An Algorithm for the Critical Pair Analysis of Amalgamated Graph Transformations *

Kristopher Born, Gabriele Taentzer

Philipps-Universität Marburg, Germany
{born,taentzer}@informatik.uni-marburg.de

Abstract. Graph transformation has been shown to be well suited as formal foundation for model transformations. While simple model changes may be specified by simple transformation rules, this is usually not sufficient for more complex changes. In these situations, the concept of amalgamated transformation has been increasingly often used to model *for each* loops of rule applications which coincide in common core actions. Such a loop can be specified by a kernel rule and a set of extending multi-rules forming an interaction scheme.
The Critical Pair Analysis (CPA) can be used to show local confluence of graph transformation systems. Each critical pair reports on a potential conflict between two rules. It has been shown recently that the generally infinite set of critical pairs for interaction schemes can be reduced to a finite set of non-redundant pairs being sufficient to show local confluence of the transformation system. Building on this basic result, we present an algorithm that is able to compute all non-redundant critical pairs for two given interaction schemes. The algorithm is implemented for Henshin, a model transformation environment based on graph transformation concepts.

1 Introduction

In model-based software development, models play a primary role w.r.t. requirements elicitation, software design and software validation. Model changes can be well specified as model transformations. If several developers work concurrently on the same model, they may run into conflicts that have to be resolved. For the execution of several model changes, a specific order may be necessary due to causal dependencies. To analyze such conflicts and dependencies as early as possible, critical pair analysis (CPA) [18, 8] has been used. This analysis allows to check transformation rules for potential conflicts and dependencies at specification time, i.e., before run time. A critical pair describes a minimal conflicting situation that may occur in the transformation system. If every critical pair can be resolved by finitely many transformation steps, the system is locally confluent. Potential dependencies between rules can be discovered by inverting the first

* This work was partially funded the German Research Foundation, Priority Program SPP 1593 "Design for Future - Managed Software Evolution".

rule and using it as input to the CPA, together with the second rule. In that case, local confluence of critical pairs show how resulting models of dependent transformations can be reached alternatively.

Conflicts as well as dependencies of model transformations have been analyzed by the CPA for several different applications as, e.g., finding conflicts and dependencies in functional requirement specifications of software systems [11], analyzing conflicts and dependencies of model refactorings [17] as well as in aspect-oriented modeling [16], and using conflict and dependency results to find the right order of edit operations for reporting model differences on an application-specific abstraction level [13].

While simple model changes can be well specified using simple rules, this is usually not sufficient for more complex model changes. Amalgamated graph transformation has proven to be suitable for specifying core actions equipped with a number of optional or context-dependent actions. Considered applications are, e.g., an interpreter semantics for statecharts [4], automatic model migration [15], and the specification of complex model edit operations [13]. A typical example of such complex model changes are model refactorings where, e.g., equal attributes in subclasses are pulled up to one attribute in their super class. Collaborative working developers are interested in understanding when model changes can be applied in parallel and when they are a potential source for conflicts. Being in conflict, it would be interesting to understand if and how these conflicts can be resolved. Hence, the notions of parallel independence, conflict and conflict resolution have to be lifted to amalgamated graph transformation.

An amalgamated transformation is specified by a interaction scheme containing a kernel rule and a set of extending multi-rules. While the kernel rule is intended to be matched exactly once, each multi-rule is matched as often as possible – in the general case, a fixed, but arbitrary number of times. An amalgamated rule over an interaction scheme contains at least the kernel rule and arbitrary many copies of multi-rules overlapping at the kernel rule. Hence, an interaction scheme specifies infinitely many amalgamated rules in general.

Applying the CPA to analyze conflicts and dependencies between interaction schemes confronts us with the problem to check infinitely many rule pairs and therefore, critical pairs. [22] shows that a finite set of CPs is enough to show local confluence of the overall transformation system. This result is proven for algebraic graph transformation [8]. Model transformations that are based on the Eclipse Modeling Framework (EMF) have been formally based on graph transformation in [5].

To apply the CPA to amalgamated transformations in practice, we need an algorithm that implements it. The main challenge is to find out an effective termination criterion when enumerating pairs of amalgamated rules and their critical pairs. The main contributions of this paper are the following:

1. An algorithm for the CPA of pairs of interaction schemes. We argue for the correctness of the presented algorithm w.r.t. the underlying theory. In particular, we focus on the termination of this algorithm.

2. An implementation of the algorithm within the model transformation tool Henshin based on EMF models.
3. First tests of the algorithm: We report on the CPA of an example transformation system, focussing on termination issues.

The paper is organized as follows: The main concepts of amalgamated graph transformation are recalled in Section 2. The main ideas for the CPA for amalgamated graph transformation are summarized in Section 3. Thereafter, we present our algorithm and argue for its correctness in Section 4.

2 Amalgamated Transformations

In the context of graph transformation, amalgamated transformation has been introduced to perform a kernel action exactly once and context-dependent actions as often as possible. In this section, we consider amalgamated transformations based on EMF [21] and use model refactorings as running example. The formal basis is given by amalgamated graph transformation as presented in [9] and the conflict analysis in [22]. Since the subtle differences do not play a role throughout this paper, we use the notions model and graph as synonyms in the following.

EMF is a common and widely-used open source technology in model-based software development. It extends Eclipse by modeling facilities and allows for defining (meta-)models and modeling languages by means of structured data models. An EMF-model can formally be considered as an instance graph with a prominent containment hierarchy.

Example 1 (Simple class models). In the running example, we consider selected refactorings of simple class models. A simple class model consists of a package being the container of all classes. A class is named and may have any number of attributes just given by their names. Classes may be related in two ways: A class may have a superclass and any number of references to other classes. The meta-model for simple class models is shown in Figure 1.

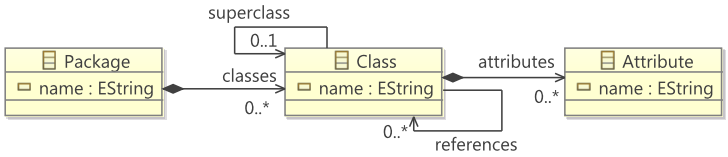

Fig. 1. Meta-model for class models

A very simple instance model to this meta-model is shown in Figure 2; it represents two classes “List” and “Stack” where the stack is inheriting from the list. Class “List” has an attribute called “first”. Since this design is not optimal, it will be refactored later on.

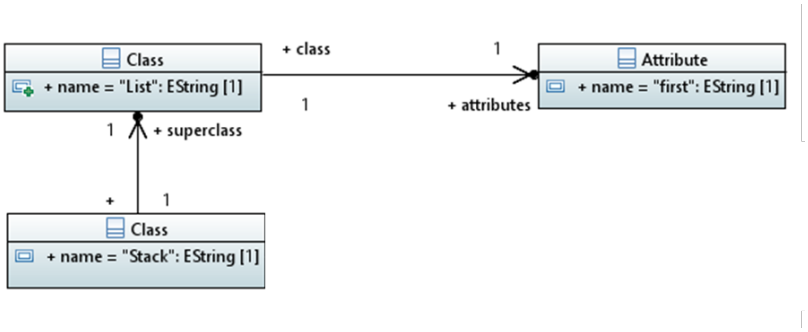

Fig. 2. Example instance model

In the context of EMF, refactorings are specified by model transformations. See e.g. [14] and [3]. In the following, we consider rule-based model transformations formulated in Henshin [2], a model transformation language based on graph transformation concepts. In Henshin, rules may be depicted in an integrated form annotating each model element node and reference edge by a change action. Nodes and edges that have to exist but are not changed during transformation are annotated with << preserve>> while others may be deleted or created dependent on their annotations. In addition, rule may have application conditions. In negative application conditions, nodes and edges may be forbidden meaning that they must not occur in the specified form for applying the rule. In contrast, positive application conditions may require model elements.

When performing model refactorings, a restructuring action is often accompanied by update actions on all involved model elements. For example, pulling up an attribute to a superclass implies the deletion of such an attribute from all subclasses. Such *for all* actions are specified by additional multi-rules comprising the basic rule (also called kernel rule). The overall rule (with optionally contained multi-rules) is also called *interaction scheme*; its semantics is defined by a set of rules (see below). A rule without any multi-rule is a special case of interaction scheme consisting of just one rule. In the following, an interaction scheme is represented in an integrated way, i.e., all multi-rules are represented in one diagram overlapping in the kernel rule. Note that - given an interaction scheme - the kernel rule always performs a subset of actions specified in a multi-rule of that scheme. If the kernel rule deletes a node, adjacent edges specified in including multi-rule have to be deleted as well. If this condition is fulfilled, interaction schemes follow their formal definition as presented in [9].

In the following, we present several interaction schemes for the refactoring of simple class models. We will see that they all include *for all* actions specified by multi-rules.

Example 2 (Interaction scheme "Replacing inheritance with delegation"). If we find out that an inheritance relation between two classes is not adequate as, e.g., pointed out in Example 1 for class "Stack" inheriting from class "List", the inheritance relation might be replaced by a reference. Formerly inherited attributes have to be copied in that case. This refactoring can be specified by a

kernel rule just replacing the superclass reference while the extending multi-rule copies all attributes. Figure 3 shows the corresponding specification in Henshin. All cascaded nodes and adjacent edges are in the multi-rule only while all other nodes and edges are also contained in the kernel rule.

Given an interaction scheme, i.e. a kernel rule with multi-rules, its semantics consists of an infinite set of simple rules called *amalgamated rules*. Each rule of this set consists of the kernel rule extended by 0, 1, 2 or more copies of its multi-rules. For each multi-rule, the exact number of copies depends on the number of different matches found in the instance graph the interaction scheme is applied to. It is assumed that all multi-rule matches overlap in the match of their common kernel rule. In the following, we show example amalgamated rules for the refactoring "Replace Inheritance With Delegation".

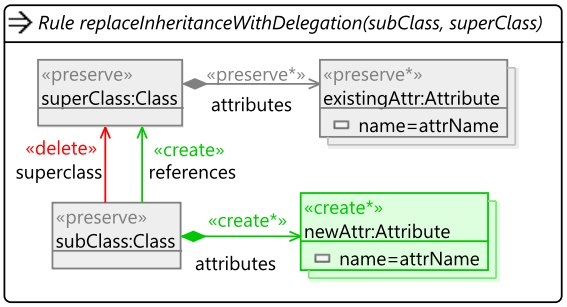

Fig. 3. Interaction scheme for refactoring "Replace Inheritance With Delegation" (RIWD)

Example 3 (Amalgamated rules and their application). Given the interaction scheme "replaceInheritanceWithDelegation" as in Figure 3, Figure 4 shows three amalgamated rules as concrete examples using 0, 1 or 2 copies of the multi-rule.

Considering the instance model in Figure 2, the inheritance between classes "List" and "Stack" shall be replaced by a delegation. Hence, we apply the refactoring "Replace Inheritance With Delegation" here. Since class "List" has one attribute, the multi-rule is applied exactly once which means that amalgamated rule riwd_1 is selected for application. The result is the model in Figure 5. The effect is that the inheritance relation between classes "List" and "Stack" is replaced by a reference and an attribute with name "first" is added to class "Stack".

After having specified one refactoring we consider three further refactoring specifications below. They are used to investigate selected refactoring conflicts and their potential resolutions below. They all use multi-rules.

Example 4 (Interaction scheme "Push down attribute"). An attribute of a given superclass may be pushed down to all its subclasses. This refactoring is needed if the modeled attribute shall be modified in its subclasses in different ways. This refactoring is the opposite of "Pull up attribute" which is not considered in detail in this paper. The diagram in Figure 6 shows the specifying interaction scheme. The kernel rule pushes down an attribute to one subclass while the multi-rule pushes down the attribute to all further subclasses.

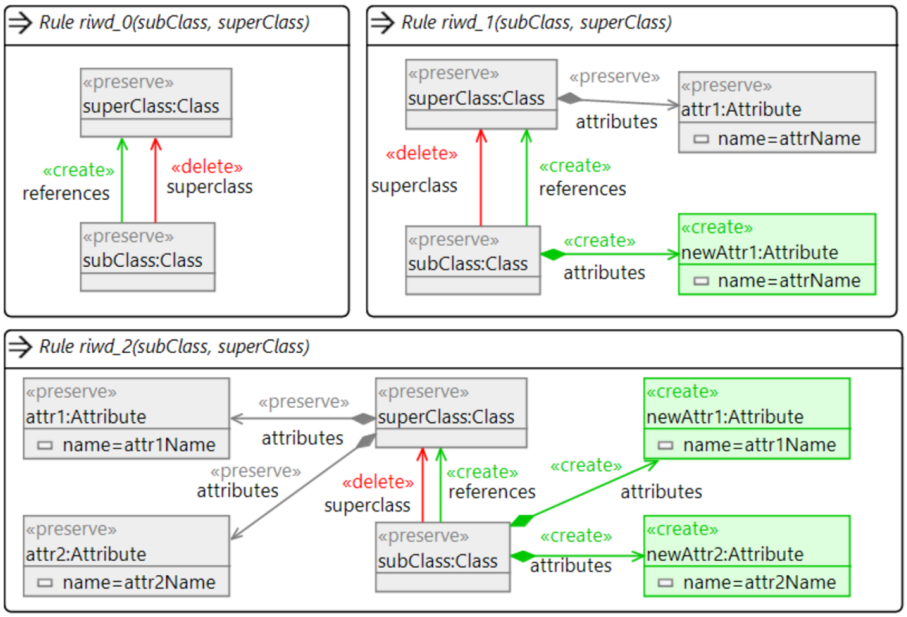

Fig. 4. Amalgamated rules which replace inheritance with delegation for classes with 0 to 2 attributes

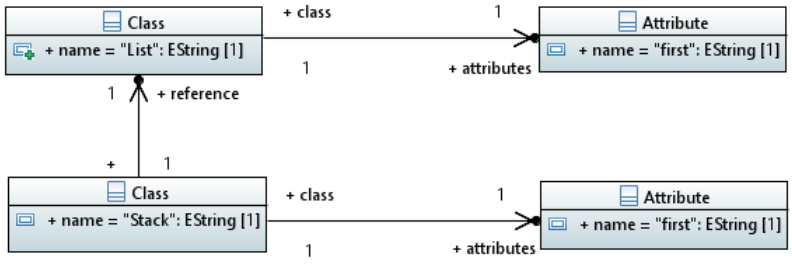

Fig. 5. Example instance model after refactoring

Example 5 (Further refactoring specifications). The interaction scheme in Figure 7 deletes all empty subclasses of a selected class indicated as superClass. Note that model nodes may only be deleted if they do not leave any edges dangling. This means for a class that it must not have attributes, references or subclasses. I.e., the interaction scheme deletes all empty subclasses of a given superclass.

Another class refactoring is the inlining of classes shown in Figure 8. If all the attributes of a class A have corresponding attributes (with the same names) in a referenced class B then class A can be inlined into class B. This means that class A and all its attributes are deleted. Again, the dangling condition checks

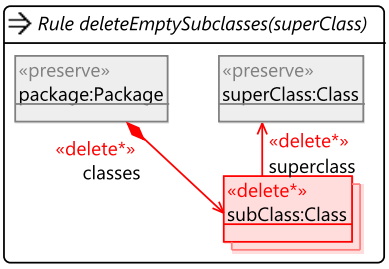

Fig. 7. Interaction scheme for refactoring "Delete Empty Subclasses" (DES)

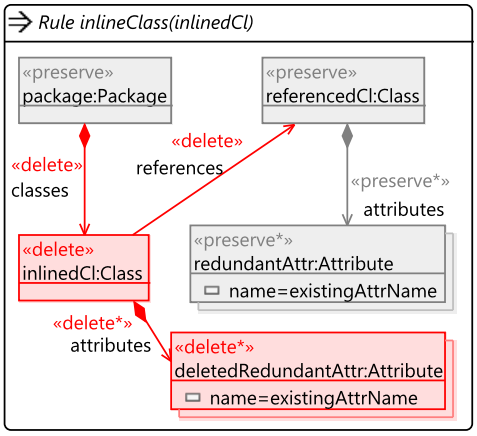

Fig. 8. Interaction scheme for refactoring "Inline Class" (IC)

if there are no further attributes (with different names), adjacent references or inheritance relations. In that case, the inlining must not take place.

3 Critical Pair Analysis

The critical pair analysis (CPA) is a well-known technique to analyze potential conflicts and dependencies of transformation systems. It has first been introduced for term rewriting and later generalized to graph transformation [18, 8]. A critical pair describes a minimal conflicting situation that may occur in the transformation system. It is well-known that if all critical pairs can be shown to be strictly confluent, the transformation system is locally confluent. This means that each pair of direct transformation steps can be resolved by arbitrary many steps to a common graph. The notion of strict confluence means

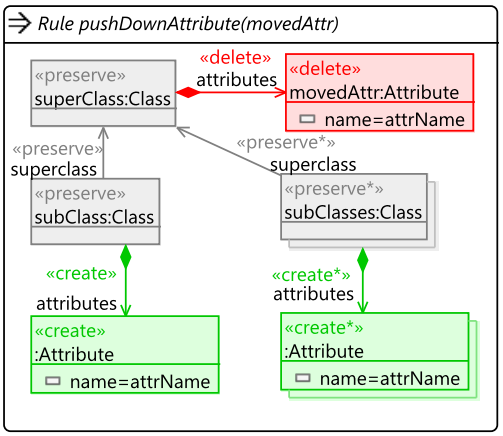

Fig. 6. Interaction scheme for refactoring "Push Down Attribute" (PDA)

that the jointly preserved part of a critical pair is also preserved by its resolution [19].

This theory has been extended to amalgamated graph transformation in [22]. Here, we have to face the problem that an interaction scheme generally describes infinite many rules and therefore, also infinite many critical pairs may exist for a given interaction scheme. In [22], we show that it is enough to check finitely many critical pairs to decide for local confluence. The key observation is that, from a certain number n of multi-rule copies, critical pairs over amalgamated rules do not lead to new kinds of conflict resolutions, i.e., all larger critical pairs are redundant to smaller ones. Up to now, however, there does not exist a construction to determine this number n. As main contribution of this paper, we present an algorithm for the CPA of amalgamated transformations below. As a prerequisite, the main definitions are recalled and illustrated at the running example here.

Two transformations are *conflicting* if (1) one transformation deletes a graph element the other uses (delete/use conflict), (2) one transformation produces a graph element the other forbids (produce/forbid conflict), or (3) one transformation changes an attribute the other uses (change/use conflict). A *critical pair*, short CP, consists of two conflicting transformations $G \overset{r1,m1}{\Longrightarrow} H1$ and $G \overset{r2,m2}{\Longrightarrow} H2$ applying rules $r1, r2$ at matches $m1, m2$ such that G is minimal. If rules $r1$ and $r2$ do not have application conditions, G is just an overlap graph of their left-hand sides. For rules with negative application conditions (NACs), also slightly larger graphs have to be considered taking parts of their NACs into account as well.

Example 6 (Critical pair). Applying refactorings "Delete Empty Subclass" and "Replace Inheritance With Delegation" in parallel may lead to conflicts. Figure 9 shows a CP over corresponding amalgamated rules, each one applying exactly one multi-rule copy. In this case, exactly one empty subclass is deleted and one attribute is copied to a referring class. This CP shows a delete/use conflict since a subclass that is deleted cannot be changed to be a delegating class. This is a potential conflict that may occur during transformations. It may be resolved by inlining the delegating class on the right yielding the model graph on the left. Note that the potential conflict shown here becomes concrete when all the variables are instantiated by concrete values.

Given two critical pairs $cp_s = (ts1 : G_s \Longrightarrow H_1s, ts2 : G_s \Longrightarrow H_2s)$ and $cp_l = (tl1 : G_l \Longrightarrow H_1l, tl2 : G_l \Longrightarrow H_2l)$ of set $CP(is1, is2)$ such that cp_l is an extension of cp_s, i.e., they distinguish just in the number of applied multi-rule copies (on at least one side). Then these CPs are considered to be *redundant* if their corresponding graphs $H1_s$ and $H1_l$ (as well as $H2_s$ and $H2_l$) allow for equivalent partial matches only, considering all rules of a given transformation system. Two partial matches m and m' to a graph H are *equivalent* if each pair of isomorphic range elements has the same history, i.e., both are newly created or both do already exist. Due to this definition, partial matches are considered equivalent if they differ only in range elements stemming from different multi-

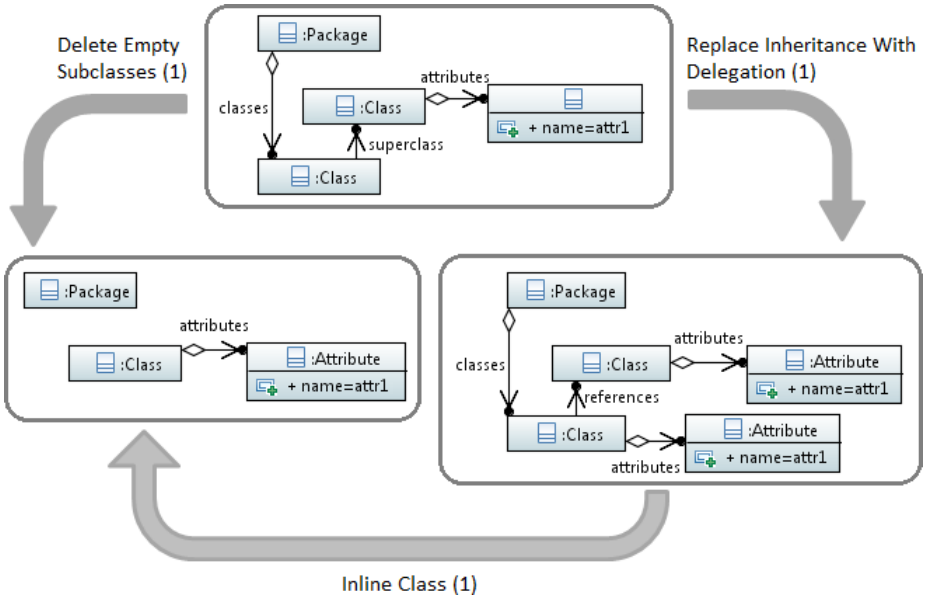

Fig. 9. Critical pair applying refactorings DES and RIWD with one multi-rule copy each

rule copies. If the dangling condition is set for a rule, the equivalence check comprises the satisfaction check of this condition as well.

Example 7 (Redundancy of critical pairs). Considering the critical pairs in Figures 9 and 10, we can notice that the CP in Figure 9 applies one multi-rule copy on each side while the one in Figure 10 applies two copies on each side. Basically, the same potential delete/use conflict is reported: A subclass that is deleted cannot be changed to a delegating class. However, the contexts are different. Although this is the case, the conflict resolution for the larger CP can be similar to the one for the smaller CP. Inlining the delegating class (with two attributes now) on the right followed by deleting the remaining empty subclass yields the model graph on the left. Any other interaction scheme is not applicable on the left or right. Comparing all the partial matches that exist in both cases and check whether they are equivalent w.r.t. the above definition, we find out that this is the case for all interaction schemes except of DES. Since the right graph in Figure 10 still contains a generalization relation, some new partial matches can be found here. Hence, the CP in Figure 10 is not redundant to the one in Figure 9 although a very similar conflict is reported.

4 Algorithm for the Critical Pair Analysis

In the following, we present the core algorithm for computing all relevant critical pairs between two interaction schemes. As shown in [22], a finite set of critical

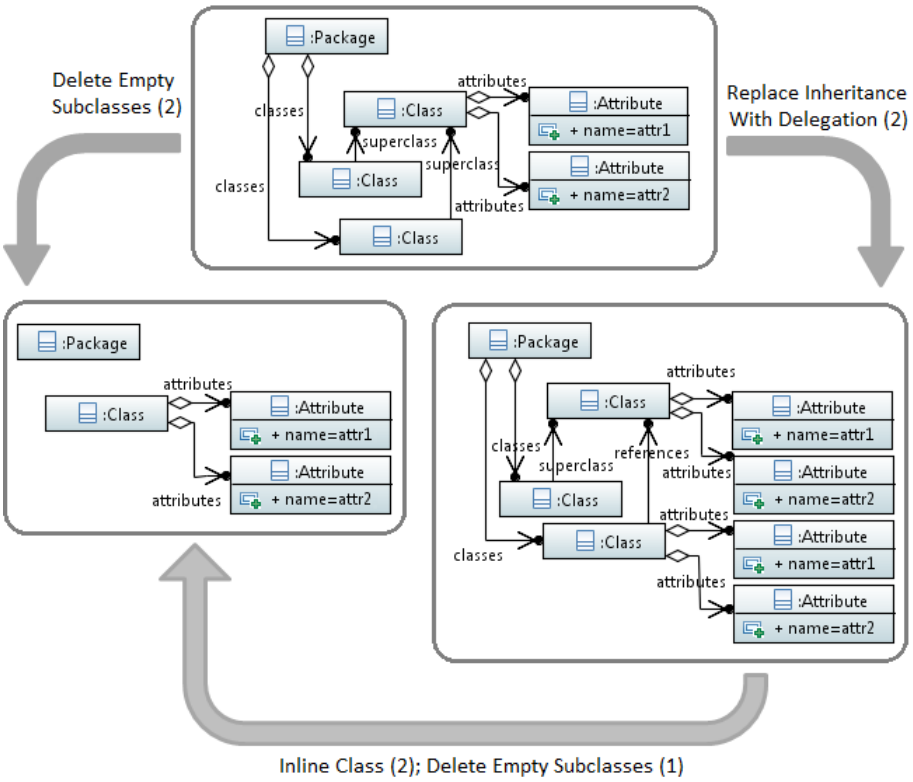

Fig. 10. Critical pair applying refactorings DES and RIWD with two multi-rule copies each

pairs is enough to decide for local confluence of a given transformation system. The computation is performed for increasingly larger amalgamated rules. The maximal number of multi-rules applied define the level of computation. The stop criterion is met if all critical pairs that are computed for the current level turn out to be redundant to critical pairs of lower levels. The main algorithm is presented in Figure 11.

Class *RulePairHandler* is a container for rule pairs and their critical pairs and the associated partial matches of the transformation system. Function *getRulePairsOfLevel* computes all pairs of amalgamated rules where each rule has at most as many multi-rule copies as *level* prescribes. Given a concrete rule pair, function *computeCps* computes all critical pairs of this rule pair. Each critical pair is reported by a minimal model with two matches of participating rules. After having computed all critical pairs of a given level, function *extractNonRedundantCps* filters out all those critical pairs that are not redundant to already existing ones computed in lower levels. The identification of redundant critical

```
Input: is1,is2: Input interaction schemes
Output: resultCps: Output set containing all non-redundant critical pairs
 1: function computeCriticalPairs(InteractionScheme is1, InteractionScheme
    is2): CpaResult
 2:     rulePairHandler = new RulePairHandler(is1,is2);
 3:     resultCps = new CpaResult();
 4:     levelCps = analyseLevelForNonRedundantCps(0, rulePairHandler);
 5:     resultCps.add(levelCps);
 6:     return resultCps;
 7: function analyseLevelForNonRedundantCps(int level, RulePairHandler
    rph): CpaResult
 8:     resultCps = new CpaResult();
 9:     currentRulePairs = rph.getRulePairsOfLevel(level);
10:     for rulePair : currentRulePairs do
11:         currentCps = computeCps(rulePair);
12:         newCps = extractNonRedundantCps(currentCps,level);
13:         rulePair.setCps(newCps);
14:         resultCps.add(newCps);
15:     if resultCps.size() != 0 || level == 0 then
16:         resultOfNextLevel = analyseLevelForNonRedundantCps(level+1);
17:         resultCps.add(resultOfNextLevel);
18:     return resultCps;
```

Fig. 11. Pseudocode for computing critical pairs of interaction schemes.

pairs is achieved by comparing the partial matches of the whole transformation system for each new critical pair against the already known.

Correctness. Given two interaction schemes $is1$ and $is2$, we have to show that computeCriticalPairs yields the set of all non-redundant CPs between these two rules schemes. The main design decision is here that CPs are computed level-wise starting with level 0. All CPs of level n are computed if $n \leq 1$ or new non-redundant CPs have been computed for level $n-1$. Level one has to be considered anytime due to the fact that level zero doesn't involve the amalgamations at all. Given level n, function analyseLevelForNonRedundantCps computes all non-redundant CPs of rule pairs of that level. All rule pairs for that level where each rule has at most n multi-rule copies, are collected in *currentRulePairs*. All their CPs are collected in *currentCps*. Function *extractNonRedundantCps* directly implements the check for non-redundant CPs based on the definition given in [22] which is informally recalled above. If any rule pair of a level yields new non-redundant CPs, the set of resulting CPs becomes non-empty and the next level has to be considered. Finally, the non-redundant CPs of all levels are joined to the set *resultCps*.

In the algorithm in 12 for each potential new critical pair (cP) all partial matches (*pMoOfR1, pMoOfR2*) of the transformation system on the two involved transformations (*tl1, tl2*) are analysed. If all the partial matches (*pMoOfR1, pMoOfR2*) are already known by the critical pairs which are extended by the evaluated critical pair (cP), then it's a redundant critical pair and won't be re-

```
1: function extractNonRedundantCps(CpaResult cPs): CpaResult
2:     nonRedundantCps = new CpaResult();
3:     for cP : cPs do
4:         tl1 = cP.getResultOfR1();
5:         pMOfR1 = findAllNonEquivPartialMatches(tl1);
6:         tl2 = cP.getResultOfR2();
7:         pMOfR2 = findAllNonEquivPartialMatches(tl2);
8:         rulePairs = rph.getExtendedRulePairs(cP.getRulePair());
9:         for rulePair : rulePairs do
10:            reducedCPs = rulePair.getCriticalPairs();
11:            for reducedCP : reducedCPs do
12:                if isExtension(cP.minimalModel(), reducedCP.minimalModel()) &&
   (pMOfR1.size()>0 || pMOfR2.size()>0) then
13:                    ts1 = reducedCP.getTransformation1();
14:                    alreadyKnownPMOfR1 = extractEquivParMatches(ts1, tl1);
15:                    pMOfR1.removeAll(alreadyKnownPMOfR1);
16:                    ts2 = reducedCP.getTransformation2();
17:                    alreadyKnownPMOfR2 = extractEquivParMatches(ts2, tl2);
18:                    pMOfR2.removeAll(alreadyKnownPMOfR2);
19:        if pMOfR1.size() != 0 || pMOfR2.size() != 0 then
20:            nonRedundantCps.addResult(criticalPair);
21:    return nonRedundantCps;
```

Fig. 12. Pseudocode of function *extractNonRedundantCps*.

turned. To do so for each extended critical pair (*reducedCp*) the already known critical pair get extracted by *extractEquivParMatches*. The set of partial matches (*pMoOfR1, pMoOfR2*) of the investigated critical pair (*cP*) gets reduced by them. This repeats until none of them are left and the critical pair (*cP*) is identified as redundant. Otherwise this repeats until all extended critical pairs (*reducedCp*) are investigated and the critical pair (*cP*) is identified as non-redundant and part of the returned set of the function.

The central question to be answered here is: Does this algorithm terminate? The answer should be yes due to the main result in [22]. The proof of this result contains the following key idea: For each pair of interaction schemes, there are two finite numbers c and d such that all rule pairs of amalgamated rules with at most c and d multi-rule copies yield redundant CPs only. If we take $max(c, d)$ as current level, there would not be any further non-redundant conflict found. The key idea for termination is that new CPs do not provide new partial matches for the rules of our transformation system. An extreme over-approximation can go like this: Given a transformation system with interaction schemes, let $|L| = x$ be the number of graph elements of the left-hand side of the largest (multi-)rule. There are at most $|\mathcal{P}(L)| = 2^x$ different subsets of elements, i.e., domains for partial matches, over L. The different ranges are not interesting in detail. We only check if range elements are preserved or newly created. If two partial matches with the same domain are equal w.r.t. this range classification, they are considered equivalent. Hence, the largest number of non-equivalent partial matches is $2^{2x} = 4^x$. Although this number is extremely high, it tells us that

there is an upper limit for non-equivalent partial matches independent of the result graphs $H1$ and $H2$ occurring in concrete CPs. Usually, the number of non-equivalent partial matches is much smaller since element types, attributes and graph structures have to be taken into account as well. Moreover, partial matches cannot exist if sub-matches do not exist as well. The following examples produce numbers of partial matches being smaller than 200 for any CP result graph although the extreme over-approximation yields 4^9. The value nine is based on the interaction scheme "Push down attribute", which has the most model elements in its left-hand side compared to the other ones in the transformation system.

Example 8 (Algorithm run). To illustrate the algorithm, we consider an example run now: Given the interaction schemes for refactorings DES and RIWD in Figures 7 and 3, all non-redundant CPs with a DES rule as first and a RIWD rule as second shall be computed. As pointed out above, the CPs of levels 0 and 1 always have to be computed. For all pairs of kernel and multi-rules of participating interaction schemes, the number of CPs found are shown in Table 1. Moreover, it shows the numbers of CPs for all pairs of amalgamated rules of levels 2 and 3. This is needed since the check by *extractNonRedundantCps* finds out that RIWD and IC have new non-equivalent partial matches to CP graphs of rule pairs of level 1, as shown in Table 2. Therefore, there are non-redundant CPs on level 1 and hence, level 2 has to be considered. Note that the table entries always show the number of non-redundant CPs as well as the number of all CPs. One example CP of level 2 is shown in Figure 10. In contrast to CPs of level 1, an inheritance relation may remain after applying refactoring RIWD which leads to new non-eqivalent partial matches of refactorings DES and PDA. Therefore, new non-redundant CPs occur on level 2 and the algorithm does not yet terminate. Hence, amalgamated rules of level 3 have to be checked for non-redundant CPs as well. As explained at Table 2, it turns out that new non-equivalent partial matches do not occur and therefore, all newly found CPs are redundant leading to the termination of the algorithm for the considered interaction schemes.

CPs 1./2.	RIWD (0)	RIWD (1)	RIWD (2)	RIWD (3)
DES(0)	0/0	0/0	0/0	0/0
DES(1)	1/1	1/1	1/1	0/1
DES(2)	2/2	2/2	0/2	0/2
DES(3)	0/3	0/3	0/3	0/3

Table 1. Numbers of non-redundant/all critical pairs between refactorings DES (first rule) and RIWD (second rule)

# Part. matches	RIWD (0)	RIWD (1)	RIWD (2)	RIWD (3)
DES(0)	-	-	-	-
DES(1)	15/28	20/84	10/0	0/0
DES(2)	0/4	0/2	0/0	0/0
DES(3)	0/0	0/0	0/0	0/0

Table 2. Numbers of non-equivalent partial matches to the left and right result graphs of CPs between refactorings DES and RIWD

Given all the CPs between refactorings DES and RIWD, Table 2 shows the numbers of non-equivalent matches to the left and right result graphs of each CP. (Remember that a CP consists of two conflicting transformations both starting from the same graph and resulting in two graphs. The left one is the result after applying refactoring DES while the right one is obtained by applying refactoring RIWD.) We see that on level 3, there are no further non-equivalent matches discovered. Hence, the computation of non-redundant CPs terminates after level 3 as stated above.

In the following, we summarize the number of non-redundant CPs found in our transformation system and show for each pair of interaction schemes how many levels of amalgamation have to be considered for the CPA until the termination criterion is fulfilled. We consider two variants of our transformation system: The first one contains all presented interaction schemes without PDA while the second one includes PDA. Tables 3 and 4 show the results. In both tables, we see that the number of levels needed is moderate. Often, the consideration of levels 0 and 1 is already enough. Tables 3 and 4 do not only differ in the numbers of rows and columns but also w.r.t. their entries. As an example, the CPA for DES and RIWD needs 3 or 4 levels, resp. The tables also show the numbers of non-redundant CPs found for pairs of interaction schemes which is 7 for pairs (DES,RIWD) as well as (RIWD,DES) on level 3 and 13 on level 4. The differences arise due to the fact that PDA causes new kinds of partial matches. These additional conflicts have to be taken into account for future confluence check. A more detailed view of the results can be found at [12].

level of amalg./ # non-red. CPs	DC	RIWD	DES	IC
DC	1/0	1/0	1/2	1/0
RIWD	1/0	2/1	3/7	1/0
DES	1/0	3/7	1/0	1/0
IC	1/0	1/0	2/4	1/0

Table 3. Level of termination and number of non-redundant critical pairs with four interaction schemes

level of amalg. / # non-red. CPs	DC	RIWD	DES	IC	PDA
DC	1/0	1/0	1/2	1/0	1/0
RIWD	1/0	2/1	4/13	1/0	2/10
DES	1/0	4/13	1/0	1/0	2/3
IC	1/0	1/0	2/4	1/0	1/0
PDA	1/0	3/4	3/8	3/4	2/6

Table 4. Level of termination and number of non-redundant critical pairs with five interaction schemes

The presented algorithm for the CPA of interaction schemes has been prototypically implemented for rules specified in Henshin. It relies on the CPA implementation for basic rules as presented in [6]. The current CPA implementation for interaction schemes supports the conflict detection only (i.e. does not support the detection of dependencies yet). Furthermore, rules with application conditions are not supported yet. These limitations are easy to erase which will be done in the near future.

5 Related Work and Conclusion

Multi-objects and other variants that match graph parts as often as possible have been considered in several graph transformation approaches: in tool environments such as PROGRES [20] and Fujaba [1] as well as in conceptual approaches by Grönmo [10] and Drewes et.al. [7]. These tools and approaches, however, do not support the critical pair analysis (CPA) for graph transformation systems expressing such variability.

While a basic graph transformation approach is taken in [22] to develop the necessary theory for the CPA for amalgamated graph transformation, we switch to model transformation based on the Eclipse Modeling Framework (EMF) and Henshin here. EMF models have typed, attributed graphs as conceptual basis while Henshin is based on graph transformation concepts. Hence, we have developed the CPA for the amalgamated transformation of typed, attributed graphs here. However, we do not yet consider application conditions for rules.

The main contribution of this paper is an algorithm for computing all non-redundant critical pairs of two given interaction schemes. It shows that the CPA for pairs of simple rules can be reused. The key idea is to compute all critical pairs for pairs of small amalgamated rules. This computation stops at level n when all pairs of rules with at most n copies of multi-rules yield redundant critical pairs only. We have implemented this algorithm in Henshin. First tests with a set of refactoring interaction schemes have shown that the CPA for interaction schemes is performed in a reasonable amount of time. An extensive evaluation is planned for future work.

References

1. The Fujaba tool suite. `www.fujaba.de`
2. 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)
3. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the eclipse modeling framework. Autom. Softw. Eng. 20(2), 141–184 (2013)
4. Biermann, E., Ehrig, H., Ermel, C., Golas, U., Taentzer, G.: Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation. In: Graph Transformations and Model-Driven Engineering. LNCS, vol. 5765, pp. 121–140. Springer (2010)
5. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software and System Modeling 11(2), 227–250 (2012), `http://dx.doi.org/10.1007/s10270-011-0199-7`
6. Born, K., Arendt, T., Heß, F., Taentzer, G.: Analyzing conflicts and dependencies of rule-based transformations in Henshin. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. vol. LNCS 9033, pp. 165–168. Springer (2015)

7. Drewes, F., Hoffmann, B., Janssens, D., Minas, M.: Adaptive star grammars and their languages. Theor. Comput. Sci. 411(34-36), 3090–3109 (2010)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
9. Golas, U., Habel, A., Ehrig, H.: Multi-amalgamation of rules with application conditions in $\mathcal{M}$-adhesive categories. Mathematical Structures in Computer Science 24(4) (2014)
10. Grönmo, R., Krogdahl, S., Möller-Pedersen, B.: A collection operator for graph transformation. In: Proc. of ICMT 2009. LNCS, vol. 5563, pp. 67–82. Springer (2009)
11. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA. pp. 105–115. ACM (2002)
12. More details on the results. `http://www.uni-marburg.de/fb12/swt/cpa_amal`
13. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. pp. 191–201. IEEE (2013)
14. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the small with the epsilon wizard language. Journal of Object Technology 6(9), 53–69 (2007), `http://dx.doi.org/10.5381/jot.2007.6.9.a3`
15. Mantz, F., Taentzer, G., Lamo, Y., Wolter, U.: Co-evolving meta-models and their instance models: A formal approach based on graph transformation. Sci. Comput. Program. 104, 2–43 (2015)
16. Mehner-Heindl, K., Monga, M., Taentzer, G.: Analysis of aspect-oriented models using graph transformation systems. In: Moreira, A., Chitchyan, R., Araújo, J., Rashid, A. (eds.) Aspect-Oriented Requirements Engineering, pp. 243–270. Springer (2013)
17. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling 6(3), 269–285 (2007)
18. Plump, D.: Critical Pairs in Term Graph Rewriting. In: Mathematical Foundations of Computer Science. LNCS, vol. 841, pp. 556–566. Springer (1994)
19. Plump, D.: On termination of graph rewriting. In: Graph-Theoretic Concepts in Computer Science, 21st Int. Workshop, WG '95. LNCS, vol. 1017, pp. 88–100. Springer (1995)
20. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pp. 487–550. World Scientific (1999)
21. Steinberg, D., Budinsky, F., Patenostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition. Pearson Eduction (2009)
22. Taentzer, G., Golas, U.: Towards Local Confluence Analysis for Amalgamated Graph Transformation. In: Parisi-Presicce, F., Westfechtel, B. (eds.) Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings. LNCS, vol. 9151, pp. 69–86 (2015), Long Version as Technical Report at Zuse Institute Berlin, no. 15-29, at https://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/5494