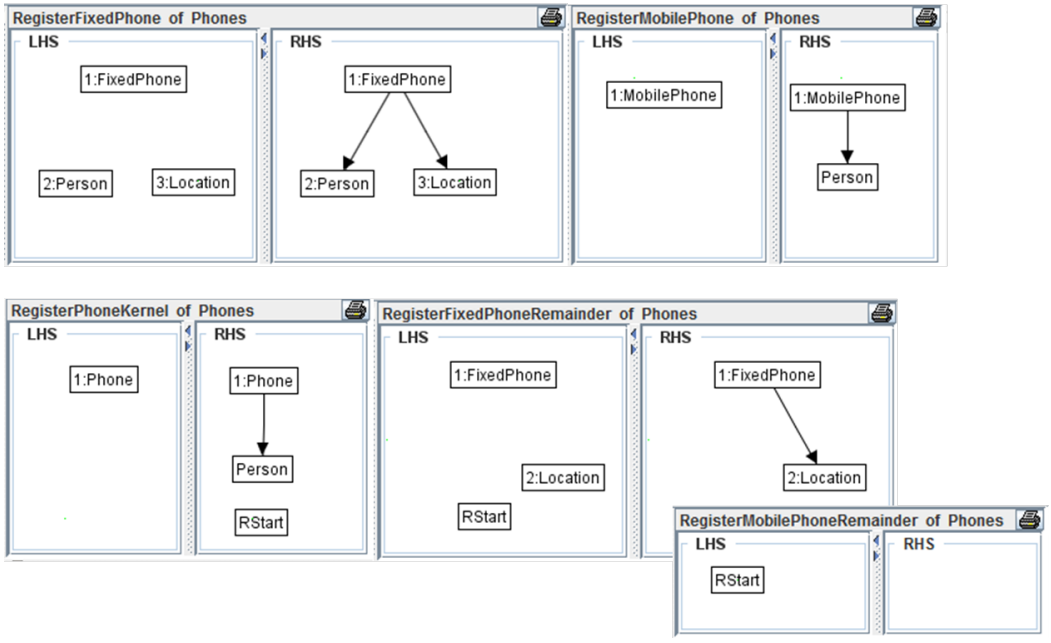


Fig. 7: Unify common register phone actions

Pre-condition: None
Strategy:

1. Identify the set of actions common to all rules in the given set. Identify also the common preserved part. This identification abstracts from concrete node types if there are common super types.
2. Create a new rule, called *kernel rule*, which contains all identified actions and the identified preserved part. If common actions and preserved parts differ in node types, their common super types are used instead.
3. Reduce each of the original rules, called *remainder rule*, by the identified set of actions. Reduce the preserved part if it is common and not needed for the remaining actions.
4. Add a flag to the kernel rule. The flag is created by this rule.
5. For each remainder rule: The flag created by the kernel rule is deleted here.

Post-condition: There is a new rule, the kernel rule, that contains all common actions. All remainder rules do not contain common actions anymore. A remainder rule is not applicable without applying the kernel rule beforehand.
Quality improvement: The redundancy within the given rule set is reduced.
Semantics: A concurrent rule consisting of the kernel rule and one remainder rule yields an original rule. There may be more transformation sequences than before, since the resulting transformations allow for more interleaving with other rule applications.

5 Preservation of Semantics

Refactorings in object-oriented programming are usually defined as *behavior-preserving* transformations [12]. The idea is that it is possible to change the internal implementation of a system (to improve its performance, maintainability, etc.) without affecting the externally visible behavior. Since graph transformation systems and graph grammars can be used for a number of different purposes, their notions of "behavior" are varied as well. If a graph grammar $\mathcal{GG}$ is used for the specification of a *graph language*, representing e.g. the abstract syntax of visual languages, its semantics is given by the set $\mathrm{L}(\mathcal{GG})$ of graphs reachable from the start graph.

If instead we are interested in the specification and implementation of *model transformations* as mappings between graphs, the semantics of a graph transformation system $\mathcal{G}$ can be given by a relation $MT_{\mathcal{G}} \subseteq Gra_{TG_S} \times Gra_{TG_T}$ between sets of graphs over type graphs $TG_S, TG_T \subseteq TG$ representing the abstract syntax of source and target languages. The relation is defined by applying to an input graph G_S typed over the source fragment TG_S of the type graph a set of rules until no further transformation is possible. Then, the projection G_T of the resulting graph to the target fragment TG_T is the output of the transformation [7], making (G_S, G_T) an element of the transformation relation $MT_{\mathcal{G}}$. In the case of a graph transformation system *modelling a process*, such as a workflow, computation, or network protocol, we are interested in the ongoing behavior

rather than the end result of the transformation. Here it makes sense to distinguish two cases, i.e., whether the structure of states is part of the semantics or if we focus on observable behavior only, hiding the representation of states. In the first case, semantic models such as (shift-equivalence classes of) transformation sequences, graph process and unfoldings are appropriate. In the second case, a notion of observation or label is required, to be associated with transformations. Then, models based on traces and labelled transition systems can be derived.

The question, whether a given refactoring is semantics preserving, therefore depends on the semantic model chosen. However, a common feature of most of the models is that they are based on transformation steps as basic building blocks. We will therefore consider preservation of this single-step semantics as an elementary requirement and discuss the more elaborate notions based on that assumption. In general, a refactoring step can lead to a change of representation $r : Gra_{TG_1} \to Gra_{TG_2}$ mapping graphs over TG_1 to graphs over TG_2. For example, in changing loops used as markers into Boolean attributes TG_1, TG_2 are the type graphs declaring the loop or the attribute only. In many examples, however, such as the Extract Precondition refactoring, this mapping r is the identity, i.e., the graph representation is unchanged at the instance level. Let $\mathcal{G}_1/\mathcal{GG}_1$ and $\mathcal{G}_2/\mathcal{GG}_2$ be graph transformation systems/grammars before and after refactoring. Preserving semantics in this case means to

- preserve the generated graph language, i.e., $r(L(\mathcal{GG}_1)) = L(\mathcal{GG}_2)$;
- preserve the transformation relation generated by a model transformation system, i.e., $r(MT_{\mathcal{G}_1}) = MT_{\mathcal{G}_2}$ where r extends to pairs of graphs in the obvious way;
- ensure that r is a suitable bisimulation function between $LTS(\mathcal{GG}_1)$ and $LTS(\mathcal{GG}_2)$, stating (in the case of the most common notion of strong bisimulation [14]) that for states s_1 and s_2 with $r(s_1) = s_2$, each transition $s_1 \xrightarrow{l} s_1'$ in $LTS(\mathcal{GG}_1)$ is matched by a transition $s_2 \xrightarrow{l} s_2'$ in $LTS(\mathcal{GG}_2)$, and vice versa, such that $r(s_1') = s_2'$.

Let us consider the *Extract Pre-condition* refactoring (see Section 4.2). For any graph G, there exists a transformation $G \overset{p,m}{\Longrightarrow} H$ if and only if there exists a transformation $G \overset{p',m'}{\Longrightarrow} H$ where p' is the result of applying the refactoring to p and $m' : L' \to G$ is the restriction of match $m : L \to G$ to the smaller left-hand side $L' \subseteq L$. Based on this preservation of the single-step semantics, it is obvious that reachability as well as transformation relations are preserved. In order to establish a bisimulation, we have to assume that $G \overset{p,m}{\Longrightarrow} H$ and $G \overset{p',m'}{\Longrightarrow} H$ carry the same labels (or inject a suitable relabelling). It is worth pointing out that this refactoring may not be consistent with concurrent models such as shift-equivalence since moving elements from the left-hand side into a positive precondition may affect notions of independence of transformation steps.

In refactoring *Pull Up Attribute* we modify the type graph by moving an attribute shared between all subclasses of a common superclass to that superclass. This change at the type level does not affect the instance graphs or rules (but

allows for more general rules). Consequently, the step semantics does not change. As mentioned before, refactoring *Loops to Boolean Attribute* involves a change of representation affecting both type and instance level, but should preserve the step semantics. Finally, refactoring *Split Rule into Set Of Action Rules* does not preserve the step semantics because a single step using the original rule is broken up into a sequence of steps using action rules. In the case of the LTS semantics this would lead to a transition system having additional states and transitions. The language and relational semantics, both based on reachability only, can be preserved if we can control action rules to ensure that they faithfully implement all and only the original transformations.

6 Prototypical Implementation

This section shortly presents the implementation of a sample refactoring for EMF model transformation by Henshin being based on graph transformation concepts. After introducing Henshin, we summarize the implementation of refactoring *Extract Pre-condition* as specified in Section 4. In particular, we discuss the overall algorithm and describe one single transformation rule used for refactoring execution in detail.

6.1 Henshin and EMF Refactor

Henshin is a language and associated tool set for in-place transformations of models that are based on the Eclipse Modeling Framework (EMF). Henshin itself is based on graph transformation concepts where rules can be equipped with nested application conditions and structured into nested transformation units [1]. EMF Refactor [18] supports the specification and application of refactorings to EMF-based models. Since the application module of EMF Refactor uses the Eclipse Language Toolkit (LTK) technology [23], a refactoring requires up to three parts. They reflect a primary application check for a selected refactoring without input parameters, a second one with parameters, and the proper refactoring execution. Currently, EMF Refactor supports refactoring specifications using Java respectively Henshin transformations.

Figure 8 shows a small part of the Henshin meta-model consisting of concepts which are affected by refactoring *Extract Pre-condition*. A transformation rule consists of left and right-hand side graphs (LHS and RHS) which describe model patterns by their underlying (graph) structure. Nodes refer to EClass objects while Edges refer to EReferences between objects via references called type (not shown in Figure 8).

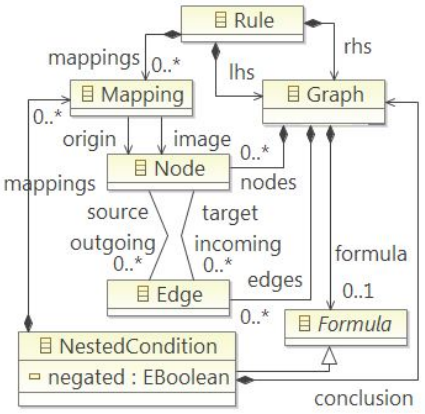


Fig. 8: Part of the Henshin meta model

These type references are used for explicit typing. Mappings between LHS and RHS can be defined based on nodes. Edge mappings are implicitly given if both, their source and target nodes are mapped. To conveniently determine where a specified rule should be applied, application conditions can be defined. Graphs can be annotated with application conditions using a Formula that is either a logical expression (not shown in Figure 8) or a Nested Condition which is an extension of the original LHS graph structure[5]. The Henshin model supports a special kind of transformation units, called Amalgamation Units, which is useful to specify *forall*-quantified operations on recurring model patterns. An amalgamation unit consists of one rule which acts as a kernel rule and multiple rules which act as multi-rules. The effect is that the modification defined in the kernel rule is applied exactly once while modifications defined in the multi-rules are applied as often as suitable matches are found. For details on amalgamation concepts we refer to [5].

6.2 Refactoring of Henshin transformation systems

Henshin transformation systems can be refactored in a straightforward way. A Henshin transformation model instance is considered as model to be refactored. The refactorings are implemented based on Henshin rules typed over the Henshin meta model, i.e. Henshin transformation systems are refactored by Henshin again. Since the Henshin model is an ordinary EMF model, EMF Refactor can be used to generate refactoring operations for EMF-based editors for Henshin transformation systems.

The prototype implementation of refactoring *Extract Pre-condition* addresses the following two modifications of the description made in Section 4:

- First, we add a precondition that checks whether the original rule does not have any application conditions.
- Second, we omit attributes on graph nodes, i.e. we consider typed graphs instead of attributed typed graphs only.

Please note that these limitations are just done in order to keep the refactoring specification simple. In the following, we concentrate on the execution steps of the refactoring. This means that the non-existence of application conditions is already assumed. The entire refactoring specification including a comprehensive discussion can be found at [2].

[5] Note that PACs and NACs are special nested conditions.

Figure 9 shows the overall structure of the execution part of the example refactoring. It consists of a so-called Sequential Unit (named *Extract Pre-condition*) that in turn consists of three further transformation units, more precisely amalgamation units. These child units are meant to be executed in a sequential order.

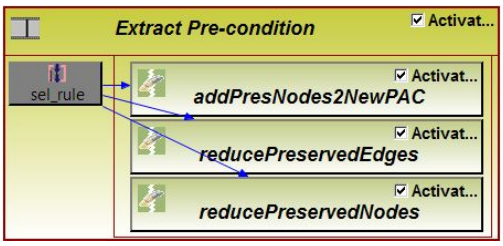


Fig. 9: Unit structure of Henshin refactoring *Extract Pre-condition*

The first amalgamation unit, *addPresNodes2NewPAC*, creates a new PAC (kernel rule) and adds a copy of each preserved LHS node to it (multi rule). In the second amalgamation unit, *reducePreservedEdges*, the preserved edges of the original rule (presented as parameter *sel_rule* in Figure 9) are moved to the application condition, i.e. afterwards there are only those edges left which have to be deleted (LHS edges) or created (RHS edges). Finally, amalgamation unit *reducePreservedNodes* deletes those mapped nodes from the LHS and RHS graphs that are not associated to any edge left.

Figure 10 shows the multi-rule of amalgamation unit *reducePreservedNodes* in an integrated view consisting of LHS, RHS, and four NACs altogether. LHS objects (nodes and edges) can be identified by tags $\langle\langle preserve \rangle\rangle$ or $\langle\langle delete \rangle\rangle$, objects tagged by $\langle\langle preserve \rangle\rangle$ or $\langle\langle create \rangle\rangle$ from the RHS of the rule (since the multi-rule does not add any objects to the model, there are no $\langle\langle create \rangle\rangle$ tags in Figure 10). Finally, NAC objects are tagged by $\langle\langle forbid \rangle\rangle$. Nodes that are mapped to corresponding nodes of the kernel rule have a gray background, whereas additional multi-rule nodes have a white background. They represent so-called multi-objects. The multi-rule matches preserved nodes (the right and center nodes in Figure 10) which are not associated to (i) an edge that has to be created by the rule (NACs *AC0* and *AC1*), or (ii) an edge that has to be deleted by the rule (NACs *AC2* and *AC3*). Finally, these nodes as well as the corresponding mappings are deleted from the according LHS respectively RHS graphs.

7 Related Work

There are various approaches using graph transformation to specify model and program refactorings, but to the best of our knowledge, there is no other work specifying refactorings for graph transformation systems. Therefore, we concentrate our consideration of related work on two aspects: theoretical results for graph transformations that form the basis for some of our refactorings, and related approaches and tools for EMF model refactoring.

Refactoring "Split Rule into Set of Action Rules", for instance, relies on the concurrency theorem [7] which states how rules can be composed. Refactoring "Erase Non-injective Matchings" goes back to results for graph transformation

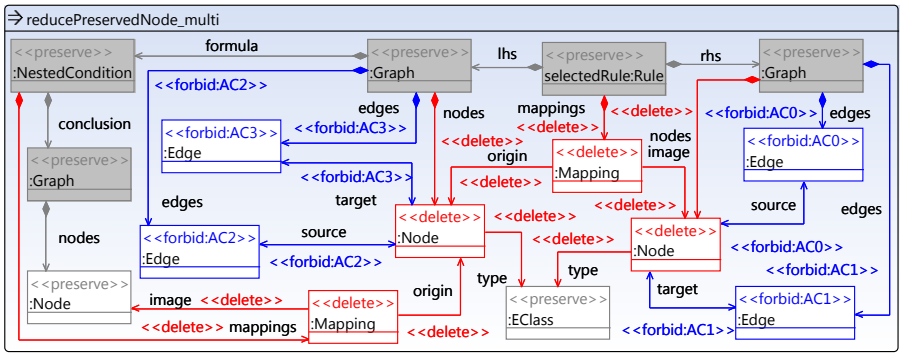


Fig. 10: Multi-rule of amalgamation unit *reducePreservedNodes*

systems (dis-)allowing non-injective matchings shown in [13]. Based on these results we can clearly specify the intended refactorings and can argue that they are semantics preserving in a certain sense.

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. To specify EMF model refactorings, there are further tools such as the Epsilon Wizard Language (EWL) [15] and a generic approach presented in [19] available. In contrast to EWL, EMF Refactor provides a specification framework for refactorings which allows different concrete specification mechanisms. In particular, EMF Refactor supports Henshin which supports more correctness checks than EWL (see also [4]). In contrast to EWL, EMF Refactor further uses the LTK technology for homogeneous refactoring execution in Eclipse including e.g. a refactoring preview. In [19], the authors propose the definition of EMF-based refactoring in a generic way, however do not consider the comprehensive specification of preconditions. Our experiences in refactoring specification show that it is mainly the preconditions that cannot be defined generically.

8 Conclusions and Future Work

Refactorings are a well-established means to make development experiences explicit such that further developers can benefit from these experiences. Therefore, we start with a selection of interesting refactorings for graph transformation systems. In this paper, we explicitly restrict this approach to kernel features. However, further features should be considered, primarily rule parameters and control structures for rule applications. In addition, it is certainly worthwhile to define not only metric-based but also pattern-based graph transformation smells. It is up to future work, to enable developers to define purpose-specific quality assurance processes for graph transformation systems.

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. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, vol. 6394, pp. 121–135 (2010)
2. Arendt, T.: Specification of Refactoring Extract Pre-condition for Henshin Transformation Systems, http://www.mathematik.uni-marburg.de/~arendt/agtive11/
3. Arendt, T., Kranz, S., Mantz, F., Regnat, N., Taentzer, G.: Towards syntactical model quality assurance in industrial software development: Process definition and tool support. In: Software Engineering 2011. LNI, vol. 183, pp. 63–74. GI (2011)
4. Arendt, T., Mantz, F., Schneider, L., Taentzer, G.: Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. In: Model-Driven Software Evolution, Workshop Models and Evolution (2009)
5. Biermann, E., Ermel, C., Taentzer, G.: Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework. ECEASST 26, 59–76 (2010), http://easst.org/eceasst
6. Booch, G., Maksimchuk, R., Engel, M., Young, B., Conallen, J., Houston, K.: Object-Oriented Analysis and Design with Applications. Addison-Wes (2007)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science, Springer (2006)
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools. World Scientific (1999)
9. Ehrig, H., Ermel, C., Ehrig, K.: Refactoring of Model Transformations. ECEASST 18 (2009), http://easst.org/eceasst
10. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Proc. Int. Conf. on Fundamental Aspects of Softw. Eng. (FASE'10). LNCS, vol. 6013, pp. 139–153. Springer (2010)
11. Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.): Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, LNCS, vol. 5765. Springer (2010)
12. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
13. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Mathematical Structures in Computer Science 11(5), 637–688 (2001)
14. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing full semantics preservation in model transformation - a comparison of techniques. In: Proc. of Int. Conf. on Integrated Formal Methods. LNCS, vol. 6396, pp. 183–198. Springer (2010)
15. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. Object Technology 6(9), 53–69 (2007)
16. Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed Graph Transformation with Node Type Inheritance. Theor. Comput. Sci. 376(3), 139–163 (2007), http://dx.doi.org/10.1016/j.tcs.2007.02.001
17. Object Management Group: Unified Modeling Language: Superstructure – Version 2.3 (2010), http://www.omg.org/technology/documents/modeling_spec_catalog.htm
18. EMF Refactor, http://www.eclipse.org/modeling/emft/refactor/

19. Reimann, J., Seifert, M., Aßmann, U.: Role-Based Generic Model Refactoring. In: Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, vol. 6394, pp. 78–92. Springer (2010)
20. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
21. Schürr, A., Nagl, M., Zündorf, A. (eds.): Applications of Graph Transformations with Industrial Relevance, LNCS, vol. 5088. Springer (2008)
22. The Eclipse Foundation: Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf/
23. The Eclipse Foundation: The Language Toolkit (LTK), http://www.eclipse.org/articles/Article-LTK/ltk.html