Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Comparison of graph-based model transformation rules

Alexander Schultheiß[a] Alexander Boll[a] Timo Kehrer[a]

a. Department of Computer Science, Humboldt-Universität zu Berlin

Abstract With model transformations arising to primary development artifacts in Model-Driven Engineering, dedicated tools supporting transformation developers in the development and maintenance of model transformations are strongly required. In this paper, we address the versioning of model transformations, which essentially relies on a basic service for comparing different versions of model transformations, e.g., a local workspace version and the latest version of a repository. Focusing on rule-based model transformations based on graph transformation concepts, we propose to compare such transformation rules using a maximum common subgraph (MCS) algorithm as the underlying matching engine. Although the MCS problem is known as a non-polynomial optimization problem, our research hypothesis is that using an MCS algorithm as a basis for comparing graph-based transformation rules is feasible for real-world model transformations and increases the quality of comparison results compared to standard model comparison algorithms. Experimental results obtained on a benchmark set for model transformation confirm this hypothesis.

Keywords model transformation development, transformation comparison, graph-based transformation rules, maximum common subgraphs

1 Introduction

Model transformation is a central activity and key technique in all stages of Model-Driven Engineering (MDE) [BCW12, SK03]. A plethora of model transformation approaches, languages, and frameworks have emerged to address a multitude of transformation scenarios [CH06]. On the contrary, less effort has been spent in providing dedicated concepts, techniques and tools supporting transformation developers in the development and maintenance of model transformations. With model transformations arising to primary development artifacts in the development of MDE infrastructures, such tools are strongly required to fully turn the MDE vision into reality. Recently, the MDE research community has addressed some development and maintenance tasks, such as clone detection [SPA16], variability management [SRA+16], refactoring [TAEH12] and differencing [KPS17] for and of model transformations. In general, however, model transformation development and maintenance is not nearly as broadly

Alexander Schultheiß, Alexander Boll, Timo Kehrer. *Comparison of graph-based model transformation rules.* In Journal of Object Technology, vol. XV, no. N, 2020, pages M:1–20. doi:10.5381/jot.202Y.VV.N.aN

supported as developers are used to from sophisticated development environments for classical code-centric development. This is still one of the major roadblocks to a more widespread adoption of MDE in practice [WHR+13].

Problem definition and scope. In this paper, we address the versioning of model transformations, which becomes an indispensable development support discipline with model transformations arising to primary development artifacts. The CMMI-Dev (Capability Maturity Model Integration for Development) [Tea10], for instance, mentions versioning support as one of the most essential disciplines to lift an unmanaged development process (CMMI level 1) to a managed one (CMMI level 2). Versioning essentially relies on a basic service for comparing different versions of model transformations. Transformation developers, for example, compare revisions of transformation rules because they want to know which parts of the transformation have remained unchanged and which ones were added, removed, or modified, e.g., when comparing a local workspace version with the latest repository version [KKPS12]. Another example of a typical versioning task which relies on a comparison service is the merging of parallel edits in distributed development environments. Here, an overview of the common and specific parts of both versions w.r.t. a common base version serves as an essential input for a subsequent merging step [ASW09]. In a broader sense, incremental model transformation techniques, such as studied in [BTW14], also rely on a comparison service for model transformations in order to re-execute only those parts of a transformation that have changed with respect to the latest execution.

We focus on rule-based model transformations based on graph transformation concepts [EEPT06], one of the major model transformation approaches that has been adopted by several widely used model transformation tools such as Henshin [SBG+17] and VIATRA2 [VB07]. Although it appears to be a natural choice, existing strategies for comparing graph-structured models cannot be simply applied to the problem of comparing graph-based transformation rules (see Sect. 2). Instead of adopting or tailoring existing comparison heuristics, we propose to compare graph-based transformation rules using a maximum common subgraph (MCS) algorithm as the underlying matching engine [CFSV04]. This class of algorithms has been largely ignored by the model comparison research community, mostly due to their exponential runtime behavior which does not scale for MDE models comprising hundreds or thousands of elements [TBWK07]. However, we argue that, in terms of the number of comprised elements, model transformation rules are typically much smaller than models, and thus calculating maximum common subgraphs can be achieved within reasonable time.

Contributions. Our central research hypothesis is that using a maximum common subgraph algorithm as a basis for comparing graph-based transformation rule is feasible and increases the quality of comparison results compared to standard model comparison algorithms. To validate this research hypothesis, the paper makes the following contributions:

- An approach to use maximum common subgraph algorithms for comparing graph-based transformation rules, including an analysis of the main variation points of such a comparison service (Sect. 4).
- A prototypical implementation of the approach which is based on an implementation of the McGregor algorithm and which is integrated with the graph-based model transformation language Henshin (Sect. 5).

- Experimental results of evaluating the prototype on a model transformation benchmark set and comparing its accuracy and runtime performance with those of *EMFCompare* and the clone detection facilities of *RuleMerger* (Sect. 6).

2 State of the art

The result of a comparison of two transformation rules, i.e., a *matching* [KDRPP09], should deliver a set of pairs of corresponding rule elements which are considered "the same" in both rules. Kehrer et al. [KPS17] suggest to use existing model comparison tools for that purpose. As surveyed in [KDRPP09, SC13], there are sophisticated model comparison approaches which work on a graph-based representation of models and which therefore appear to be a natural choice for comparing graph-based model transformation rules. Moreover, the abstract syntax of many model transformation languages is defined in terms of a meta-model, and transformation rules are instantiations thereof. Thus, existing model comparison tools can be applied out of the box. The Henshin model transformation language, for instance, is defined using Ecore, the meta-modeling language of the Eclipse Modeling Framework (EMF). Thus, Henshin rules can be compared using EMFCompare [BP08], the most widely used tool for comparing EMF-based models.

However, comparing graph-based transformation rules using generic model comparison tools such as EMFCompare only delivers satisfactory matching results under very restricted conditions, notably when reliable universal unique identifiers (UUIDs) are attached to rule elements. The limitations of UUIDs have been extensively discussed in the literature, e.g., in [KDRPP09]. The similarity-based matching results obtained from EMFCompare in the absence of UUIDs are only of minor quality. As an example, consider the Henshin rules *createExcludeConstraint* and *deleteExcludeConstraint* shown in Fig. 1, taken from the catalog of edit operations on FODA-like feature diagrams [KCH+90] presented in [BKL+16] and typed over the meta-model shown in Fig. 2. EMFCompare delivers only a single rule graph correspondence when comparing both rules, namely between the nodes of type *FeatureModel*, although the rules are highly similar to each other. In fact, having a look at the edit history of the rules provided as supplementary material of the work of Bürdek et al. [BKL+16], the rule *createExcludeConstraint* has been obtained by (i) creating a copy of the

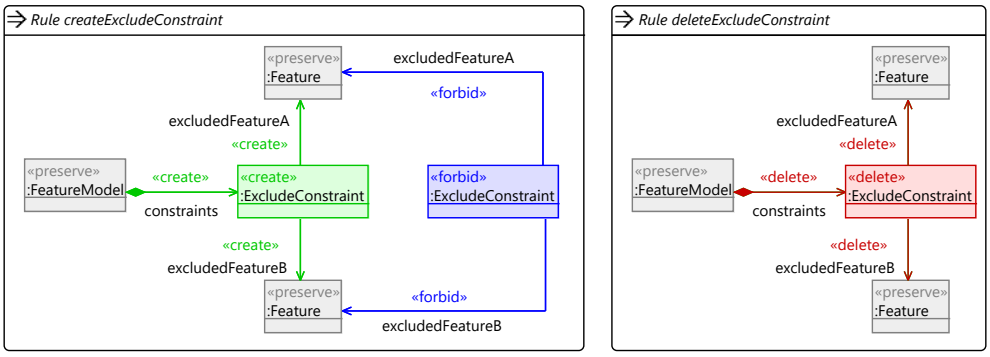

Figure 1 – Henshin rules *createExcludeConstraint* (left) and *deleteExcludeConstraint* (right) taken from the catalog of edit operations on feature diagrams presented in [BKL+16].

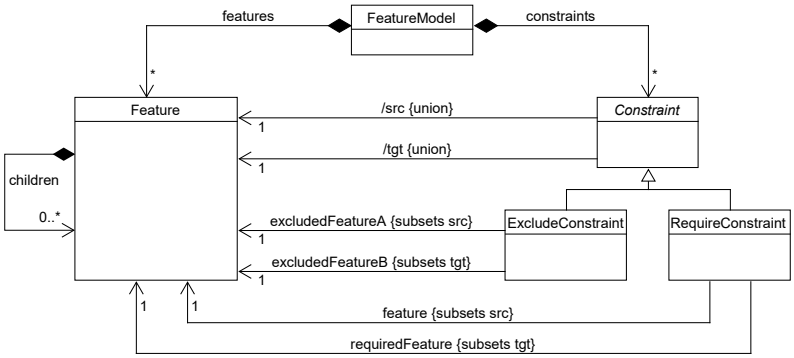

Figure 2 – Excerpt of a simple meta-model for FODA-like feature diagrams [KCH+90].

rule *deleteExcludeConstraint*, (ii) converting all elements which are to be deleted into elements to be created, and adding the negative application condition (forbidden elements) which prevents the creation of duplicated exclude constraints between the same features. Thus, matching the action elements of both rules is highly desirable.

The minor quality of the comparison results can be explained by having a look at the similarity-based matching algorithm employed by EMFCompare [XS05]. This algorithm relies on two fundamental assumptions, namely that (a) model elements are organized in a tree-like containment structure and (b) most of the model elements are named. The overall matching strategy is to traverse the tree structure in a top-down manner, and to match the top-level elements first. For each pair of corresponding elements, the algorithm tries to find further correspondences in the sets of their direct children, and so on recursively. Within such sets of child elements, correspondences are established between the mutually most similar elements having the same type. The similarity of a pair of elements is calculated based on their local properties, most notably based on the similarity of their names. This strategy works reasonably well for class diagrams and other types of UML-like models where the above assumptions (a) and (b) are met, but fails for graph-based model transformation rules. A rule exposes a plain graph structure without any hierarchical relationships. Most of the nodes do not have any distinguishing local properties. Even though transformation languages such as Henshin offer the possibility to assign logical identifiers to nodes, these identifiers are not commonly used.

The absence of hierarchical structures as well as the existence of "anonymous" elements has motivated further model matching strategies which globally search for potential correspondences [MGMR02, KKPS12]. All pairwise similarities of candidates are computed in an iterative manner, which allows similarities to be propagated (referred to as "similarity flooding" in [MGMR02]): In each iteration, the similarity of the neighborhood of two model elements is "added" to their similarity from the previous iteration. However, due to cyclic dependencies between model elements, similarity propagation is not guaranteed to converge. Most often, similarity flooding only works well if at least some "anchor points" can be established [KKPS12]. Anchor points are model elements which can be matched very reliably, such as initial or final states in UML statecharts, and which can serve as starting points from where to propagate similarities. In general, however, there are no elements which can serve as natural anchor points for the comparison of graph-based transformation rules.

Finally, we can find closely related work in the context of the *RuleMerger* project and tool which includes clone detection facilities that have been tailored to graph-based transformation rules [SPA16]. Here, given a set of rules, a clone is defined as a

subrule which can be embedded into a subset of the given rule set. That is, given two transformation rules which are to be compared with each other, clone detection can be utilized to identify the corresponding rule elements in both rules. However, the notion of a clone is way too restrictive for our purpose of comparing rules in the context of versioning. Consider again the transformation rules *createExcludeConstraint* and *deleteExcludeConstraint* shown in Fig. 1. The largest common subrule consists of the preserved elements of type *FeatureModel* and *Feature* while, analogously to the comparison result obtained by EMFCompare, all the other reasonable correspondences are missed by such a clone-based comparison.

3 Basic definitions

The formal underpinning of our approach is based on graphs and graph transformation. We draw from the work presented in [CFV07, EEPT06] and briefly recall the needed concepts in the remainder of this section.

3.1 Graphs, graph morphisms and maximum common subgraphs

As usual in MDE, we consider models as typed graphs whose types are drawn from a meta-model. Like existing model comparison tools, as studied in the previous section, we assume graphs to be correctly typed over a fixed meta-model and thus abstain from a formal definition of typing using type graphs and type morphisms (see, e.g., [BET12]). Instead, to keep our basic definitions as simple as possible, we work with a variant of labeled graphs where the node and edge label alphabets, both finite sets referred to as L_V and L_E, represent node and edge type definitions of a meta-model, respectively. Moreover, as it is common for standard modeling frameworks such as EMF, we assume that graphs do not contain parallel edges of the same type (i.e., two edges of the same type linking the same source and target node).

Definition 1 (Graph)**.** *A graph is a tuple $G = (V, E, \tau_V)$ where V is a finite set of nodes, $E \subseteq V \times V \times L_E$ is a set of labeled edges, and $\tau_V : V \to L_V$ is a node labeling function assigning every node in V its label in L_V.*

An edge is a tuple (s, t, l) having its source in s and its target in t, and the label l represents the name of its type. Given an edge $e = (s, t, l)$, we use the notations $src(e)$, $tgt(e)$ and $\tau_E(e)$ to refer to s, t and l, respectively.

Definition 2 (Subgraph)**.** *Let $G = (V, E, \tau_V)$ and $G' = (V', E', \tau'_V)$ be graphs. Graph G' is a subgraph of G, written $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and $\tau'_V(v) = \tau_V(v)$ for every $v \in V'$.*

G' is called an induced subgraph of G if $E' = E \cap (V' \times V' \times L_E)$. It is uniquely induced by V', i.e., every edge in G with source and target nodes in V' is also an edge in G'.

While a graph and its subgraph share elements, a graph morphism relates two graphs by solely relying on structural properties rather than on globally defined element identifiers. In our notion of a graph, a graph morphism maps a graph G to a graph G' by associating the nodes of G with those of G' in a structure- and label-preserving manner.

Definition 3 (Graph morphism)**.** *Let $G = (V, E, \tau_V)$ and $G' = (V', E', \tau'_V)$ be graphs. A graph morphism from G to G' is a function $f : V \to V'$ such that*

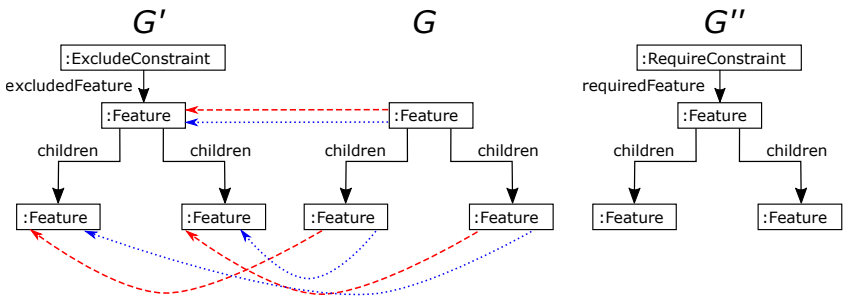

Figure 3 – Three example graphs where G is a maximum common subgraph of G' and G''. Two variants of subgraph isomorphisms $G \hookrightarrow G'$ are illustrated by red dashed and blue dotted arrows, respectively.

(1) $\tau_V(v) = \tau'_V(f(v))$ *for every* $v \in V$,

(2) for every edge $e = (u, v, l) \in E$ *there exists an edge* $e' = (f(u), f(v), l) \in E'$.

Note that the edges of G are implicitly mapped to those of G' since we do not support parallel edges of the same type.

A morphism is called isomorphism if f is a bijection. If f is an isomorphism between graphs G and G', and G' is an induced subgraph of another graph G'', i.e., $G' \subseteq G''$, then f is called a subgraph isomorphism from G to G'', written $G \hookrightarrow G''$.

Definition 4 (Maximum common subgraph)**.** *Let* $G' = (V', E', \tau'_V)$ *and* $G'' = (V'', E'', \tau''_V)$ *be graphs. A graph* $G = (V, E, \tau)$ *is a common subgraph of* G' *and* G''*, written* $CS(G', G'')$*, if there are subgraph isomorphisms* $cs' : G \hookrightarrow G'$ *and* $cs'' : G \hookrightarrow G''$*. G is the maximum common subgraph of* G' *and* G''*, written* $MCS(G', G'')$*, if there is no other common subgraph of* G' *and* G'' *having a larger set of nodes than* $MCS(G', G'')$.

Note that, for a maximum common subgraph $MCS(G', G'')$, the subgraph isomorphisms cs' and cs'' are not necessarily unique.

Example 1. *Our basic notions are illustrated in Fig. 3. Graph G is the maximum common subgraph of* G' *and* G''*, type labels are drawn from the meta-model shown in Fig. 2. The "feature" nodes of G can be mapped in two ways to the "feature" nodes of* G' *and* G''*, respectively, as illustrated for the two variants of* $G \hookrightarrow G'$.

3.2 Graph-based transformation rules

Since we aim at a syntactic comparison of graph-based transformation rules, we restrict ourselves to the formalization of transformation rule structures, while their operational semantics will be sketched briefly and largely informally. Our definition assumes a double-pushout (DPO) approach to algebraic graph transformation [EEPT06].

Definition 5 (Transformation rule)**.** *A transformation rule is a tuple* $r = (L \supseteq K \subseteq R, NAC, PAC)$ *where*

- L, K *and* R *are graphs called the left-hand side, intersection and right-hand side of a rule, respectively,*

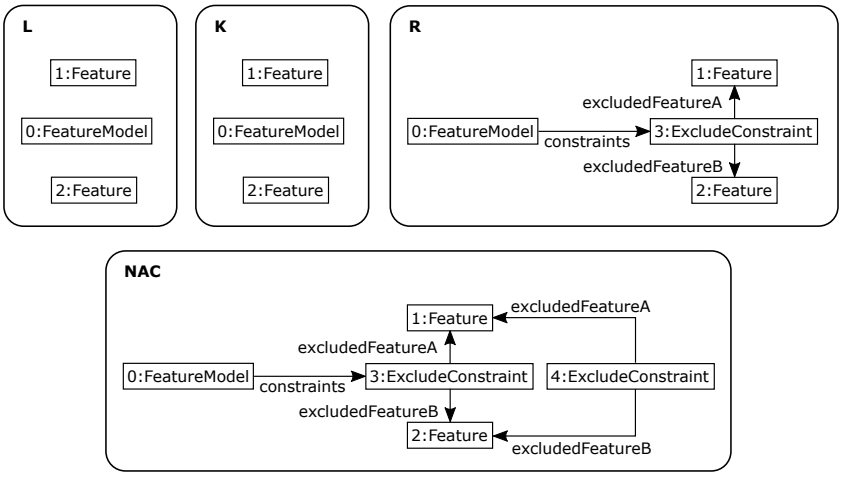

Figure 4 – Formal representation of the rule *createExcludeConstraint* from Fig. 1 (left).

- *NAC is a set of negative application conditions of the form* $(X \supset L)$ *embedding the left-hand side graph* L *of the rule into a NAC graph* X*, and*
- *PAC is a set of positive application conditions of the form* $(X \supset L)$ *embedding the left-hand side graph* L *of the rule into a PAC graph* X*.*

The left-hand side of a rule can have several occurrences in a graph G, an occurrence o being a subgraph isomorphism $o : L \hookrightarrow G$. A rule is applicable at occurrence o if all application conditions are satisfied. This is the case if (i) for every positive application condition $(X_i \supset L) \in PAC$ the subgraph $o(L) \subseteq G$ induced by o can be extended to a subgraph $X_i' \in G$ which is isomorphic to the PAC graph X_i, and (ii) for every negative application condition $(X_j \supset L) \in NAC$ the subgraph $o(L) \subseteq G$ can not be extended to a subgraph $X_j' \in G$ which is isomorphic to the NAC graph X_j. The effects of applying a rule r at occurrence o can be described as follows: The fragment $o(L \setminus K) \subseteq G$ is deleted from G, while the fragment $R \setminus K$ is inserted into G as a fresh copy and connected with $o(K)$ as in the rule r.

Example 2. *As an example, Fig. 4 shows the formal representation of the rule createExcludeConstraint from Fig. 1 (left). Opposite edges of types excludeConstraintsA and excludeConstraintsB are omitted for the sake of brevity. The common elements of the graphs L, K, R and NAC are indicated by symbolic identifiers. Note that in the visual syntax of the Henshin transformation language used in Fig. 1, the graphs L, K, R and NAC are merged to a unified graph, elements to be preserved, created, deleted and forbidden are indicated by respective stereotype notations.*

4 Approach

In this section, we present our approach of using maximum common subgraph algorithms for comparing graph-based transformation rules. The basic idea of obtaining rule graph matchings from maximum common subgraphs is presented in Sect. 4.1, before variation points of a dedicated comparison service are analyzed in Sect. 4.2.

4.1 MCS-induced graph matching

Following common definitions from the field of model comparison, the result of a comparison of two graphs, i.e., a graph matching, is a set of corresponding graph elements which are considered the same in both graphs. Every graph element may have at most one corresponding partner.

Definition 6. *Let $G' = (V', E', \tau'_V)$ and $G'' = (V'', E'', \tau''_V)$ be graphs. A partial injective function $m\colon V' \to V''$ is called a matching. A pair of elements (x, y) with $m(x) = y$ is called a correspondence, the elements x and y are said to correspond to each other.*

The basic idea of using a maximum common subgraph algorithm for calculating a graph matching according to Definition 6 is that, given two graphs $G' = (V', E', \tau'_V)$ and $G'' = (V'', E'', \tau''_V)$, a maximum common subgraph $MCS(G', G'')$ uniquely induces a graph matching $m\colon V' \to V''$: For every $x \in V'$ and $y \in V''$, $m(x) = y$ if and only if there is a $z \in MCS(G', G'')$ such that $cs'(z) = x$ and $cs''(z) = y$.

4.2 Variation points

So far, we have deliberately left open (i) which rule graphs of a transformation rule shall be compared with each other, and (ii) how to select rule graph elements as matching candidates. These variation points are documented in the feature diagram shown in Fig. 5 and analyzed in more detail in the remainder of this section.

4.2.1 Rule graph representation

A first variation point when comparing graph-based transformation rules is the *Rule Graph Representation*. In the *Separate* variant, all the rule graphs (i.e., left- and right hand sides as well as NAC and PAC graphs) are compared separately with each other. On the contrary, in the *Integrated* variant, all the rule graphs are merged into a single unified graph (as in the visual syntax of the Henshin transformation language, cf. Fig. 1 and Fig. 6) and two rules are compared based on their unified rule graphs.

Comparing two rules based on their integrated rule graphs has the advantage that all the graph elements are embedded into the overall context of a transformation rule. Exploiting this context information during comparison reduces the risk of establishing wrong correspondences (i.e., false positives). More generally, our hypothesis is that the *integrated* variant leads to matching results exposing a higher precision than those

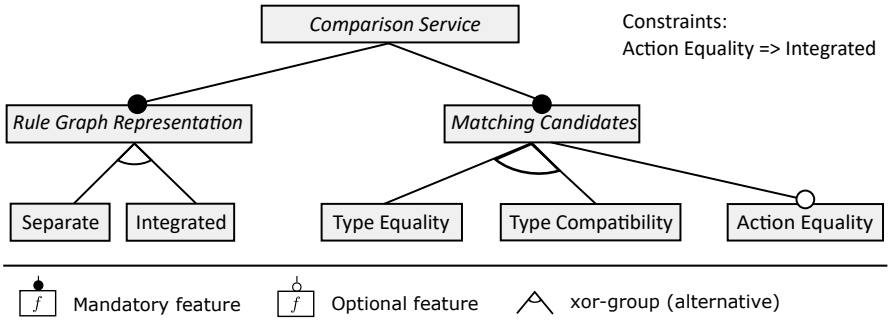

Figure 5 – Variation points of an MCS-based comparison service for graph-based model transformation rules.

obtained in the *Separate* variant. On the contrary, the *Separate* variant may lead to a better runtime performance in the case of large transformation rules which define many actions. In such cases, the integrated rule graph is significantly larger than each of the single rule graphs. Consequently, since the complexity of calculating an MCS is exponential in the size of the input graphs, comparing several smaller rule graphs may be more efficient than comparing a single integrated yet significantly larger one.

Formally, an integrated rule graph is a doubly labeled graph, where labels represent (i) a rule graph element's type as usual and (ii) the action which is to be performed on that element (i.e., whether it is to be preserved, created, deleted, forbidden or required when the rule is applied). As in Definition 1, type labels are drawn from two sets representing the node and edge type definitions of a meta-model, referred to as L_V and L_E. Action labels are drawn from the set $L_A = \{CRE, DEL, PRE, FRB, REQ\}$.

Definition 7 (Integrated rule graph)**.** *An integrated rule graph is a tuple* $G = (V, E, \tau_V, \alpha_V, \alpha_E)$ *where* V *is a finite set of nodes,* $E \subseteq V \times V \times L_E$ *is a set of labeled edges, and* $\tau_V : V \to L_V$ *is a type labeling function assigning every node in* V *its type in* L_V*. Moreover,* $\alpha_V : V \to A$ *and* $\alpha_E : E \to A$ *are action labeling functions assigning every node in* V *and every edge in* E *their action in* A*.*

Given a transformation rule $r = (L \supseteq K \subseteq R, NAC, PAC)$, the construction of its integrated rule graph $G = (V, E, \tau_V, \alpha_V, \alpha_E)$ is largely straightforward. The nodes and edges of G are determined as a unification of all the rule graphs in r, preserving the elements' types. Elements originating from $R \setminus K$, $L \setminus K$, and K are assigned with action labels CRE, DEL and PRE, respectively. Elements that are specific to a NAC graph X_i (i.e., $X_i \setminus L$) are labeled as FRB, while elements specific to a PAC graph X_j (i.e., $X_j \setminus L$) are labeled as REQ.

Note that all the basic definitions presented in Sect. 3 can be easily transferred to integrated rule graphs by simply ignoring its action labels.

4.2.2 Matching candidates

Type equality vs. type compatibility. As described in Sect. 3, the graphs of a graph-based transformation rule are labeled using the node and edge types as labels. Thus, following our basic definitions of Sect. 3, only graph elements having the same type are considered as possible matching candidates during rule graph comparison. However, there are cases where it is reasonable to weaken this *Type Equality* condition into what we refer to as *Type Compatibility* condition. In the latter variant, labels of graph elements are not compared literally, but they are interpreted w.r.t. the underlying meta-model. Two types are compatible if they are equal or specializations of a common supertype. The question of which variant shall be selected depends on whether one prefers a more liberal (*Type Compatibility*) or more conservative matching behavior (*Type Equality*).

To give an illustration, consider the transformation rules *createExcludeConstraint* (see Fig. 1) and *createRequireConstraint* (see Fig. 6). Since the types *ExcludeConstraint* and *RequireConstraint* share a common supertype (cf. meta-model shown in Fig. 2), these nodes will be considered to be corresponding according to the *Type Compatibility* strategy, while no correspondence will be established if the *Type Equality* strategy is applied. Analogously, the *Type Compatibility* variant leads to correspondences between the edges of type *excludedFeatureA* and *feature* as well as *excludedFeatureB* and *requiredFeature*. Since MOF-based meta-modeling frameworks as assumed by our approach do not support edge type inheritance, specialization of

edge types is defined using the MOF 2.0 properties *subset* and *derived union* [AP05]. As we can see in Fig. 2, the edge types *excludedFeatureA* and *feature* (resp. *excludedFeatureB* and *requiredFeature*) are compatible since they are subsetting the edge type *src* (resp. *tgt*) which is defined as their derived union.

Formally, the *Type Compatibility* variant relies on a slightly modified definition of a graph morphism which we refer to as type-compatible morphism.

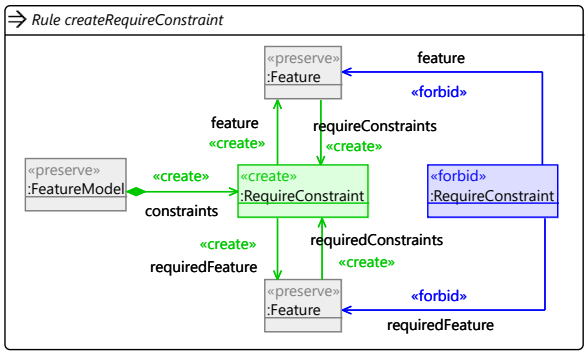

Figure 6 – Henshin rule *createRequireConstraint* taken from the catalog of edit operations on feature models presented in [BKL+16].

Definition 8 (Type-compatible morphism)**.** *Let $G = (V, E, \tau_V)$ and $G' = (V', E', \tau'_V)$ be graphs. A type-compatible graph morphism from G to G' is a function $f : V \to V'$ such that*

(1) for every $v \in V$ we have

(1.a) $\tau_V(v) = \tau'_V(f(v))$, or

(1.b) the node types represented by $\tau_V(v)$ and $\tau'_V(f(v))$ have a common supertype according to the type hierarchy defined by the meta-model

(2) for every edge $e =\in E$ there is an edge $e' \in E'$ such that $f(src(e)) = src(e')$ and $f(tgt(e)) = tgt(e')$, and

(2.a) $\tau_E(e) = \tau'_E(e')$ or

(2.b) there is an edge type in the underlying meta-model which is defined as a derived union of the subsetting edge types represented by $\tau_E(e)$ and $\tau'_E(e')$.

Type-compatible morphisms differ from type-preserving morphisms (see Definition 3) in the relaxed conditions (1) and (2) which are extended by the disjunctions (1.b) (compatibility of node types) and (2.b) (compatibility of edge types). A formal treatment of typing and type hierarchies (cf. condition (1.b)) as well as of subset and derived union properties (cf. condition (2.b)) can be found in [BET12] and [AP05], respectively. Analogously to the basic definitions presented in Sect. 3, the definition of a type-compatible graph morphism can be transferred to integrated rule graphs by simply ignoring its action labels.

Action equality. An optional feature is whether the elements of an integrated rule graph must have the same action in order to be considered as matching candidates. Note that *Action Equality* may only be selected if rule graphs are represented in the *Integrated* variant (see propositional formula over feature variables in Fig. 5).

One typically abstains from selecting the feature *Action Equality* if one wants to match larger structural patterns, regardless of the actions performed by the individual elements of that pattern. An example has been already discussed in Sect. 2, where large parts of the rules *createExcludeConstraint* and *deleteExcludeConstraint* (cf. Fig. 1)

are structurally equivalent yet performing different actions (creation vs. deletion of an *ExcludeConstraint*, cf. Fig. 1). On the contrary, requiring *Action Equality* leads to a more conservative selection of matching candidates, and the matching behavior is similar to the detection of clones in the *RuleMerger* approach which relies on the notion of subrules.

Formally, analogously to the *Type Compatibility* feature, an implementation of the optional feature *Action Equality* is based on a slightly modified definition of a graph morphism to which we refer to as action-preserving morphism.

Definition 9 (Action-preserving morphism)**.** *Let $G = (V, E, \tau_V, \alpha_V, \alpha_E)$ and $G' = (V', E', \tau'_V, \alpha'_V, \alpha'_E)$ be integrated rule graphs. An action-preserving graph morphism from G to G' is a function $f : V \to V'$ such that*

(1): for every $v \in V$ we have $\alpha_V(v) = \alpha'_V(f(v))$, and

(2): for every edge $e =\in E$ there is an edge $e' \in E'$ such that $f(src(e)) = src(e')$ and $f(tgt(e)) = tgt(e')$, and $\alpha_E(e) = \alpha'_E(e')$.

Note that the definition of an action-preserving morphism ignores the types attached to nodes and edges. This way, the interaction *Action Equality* & *Type Equality* can be easily realized by conjunctions of the conditions (1) and (2) of Definition 9 and Definition 3. Likewise, the interaction *Action Equality* & *Type Compatibility* can be realized by conjunctions of the conditions (1) and (2) of Definition 9 and Definition 8.

5 Prototypical implementation

The prototypical implementation of our approach supports the comparison of Henshin transformation rules and is included in the replication package of this paper [SBK19], including instructions of how to use the comparison service as a Java library.

Sect. 5.1 motivates our selection of a concrete implementation of a maximum common subgraph algorithm. Variation points identified in the previous section are implemented by integrating the algorithmic components with Henshin and EMF, as we will briefly outline in Sect. 5.2.

5.1 Maximum common subgraph algorithm

Conte et al. [CFV07] report about a performance benchmark of three state-of-the-art MCS-algorithms — McGregor, Durand-Pasari and Balas Yu — on a large set of randomly connected graphs with varying size and density. According to the experimental results, McGregor always performs best on sparse graphs (n = 0.05) and, for an increasing density (n = 0.1), it is still outperforming Durand-Pasari and Balas-Yu when the graphs are small or the label alphabets are large.

For our application scenario of comparing graph-based transformation rules, we expect that graphs are typically sparse and small, which suggests to select the McGregor algorithm for our purpose. Specifically, we use the implementation of the McGregor algorithm presented in [Wel11], which is written in Java and based on the widely used graph library JGraphT, and thus can be easily integrated with the other technologies used by our prototypical implementation.

5.2 Integration with Henshin and EMF

The choice of whether rule graphs are compared based on an integrated representation or separately is made when converting the Henshin rule graphs to JGraphT. To that end, we provide two converters covering both rule graph representations. Each of the converters supports a bidirectional transformation in order to transfer the comparison results back to the Henshin representation.

As for the selection of matching candidates, we slightly extend the implementation of the McGregor algorithm of Welling [Wel11]. An extension point enables to register a custom candidate selection strategy. The check for type and action equality boils down to a simple check of the equality of labels. The check for type compatibility is implemented using the reflective API of the EMF runtime.

6 Evaluation

For the evaluation of our approach, we conduct experiments on several sets of transformation rules with the aim of answering the following research questions:

- **RQ1**: How do the variation points of our approach affect the matching quality and is there an optimal configuration?
- **RQ2**: How does our approach perform w.r.t. to competing approaches for the comparison of model transformations?
- **RQ3**: Does our approach produce satisfactory results within reasonable time?

6.1 Experimental Setup

6.1.1 Subject Selection.

Strüber et al. [SKA+16] present a model transformation benchmark set which we use as subject for our experimental evaluation. The benchmark set comprises different kinds of graph-based model transformation rules, written in the model transformation language Henshin. In total, we have 9 sets of different kinds of transformation rules.

First, *edit rules* specify edit operations offered by visual model editors and model refactoring tools, covering feature models and parts of the UML. Both sets are further classified into so-called elementary (*fm-edit-atomic* and *uml-edit-generated*) and composite (*fm-edit-complex* and *uml-edit-manual*) edit rules. Second, edit operation *recognition rules* derived from these edit rules specify change patterns that can be observed as a result of executing the respective edit operation [KKT11]. Consequently, there are four different kinds of recognition rules, namely *fm-recog-atomic*, *fm-recog-complex*, *uml-recog-generated* and *uml-recog-manual*. Finally, constraint *translation rules* (*ocl2ngc*) support the translation of OCL constraints to nested graph constraints [AHRT14], which can then be translated to application conditions of transformation rules.

As argued in related work [SPA16], the rule sets of the selected benchmark comprise realistic, non-trivial transformation rules of varying size and complexity. The average number of rule graph nodes ranges from 4.62 (*uml-edit-generated*) to 58.65 (*fm-recog-complex*), while the average number of edges ranges from 1.90 to 111.39. For the integrated rule graphs which are used internally by some configurations of our matching algorithm, average numbers (#nodes/#edges) of these rule sets range from 2.53/1.90 to 30.26/62.03.

6.1.2 Preparation of Rule Pairs.

To obtain pairs of transformation rules which are supposed to be compared, each of the transformation rules in our 9 rule sets is first modified by a single *permutation*, followed by an increasing number (0, 1, and 3) of *mutations*. While the permutation only affects the physical representation of a rule (i.e., the ordering of rule graph elements), mutations perform (a sequence of) modifications of a rule's conceptual contents (i.e., edit operations available in the visual Henshin editor). In other words, an original rule and its permutated copy are structurally identical (up to their physical representation) and thus semantically equivalent, while mutations lead to real structural and thus semantic changes. For each rule, this modification process is *repeated* 10 times in order to mitigate both performance benchmarking issues resulting from the Java just-in-time compilation, and randomization issues resulting from the permutation and mutation of rule graph elements. Each of the modified copies is supposed to be compared with the original version of the rule, leading to $3 \times 10 = 30$ comparison cases (i.e., rule pairs) per transformation rule.

Permutation. Although there is no reasonable natural order of how the rule graph elements of a Henshin transformation rule are arranged, all the elements are implicitly ordered due to the EMF-based implementation which basically uses an extension of `java.util.List` as container class for any element. Since the order in which the elements are processed may influence both the accuracy of the matching results and the performance of the matching algorithm, we randomly permute the rule graph elements of a copied rule to avoid this bias.

Mutation. Henshin rules are mutated by applying three kinds of edit operations (insert, update, delete) that are also available in the visual editor of the Henshin tool suite. A mutation step can be considered as one of the atomic changes a user could apply to a Henshin rule using its visual editor. Technically, the mutator takes a rule as input and applies a specified number of edit operations. Insert operations add a new element to the rule (e.g., a new node), update operations locally change an existing rule element (e.g., the type of an existing node), and delete operations remove an existing rule element. Inserts and updates of rule graph elements are further characterized by a new action (create, delete, preserve, require, forbid) which is to be performed by the inserted or updated element when the rule is applied to a model. Edit operations as well as the context in which they are to be applied are chosen randomly. Our mutator applies an edit operation only if its application results in a graph which is properly typed. Otherwise, another operation is selected randomly. Since some of the rules of our benchmark set comprise only a very few rule graph elements, we limit the maximum number of mutations applied to a rule to 3.

6.2 RQ1: Assessing the accuracy of different configurations

To answer RQ1, we assess the accuracy of the different configurations of our matching algorithm using our prototypical implementation and the rule pairs obtained from our selected benchmark set as described in the previous section.

Since we aim at experimenting under realistic conditions, we limit the calculation time for each comparison run to one second. If a timeout is reached, the matching result is calculated based on the largest common subgraph the algorithm has found so far. We argue that the selected timeout is reasonable for our envisioned versioning

scenario. For example, if a developer compares a local workspace version with its latest repository version, we believe that latency times of a second or less are acceptable.

Configurations. According to the variability analysis conducted in Sect. 4.2, binding the variation points *Rule Graph Representation* and *Matching Candidates* leads to different variants of our matching algorithm. The feature diagram presented in Fig. 5 yields the following six valid configurations, each of which is supported by our prototypical implementation: Separate-TE := {*Separate*, *Type Equality*}, Separate-TC := {*Separate*, *Type Compability*}, Integrated-TE := {*Integrated*, *Type Equality*}, Integrated-TC := {*Integrated*, *Type Compability*}, Integrated-TE-AE := {*Integrated*, *Type Equality*, *Action Equality*}, and Integrated-TC-AE := {*Integrated*, *Type Compability*, *Action Equality*}.

Accuracy measures. As already mentioned in Sect. 2, reliable and persistent UUIDs attached to rule graph elements are not generally available, e.g., due to exchange across tool boundaries, parallel edits, etc. [KDRPP09]. For our controlled experiments, however, we attached UUIDs to all graph elements, and the permutations and mutations described in Sect. 6.1.2 have been implemented in a UUID-preserving manner. This way, UUIDs can be exploited to serve as an oracle in order to assess the accuracy of the matching results, according to the following classification scheme:

- *True Positive (TP)*: The comparison algorithm delivers a correspondence and the corresponding rule graph elements have the same UUID.
- *False Positive (FP)*: The comparison algorithm delivers a correspondence and the corresponding rule graph elements have different UUIDs.
- *False Negative (FN)*: Two rule graph elements have the same UUID and the comparison algorithm does not deliver a correspondence between these elements.

Based on the classification into TPs, FPs and FNs, we obtain the following well-known accuracy metrics:

$$\text{Precision} := \frac{\#TP}{\#TP+\#FP} \qquad \text{Recall} := \frac{\#TP}{\#TP+\#FN} \qquad \text{F-Measure} := \frac{2 \cdot Precision \cdot Recall}{Precision+Recall}$$

Results. The results of applying all the six configurations of our comparison algorithm to rule pairs exposing increasing differences (0, 1 and 3 mutations) are summarized in Table 1. Precision, recall and F-Measure denote average values over comparison runs on all rule pairs (cf. Sect. 6.1.2).

A first observation when comparing a rule with a modified copy of itself (0 mutations) is that none of the configurations achieves perfect scores for every rule set. The reason for this is that not all the correspondences are found if the calculation of the MCS reaches its timeout of one second and thus has to be aborted. Specifically, the *fm-recog-complex* rule set causes several timeouts of the MCS calculation, regardless of the concrete configuration. Moreover, rule graphs may share several maximum common subgraph mappings that are semantically equivalent. Arguably, each of them yields a perfect matching, but only one of them is considered to be correct and our algorithm may choose the "wrong" one. Such deviations from perfect accuracy scores, however, are rather a problem of our measurement methodology which simply lacks a better oracle than UUIDs.

Not very surprisingly, although not significantly, the highest accuracy when comparing a rule with a copy of itself is achieved by Integrated-TE-AE, which is the most

Table 1 – Results of applying all the six configurations of our comparison algorithm to rule pairs exposing increasing differences.

	0 Mutations			1 Mutation			3 Mutations		
Algorithm	**Prec.**	**Rec.**	**FM**	**Prec.**	**Rec.**	**FM**	**Prec.**	**Rec.**	**FM**
Integrated-TE	0.89	0.88	0.88	0.86	0.86	0.86	0.81	0.80	0.80
Integrated-TC	0.88	0.87	0.88	0.86	0.86	0.86	0.81	0.80	0.80
Integrated-TE-AE	0.90	0.89	0.90	0.86	0.75	0.78	0.78	0.55	0.62
Integrated-TC-AE	0.90	0.89	0.89	0.86	0.74	0.78	0.77	0.55	0.61
Separate-TE	0.90	0.89	0.89	0.86	0.86	0.86	0.81	0.81	0.80
Separate-TC	0.90	0.88	0.89	0.86	0.86	0.86	0.81	0.81	0.81

conservative of our six configurations. However, after the first mutation, it turns out that requiring action equality for integrated rule graphs (*Integrated-TE-AE* and *Integrated-TC-AE*) is way too restrictive (cf. discussion in Sect. 2). The more liberal variants (*Integrated-TE* and *Integrated-TC*) and the variants working on separate rule graphs (*Separate-TE* and *Separate-TC*) are fairly comparable. In sum, similar results are achieved by all of these configurations. However, a more detailed investigation shows that the separate variants perform better on rule sets with simple rule graphs, and the integrated variants perform better on sets with more complex rules. For complex rules, the integrated graph contains more structural information that can be used in order to find the correct matching. For simple rule graphs, the integrated rule graph adds only little structural information.

Finally, there is no significant difference between the integrated variants *Integrated-TE* and *Integrated-TC*. This may be explained by our experimental setup. Our mutator applies an edit operation only if its application results in a graph which is properly typed. However, the random selection of a new node type which is permitted in the given context is not likely.

Integrated representations of rule graphs lead to slightly higher accuracies of matching results on complex rules than comparing rule graphs separately, due to additional context information which may be exploited by the matcher. However, integrated representations lead to lower accuracies on simple rules due to adding only little structural information. In the case of comparing mutated rules with each other, a liberal matching strategy needs to be selected which does not require action equality.

6.3 RQ2: Comparison to EMFCompare and RuleMerger

Baseline selection. For answering RQ2, we compare our most accurate configurations *Integrated-TE* and *Separate-TE* (see RQ1) with *EMFCompare* and the three clone detection algorithms ConQat, EScan and ScanQat supported by *RuleMerger* [SPA16]. For the latter, correspondences are derived from the largest clone detected for a rule pair, similar to calculating an MCS-induced matching. For each comparison run, timeouts are set to one second, analogously to RQ1. Moreover, accuracy measures are obtained in the same way as for RQ1.

Results. Table 2 presents the average results of the six algorithms over all comparison runs, grouped by our 9 rule sets. As expected, due to its heuristic approach, which does not apply well to graph-based transformation rules (see Sect. 2), *EMFCompare* delivers highly inaccurate results already for the trivial case of comparing a rule with a permutated copy of itself. This applies in particular to larger rule graphs with more than 10 elements. Interestingly, as opposed to the MCS-based algorithms, its accuracy

Table 2 – Results of the comparison algorithms on the 9 rule sets. All results are average values over the rules of the corresponding rule set and the 10 independent comparisons of each rule with its modified copies.

	0 Mutations			1 Mutation			3 Mutations		
Algorithm	**Prec.**	**Rec.**	**FM**	**Prec.**	**Rec.**	**FM**	**Prec.**	**Rec.**	**FM**
fm-edit-atomic - 27 rules, 5.7 Nodes, 4.6 Edges									
MCS-Integrated-TE	**0.90**	**0.90**	**0.90**	0.79	0.81	0.80	0.71	**0.75**	**0.73**
MCS-Separate-TE	0.89	0.89	0.89	0.80	**0.83**	**0.81**	0.69	0.74	0.71
EMFCompare	0.85	0.77	0.80	**0.82**	0.65	0.71	**0.75**	0.52	0.60
ConQat	0.81	0.73	0.76	0.54	0.45	0.48	0.36	0.27	0.30
EScan	0.70	0.63	0.66	0.56	0.46	0.50	0.46	0.36	0.39
ScanQat	0.39	0.38	0.38	0.32	0.27	0.29	0.22	0.18	0.19
fm-edit-complex - 31 rules, 11.5 Nodes, 15.6 Edges									
MCS-Integrated-TE	0.80	**0.80**	**0.80**	0.77	**0.78**	**0.77**	**0.69**	0.69	0.68
MCS-Separate-TE	0.78	0.78	0.78	0.73	0.75	0.74	**0.69**	**0.72**	**0.70**
EMFCompare	0.68	0.50	0.57	0.67	0.45	0.54	0.68	0.42	0.51
ConQat	**0.83**	0.61	0.69	**0.79**	0.54	0.63	**0.69**	0.43	0.51
EScan	0.57	0.40	0.46	0.58	0.40	0.47	0.52	0.35	0.41
ScanQat	0.28	0.20	0.23	0.29	0.21	0.24	0.28	0.19	0.22
fm-recog-atomic - 27 rules, 21.7 Nodes, 29.6 Edges									
MCS-Integrated-TE	0.92	0.92	0.92	0.93	**0.93**	**0.93**	0.89	0.86	0.87
MCS-Separate-TE	0.93	**0.93**	**0.93**	0.91	0.92	0.91	0.88	**0.89**	**0.88**
EMFCompare	0.64	0.40	0.49	0.63	0.38	0.47	0.65	0.39	0.48
ConQat	**0.95**	0.91	**0.93**	**0.96**	0.87	0.91	**0.96**	0.76	0.83
EScan	0.00	0.00	0.00	0.03	0.02	0.02	0.16	0.09	0.11
ScanQat	0.14	0.14	0.14	0.20	0.18	0.19	0.30	0.22	0.25
fm-recog-complex - 31 rules, 58.65 Nodes, 111.39 Edges									
MCS-Integrated-TE	0.72	**0.66**	**0.68**	**0.73**	**0.67**	**0.69**	0.69	**0.60**	**0.64**
MCS-Separate-TE	**0.73**	0.65	**0.68**	0.72	0.64	0.67	**0.70**	**0.60**	**0.64**
EMFCompare	0.33	0.17	0.22	0.33	0.17	0.22	0.34	0.17	0.22
ConQat	0.70	0.57	0.62	0.71	0.57	0.62	**0.70**	0.53	0.59
EScan	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.01	0.01
ScanQat	0.05	0.04	0.04	0.05	0.03	0.04	0.05	0.03	0.04
ocl2ngc - 52 rules, 31.9 Nodes, 42.5 Edges									
MCS-Integrated-TE	0.94	0.94	0.94	0.93	0.93	0.93	0.88	0.87	0.87
MCS-Separate-TE	0.95	0.95	0.95	0.94	**0.95**	0.94	0.90	**0.88**	**0.89**
EMFCompare	0.62	0.35	0.44	0.62	0.35	0.44	0.63	0.33	0.43
ConQat	**1.00**	**1.00**	**1.00**	**1.00**	0.94	**0.96**	**0.99**	0.81	0.88
EScan	0.08	0.08	0.08	0.14	0.13	0.13	0.21	0.17	0.19
ScanQat	0.07	0.09	0.08	0.18	0.18	0.18	0.30	0.27	0.28
uml-edit-generated - 1379 rules, 4.6 Nodes, 1.9 Edges									
MCS-Integrated-TE	0.91	0.91	0.91	0.89	0.90	0.90	**0.82**	**0.85**	**0.83**
MCS-Separate-TE	**1.00**	**1.00**	**1.00**	**0.94**	**0.96**	**0.95**	0.81	**0.85**	**0.83**
EMFCompare	0.93	0.86	0.89	0.90	0.79	0.83	0.81	0.65	0.71
ConQat	0.72	0.64	0.67	0.54	0.42	0.46	0.24	0.18	0.20
EScan	0.71	0.63	0.66	0.57	0.46	0.50	0.32	0.24	0.27
ScanQat	0.43	0.39	0.40	0.39	0.31	0.34	0.20	0.15	0.17
uml-edit-manual - 25 rules, 5.2 Nodes, 4.1 Edges									
MCS-Integrated-TE	0.92	0.92	0.92	**0.90**	**0.91**	**0.91**	0.82	0.85	0.83
MCS-Separate-TE	**0.95**	**0.95**	**0.95**	**0.90**	**0.91**	**0.91**	**0.86**	**0.90**	**0.87**
EMFCompare	0.81	0.70	0.74	0.81	0.66	0.72	0.82	0.58	0.67
ConQat	0.92	0.85	0.88	0.59	0.48	0.52	0.23	0.16	0.18
EScan	0.74	0.69	0.71	0.59	0.51	0.54	0.27	0.22	0.24
ScanQat	0.49	0.47	0.48	0.38	0.32	0.34	0.18	0.15	0.16
uml-recog-generated - 1379 rules, 18.2 Nodes, 21.5 Edges									
MCS-Integrated-TE	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.97	**0.93**	**0.94**
MCS-Separate-TE	**1.00**	**1.00**	**1.00**	0.99	**1.00**	0.99	0.94	**0.93**	0.93
EMFCompare	0.71	0.44	0.54	0.72	0.43	0.54	0.72	0.42	0.53
ConQat	**1.00**	0.98	0.99	**1.00**	0.91	0.94	**0.99**	0.76	0.84
EScan	0.00	0.00	0.00	0.08	0.05	0.06	0.25	0.14	0.17
ScanQat	0.13	0.14	0.14	0.21	0.18	0.19	0.33	0.23	0.26
uml-recog-manual - 25 rules, 26.6 Nodes, 42.2 Edges									
MCS-Integrated-TE	0.86	0.84	0.85	0.84	0.83	0.83	0.83	**0.80**	**0.81**
MCS-Separate-TE	0.85	0.82	0.83	0.83	0.82	0.82	0.80	0.78	0.78
EMFCompare	0.57	0.35	0.43	0.57	0.35	0.43	0.59	0.34	0.43
ConQat	**0.98**	**0.96**	**0.97**	**0.96**	**0.88**	**0.91**	**0.94**	0.73	**0.81**
EScan	0.00	0.00	0.00	0.06	0.04	0.05	0.26	0.16	0.19
ScanQat	0.10	0.11	0.10	0.15	0.14	0.14	0.25	0.17	0.19

does not continuously decrease with an increasing amount of mutations. This is due to the fact that the heuristic approach employed by *EMFCompare* is less sensitive against mutations than our MCS-based approach. Moreover, none of the comparisons with *EMFCompare* was aborted by a timeout, demonstrating the runtime efficiency of its matching algorithm. Nonetheless, MCS-based comparison still outperforms *EMFCompare* w.r.t. precision and recall in all the cases, even in those cases where matchings are derived from an incomplete MCS-calculation.

Among the clone detection variants, the most accurate results are delivered by ConQat. Given the timeout of one second, EScan suffers from being interrupted for most larger rules. In such cases, no correspondences are derived at all since EScan does not deliver any intermediate results. This confirms the theoretical expectation and experimental results presented in [SPA16] that ConQat tends to provide the best compromise between accuracy and performance among the three clone detection algorithms. For most of the cases without any mutation, ConQat even delivers slightly better results than our MCS-based approach, however, potentially biased by the oracle problem with UUIDs (see RQ1). On the contrary, compared to our approach, clone detection is highly sensitive against mutations. Looking at the results for 3 mutations, ConQat delivers lower accuracy values than our approach, a trend which we expect to be even more significant with an increasing number of mutations. We see this as a confirmation of our hypothesis that clone detection is too restrictive in order to be exploited for the sake of transformation rule comparison in the context of model transformation versioning.

The MCS-based comparison of graph-based model transformation rules significantly increases the accuracy of comparison results compared to generic model comparison with *EMFCompare*. Clone detection, even when customized to graph-based model transformations, is too restrictive when the rules to be compared with each each other expose an edit distance of only a very few edit operations.

6.4 RQ3: Runtime Performance of the Matching Algorithm

In order to answer RQ3, we first investigated whether it is possible to improve the matching results of our MCS-based approach by increasing the timeout. The experimental subject *fm-recog-complex* contains several complex rules with more than 50 nodes and 100 edges on average. For these rules, the calculation of the MCS often fails to finish in time and only a intermediate result can be retrieved. Hence, there was reason to assume that our approach could achieve better matchings if more

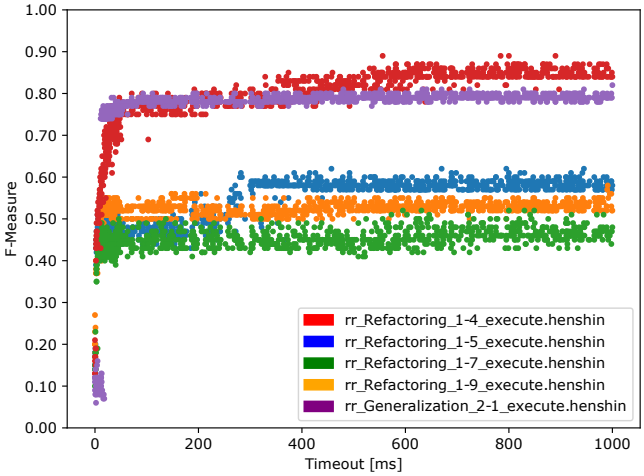

Figure 7 – Quality of intermediate matches provided by MCS-Integrated-TE on five complex rules after different timeouts.

computation time is given. However, we were not able to observe a significant improvement, even after setting the timeout from one second to 60 seconds. This leaves us with the question how far we can reduce the timeout without negatively affecting the matching quality.

Therefore, we selected a subset of *fm-recog-complex* that contains five of the most complex rules in this set and evaluated the matchings provided by MCS-Integrated-TE after varying timeouts. The timeout of each run is selected randomly from the range of 1ms to 1,000ms. Figure 7 shows the F-Measures of 1,000 runs per rule w.r.t. the selected timeout. The results show that the matching quality increases significantly in the first 100ms and then starts to converge. This is because the MCS algorithm quickly finds a large number of matches in a rule for those elements having only a single matching candidate. Thereafter, the search for further matches among larger candidate sets requires more time as the algorithm checks each possibility through backtracking [Wel11]. For four of the rules, there is no noticeable increase in F-Measure after 800ms, while the F-Measure of the results on *Refactoring_1-7* increases only slightly. In conclusion, a timeout of 1,000ms provides enough time for our approach to achieve satisfactory results on even the most complex rules of the benchmark set.

Increasing the timeout from one second to 60 seconds for complex rules shows no improvement in the match quality of our MCS-based approach. Our approach requires only a few hundred milliseconds to achieve satisfactory results w.r.t. its best possible results.

6.5 Threats to Validity

A threat to the external validity pertains the question whether our benchmark set is representative. However, up to the best of our knowledge, the chosen benchmark set is the most comprehensive collection of graph-based transformation rules. It contains a selection of transformation rules which stem from different application scenarios and are typed over different meta-models.

As for construct validity, the randomly chosen permutation and mutation operators and their parameters may impact the accuracy of our matching algorithm in the conducted experiments. We mitigate this by experimenting on many rules and several runs on each of them.

7 Conclusion

A dedicated comparison service is a key technique to effectively support the versioning of model transformations. However, providing such a service for graph-based transformation languages is highly challenging and hard, if not impossible to realize using generic model matchers. In this paper, we proposed to use maximum common subgraph algorithms as a technical basis and discussed major variation points of such an approach. Our central hypothesis that an MCS-based comparison approach is feasible for typical graph-based transformation rules is confirmed by our experimental results. In a comparative study, our approach outperforms the most closely related competitors for typical comparison scenarios in the context of model versioning.

As for future work, we want to further mitigate the threats to validity of our experimental results by incorporating additional experimental subjects including larger rule graphs, and by studying the correlation between the applied mutation

operators and the accuracy of the matching algorithms in more detail. As a technical enhancement, we pursue a hybrid approach in which MCS-based and similarity-based comparison are interwined. To support the comparison of transformation systems in which rules are arranged in "programmed" workflows, correspondences identified by an MCS-based matching may serve as anchor points for the similarity propagation of a subsequent similarity-based comparison of an entire transformation system. Moreover, in order to deal with the problem of meta-model evolution, we want to weaken our basic assumption that rule graphs are correctly typed over a fixed meta-model.

References

[AHRT14] Thorsten Arendt, Annegret Habel, Hendrik Radke, and Gabriele Taentzer. From core OCL invariants to nested graph constraints. In *ICGT*, 2014.

[AP05] Marcus Alanen and Ivan Porres. Subset and union properties in modeling languages. Technical report, Technical Report 731, TUCS, 2005.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *Journal of Web Information Systems*, 5(3):271–304, 2009.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[BET12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.

[BKL+16] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about product-line evolution using complex feature model differences. *ASE J.*, 23(4):687–733, 2016.

[BP08] Cédric Brun and Alfonso Pierantonio. Model differences in the Eclipse Modeling Framework. *UPGRADE*, 9(2), 2008.

[BTW14] Alexander Bergmayr, Javier Troya, and Manuel Wimmer. From out-place transformation evolution to in-place model patching. In *ASE*, pages 647–652. ACM, 2014.

[CFSV04] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *Journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 11(1):99–143, 2007.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie Mellon University, 1990.

[KDRPP09] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *CVSM@ICSE*, 2009.

[KKPS12] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of model comparison tools. In *ASE*, 2012.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ASE*, 2011.

[KPS17] Timo Kehrer, Christopher Pietsch, and Daniel Strüber. Differencing of model transformation rules: Towards versioning support in the development and maintenance of model transformations. In *ICMT*, 2017.

[MGMR02] S Melnik, H Garcia-Molina, and E Rahm. A versatile graph matching algorithm and its application to schema matching. In *Intl. Conf. on Data Engineering (ICDE)*, 2002.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for EMF model transformation development. In *ICMT*, 2017.

[SBK19] Alexander Schultheiß, Alexander Boll, and Timo Kehrer. Supplementary material of this paper (replication package). *https://www.informatik.hu-berlin.de/de/forschung/gebiete/mse/MCS.zip*, 2019.

[SC13] Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. In *Modelsward*, 2013.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5), 2003.

[SKA+16] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. Scalability of model transformations: Position paper and benchmark set. In *BigMDE@STAF*, 2016.

[SPA16] Daniel Strüber, Jennifer Plöger, and Vlad Acreţoaie. Clone detection for graph-based model transformation languages. In *ICMT*, 2016.

[SRA+16] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. Rulemerger: Automatic construction of variability-based model transformation rules. In *FASE*, 2016.

[TAEH12] Gabriele Taentzer, Thorsten Arendt, Claudia Ermel, and Reiko Heckel. Towards refactoring of rule-based, in-place model transformation systems. In *AMT@MoDELS*, 2012.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *ESEC/FSE*, 2007.

[Tea10] CMMI Product Team. Cmmi for development, version 1.3. Technical Report CMU/SEI-2010-TR-033, Carnegie Mellon University, 2010.

[VB07] Dániel Varró and András Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[Wel11] Ruud Welling. A performance analysis on maximal common subgraph algorithms. In *15th Twente Student Conference on IT, University of Twente*, volume 14, 2011.

[WHR+13] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *MoDELS*, 2013.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE*, 2005.

Acknowledgments This work has been supported by the German Research Foundation (DFG) under grant KE 2267/1-1 and the German Ministry of Research and Education (BMBF) under grant 01IS18091B.