Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin *

Kristopher Born, Thorsten Arendt, Florian Heß, Gabriele Taentzer

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

Abstract. Rule-based model transformation approaches show two kinds of non-determinism: (1) Several rules may be applicable to the same model and (2) a rule may be applicable at several different matches. If two rule applications to the same model exist, they may be in conflict, i.e., one application may disable the other one. Furthermore, rule applications may enable others leading to dependencies. The critical pair analysis (CPA) can report all potential conflicts and dependencies of rule applications that may occur during model transformation processes. This paper presents the CPA integrated in Henshin, a model transformation environment based on the Eclipse Modeling Framework (EMF).

1 Introduction

Rule-based model transformation systems can control the application of rules not only by explicit control mechanisms but also by causal dependencies of rule applications. Hence, these causal dependencies influence their execution order. If, e.g., a rule creates a model element, it can be used in subsequent rule applications. It can also happen that two rule applications overlap in a model element and one rule is to delete it while the other one requires its existence. For a better understanding of this implicit control flow, it is interesting to analyze all potential causal dependencies of rule applications for a given rule set.

The *critical pair analysis* (CPA) for graph rewriting [6] can be adapted to rule-based model transformation, e.g., to find conflicting functional requirements for software systems [7], or to analyze potential causal dependencies between model refactorings [9] which helps to make informed decisions on the most suitable refactoring to apply next. The CPA reports two different forms of potential causal dependencies, called conflicts and dependencies.
The application of a rule r_1 is in *conflict* with the application of a rule r_2 if

- r_1 deletes a model element used by the application of r_2 **(delete/use)**, or
- r_1 produces a model element that r_2 forbids **(produce/forbid)**, or
- r_1 changes an attribute value used by r_2 **(change/use)**. [1]

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

[1] Dependencies between rule applications can be characterized analogously.

In our work, we extended Henshin [2], a rule-based model transformation language adapting graph transformation concepts and being based on the Eclipse Modeling Framework (EMF) [5]. Our extension computes all potential conflicts and dependencies of a set of rules and reports them in form of critical pairs. Each critical pair consists of the respective pair of rules, the kind of potential conflict or dependency found, and a minimal instance model illustrating the conflict or dependency. The analysis can be fine-tuned by a number of additional options to be set. The adoption of graph transformation theory to EMF model transformation requires to check the transformation rules for preserving model consistency and the resulting minimal model for being a valid EMF model [4].

The next section introduces a running example and discusses expected results; afterwards the new analysis tool is presented.

2 Model transformation with Henshin

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.

Henshin is an EMF model transformation engine based on graph transformation concepts. Since refactoring is a specific kind of model transformation, refactorings of EMF-based models can be specified in Henshin and then integrated into a refactoring framework such as EMF Refactor [3]. To demonstrate the main idea, we limit ourselves to one rule of a refactoring example for class modeling [8]. Rule `Move_Attribute` (Figure 1(a)) specifies the shift of an attribute from its owning class to an associated one along a reference. It is shown in abstract syntax. Objects and references tagged by ⟨⟨preserve⟩⟩ represent unchanged model elements, elements tagged by ⟨⟨create⟩⟩ represent new ones whereas those tagged by ⟨⟨delete⟩⟩ are removed by the transformation.

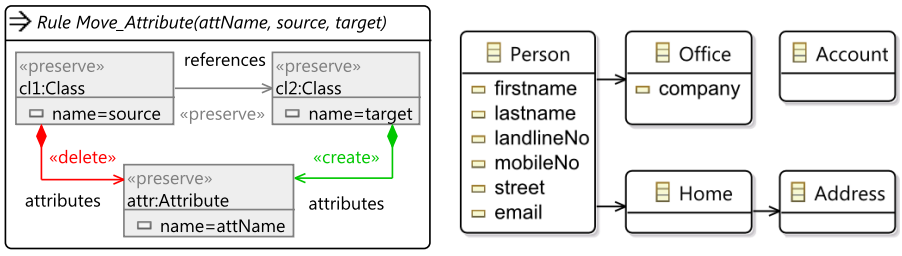

Fig. 1. Henshin refactoring rule (a) and class model `Address Book`(b)

Modifying the class model in Figure 1(b) by the refactoring specified in Figure 1(a), we observe two potential problems: (1) The attribute `landlineNo` of class `Person` can be shifted to either class `Home` or class `Office` (by refactoring `Move_Attribute`). However, if it is shifted to class `Home` the other refactoring becomes inapplicable (and vice versa). This means, refactoring `Move_Attribute`

is in conflict with itself. (2) The attribute `street` of class `Person` can be shifted to class `Address` via class `Home` (by two applications of `Move_Attribute` along existing references). The second shift is currently not possible since class `Home` does not have an attribute so far, i.e., refactoring `Move_Attribute` may depend on itself. Graph transformation theory allows us to analyze such conflicts and dependencies at specification time by relying on the idea of the CPA.

3 Tooling

The provided CPA extension of Henshin can be used in two different ways: Its application programming interface (API) can be used to integrate the CPA into other tools and a user interface (UI) is provided supporting domain experts in developing rules by using the CPA interactively.

After invoking the analysis, the rule set and the kind of critical pairs to be analyzed have to be specified. Furthermore, options can be customized to stop the calculation after finding a first critical pair, to ignore critical pairs of the same rules, etc. The resulting list of critical pairs is shown and ordered along rule pairs. Figure 2 depicts an example for the analysis of rule `Move_Attribute`, in which the delete/use-conflict (1) corresponds to the example discussed above.

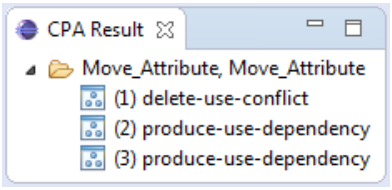

Fig. 2. The result view

The subsequent dependency results differ in their target of the second attribute movement. The first produce/use-dependency (2) represents the case of moving the attribute back to the original class, which leads to a smaller minimal model with only two classes referencing each other, as depicted in Figure 3. The highlighting by enclosing hash marks is the most important information, since the enclosing element is the cause of the dependency. The link between `2:Class` and `3:Attribute` is created by the first rule application and is required by the second application. Since all elements and values in the minimal model may be matched by the first and the second rule application, there is a generic approach to represent attribute values. Value `r1_source_r2_target`, e.g., means that it must conform to value `source` in rule `r1` and value `target` in rule `r2`, respectively (compare Fig. 1(a)).

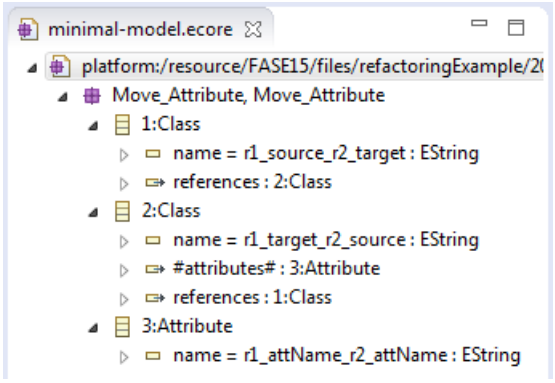

Fig. 3. Minimal model of a dependency

The second dependency reported in Figure 2 is the handling of two consecutive attribute shifts, also described in Section 2.

The current version of the tool can analyze rules with negative application conditions and attributes of primitive data types. Positive application conditions shall be supported in the future. In order to avoid improper results, the rules are checked regarding these prerequisites. Further checks ensure that the rules are consistent to the properties defined in. The LHS, RHS and intersection graphs of each rule are checked to comply to Definition 3 in [4], e.g., each node must have at most one container, there is no containment cycle. Furthermore, rules have to ensure consistent results, i.e., have to comply to Def. 6 in [4], ensuring e.g. that containment edge deletion and creation is restricted to edge redirection. The rule shown in Figure 1(a) is consistent to this definition. Internally, the CPA extension of Henshin is based on the graph transformation tool AGG [1]. Dedicated exporter and importer translate EMF meta-models and Henshin rules to AGG and CPA results back to EMF models.

4 Conclusion

The model transformation tool Henshin has been extended by a critical pair analysis to inspect rule sets for dependencies and conflicts. An interactive user interface is provided allowing the inspection of each critical pair in detail.

For the future, we intend to support also a confluence check of critical pairs, for which the CPA is a first step. The tool download as well as additional information on the CPA in Henshin, especially with respect to the translation between Henshin and AGG, can be found at [8].

References

1. AGG: `http://user.cs.tu-berlin.de/~gragra/agg/`
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: MoDELS. vol. 6394, pp. 125–135 (2010), `http://www.eclipse.org/henshin/`
3. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering 20(2), 141–184 (2013), `http://www.eclipse.org/emf-refactor`
4. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. SoSyM 11(2), 227–250 (2012)
5. Eclipse: Eclipse Modeling Framework (EMF). `http://www.eclipse.org/emf`
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
7. 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: ICSE. pp. 105–115. ACM (2002)
8. Tool download and installation. `http://www.uni-marburg.de/fb12/swt/cpa`
9. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling 6(3), 269–285 (2007)

A Demonstration

Our demonstration will accompany domain experts along their tasks and experiences. They define refactoring rules for a domain, analyze their interdependencies and consider to integrate this analysis into another tool.

A.1 The complete example

Given a meta-model defining a modeling language, the domain expert develops refactoring rules. Starting with the rule in Figure 1(a), two further rules `Add_Attribute` and `Remove_Empty_Class` are introduced in Figure 4.

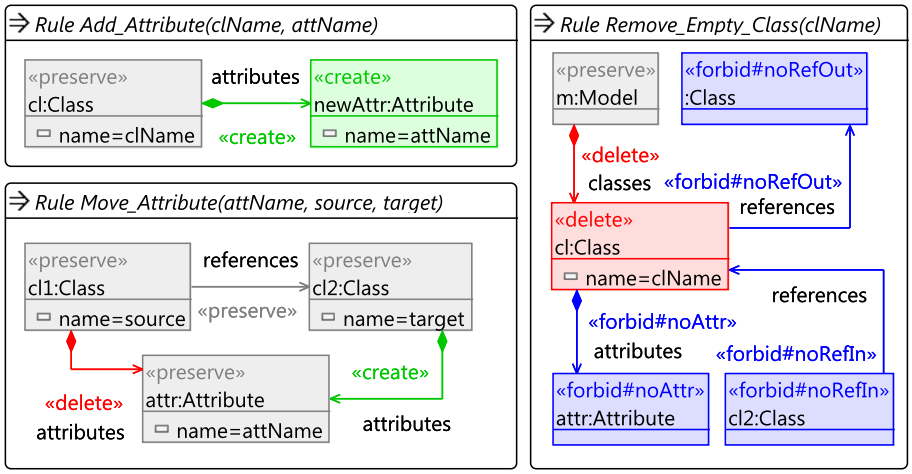

Fig. 4. Example set of rules

Rule `Add_Attribute` inserts a new attribute into an existing class with name *clName*. The third rule `Remove_Empty_Class` deletes an existing class if it has no relation to further model elements, i.e., the class has neither attributes nor incoming nor outgoing references. Elements tagged by ⟨⟨forbid⟩⟩ specify negative application conditions (NACs) that prevent the rule from being executed when found. All these rules will be shortly introduced in the demonstration.

A.2 Analyzing critical pairs

After setting up the rules, domain experts may want to know all potential interrelations between applications of these rules. After realizing, that they could miss some of them, they decide to use our tool. The analysis is started on a Henshin file which contains a set of rules (see Figure 5).

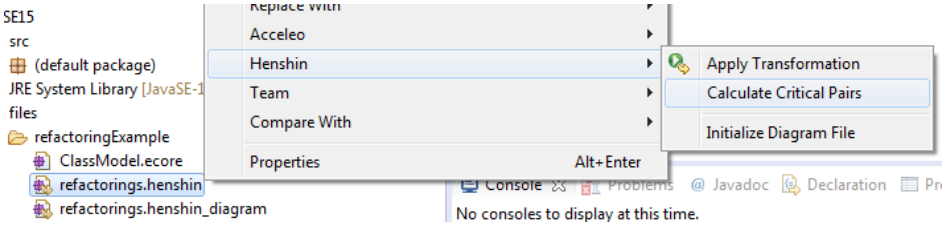

Fig. 5. Access to CPA tool within the user interface in Eclipse

The wizard depicted in Figure 6 requires the selection of at least one rule. Furthermore, the kind of critical pair analysis, conflicts or dependencies, has to be set.

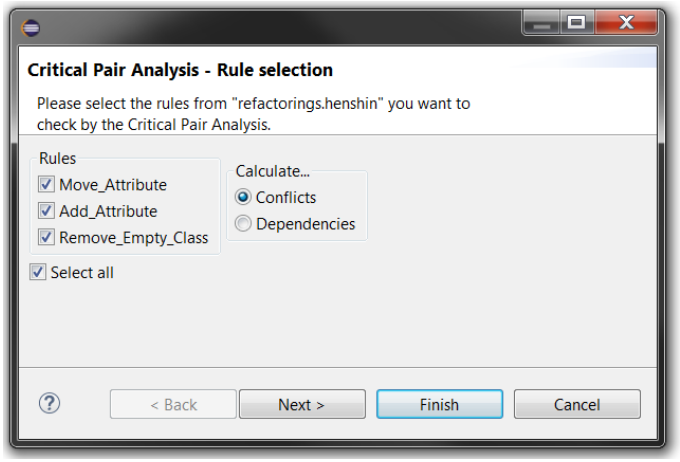

Fig. 6. Wizard for selection of rules and analysis kind

The second page of the wizard, depicted in Figure 7, provides the opportunity to adjust options for CPA calculation.

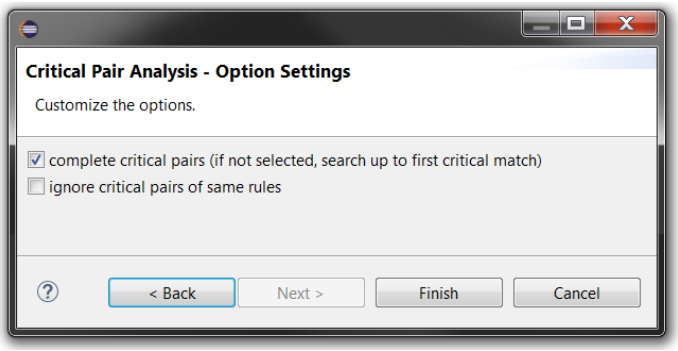

Fig. 7. Second wizard page for adjusting CPA options

Finishing the wizard, the calculation of all critical pairs starts leading to the representation of the found critical pairs within the CPA Result view in Figure 8. Comparing the results with those in Figure 2 it is obvious that the set of rules of the enlarged example contains more critical pairs than the analysis of rule `Move_Attribute` with itself. Nevertheless, all the critical pair results of rule `Move_Attribute` with itself do occur as well.

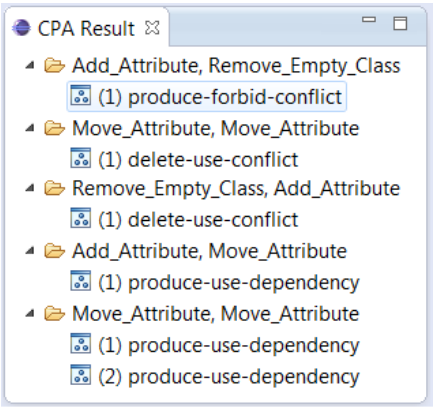

Fig. 8. CPA Result view for enlarged example

Note that similarly to different kinds of conflicts, as presented in Section 1, we distinguish different kinds of dependencies. The application of a rule r_2 depends on applying a rule r_1 if:

- **produce/use:** r_1 produces a model element that is needed by r_2.
- **delete/forbid:** r_1 deletes a model element that a NAC of r_2 forbids.
- **change/use:** r_1 changes the value of an attribute that is used by the match of r_2 or checked by one of its NACs.

A.3 Inspecting the results

Detailed information to each critical pair is provided by a minimal model that shows the occurrence of a conflict or a dependency. A conflict is shown at a minimal model to which rules are applied, while a minimal model for a dependency shows the intermediate result of rule application. Besides the CPA Result view, critical pairs can also be persisted within the project to inspect them in the future. Figure 9 depicts a minimal model for a produce-use-dependency at its top

and in addition a graphical representation of the minimal model in the center. For better understanding, both involved rules, each one being `Move_Attribute`, are depicted. Their matches are shown by framing parts of the minimal model, pointing to the applied rule.

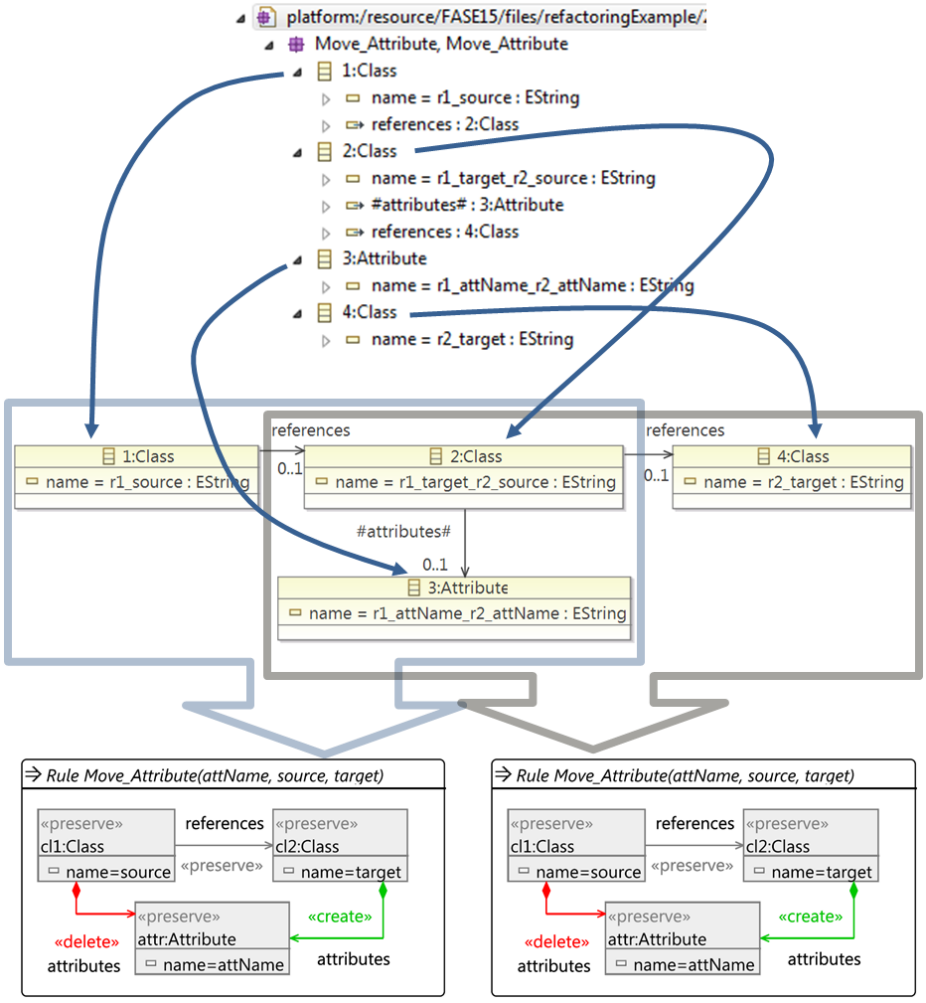

Fig. 9. Detailed representation of a critical pair showing a dependency

The minimal model depends on the kind of critical pair. According to its name, a minimal model is as small as possible.

For conflicts, this means that both rules must be applied to it, which implies that the LHS of both rules can be matched to the minimal model. Minimal models of dependencies differ in regard to the first rule. They show minimal

intermediate results. While the minimal model of a dependency still provides the capability to apply the second rule, it represents a result of applying the first rule. Accordingly, the minimal model must contain an image of the RHS of the first rule. Furthermore, the LHS of the second rule has to occur. In Figure 9 this can be observed since the minimal model is an overlapping of the LHS and the RHS of rule `Move_Attribute`. Application conditions like PAC and NAC, which are parts of the LHS, are relevant as well. To understand this consider, for example, the produce-forbid-conflict between rules `Add_Attribute` and `Remove_Empty_Class`. The `Attribute` (see Figure 4) is an image of the RHS of the first rule and of the NAC of the second rule.

To exemplify an interrelationship between rule applications, the domain experts may use a class model like the one in Figure 1(b). In addition to the critical pairs described in Section 2 there are further ones wrt. the additional rules in Figure 4:

In our example class model in Figure 1(b), class `Account` is not used so far and makes no sense. Two refactorings are applicable here. Either this class is extended by a new attribute, e.g. *accountnumber*, or it is removed from the model (refactorings `Add_Attribute` respectively `Remove_Empty_Class`). However, if one refactoring is executed, the other one becomes inapplicable, i.e., these refactorings are in conflict. According to the sequence we have a produce/forbid conflict for `Add_Attribute` followed by `Remove_Empty_Class` and a delete/use conflict for the opposite order.

A missing information within class model `Address Book` is the specification of the city a person lives in. It can be inserted by first applying refactoring `Add_Attribute` to class `Home` followed by refactoring `Move_Attribute` (to class `Address`) which can not be applied before since class `Home` does not have an attribute in the beginning. This means, refactoring `Move_Attribute` may depend on refactoring `Add_Attribute` by a produce/use dependency.

A.4 Using the Critical Pair Analysis via an API

Domain experts may also want to integrate the CPA into other tools, for example, to provide recommendations for refactorings interactively. The integration by its API is illustrated by the code-snippet shown in Listing 1. Lines 1-4 are concerned with loading our example; in lines 6-10 the rules are extracted from transformation units, since the CPA interface strictly limits to rules. In lines 12-15 the CPA is prepared and started. The analysis is done independently for conflicts and dependencies. The resulting `CPAResult` object complies to the data structure in Figure 11, which provides the necessary information for further processing by public access modifiers.

Listing 1. Code snippet for using the CPA by its API

```
1  String PATH = "files/";
2  String henshinFileName = "refactorings.henshin";
3  HenshinResourceSet rS = new HenshinResourceSet(PATH);
4  Module module = rS.getModule(henshinFileName, false);
5
6  List<Rule> rules = new LinkedList<Rule>();
7  for(Unit unit : module.getUnits()){
8          if(unit instanceof Rule)
9                  rules.add((Rule) unit);
10 }
11
12 ICriticalPairAnalysis cpa = new AGGCPA();
13 cpa.init(rules, new CPAOptions());
14 CPAResult dependencies = cpa.runDependencyAnalysis();
15 CPAResult conflicts = cpa.runConflictAnalysis();
```

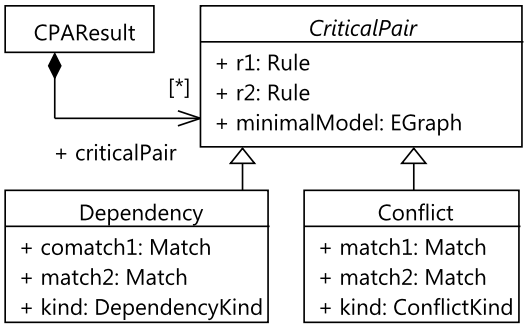

Fig. 10. Interface for CPA results

Figure 11 shows a class model where `CPAResult` contains all `Conflicts` and `Dependencies`, which specialize the general concept `CriticalPair`. It always consists of two rules and a minimal model showing the conflict or dependency when applying these rules. Two mappings are required to map rule elements to corresponding ones in the minimal model. Although `Dependency` and `Conflict` share this requirement, it is not part of a critical pair in general, since different mappings are reported, as described in [6]. A conflict is based on the left-hand sides (LHSs) of both rules, such that both mappings are *matches*. A dependency is based on the right-hand side (RHS) of the first rule *r1* and the LHS of the second rule *r2*. Therefore, the first mapping of a dependency is a *comatch* of RHS to the minimal model. Finally, there are different kinds of conflicts and dependencies, as introduced in Section 2.

As we have seen the usage of the API is very handy, so that it is one of our next aims to integrate the CPA into other tools and to evaluate it for, e.g., model versioning, refactoring and model transformation by triple graph grammars.