Logical Methods in Computer Science
Volume 22, Issue 1, 2026, pp. 10:1–10:51
https://lmcs.episciences.org/

Submitted Jan. 22, 2025
Published Feb. 27, 2026

USING WEAKEST APPLICATION CONDITIONS TO RANK GRAPH TRANSFORMATIONS FOR GRAPH REPAIR

LARS FRITSCHE [c], ALEXANDER LAUER [b], MAXIMILIAN KRATZ [a], ANDY SCHÜRR [a], AND GABRIELE TAENTZER [b]

[a] Technical University Darmstadt, Darmstadt, Germany
e-mail address: {maximilian.kratz, andy.schuerr}@es.tu-darmstadt.de

[b] Philipps-Universität Marburg, Marburg, Germany
e-mail address: alexander.lauer@uni-marburg.de, taentzer@mathematik.uni-marburg.de

[c] Faktor Zehn GmbH, Germany
e-mail address: lars.fritsche@faktorzehn.de

Abstract. When using graphs and graph transformations to model systems, consistency is an important concern. While consistency has primarily been viewed as a binary property, i.e., a graph is consistent or inconsistent with respect to a set of constraints, recent work has presented an approach to consistency as a graduated property. This allows living with inconsistencies for a while and repairing them when necessary. For repairing inconsistencies in a graph, we use graph transformation rules with so-called *impairment-indicating and repair-indicating application conditions* to understand how much repair gain certain rule applications would bring. Both types of conditions can be derived from given graph constraints. Our main theorem shows that the difference between the number of actual constraint violations before and after a graph transformation step can be characterised by the difference between the numbers of violated impairment-indicating and repair-indicating application conditions. This theory forms the basis for algorithms with look-ahead that rank graph transformations according to their potential for graph repair. An evaluation shows that graph repair can be well-supported by rules with these new types of application conditions in terms of effectiveness and scalability. This paper provides further theoretical results and an extended evaluation of the results presented in [FLST24].

1. Introduction

Graph transformation has proven to be a versatile approach for specifying and validating software engineering problems [HT20]. This is true because graphs are an appropriate means for representing complex structures of interest, the constant change of structures can be specified by graph transformations, and there is a strong theory of graph transformation [EEPT06] that has been used to validate software engineering problems. When applying graph transformations, it is typically important that the processed graphs are consistent with respect to a given set of constraints. Ensuring graph consistency involves two tasks. First, to specify what a consistent graph is and to check whether a graph is indeed consistent, and second, to ensure that graph transformations preserve or even improve consistency.

Throughout this paper, we consider the Class Responsibility Assignment (CRA) problem as running example [BBL10]. It concerns an optimal assignment of features (i.e., methods

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-22(1:10)2026

and attributes) to classes. The constraints enforcing that each feature belongs to one and only one class are invariants for all operations modifying the class model. To validate the quality of a class model, coupling and cohesion metrics are often used. Related design guidelines, such as "minimize dependencies across class boundaries", can also be formulated as constraints. This example shows that constraints can serve different purposes; some are considered so essential that they must always be satisfied, while others are used for optimization; they may be violated to a certain extent, but the number of violations should be kept as small as possible.

Nested graph constraints [HP09] provide a viable means of formulating graph properties. The related notion of constraint consistency introduced in [HP09] is binary: a graph is either consistent or inconsistent. Since graph repair is often only gradual, it is also interesting to consider graph consistency as a graduated property, as was done in [KSTZ22]. To support gradual repair, graph transformations are analyzed in [KSTZ22] with respect to their potential to improve (sustain) the consistency level of a processed graph, i.e., to strictly reduce (preserve) the number of constraint violations in a graph. A *static analysis approach* is presented in [KSTZ22] for checking whether or not a graph transformation rule is always consistency-sustaining or -improving. That approach does not yet support graph constraints with different priorities or propositional logic operators. In addition, it does not support scenarios where a rule either improves or degrades graph consistency depending on the context of the matched and rewritten subgraph.

To mitigate these problems, we introduce a *new dynamic analysis approach that ranks rule matches and related rule applications according to their effect on improving graph consistency* as follows:

(1) Graph transformation rules are equipped with *impairment-indicating and repair-indicating application conditions*. These conditions no longer block rule applications, but count the number of constraint violations introduced or removed by a given graph transformation step. They are derived from nested graph constraints based on the constructions presented in [HP09].
(2) Our main theorem shows that *the number of additional constraint violations caused by a rule application can be characterized by the difference between the numbers of violations of the associated impairment-indicating and repair-indicating application conditions.*
(3) This theory forms the basis for *graph repair algorithms with a look-ahead* for the graph-consistency-improving potential of selectable rule applications. Based on a prototypical implementation of a greedy algorithm, we show in an initial evaluation that our approach is effective in the sense that it can reduce the number of violations, and it scales well with the available number of rule applications for each consistency-improving transformation step.

This paper extends the approach presented in [FLST24] as follows:

(1) In Section 5 we present a new method for computing *repair-* and *impairment-indicating application conditions* that (a) reduces the total number of overlaps of the left-hand side of a rule and the *premise* of a constraint that need to be considered, (b) results in a reduction of the total number of application conditions, (c) provides a simpler way to compute the change in consistency during a transformation through repair- and impairment-indicating application conditions, and (d) relies solely on *nested graph condition*, whereas in the theory presented in [FLST24], additional information about

the overlap with which an application condition was computed must be considered to correctly determine the change in consistency.

(2) We compare *repair-* and *impairment-indicating application conditions* with *(direct) consistency-sustaining* and *(direct) consistency-improving* transformations as introduced in [KSTZ22] in Section 5.4 and show that *repair-* and *impairment-indicating application conditions* can be used to construct weakest direct consistency-sustaining and weakest direct consistency-improving application conditions, i.e., we obtain application conditions that are satisfied if and only if the transformation is direct consistency-sustaining (-improving).

(3) We further extend our evaluation using weak constraints, which model good patterns for class structures, and evaluate them on the well-known test data of the TTC16 [FTW16] CRA case study and compared it to another tool using the **G**raph-Based (M)**I**LP **P**roblem **S**pecification (GIPS) tool [EKS22], which uses Integer Linear Programming (ILP) to compute the optimal solution for a given input graph.

2. Running Example

To illustrate the problem addressed, we consider a variant of the CRA problem [BBL10, FTW16] for our running example. The CRA aims to provide a high-quality design for object-oriented class structures. Features, i.e., methods and attributes, with dependencies between them are assigned to classes so that the class design achieves high cohesion within classes and low coupling between classes. We slightly adapt the CRA problem by starting with a predefined class diagram and refactoring it by moving features between classes. These refactoring steps aim to reduce dependencies between methods and attributes as well as methods and methods across different classes. They can also group methods with similar attribute dependencies within the same class.

The amount of coupling and cohesion is well reflected by the CRA-index, which is the cohesion ratio minus the coupling ratio within a class diagram [MJ14]. This means that dependencies should be maximised within classes, while minimising dependencies between different classes.

Class diagrams and feature dependencies. Figure 1 shows a class diagram to be refactored on the left, consisting of three classes that model a snippet of an online shopping session. On the right, we see a graph-like representation of the class diagram, in which the classes, methods, and attributes are represented as graph nodes. The names of the classes, methods and attributes are used as node identifiers. There are also edges between methods and attributes, which model access dependencies of methods on attributes.

Rules and constraints. Next, we define two simple refactoring rules and the constraints that they must obey, as shown in Figure 2. The moveAttribute rule moves an attribute from one class to another one, while the moveMethod rule does the same for methods. Graph elements annotated with "- -" are to be deleted, while graph elements annotated with "++" are to be created. From the set of language constraints on class structures, we select two constraints, h_1 and h_2, which impose two basic properties on class models. We consider these constraints to be *hard*, i.e., constraints that must not be violated. These constraints state that methods and attributes must not be contained in more than one class.

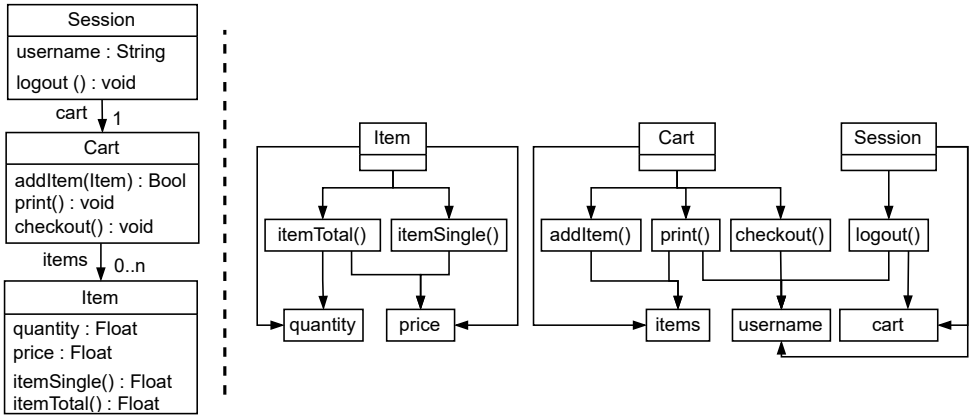

Figure 1: Class diagram (left) and feature dependencies (right)

To model the CRA-index as accurately as possible, we must represent the desired and undesired dependencies as graph constraints that state the following properties:

(1) For each pair of a method and an attribute contained within the same class, the method uses the attribute.
(2) A method cannot use an attribute contained in another class.
(3) Methods contained in the same class must depend on each other.
(4) A method cannot call a method contained in another class.

We will formalise these properties as *weak constraints* that can be violated, but which must be satisfied to the greatest possible extent. We start with the constraint w_2, since the constraint w_1 is special and will be explained later. In Figure 2, the constraint w_2 says that methods and attributes contained in different classes must not be dependent on each other (reflecting the property described in (2) above). Constraint w_3 says that methods contained in the same class must depend on each other (this reflects the property described in (3) above), and constraint w_4 says that methods contained in different classes must not depend on each other (this reflects the property described in (4) above).

A constraint that describes the property stated in (1) would have the same structure as w_3. Such a constraint could be achieved by replacing the method m_2' in w_3 with an attribute and deleting the right pattern within the brackets. However, to illustrate the concepts presented in the theory section as much as possible, we have selected a slightly different constraint w_1. This constraint states that each pair of methods contained in the same class must have at least one common dependency on an attribute within the same class. We use this constraint because it leads to application conditions that can illustrate each part of our construction nicely, in contrast to the application conditions for constraints w_2, w_3, and w_4.

Please note that we will only use w_1 and w_2 in the examples throughout the theoretical part of the paper. Constraints w_3 and w_4 are only used in our extended evaluation.

Obviously, neither rule can violate the hard constraints, but moving features between classes can remove or add violations of the weak constraints, i.e., repair or impair the consistency of the graph under consideration. For example, moving the print() method from the Cart to the Session class would repair a violation of w_2 because the print() depends on the username attribute. However, this would introduce a new violation of the same constraint due to its dependency on the items attribute in the Cart class.

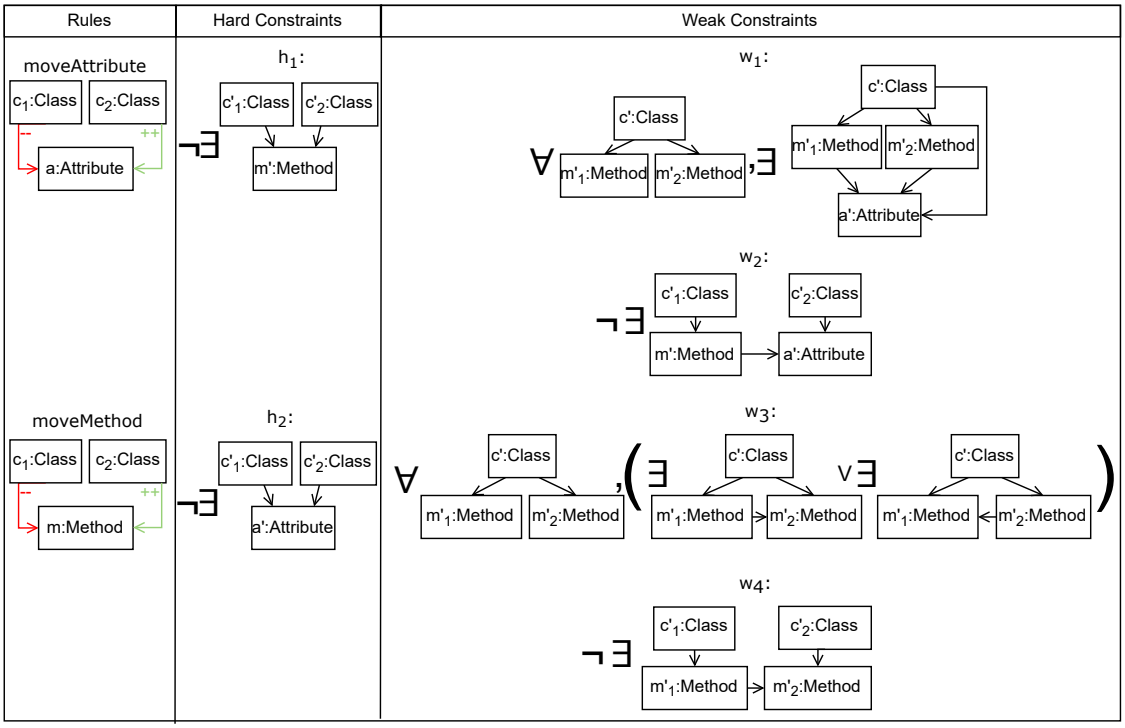

Figure 2: Refactoring rules moveAttribute and moveMethod and constraints: h_1: A method must not be contained in more than one class. h_2: An attribute must not be contained in more than one class. w_1: Two methods within the same class c' should have at least one dependency on a common attribute within the same class c'. w_2: A method should not depend on an attribute from another class. w_3: Methods contained within the same class must have at least one dependency on each other. w_4: A method should not depend on a method of another class.

Ranking of graph transformations for graph repair. We will now examine the impact of graph transformations on the consistency of the graph shown in Figure 1 with respect to the weak constraints w_1 and w_2 by applying the moveMethod and moveAttribute rules. Table 1 shows the actions of the applied rules and the number of impairments and repairs for the constraints w_1 and w_2.

The first rule application uses the rule moveMethod to move checkout() from Cart to Session. As checkout() does not share a common attribute with print() and addItem(), which are contained within the same class, the rule application will perform four repairs for the constraint w_1 (due to the symmetry of w_1, the pairs (checkout(), print()) and (checkout(), addItem()) are counted twice). Additionally, one repair is performed for constraint w_2 because checkout() has a dependency on the username attribute, which is contained in Session.

In the second transformation, moveAttribute is used to move username from Session to Cart. This transformation makes two repairs to w_1. Once the transformation has been performed, both methods print() and checkout() have a dependency on username, which is now contained in Cart. Additionally, there are two repairs of w_2: After the transformation, neither print() nor checkout() have a dependency on an attribute not contained in Cart.

Table 1: Ranking of rule applications of moveMethod and moveAttribute w.r.t. the consistency of the weak constraints w_1 and w_2

Rule	Action			constraint w_1		constraint w_2	
	Element	From	To	rep.	imp.	rep.	imp.
moveMethod	checkout()	Cart	Session	4	0	1	0
moveAttribute	username	Session	Cart	2	0	2	1
moveMethod	print()	Cart	Session	2	0	1	1
moveMethod	addItem()	Cart	Session	2	2	0	1

However, there is also an impairment of w_2 since logout() (contained in Session) has a dependency on username (which is now in Cart).

In the third transformation the method print() is moved from Cart to Session. This transformation performs two repairs of w_1. These are exactly the same repairs as described for the second transformation. Additionally, there is a repair and an impairment of w_2. The repair arises because both print() and username are contained in Session after the transformation. However, items (on which print() depends) remains in Cart. Therefore, there is also an impairment of w_2.

In the fourth transformation, the addItem() method is moved from Cart to Session. This transformation also performs two repairs of w_1. After the transformation, addItem() and checkout() (which do not share an attribute) are no longer contained in the same class. Since Session will contain addItem() and logout(), which do not share a common attribute, this transformation also performs two impairments. Additionally, there is an impairment of w_2, since items and addItem() will be contained in different classes, even though addItem() has a dependency on items.

During a repair process, we would apply the moveMethod rule to move checkout() from Cart to Session, as this maximises the consistency of w_1 and w_2.

Based on this ranking information of rules, various optimisation algorithms can be implemented, including a greedy algorithm that always selects refactoring rule applications with the greatest possible consistency gain. We will report on a greedy implementation of the CRA in Section 6.

Repair-indicating application conditions. As we have seen, there are many ways to apply the moveMethod and moveAttribute rules, even in this small example. Therefore, it is not practically feasible to apply each possible transformation in order to find the one that increases consistency the most. This is why we will present a new methodology that derives application conditions for each refactoring rule. Based on the application conditions, we can calculate a look-ahead for each refactoring transformation (i.e., rule application) to determine how many repairs and impairments will be caused. These application conditions will not prevent rule applications, but will indicate which parts of a graph will be repaired or impaired. More specifically, a rule can have repair- and impairment-indicating application conditions. The set of violated (i.e., not satisfying) occurrences of repair-indicating application conditions shows where the graph would be repaired by the rule match under consideration. Similarly, the set of violated (i.e., not holding) occurrences of impairment-indicating application conditions shows where the graph would be impaired.

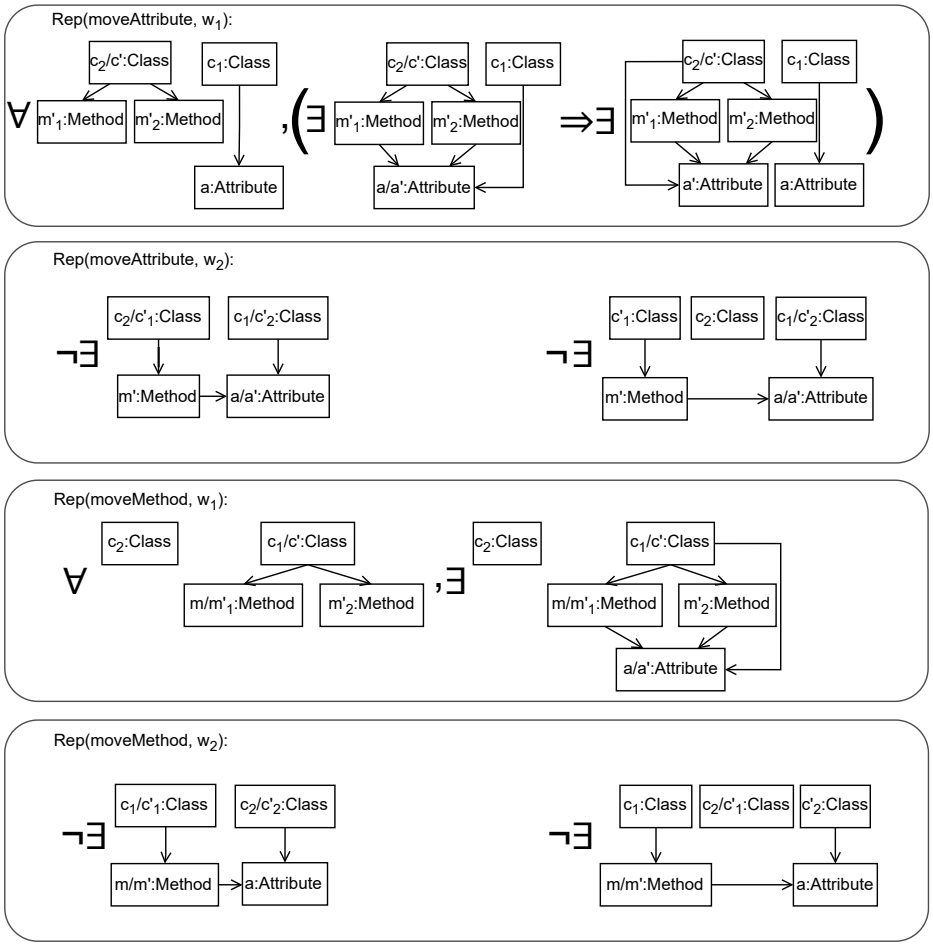

Figure 3: Repair-indicating application conditions for the rules moveAttribute and moveMethod w.r.t. the constraints w_1 and w_2. The node labels implicitly denote the mappings of the LHS of the rule and the premise of the constraint. For example, the label c_2/c' indicates that the node c_2 from the LHS of moveAttribute and c' from the premise of w_1 are mapped to the node c_2/c'.

Figure 3 shows the repair-indicating application conditions for the rules moveAttribute and moveMethod with respect to the constraints w_1 and w_2. The set of repair-indicating application conditions for moveAttribute and w_1, Rep(moveAttribute, w_1), contains one application condition. This condition checks that, for each pair of methods contained within the class to which the attribute is moved, they share a common attribute after the transformation has been performed and they do not share a common attribute before the transformation. This can only happen if they both have a dependency on the moved attribute (as modelled by the premise of the implication) and if they do not both have a dependency on another attribute contained in the same class (as modelled by the conclusion of the implication). Therefore, a repair of w_1 will be found if this implication is not satisfied, i.e., if the premise of the implication is satisfied but the conclusion is not.

The set Rep(moveAttribute, w_2) consists of two repair-indicating application conditions. The left application condition has the same form as the constraint w_2. This application condition checks whether the moved attribute has been moved to a class containing a method that depends on it. If so, a repair of w_2 will be found. The application condition on the right checks whether a method that depends on the moved attribute exists that is not contained within either the attribute's original class nor its new class. An occurrence of this pattern in a transformation does not affect the consistency of w_2 w.r.t. that method and the moved attribute. We will see later on that this application condition is also included in the set of impairment-indicating application conditions for moveAttribute and w_2, which implies that we can discard it.

The set Rep(moveMethod, w_1) contains one application condition. This application condition checks whether the moved method shares a common attribute with every other method that is contained in the same class. If not, a repair is found, since w_1 states that methods contained in the same class must share a common attribute.

The set Rep(moveMethod, w_2) consists of two application conditions. Similar to the application conditions contained in the set Rep(moveAttribute, w_2), the left application condition checks whether the method has been moved to a class containing an attribute on which it depends. If so, a repair of w_2 will be found. The application condition on the right checks whether the moved method depends on an attribute that is not contained in either the method's original class nor its new class. An occurrence of this pattern within a transformation will not affect the consistency of w_2 w.r.t. this attribute and the moved method. We will see later that this application condition is also included in the set of impairment-indicating application conditions for moveMethod and w_2, which implies that we can discard it.

3. Preliminaries

In this section, we recall key notions that are used throughout this paper. Our theory for the construction of graph repair algorithms is based on typed graphs, as introduced for graph transformations in [Ehr78, EEPT06].

Typed graph and graph morphism. A graph consists of nodes and edges, where edges connect nodes. In our running example, classes, methods, and attributes are represented as nodes. Object references, such as the attribute dependencies between a method and an attribute, are represented as edges. *Graph morphisms* are mappings between graphs that preserve the structural properties of their domain graph. In this paper, we will only consider injective graph morphisms. Intuitively, the existence of an injective graph morphism from a graph A to a graph B means that A is a subgraph of B.

Definition 3.1 (Graph and graph morphism)**.** A *graph* $G = (G_V, G_E, s_G, t_G)$ consists of a set G_V of nodes, a set G_E of edges and two mappings $s_G\colon G_E \to G_V$ and $t_G\colon G_E \to G_V$ that assign the source and target nodes for each edge of G. If a tuple as above is not given explicitly, the set of nodes (edges) is denoted by G_V (G_E) and the source (target) mapping is denoted by s_G (t_G).

A *graph morphism* $f\colon G \to H$ between two graphs G and H consists of two mappings $f_V\colon G_V \to H_V$ and $f_E\colon G_E \to H_E$ that preserve the source and target mappings, i.e., $f_V \circ s_G = s_H \circ f_E$ and $f_C \circ t_G = t_H \circ f_E$. A graph morphism is called *injective* (*surjective*)

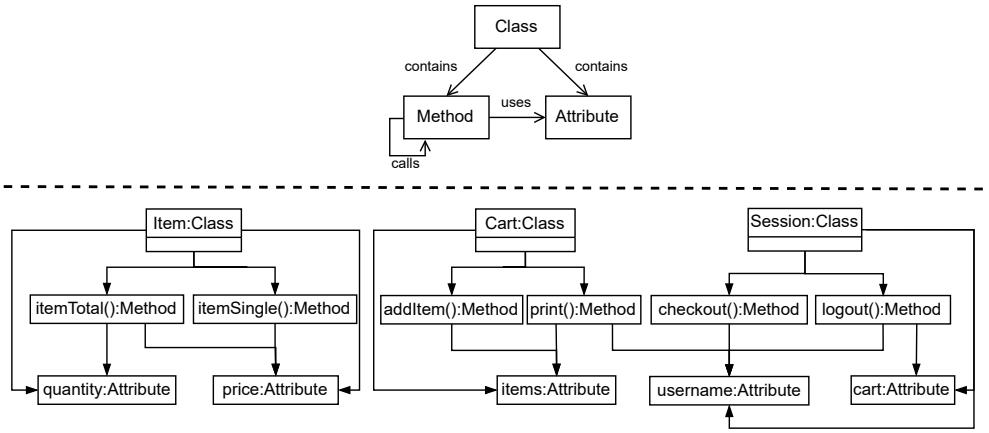

Figure 4: Type graph (top) and typed graph (bottom)

if both mappings f_V and f_E are injective (surjective). An injective morphism f from G to H is denoted by $f\colon G \hookrightarrow H$. If $f\colon G \hookrightarrow H$ is injective, we call f an *occurrence of G in H*. A graph morphism $f\colon G \to H$ is called an *isomorphism* if there is a graph morphism $f^{-1}\colon H \to G$ such that $\mathrm{id}_G = f^{-1} \circ f$ and $\mathrm{id}_H = f \circ f^{-1}$.

Note that, for each graph G, there is a unique morphism $f\colon \emptyset \to G$ originating in the empty graph $\emptyset$.

Example 3.2. For the graph at the bottom in Figure 4, the set of nodes includes Item, Cart, and Session, as well as any nodes representing features. The set of edges contains the arrows in between them. The source and target nodes of each edge are determined by the source and target of the arrow.

If we consider the graph that contains a class c' that is connected to the methods $\mathsf{m'}_1$ and $\mathsf{m'}_2$ as shown in the constraint w_1 in Figure 2, there are several injective graph morphisms from this graph into the graph at the bottom of Figure 4. For example, the node c' could be mapped to Item, $\mathsf{m'}_1$ to itemTotal(), and $\mathsf{m'}_2$ to itemSingle(). The edge from c' to $\mathsf{m'}_1$ is mapped to the edge starting in Item and ending in itemTotal(), and the edge from c' to $\mathsf{m'}_2$ is mapped to the edge starting in Item and ending in itemTotal().

Throughout the paper, we assume that a graph is always finite, i.e., the set of nodes and the set of edges are finite.

Typing provides a way to assign meaning to graph elements. For example, we can assign each node and edge of the graph shown at the top of Figure 4 their respective roles in a class diagram. I.e., the nodes Item, Cart, and Session are typed as classes, the nodes on the second layer itemTotal(), itemSingle(), ... are typed as methods, and the nodes quantity, price, item, username, and cart are typed as attributes. An edge from a class to a method signals that the method is contained in the class, and an edge from a method to an attribute signals that the method uses the attribute. This can be formalised by *typed graphs* and *typed graph morphisms*.

Definition 3.3 (Typed graph, typed graph morphism)**.** Given a graph TG, called the *type graph*, a *typed graph* over TG is a tuple $(G, type)$ consisting of a graph G and a graph morphism $type\colon G \to TG$. Given two typed graphs $G = (G', type_G)$ and $H = (H', type_H)$, a *typed graph morphism* $f\colon G \to H$ is a graph morphism $f\colon G' \to H'$ such that $type_H \circ f = type_G$.

The typed graph morphism $f\colon G \to H$ is *injective (surjective)* if $f\colon G' \to H'$ is injective (surjective). It is called an *isomorphism* if there is a graph morphism $f^{-1}\colon H \to G$ such that $\mathrm{id}_G = f^{-1} \circ f$ and $\mathrm{id}_H = f \circ f^{-1}$.

Example 3.4. The graph G shown at the bottom of Figure 4 is typed over the graph TG shown at the top of Figure 4. The typed graph morphism maps the nodes Item, Cart, and Session to Class. The nodes itemTotal(), itemSingle(), addItem(), print(), checkout(), and logout() are mapped to Method and quantity, price, items, username, and cart are mapped to Attribute. The edges are mapped accordingly.

Throughout the paper, we assume that all graphs are typed graphs. Additionally, all graphs used in our examples and evaluation are typed over the graph shown at the top of Figure 4. For a more compact representation of graphs, we will not specify edge types in the figures for the rest of the paper if they can be identified unambiguously.

Nested graph constraints. To verify that a graph satisfies a desired structural property, a concept of formulating these properties is needed. An example property for class diagrams is that a method should not be contained in two classes. In particular, we want to derive graphs that satisfy these properties or are more consistent with respect to these properties. For example, considering the property described above, a class diagram H is more consistent with respect to this property than another class diagram G if H contains fewer methods that are contained in two classes. To formulate such properties, we use *nested graph conditions* introduced by Habel and Pennemann [HP09]. Rensink has shown that the class of nested graph conditions is equivalent to first-order logic [Ren04]. In addition, almost all Object Constraint Language (OCL) formulas can be translated into nested graph constraints [RAB+18]. Nested graph constraints, or constraints for short, are nested graph conditions that can be evaluated directly on a given graph, whereas graph conditions are generally evaluated with respect to graph morphisms.

Definition 3.5 (Nested graph conditions)**.** A *nested graph condition* over a graph P is of the form

- true, or
- $\exists(e\colon P \hookrightarrow Q, d)$ where d is a condition over Q, or
- $d_1 \vee d_2$ or $\neg d_1$ where d_1 and d_2 are conditions over P.

A condition over the empty graph $\emptyset$ is called *constraint*. We use the abbreviations $\mathsf{false} := \neg\mathsf{true}$, $d_1 \wedge d_2 := \neg(\neg d_1 \vee \neg d_2)$, $d_1 \implies d_2 := \neg d_1 \vee d_2$, and $\forall(e\colon P \hookrightarrow Q, d) := \neg\exists(e\colon P \hookrightarrow Q, \neg d)$. When e is of the form $e\colon \emptyset \hookrightarrow Q$, we use the short notations $\exists(Q, d)$ and $\forall(Q, d)$. For a condition $c = \forall(Q, d)$, we call Q the *premise* and d the *conclusion* of c.

Throughout the paper, we assume that each condition is finite, i.e., the graphs used and the number of nesting levels are finite.

Example 3.6. Figure 2 shows the constraints $\mathsf{h}_1, \mathsf{h}_2, \mathsf{w}_1$, and w_2. The constraints $\mathsf{h}_1, \mathsf{h}_2$, and w_2 specify forbidden patterns where

- h_1 states that a method cannot be contained in two different classes,
- h_2 states that an attribute cannot be contained in two different classes, and
- w_2 states that if a method uses an attribute, they cannot be contained in different classes.

The constraint w_1 states that each pair of methods contained in the same class must use at least one common attribute that is also contained in that class. The graph morphism from the first graph of w_1 to the second graph is implicitly given by mapping nodes with the same identifiers and the edges are mapped accordingly.

Definition 3.7 (Semantics of condition)**.** Given a graph morphism $p\colon P \hookrightarrow G$ and a condition c over P, then p satisfies c, denoted by $p \models d$, if

- $c = \mathsf{true}$, or
- $c = \exists(e\colon P \hookrightarrow P, d)$ and there is a morphism $q\colon Q \hookrightarrow G$ with $p = q \circ e$ and $q \models d$, or
- $c = d_1 \vee d_2$ and $p \models d_1$ or $p \models d_2$, or
- $c = \neg d$ and $p \not\models d$.

A graph G satisfies a constraint c, denoted by $G \models c$ if the unique morphism $p\colon \emptyset \hookrightarrow G$ satisfies c. Two conditions c_1 and c_2 over a graph P are equivalent, denoted by $c_1 \equiv c_2$, if for each morphism $p\colon P \hookrightarrow G$ we have $p \models c_1 \iff p \models c_2$.

Example 3.8. The graph shown in Figure 1 satisfies the hard constraints h_1 and h_2. There is no attribute and no method in two classes. However, the graph does not satisfy the weak constraint w_1 shown in Figure 2. The methods addItem() and checkout() are contained in the same class Cart, but Cart does not contain an attribute used by both methods. The graph also does not satisfy the weak constraint w_2. This is because checkout() uses the attribute username, which is contained in the class Session.

Graph rules and transformations. Graph transformation rules are used to specify state-changing operations for a system of interest. They can be used to specify graph elements that are to be deleted or created when the rule is applied to a graph. In the case of our CRA example, rules are used to define the set of all available refactoring operations. Their associated application conditions are unsatisfied whenever a rule application changes the consistency state of a rewritten graph. In the following, we recall the formal definitions of *graph transformation rules* and *graph transformations* [EEPT06, Ehr78].

Definition 3.9 (Graph transformation rule and application condition)**.** A *graph transformation rule*, or *rule* for short, $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ consists of graph L, called the *Left-Hand Side (LHS)*, K, called the *context*, R, called the *Right-Hand Side (RHS)*, and injective morphisms $l\colon K \hookrightarrow L$ and $r\colon K \hookrightarrow R$. The *inverse rule* of ρ, denoted by ρ^{-1} is defined as $\rho^{-1} := R \xhookleftarrow{r} K \xhookrightarrow{l} L$. An *application condition* for a rule is a nested condition over its LHS.

Example 3.10. Figure 2 shows the rules moveAttribute and moveMethod that move an attribute or a method from one class to another. Elements to be deleted are coloured red, elements to be preserved are coloured black, and elements to be created are coloured green. The LHS of the rules contains the black and red elements, the context contains the black elements, and the RHS contains the black and green elements. When the moveAttribute rule is applied to a graph, the red edge is deleted and the green edge is created.

The condition contained in $\mathrm{Rep}(\mathsf{moveAttribute}, \mathsf{w}_1)$ (Figure 3) is an application condition for the rule moveAttribute. The embedding of the LHS in the first graph of the application condition is implicitly given by the node identifiers, i.e., c_1 is mapped to c_1, c_2 is mapped to c_2/c', and a is mapped to a.

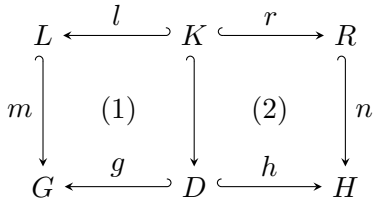

Figure 5: Graph transformation

For simplicity, we present the more constructive, set-theoretic definition of graph transformation, which has been shown to be equivalent to the commonly used double-pushout approach, based on category theory [EEPT06]. A brief introduction to the double-pushout approach is given in Section A, as we need some of its properties for our proofs. Intuitively, a graph transformation consists of two steps: First, elements that are exclusively contained in the LHS of the rule are deleted. The second step is to create elements exclusively contained in the RHS of the rule. Note that a rule is only applicable to a graph if deleting a node does not lead to a dangling edge, i.e., every edge in the resulting graph has a source and a target node.

Definition 3.11 (Graph transformation and derived rule)**.** Given a graph G, a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, and an injective morphism $m\colon L \hookrightarrow G$ (see Figure 5), a *graph transformation* t, denoted by $t\colon G \Longrightarrow_{\rho,m} H$, via ρ at m can be constructed by (a) deleting all nodes and edges of L that do not have a preimage in K, i.e., construct the graph $D = G \setminus m(L \setminus (l(K))$ and (b) adding all nodes and edges of R that do not have a preimage in K, i.e., construct the graph $H = D \dot\cup R \setminus r(K)$, where $\dot\cup$ denotes the disjoint union.

The rule ρ is applicable at m if and only if D is a graph, i.e., if it does not contain any dangling edges. In this case, m is called *match* and the newly created morphism $n\colon R \hookrightarrow H$ is called *comatch*. We call G the *original graph*, D the *interface* and H the *result graph* of the transformation t. The *derived rule* of a transformation $t\colon G \Longrightarrow_{\rho,m} H$, denoted by $\mathrm{der}(t)$, is defined as $\mathrm{der}(t) = G \xhookleftarrow{g} D \xhookrightarrow{h} H$.

Example 3.12. For the rule moveMethod, there is a match m from the LHS in the graph G shown in Figure 1. We can map c_1 to Cart, c_2 to Session, and m to checkout(). When moveMethod is applied to m, we get the graph H shown at the bottom of Figure 4. In the resulting graph, checkout() has been moved from Cart to Session, i.e., first the edge from Cart to checkout() is deleted and then a new edge from Session to checkout() is created.

In addition, we need a method for tracking graph elements throughout a transformation, i.e., we want to know which elements in the original graph of the transformation correspond to which elements in the resulting graph of the transformations, and which elements have been deleted or created. In particular, we want to decide whether occurrences of graphs in other graphs are deleted or created by a transformation.

Definition 3.13 (Track morphism [Plu05])**.** The *track morphism* of a transformation $t\colon G \Longrightarrow H$ (see Figure 5), denoted by $\mathrm{tr}_t\colon G \dashrightarrow H$, is a partial morphism, defined as

$$\mathrm{tr}_t(e) := \begin{cases} h(g^{-1}(e)) & \text{if } e \in g(D) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Example 3.14. If we consider the transformation $t\colon G \Longrightarrow H$ described in Example 3.12, we have $\mathrm{tr}(e) = e$ for each element that is preserved (i.e., for every element except the edge from Cart to checkout()). Also, there is no element $e \in G$, so that the track morphism maps e to the newly created edge from Session to checkout() in H.

4. Counting Constraint Violations

When large graphs are transformed using small rules, such as MoveMethod or MoveAttribute, it is unlikely that the consistency of a constraint will be fully restored by a single transformation. However, consistency can be increased in the sense that the original graph of the transformation contains more violations than the resulting graph.

For example, if we consider the transformation described in Example 3.12, where the method checkout is moved from Card to Session (the original graph is shown in Figure 1; the resulting graph is shown in Figure 4) and the weak constraint w_2 (shown in Figure 2), the consistency increases even though the resulting graph still does not satisfy w_2. The original graph contains two methods print() and checkout() that use an attribute that is not in the same class, whereas the resulting graph contains only one such method (which is print()). Therefore, the resulting graph is more consistent w.r.t. w_2 than the original graph of this transformation.

To rank rule applications in terms of their potential to increase the consistency of a constraint, we need a method to evaluate this increase or decrease in the consistency of weak constraints. Here, we rely on the *number of violations* introduced by Kosiol et al. in [KSTZ22]. This notion enables an increase in consistency to be detected even if it is not fully recovered. For example, it can detect whether an occurrence of the premise that does not satisfy the conclusion has been deleted or extended so that it satisfies the conclusion in the result graph of the transformation. Although this notion was originally introduced for constraints in so-called *alternating quantifier normal form*, i.e., the set of all nested constraints that do not use conjunctions or disjunctions, we have extended it to support universally bounded nested conditions that may use these operators in the conclusion.

We allow hard constraints and weak constraints to be arbitrary nested graph conditions.

Although our theory is developed for universal weak constraints (i.e., weak constraints of the form $\forall(P, d)$), this does not limit the range of supported constraints: *Every condition c over a graph P is equivalent to the condition* $\forall(\mathrm{id}_P\colon P \hookrightarrow P, c)$. This means that any nested graph condition can be moulded into the required form. For example, every existential constraint $c = \exists(e\colon \emptyset \hookrightarrow P, d)$ is equivalent to the condition $\forall(id_\emptyset\colon \emptyset \hookrightarrow \emptyset, c)$. In this case, the number of violations acts as a binary property, i.e., it is equal to 1 if the constraint is unsatisfied and equal to 0 if the constraint is satisfied. This is because there is only one morphism $p\colon \emptyset \hookrightarrow G$ into any graph G that either satisfies or does not satisfy the constraint c. This effect complies with the semantics of existential graph conditions, since only one repair is sufficient for the resulting graph of a transformation to satisfy the constraint. Therefore, inserting occurrences of P that do not satisfy d does not decrease the consistency of a graph, since only one of these occurrences needs to be repaired, or a new occurrence of P that satisfies d needs to be inserted.

Definition 4.1 (Set and number of violations)**.** Given a condition $c = \forall(e\colon C \hookrightarrow P, d)$ and a graph morphism $p\colon C \hookrightarrow G$ (Figure 6), the *set of violations of c in p*, denoted by $\mathrm{sv}_p(c)$, is

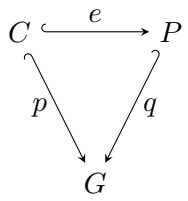

Figure 6: Morphisms used to evaluate the *set of violations*

defined as

$$\mathrm{sv}_p(c) := \{q\colon P \hookrightarrow G \mid p = q \circ e \text{ and } q \not\models d\}.$$

The set of violations of a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$ in a graph G, denoted by $\mathrm{sv}_G(c)$, is defined as $\mathrm{sv}_G(c) := \mathrm{sv}_p(c)$ where $p\colon \emptyset \hookrightarrow G$ is the unique morphism into G.

The *number of violations of a condition c in p*, denoted by $\mathrm{nv}_p(c)$, is defined as $\mathrm{nv}_p(c) := |\mathrm{sv}_p(c)|$. The *number of violations of a constraint c in a graph G*, denoted with $\mathrm{nv}_G(c)$, is defined as $\mathrm{nv}_G(c) := |\mathrm{sv}_G(c)|$.

Example 4.2. As discussed in Example 3.8, the graph G in Figure 1 does not satisfy w_1 (which states that two methods contained within the same class must use a common attribute which is also contained in the same class) and w_2 (which states that a method and its used attributes cannot be contained in different classes). For w_1, there are two pairs of methods that do not use a common attribute in the same class (addItem() and checkout(); print() and checkout()). Each of these pairs, accompanied with the class they are contained in, is detected twice by the premise of the constraint. For addItem() and checkout(), we either map m_1' to addItem() and m_2' to checkout() or vice versa. So the total number of violations of w_1 in G is 4.

When considering w_2, the methods print() and checkout() contained in the class Cart use the attribute username, which is not contained in Cart. Therefore, the total number of violations of w_2 in G is 2.

5. Application Conditions for Consistency Monitoring

The main idea of our approach is to predict the change in consistency induced by the application of a rule. This allows us to apply the most consistency-increasing rule without using a trial-and-error approach. To obtain this prediction, we use application conditions that do not block the application of constraint-violating rules, but instead annotate rule matches with the number of constraint impairments and/or repairs caused by the associated transformation; thus, the conditions monitor the consistency change.

This section begins by recalling some preliminaries from the theory of graph transformation that are necessary for our new theory. To derive application conditions from a constraint, the first main step is to construct all overlaps of a rule and a constraint. The set of application conditions should be concise, so we identify equivalence classes of overlaps and work with representatives of these classes. This allows us to simplify the construction and complexity of application conditions compared to the construction presented in [FLST24]. We use the well-known techniques introduced by Habel and Pennemann to construct so-called *consistency-preserving* and *consistency-guaranteeing* application conditions [HP09] to construct the application conditions. To detect all situations in which a violation of a

constraint $c = \forall(P, d)$ is repaired by applying a rule ρ, we need to consider each overlap of the LHS of the rule ρ and the premise P of the constraint. To find repairs of the premise P, we consider overlaps where the occurrence of P is destroyed by the transformation. To find repairs of the conclusion d, we consider overlaps where the occurrence of P is preserved by the transformation. If this occurrence of P satisfies d in the resulting graph of the transformation but does not satisfy d in the original graph of the transformation, then this occurrence is a repair of d.

For the construction of impairment-indicating application conditions, we use the observation that a rule introduces a violation of a constraint c if and only if the inverse rule repairs a violation of c. Thus, repair-indicating application conditions for a rule ρ are impairment-indicating application conditions for its inverse rule ρ^{-1}. Our main theorem shows that *the number of additional constraint violations and repairs caused by a rule application can be characterized by the difference between the numbers of violations of the associated impairment-indicating and repair-indicating application conditions*. We call this difference the *gain in consistency*. Constraints can be assigned weights to prioritise certain ones over others, if desired. Finally, transformations with repair-indicating and impairment-indicating application conditions are compared with consistency-sustaining and consistency-improving transformations presented in [KSTZ22]. We found out that a transformation is consistency-sustaining if the gain in consistency is not negative, it is consistency-improving if the gain is positive. The proofs of the results presented in this section are included in Section B.

5.1. **Preliminaries.** In the following, we will briefly introduce the preliminaries that are needed for our construction of application conditions.

An *overlap* $o = (i_G, i_H, GH)$ of graphs G and H consists of jointly surjective morphisms[1] $i_G \colon G \hookrightarrow GH$ and $i_H \colon H \hookrightarrow GH$, called the *overlap morphisms*, and the common codomain GH, called the *overlap graph*. The set of all overlaps of G and H is denoted by $\mathrm{OL}(G, H)$.

The *shift along morphism* operator allows shifting a condition c over a graph P along a morphism $i_P \colon P \hookrightarrow PL$ to extend the graph P by further graph elements. This construction results in a condition over PL that is equivalent in the sense that each morphism $p' \colon PL \hookrightarrow G$ into some graph G satisfies the shifted condition if and only if $p' \circ i_P$ satisfies c. Given a condition c over a graph P and a morphism $i_P \colon P \hookrightarrow PL$, the *shift along* i_P [HP09], denoted by $\mathrm{Shift}(i_P, c)$, is defined as follows:

- if $c = \mathsf{true}$, $\mathrm{Shift}(i_P, c) = \mathsf{true}$, and
- if $c = \exists(e \colon P \hookrightarrow Q, d)$, $\mathrm{Shift}(i_P, c) = \bigvee_{(e', i_Q)} \exists(e' \colon PL \hookrightarrow QL, \mathrm{Shift}(i_Q, d))$, where (e', i_Q, QL) is an overlap of PL and Q such that $e' \circ i_P = i_Q \circ e$, i.e., the square on the right is commutative, and
- if $c = c_1 \vee c_2$, $\mathrm{Shift}(i_P, c) = \mathrm{Shift}(i_P, c_1) \vee \mathrm{Shift}(i_P, c_2)$, and
- if $c = \neg c_1$, $\mathrm{Shift}(i_P, c) = \neg\, \mathrm{Shift}(i_P, c_1)$.

$$\begin{array}{ccc} PL & \xleftarrow{\;i_P\;} & P \\ {\scriptstyle e'}\big\downarrow & & \big\downarrow{\scriptstyle e} \\ QL & \xleftarrow{\;i_Q\;} & Q \end{array}$$

Note that, when shifting a constraint $\forall(P, d)$ over a morphism $p \colon \emptyset \hookrightarrow P'$, the first step is to construct all overlaps $\mathrm{OL}(P, P')$.

The *shift over rule* operator transforms a right-application condition into an equivalent left-application condition and vice versa. Given a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ and a condition c over R, the *shift of* c *over* ρ [HP09], denoted by $\mathrm{Left}(\rho, c)$ is defined as follows:

[1] Two morphisms $p \colon P \hookrightarrow G$ and $q \colon Q \hookrightarrow G$ into the same graph G are called *jointly surjective* if for each element $e \in G$ either there is an element $e' \in P$ with $p(e') = e$ or there is an element $e' \in Q$ with $q(e') = e$.

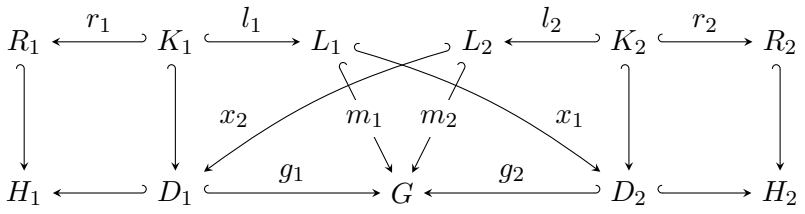

Figure 7: Parallel independent transformations

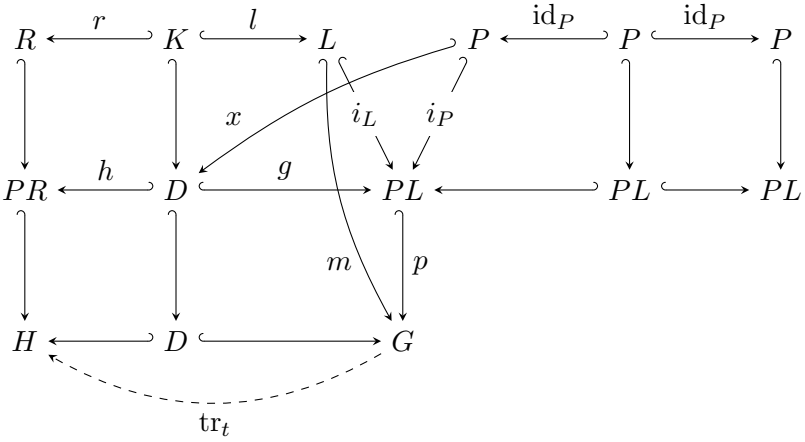

Figure 8: Induced transformations of an overlap (i_L, i_P, PL) of a rule $\rho = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ and a graph P and application to a graph G

- If $c = \mathsf{true}$, $\mathrm{Left}(\rho, c) = \mathsf{true}$, and
- if $c = \exists(e\colon R \hookrightarrow P, d)$, $\mathrm{Left}(\rho, c) := \exists(n\colon L \hookrightarrow P', \mathrm{Left}(\mathrm{der}(t)^{-1}, d))$, where P' is the resulting graph, n the comatch and $\mathrm{der}(t)$ the derived rule of the transformation $t\colon P \Longrightarrow_{\rho^{-1},e} P'$. If there is no such transformation, we set $\mathrm{Left}(\rho, c) = \mathsf{false}$.
- If $c = c_1 \vee c_2$, $\mathrm{Left}(\rho, c) := \mathrm{Left}(\rho, c_1) \vee \mathrm{Left}(\rho, c_1)$.
- If $c = \neg c_1$, $\mathrm{Left}(\rho, c) := \neg\, \mathrm{Left}(\rho, c_1)$.

Definition 5.1 (Parallel independent graph transformations)**.** Two graph transformations $t_1\colon G \Longrightarrow_{m_1,\rho_1} H_1$ and $t_2\colon G \Longrightarrow_{m_2,\rho_2} H_2$ via rules $\rho_1 = L_1 \overset{l_1}{\hookleftarrow} K_1 \overset{r_1}{\hookrightarrow} R_1$ and $\rho_2 = L_2 \overset{l_2}{\hookleftarrow} K_2 \overset{r_2}{\hookrightarrow} R_2$ are *parallel independent* if there are morphisms $x_1\colon L_1 \hookrightarrow D_2$ and $x_2\colon L_2 \hookrightarrow D_1$ such that $g_2 \circ x_1 = m_2$ and $g_1 \circ x_2 = m_1$. The transformations and morphisms are shown in Figure 7. If a pair of transformations is not parallel independent, it is called *parallel dependent*.

5.2. **Equivalence of Overlaps.** To ensure that each occurrence of the premise of a constraint is detected by the application conditions, so that each violation or repair of the constraint during the transformation can be detected, we need to consider each overlap of the LHS of the rule ρ that is to be applied and the premise of the constraint that is to be considered, called *overlaps of rule and graph*. Additionally, we require that the overlap has

induced transformations, i.e., that the embedding $i_L\colon L \hookrightarrow PL$ of the LHS in the overlap graph is a match for the rule and that the rule is applicable at this embedding (see Figure 8). This means that applying the rule at the overlap does not create any dangling edges. We do not need to consider overlaps without *induced transformations* since they cannot occur in a transformation t (which does not produce any dangling edges) so that the match m factors[2] through i_L and the occurrence of PL as this factorisation is needed for the number of violations (see Definition 4.1).

Definition 5.2 (Overlap induced transformation, Overlaps of rule and graph)**.** Given a rule $\rho := L \xleftarrow{l} K \xhookrightarrow{r} R$ and a graph P, the *set of overlaps of ρ and P*, denoted by $\mathrm{OL}(\rho, P)$ is defined as

$$\mathrm{OL}(\rho, P) := \{(i_L, i_P, PL) \in \mathrm{OL}(L, P) \mid \rho \text{ is applicable at } i_L\}.$$

The *induced transformations of an overlap* $(i_L, i_P) \in \mathrm{OL}(\rho, P)$ are given by $t_1\colon PL \Longrightarrow_{\rho, i_L} PR$ and $t_2\colon PL \Longrightarrow_{\mathrm{check}_{PL}, i_P} PL$ where $\mathrm{check}_{PL} = PL \xleftarrow{\mathrm{id}_{PL}} PL \xhookrightarrow{\mathrm{id}_{PL}} PL$ does not create or delete any elements.

Note that the induced transformation $t_2\colon PL \Longrightarrow_{\mathrm{check}_P, i_P} PL$ of an overlap (i_L, i_P, PL) always exists because no elements are deleted or created. However, we will use this transformation to determine which type of repair (impairment) is detected by an application condition constructed using this overlap: If a rule match can be extended by the overlap graph, we have found an occurrence of the premise. If this occurrence is also a violation of the constraint, it can be repaired either by deleting it or by extending it so that it satisfies the conclusion of the constraint in the resulting graph of the transformation.

Example 5.3. All overlaps (and their equivalence classes) of the rule moveAttribute and constraint w_1 are shown in Figure 9. The embeddings of the LHS and the premise in the graphs are implicitly given by the node identifiers, i.e, if we consider the overlap o_1 and the LHS of moveAttribute, the node c_1 is mapped to c_1/c', c_2 is mapped to c_2, and a is mapped to a. For the premise of w_1 and o_1, c' is mapped to c_1/c', m'_1 is mapped to m'_1, and m'_2 is mapped to m'_2. All of the overlaps have induced transformations. Applying the rule moveAttribute at an overlap will never produce a dangling edge, since moveAttribute does not delete any nodes.

When constructing overlaps of graphs, some of them are redundant in the sense that they will detect the same occurrences of the graph they were constructed with, i.e., if for two overlaps (i_G, i_H, GH) and (i'_G, i'_H, GH') of the same graphs G and H we have: If there is a morphism $p\colon GH \hookrightarrow X$ into some graph X, there is also a morphism $p'\colon GH' \hookrightarrow X$, so that $p \circ i_G = p' \circ i'_G$ and $p \circ i_H = p' \circ i'_H$. For example, the overlaps o_1 and o_2 of the LHS of moveAttribute and the premise of w_1 (see Figure 9) will detect the same occurrences of the premise of w_1 (i.e., methods contained within the same class), as they only differ in the mapping of the method nodes. When evaluating application conditions for the number of repairs (impairs), this will lead to overcounting. In the graph shown in Figure 4, there are two occurrences of the premise of w_1 involving the nodes Item, itemTotal, and itemSingle (m'_1 can be mapped to itemTotal, m'_2 can be mapped to itemSingle, and vice versa). However, both overlaps o_1 and o_2 will detect two occurrences of the premise, which results in a total of four detected occurrences of the premise. In other words, if these occurrences do not satisfy the conclusion of w_1 and are repaired during a transformation, two violations will be

[2]A morphism $e\colon G \to H$ factors through morphisms $f\colon G \to X$ and $g\colon X \to G$ if $e = g \circ f$.

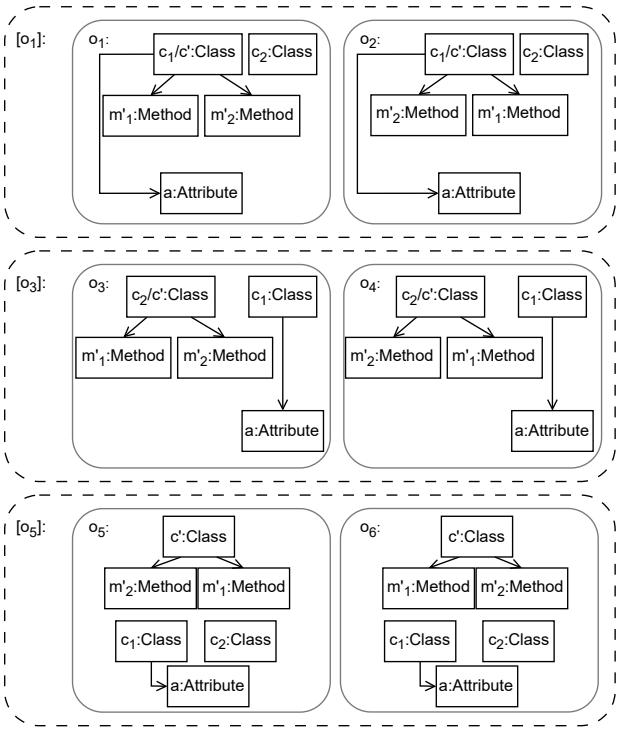

Figure 9: Overlaps $o_1, \ldots, o_6$ of the LHS of moveAttribute and the premise of w_1 and their respective equivalence classes $[o_1], [o_3]$ and $[o_5]$ (marked with the dashed boxes)

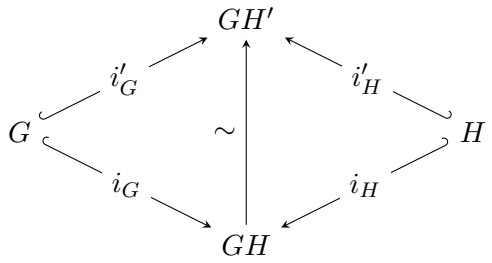

Figure 10: Isomorphism for the equivalence of overlaps

repaired, but the application condition will detect four repairs of the constraint. This will lead to a false prediction of consistency gain if this overcounting is not corrected in some way.

To overcome this, we introduce a notion of *equivalence for overlaps* and will then compute only one application condition for each of these equivalence classes, so that repairs (impairments) are only detected once. This leads to a reduction in the total number of application conditions that need to be computed, and a simpler method of calculating the total gain in consistency using application conditions. Intuitively, two overlaps of the same graphs are *equivalent* if there is an isomorphism between the overlap graphs that preserves the embeddings of the graphs with which these overlaps were computed.

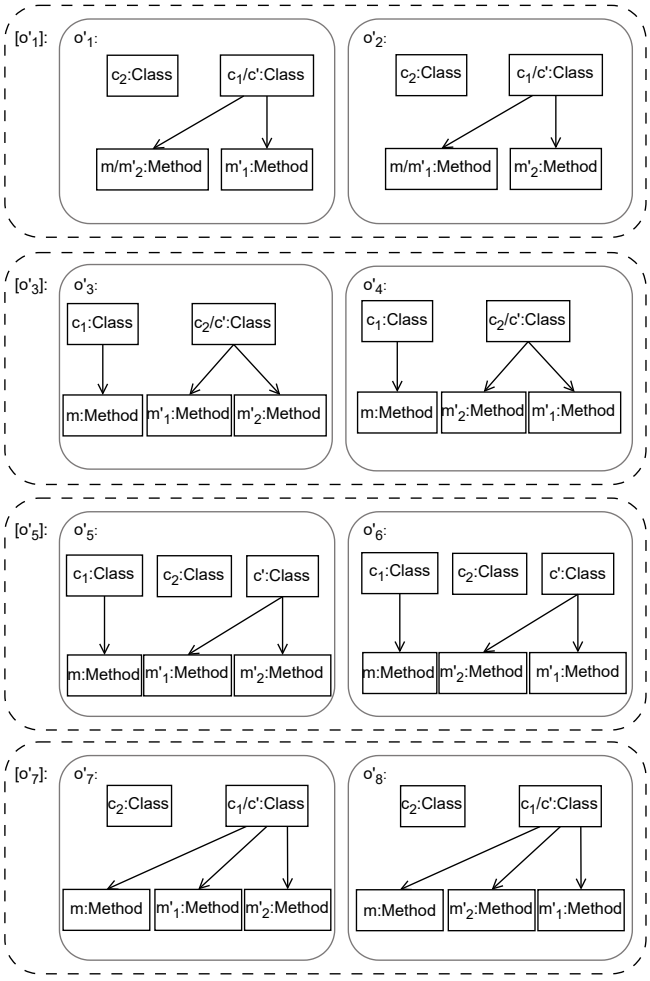

Figure 11: Overlaps $o'_1, \ldots, o'_8$ of the LHS of moveMethod and the premise of w_1 and their respective equivalence classes $[o'_1], [o'_3], [o'_5]$ and $[o'_7]$ (marked with the dashed boxes)

Definition 5.4 (Equivalence of overlaps)**.** Two overlaps $o = (i_G \colon G \hookrightarrow GH, i_H \colon H \hookrightarrow GH)$ and $o' = (i'_G \colon G \hookrightarrow GH', i'_H \colon H \hookrightarrow GH')$ over the graphs G and H are *equivalent*, denoted by $o \cong o'$, if there is an isomorphism $\sim \colon GH \hookrightarrow GH'$ so that $i'_G = \sim \circ i_G$ and $i'_H = \sim \circ i_H$, i.e., the diagram shown in Figure 10 commutes. The *equivalence class of an overlap* o is given by

$$[o] := \{o' \mid o \cong o'\}.$$

Example 5.5. The overlap equivalence classes of moveAttribute and the premise of w_1 are shown in Figure 9. The node labels implicitly denote the mappings of the LHS of the rule and the overlapped graph. For example, the label c_1/c' indicates that the node c_1 from the LHS of moveAttribute and c' from the premise of w_1 are mapped to c_1/c', i.e., $i_L(c_1) = c_1/c'$ and $i_P(c') = c_1/c'$. There are three equivalence classes, which are marked by the dashed boxes. Figure 11 shows the equivalence classes of moveMethod and the premise of w_1. Again, we obtain three equivalence classes.

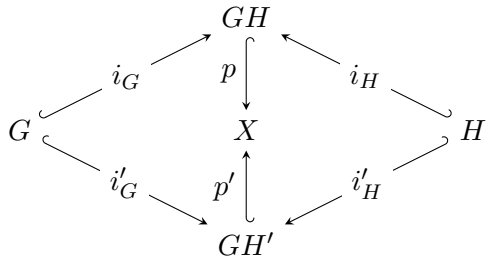

Figure 12: Diagram for the characterisation of equivalence classes of overlaps

As the following lemma shows, two overlaps are equivalent if and only if they can detect the same occurrences of the graph P. It ensures that the notion of *equivalence of overlaps* has the desired effect as described above.

Lemma 5.6 (Characterisation of equivalence classes)**.** *Given two graphs G and H, two overlaps $(i_G, i_H), (i'_G, i'_H) \in \mathrm{OL}(G, H)$ with overlap graphs GH and GH', and a morphism $p\colon GH \hookrightarrow X$ into some graph X. Then $(i_G, i_H) \cong (i'_G, i'_H)$ if and only if there is a morphism $p'\colon GH' \hookrightarrow X$ so that the diagram shown in Figure 12 commutes, i.e.,*

$$p' \circ i'_G = p \circ i_G \text{ and}$$
$$p' \circ i'_H = p \circ i_H.$$

In particular, this is a crucial observation for the correctness of our approach: If we consider a constraint $\forall(P, d)$ and evaluate an application condition constructed over some equivalence class $[(i_L, i_P, PL)]$ with respect to some transformation $t\colon G \Longrightarrow_{\rho,m} H$ using some rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ at the match m, we consider only those occurrences $p\colon PL \hookrightarrow G$ of the overlap graph, so that the match m factors through p and i_L (i.e., $m = p \circ i_L$). Lemma 5.6 implies that $p \circ i_P \neq p' \circ i'_P$ (both morphisms p and p' are occurrences of the premise in G) if $[(i'_L, i'_P, PL')]$ is another equivalence class and $p'\colon PL' \hookrightarrow G$ is an occurrence of PL', so that $m = p' \circ i'_L$. Therefore, each repair (impairment) of the constraint is detected by exactly one application condition, and we only need to consider the set of *equivalence classes of overlaps of a rule and premise of a constraint* to compute the necessary application conditions.

5.3. **Construction of Application Conditions.** When constructing impairment- and repair-indicating application conditions, all cases in which an impairment can be introduced, or a violation can be repaired must be considered. A transformation $t\colon G \Longrightarrow H$ can repair violations of a constraint $c = \forall(P, d)$ in the following two ways:

(1) *Repair of the premise:* An occurrence of P in G that does not satisfy d is deleted.
(2) *Repair of the conclusion:* There exists an occurrence of P which satisfies d in H but does not satisfy d in G.

Similarly, the transformation t can introduce impairments in the following ways:

(1) *Impairment of the premise*: A new occurrence of P is introduced that does not satisfy d.
(2) *Impairment of the conclusion*: An occurrence of P satisfies d in G and not in H.

The following notions of *repaired* and *impaired morphisms* characterise occurrences of the premise of the constraint that are either repaired or impaired.

Definition 5.7 (Repaired morphism)**.** Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a morphism $p\colon P \hookrightarrow G$ is called a *repaired morphism w.r.t. a transformation* $t\colon G \Longrightarrow_{\rho,m} H$ if $p \not\models d$ and

(1) p is a *repair of the premise*, if $\mathrm{tr}_t \circ p$ is not total, i.e., p is destroyed by the transformation, or
(2) p is a *repair of the conclusion*, if $\mathrm{tr}_t \circ p$ is total and $\mathrm{tr}_t \circ p \models d$, i.e., p satisfies the conclusion after the transformation.

Definition 5.8 (Impaired morphism)**.** Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a morphism $p\colon P \hookrightarrow H$ is called an *impaired morphism w.r.t. a transformation* $t\colon G \Longrightarrow_{\rho,m} H$ if $p \not\models d$ and

(1) p is an *impair of the premise*, if there is no morphism $p'\colon P \hookrightarrow G$ with $p = \mathrm{tr}_t \circ p'$, i.e., p is created by the transformation, or
(2) p is an *impair of the conclusion*, if there is a morphism $p'\colon P \hookrightarrow G$ with $p = \mathrm{tr}_t \circ p'$ and $p' \models d$, i.e., p satisfied the conclusion in the original graph of the transformation.

The notions of *repaired* and *impaired* morphisms of a transformation t are closely related, since a repaired morphism w.r.t. t and some constraint c is an impaired morphism w.r.t. the inverse transformation t^{-1} and c. This observation also implies that we only need an overlap-based construction for repair-indicating application conditions, since impair-indicating application conditions can then be constructed by computing the repair-indicating application conditions of the inverse rule and shifting them to the LHS.

Lemma 5.9. *Given a constraint* $c = \forall(e\colon \emptyset \hookrightarrow P, d)$*, a morphism* $p\colon P \hookrightarrow H$ *is a repaired morphism w.r.t. a transformation* $t\colon G \Longrightarrow_{\rho,m} H$ *if and only if* p *is an impaired morphism w.r.t. the inverse transformation* $t^{-1}\colon H \Longrightarrow_{\rho^{-1},n} G$*, where* n *is the comatch of* t*.*

In order to construct application conditions that are able to detect both, repairs (impairments) of the premise and repairs (impairments) of the conclusion, different overlaps of the LHS, and the premise of the constraints must be considered. For repairs of the premise during a transformation $t\colon G \Longrightarrow_{\rho,m} H$, we want to detect those occurrences of the premise that are destroyed by a transformation, i.e., we are searching for an occurrence $p\colon PL \hookrightarrow G$ of an overlap $(i_L, i_P, PL) \in \mathrm{OL}(\rho, P)$ (where P is the premise of the constraint to be considered) so that $m = p \circ i_L$ and the occurrence $p \circ i_P$ of the premise is destroyed. This means that we have to consider each overlap, so that the induced transformations are parallel dependent, since the application of ρ at m will destroy i_P (this implies, in particular, that a transformation of ρ at i_L destroys the occurrence i_P).

For repairs of the conclusion we want to detect those occurrences of the premise that are preserved by a transformation, i.e., we are searching for occurrences $p\colon PL \hookrightarrow G$ of an overlap $(i_L, i_P, PL) \in \mathrm{OL}(\rho, P)$ so that $m = p \circ i_L$ and the occurrence $p \circ i_P$ is preserved. This means that we have to consider every overlap, so that the induced transformations are parallel independent. Therefore, we decompose the set of all equivalence classes into a set containing the classes whose induced transformations are parallel independent and a set containing the equivalence classes whose induced transformations are parallel dependent. Note that if the induced transformations of an overlap o are parallel dependent (independent), then the same is true for each overlap o' that is equivalent to o.

Definition 5.10. Given a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ and a graph P, the set of *equivalence classes of overlaps of* ρ *and* P, denoted by $\mathrm{OL}_\sim(\rho, P)$, is defined as the disjoint union

$\mathrm{OL}_{\sim}(\rho, P) = \mathrm{OL}_{\mathrm{dep}}(\rho, P) \dot{\cup} \mathrm{OL}_{\mathrm{ind}}(\rho, P)$ where

$\mathrm{OL}_{\mathrm{dep}} := \{[o] \in \mathrm{OL}(\rho, P) \mid$ The induced transformations of o are parallel dependent$\}$ and

$\mathrm{OL}_{\mathrm{ind}} := \{[o] \in \mathrm{OL}(\rho, P) \mid$ The induced transformations of o are parallel independent$\}$.

Example 5.11. The overlap equivalence classes contained in $\mathrm{OL}_{\sim}(\mathsf{moveAttribute}, P)$, where P is the premise of w_1, are shown in Figure 9. There are three equivalence classes (marked by the dashed boxes). We have $\mathrm{OL}_{\sim}(\mathsf{moveAttribute}, P) = \mathrm{OL}_{\mathrm{ind}}(\mathsf{moveAttribute}, P)$, because the induced transformations are parallel independent for each of these classes: When applying moveAttribute, the edges from nodes which contain the identifier c_1 to attribute a are deleted. This edge is not mapped by the premise of w_1. Figure 11 shows the equivalence classes of moveMethod and the premise of w_1. Here, we have $\mathrm{OL}_{\mathrm{dep}}(\mathsf{moveMethod}, P) = \{[o_1']\}$ (applying moveMethod would delete the edges from $\mathsf{c}_1/\mathsf{c'}$ to m/m_1 and m/m_2 respectively; these are mapped by the premise of w_1) and $\mathrm{OL}_{\mathrm{ind}}(\mathsf{moveMethod}, P) = \{[o_3'], [o_5'], [o_7']\}$ (here, the deleted edges from classes to methods are not mapped by the premise of w_1).

To check whether an occurrence of the premise of a constraint satisfies the conclusion before and after a transformation (i.e., in the original and in the resulting graph of a transformation), we introduce *overlap-induced pre- and post-conditions*. Given an overlap (i_L, i_P, PL) of a constraint with the LHS of a given rule ρ, the overlap-induced pre-condition checks whether an occurrence of the premise satisfies the conclusion of the constraint. I.e., an occurrence $p\colon PL \hookrightarrow G$ (see Figure 8) satisfies the overlap-induced pre-condition if and only if $p \circ i_P$ satisfies the conclusion of the constraint. The overlap-induced post-condition gives a look-ahead on whether an occurrence of the premise will satisfy the conclusion of the constraint after some transformation. In particular, an occurrence $p\colon PL \hookrightarrow G$ satisfies the overlap induced post-condition if and only if $\mathrm{tr}_t \circ p \circ i_P$ satisfies the conclusion of the constraint where $t\colon G \Longrightarrow_{\rho, p \circ i_L} H$. Note that we only need the overlap-induced post-condition if the occurrence $p \circ i_P\colon P \hookrightarrow G$ is not destroyed during the transformation, i.e., if the induced transformations of the overlap are parallel independent. Otherwise, if the induced transformations of the overlap are parallel dependent, the overlap-induced post-condition is equal to true.

Definition 5.12 (Overlap-induced pre- and post-conditions)**.** Given a rule $\rho = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$, an overlap $o = (i_L, i_P, PL) \in \mathrm{OL}(\rho, P)$, and a condition d over P, the *overlap-induced pre-* and *post-condition* of ρ w.r.t. o and d, denoted by $\mathrm{Pre}_\rho(o, d)$ and $\mathrm{Post}_\rho(o, d)$ are defined as

(1) $\mathrm{Pre}_\rho(o, d) := \mathrm{Shift}(i_P, d)$, and

(2) $\mathrm{Post}_\rho(o, d) := \begin{cases} \mathrm{Left}(\rho', \mathrm{Shift}(h \circ x, d)) & \text{if } [o] \in \mathrm{OL}_{\mathrm{ind}}(\rho, P) \\ \mathsf{true} & \text{if } [o] \in \mathrm{OL}_{\mathrm{dep}}(\rho, P), \end{cases}$

where x, h are the morphisms and $\rho' = H \overset{h}{\hookleftarrow} D \overset{g}{\hookrightarrow} PL$ is the rule as shown in Figure 8.

Example 5.13. If we consider the rule moveAttribute, constraint w_1 (Figure 2) and the equivalence class $[o_3]$ shown in Figure 9, the induced pre- and post-conditions of moveAttribute w.r.t. $[o_3]$ are shown in Figure 13. If the overlaps of $[o_3]$ occur in a transformation via moveAttribute, the attribute a is moved from class c_1 to class c_2. The pre-condition checks whether each pair of methods contained in the class c_2 shares a common attribute before the transformation is performed. There are two possibilities for this: Either an additional

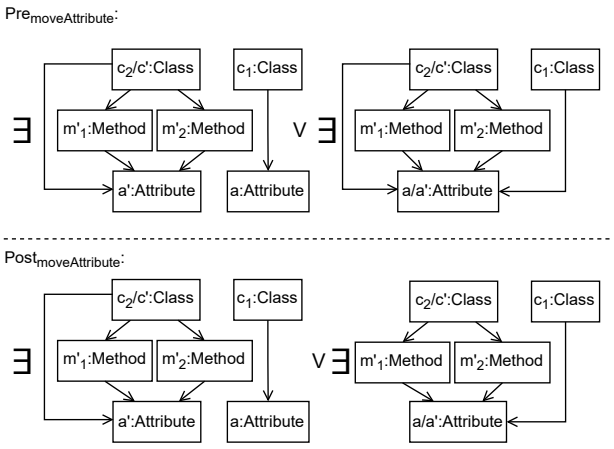

Figure 13: Induced pre- and post-conditions of moveAttribute w.r.t. $[o_3]$

attribute a' contained in c_2 is shared by both methods (as in the left graph of the pre-condition), or they share the attribute a, which is to be moved, and a is also contained in c_2 (as in the right graph of the pre-condition).

The post-condition predicts whether each pair of methods contained in the class c_2 shares a common attribute after the transformation has been performed. Again, there are two possibilities for this: Similar to the pre-condition, either an additional attribute a' contained in c_2 can be shared by both methods (as in the left graph of the post-condition) or they share the attribute a, which is to be moved (as in the right graph of the pre-condition). Note that, for the post-condition a must not be contained in c_2 before the transformation, as it will be contained in c_2 after the transformation has been performed.

Before continuing with the construction of the application condition, we present two simplifications for conditions that will reduce the number of graphs contained in the application conditions. The first lemma enables us to simplify the application conditions by exploiting the fact that hard constraints are always satisfied. If we consider a hard constraint of the form $c = \neg\exists(P)$ (which is satisfied by a graph G if P is not a subgraph of G) and a condition $\exists(e\colon P' \hookrightarrow Q, d)$, a morphism $p\colon P' \hookrightarrow G$ cannot satisfy $\exists(e\colon P' \hookrightarrow Q, d)$ if $Q \not\models c$, otherwise P is a subgraph of G. Therefore, when simplifying an application condition, we can replace any condition of the form $\exists(e\colon P' \hookrightarrow Q, d)$ with $Q \not\models c$ by false. Since $\forall(e\colon P' \hookrightarrow Q, d) = \neg\exists(e\colon P' \hookrightarrow Q, \neg d)$, conditions of the form $\forall(e\colon P' \hookrightarrow Q, d)$ can be replaced by true if $Q \not\models c$. In our running example, this simplification drastically reduces the number and complexity of derived impairment- and repair-indicating application conditions.

Lemma 5.14 (Simplification of graph conditions using hard constraints)**.** *Let a constraint* $c = \forall(e\colon \emptyset \hookrightarrow P, \mathit{false})$ *and a condition* $c' = \exists(e'\colon P' \hookrightarrow Q', d)$ *with* $Q' \not\models c$ *be given, then* $G \models c \implies p \not\models c'$*, for each morphism* $p\colon P' \hookrightarrow G$*.*

Example 5.15. If we consider the induced pre-condition of moveAttribute w.r.t. $[o_3]$ shown in Figure 13, the second graph does not satisfy the hard constraint h_2, because the attribute a is contained in two classes. Therefore, we can replace this graph with false and the induced pre-condition contains only the existentially bound graph on the left of Figure 13.

The second simplification of conditions is based on the propositional logical equivalence that $(a \vee b) \implies (a \vee c)$ is equivalent to $b \implies (a \vee c)$. This allows conditions to be removed

if they occur in both parts of the implication. This again reduces the number of graphs contained in the application condition.

Lemma 5.16 (Simplification of graph conditions using propositional logic)**.** *Given a condition* $c = (\bigvee_{i\in\mathcal{I}} c_i) \implies (\bigvee_{j\in\mathcal{J}} d_j)$*, with index sets* $\mathcal{I}$ *and* $\mathcal{J}$*, then this condition is equivalent to the condition* $c' = (\bigvee_{i\in\mathcal{I}\setminus\{i_0\}} c_i) \implies (\bigvee_{j\in\mathcal{J}} d_j)$ *for some* $i_0 \in \mathcal{I}$ *if there is a* $j_0 \in \mathcal{J}$ *with* $c_{i_0} \equiv d_{j_0}$*.*

Example 5.17. Again, consider the induced pre- and post-conditions of moveAttribute w.r.t. $[o_3]$ as described in Example 5.13. As described in Example 5.15, the pre-condition can be simplified so that it contains only the left graph shown in Figure 13. This means that the condition $\text{Post}_{\text{moveAttribute}} \implies \text{Pre}_{\text{moveAttribute}}$ is of the form $(\exists Q_1 \vee \exists Q_2) \implies \exists Q_1$, where Q_1 is the graph on the bottom left and Q_2 is the graph on the bottom right of Figure 13, respectively. Using Lemma 5.16, we can remove the occurrence of $\exists Q_1$ from the left side of the implication, since it also appears on the right side of the implication. This simplification leads to the equivalent condition $\exists Q_2 \implies \exists Q_1$, i.e., $\text{Post}_{\text{moveAttribute}} \implies \text{Pre}_{\text{moveAttribute}} \equiv \exists Q_2 \implies \exists Q_1$.

Now we are ready to present the construction of *repair-indicating application conditions*. To detect all situations in which a violation of a constraint $c = \forall(P, d)$ is repaired (i.e., to detect each occurrence of the premise), we need to consider each overlap of the LHS of the rule ρ and the premise of the constraint, such that this overlap has induced transformations (i.e., we need to consider each overlap $o = (i_L, i_P, PL)$ contained in $\text{OL}(\rho, P)$). In particular, Lemma 5.6 implies that it is sufficient to consider only the equivalence classes contained in $\text{OL}_\sim(\rho, P)$. To find repairs of the conclusion w.r.t. a transformation, we consider overlaps whose induced transformations are parallel independent (i.e., the occurrence of the premise is preserved by the transformation). We first need to check whether the occurrence of the premise satisfies the conclusion of the transformation in the resulting graph of the transformation using the overlap induced post-condition. In addition, if this occurrence of the premise does not satisfy the conclusion in the original graph of the transformation, then this occurrence is a repair of the conclusion. I.e., if an occurrence of the overlap does not satisfy the condition $\text{Post}(o, d) \implies \text{Pre}(o, d)$, we have found a repair of the conclusion (as then it satisfies the conclusion in the resulting but not in the original graph of the transformation). *This means, that each violation of the application condition* $\forall(PL, \text{Post}(o, d) \implies \text{Pre}(o, d))$ *represents a repair of the conclusion.* To find repairs of the premise w.r.t. a transformation, we consider overlaps whose induced transformations are parallel dependent (i.e., the occurrence of the premise is destroyed by the transformation). Since the overlap-induced post-condition of that overlap is equal to true and for propositional logic formula it holds that $\text{true} \implies a \equiv a$, we have found a repair of the premise if the overlap does not satisfy the condition $\text{Post}(o, d) \implies \text{Pre}(o, d)$, i.e., it does not satisfy the conclusion in the original graph of the transformation. Here, *each violation of the application condition* $\forall(PL, \text{Post}(o, d) \implies \text{Pre}(o, d))$ *represents a repair of the conclusion.* Therefore, we can compute repair-indicating application conditions for both repairs of the conclusion and repairs of the premise via the following construction.

Definition 5.18 (Repair-indicating application condition)**.** Given a rule ρ and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, the set of *repair-indicating application conditions for* ρ *w.r.t.* c, denoted

by $\mathrm{Rep}(\rho, c)$, is defined as

$$\mathrm{Rep}(\rho, c) := \{\forall(i_L \colon L \hookrightarrow PL, \mathrm{Post}_\rho(o, d) \implies \mathrm{Pre}_\rho(o, d)) \mid [o] = [(i_L, i_P, PL)] \in \mathrm{OL}_\sim(\rho, P)\}.$$

For the special case, if $c = \forall(e \colon \emptyset \hookrightarrow P, \mathsf{false})$ (a graph G satisfies this constraint if P is not a subgraph of G, i.e., there is no morphism $p \colon P \hookrightarrow G$), the constraint can only be repaired (impaired) by repairs (impairments) of the premise. For an overlap $[o] = [(i_L, i_P, PL)] \in \mathrm{OL}_{\mathrm{dep}}(\rho, P)$, i.e., if the induced transformations are parallel dependent and application conditions constructed over this overlap will detect repairs of the premise, we have $\mathrm{Post}_\rho(o, d) = \mathsf{true}$ and, by using the definition of the Shift operator, $\mathrm{Pre}_\rho(o, d) = \mathsf{false}$. This results in the application condition $\forall(i_L \colon L \hookrightarrow PL, \mathsf{false})$, which detects each destroyed occurrence of the premise as a repair of the constraint. For an overlap $[o] = [(i_L, i_P, PL)] \in \mathrm{OL}_{\mathrm{ind}}(\rho, P)$, i.e., if the induced transformations are parallel independent and application conditions constructed over this overlap will detect repairs of the conclusion, we have $\mathrm{Post}_\rho(o, d) = \mathrm{Pre}_\rho(o, d) = \mathsf{false}$ and we obtain the application condition $\forall(i_L \colon L \hookrightarrow PL, \mathsf{true})$ which is always satisfied and reflects the fact that there can be no repair of the conclusion for a constraint of the form $\forall(P, \mathsf{false})$.

Example 5.19. Figure 3 shows the repair-indicating application conditions for the rules moveAttribute and moveMethod w.r.t. the constraints w_1 and w_2. The application conditions w.r.t. w_1 are constructed for the overlap classes $[o_3]$ and $[o_1']$ shown in Figure 9 and Figure 11. The application conditions constructed with the other equivalence classes can be simplified using Lemma 5.14 and Lemma 5.16, so that it is easy to see that they are equivalent to true and can therefore be disregarded. Note also that the conditions contained in the sets $\mathrm{Rep}(\mathsf{moveAttribute}, \mathsf{w}_2)$, $\mathrm{Rep}(\mathsf{moveMethod}, \mathsf{w}_2)$, and $\mathrm{Rep}(\mathsf{moveMethod}, \mathsf{w}_1)$ will detect repairs of the premise, while the application conditions contained in $\mathrm{Rep}(\mathsf{moveAttribute}, \mathsf{w}_1)$ detect repairs of the conclusion.

As shown in Lemma 5.9, a morphism is a repaired morphism w.r.t. some transformation t if and only if it is an impaired transformation w.r.t. the inverse transformation t^{-1}, i.e., we can use the same construction to compute impair-indicating application conditions by switching the role of the LHS and the RHS of the rule. So the impair-indicating application conditions can be obtained by computing the repair-indicating application conditions of the inverse rules and shifting them to the LHS.

Definition 5.20 (Impairment-indicating application condition)**.** Given a rule ρ and a constraint $c = \forall(e \colon \emptyset \hookrightarrow P, d)$, the set of *impairment-indicating applications for ρ w.r.t. c*, denoted by $\mathrm{Imp}(\rho, c)$, is defined as

$$\mathrm{Imp}(\rho, c) := \{\mathrm{Left}(\rho, ac) \mid ac \in \mathrm{Rep}(\rho^{-1}, c)\}.$$

Example 5.21. Figure 14 shows the impairment-indicating application conditions for the rules moveAttribute and moveMethod w.r.t. the constraints w_1 and w_2. The conditions contained in the sets $\mathrm{Imp}(\mathsf{moveAttribute}, \mathsf{w}_2)$, $\mathrm{Imp}(\mathsf{moveMethod}, \mathsf{w}_2)$, and $\mathrm{Imp}(\mathsf{moveMethod}, \mathsf{w}_1)$ will detect impairments of the premise, while the application conditions contained in $\mathrm{Imp}(\mathsf{moveAttribute}, \mathsf{w}_1)$ will detect impairments of the conclusion. Note that the right application conditions in $\mathrm{Imp}(\mathsf{moveMethod}, \mathsf{w}_2)$ and $\mathrm{Imp}(\mathsf{moveAttribute}, \mathsf{w}_2)$ are also contained in the sets $\mathrm{Rep}(\mathsf{moveMethod}, \mathsf{w}_2)$ and $\mathrm{Rep}(\mathsf{moveAttribute}, \mathsf{w}_2)$, respectively. These four application conditions can be removed as their results will cancel each other out when

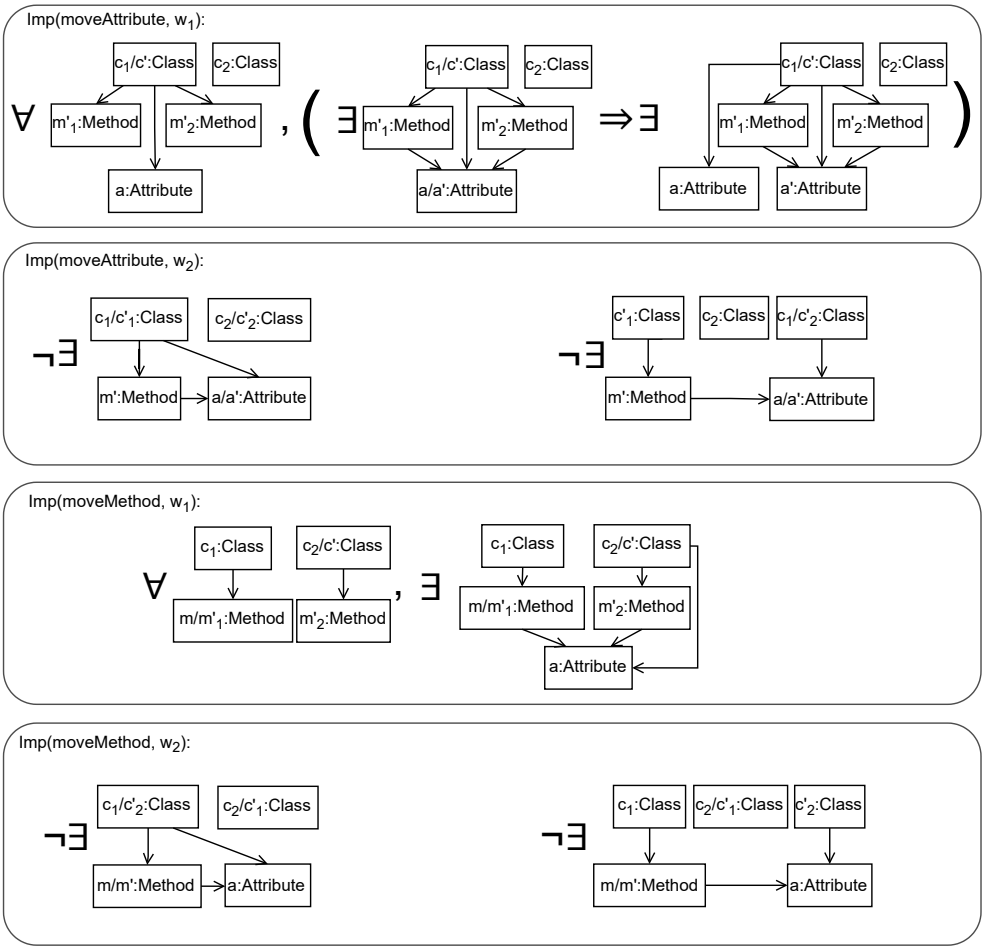

Figure 14: Impairment-indicating application conditions for the rules moveAttribute and moveMethod w.r.t. the constraints w_1 and w_2

evaluating the consistency gain, since each impairment detected by the right application condition in Imp(moveMethod, w_2) will also be detected as a repair by the right application condition in Rep(moveMethod, w_2). Therefore, these application conditions cannot influence the consistency gain, either positively or negatively.

Using the repair-indicating and impairment-indicating application conditions, we can now predict the change of consistency caused by a transformation. To evaluate the number of impairments, we consider the number of violations of the impairment-indicating application conditions, i.e., each violation of an impairment-indicating application condition corresponds to an impairment of the constraint. The total sum of these violations is the total number of impairments introduced by the transformation. For the number of repairs, we consider the number of violations of the repair-indicating application conditions, i.e., each violation of a repair-indicating application condition corresponds to a repair of the constraint. Again, the total sum of these violations is the total number of repairs introduced by the transformation.

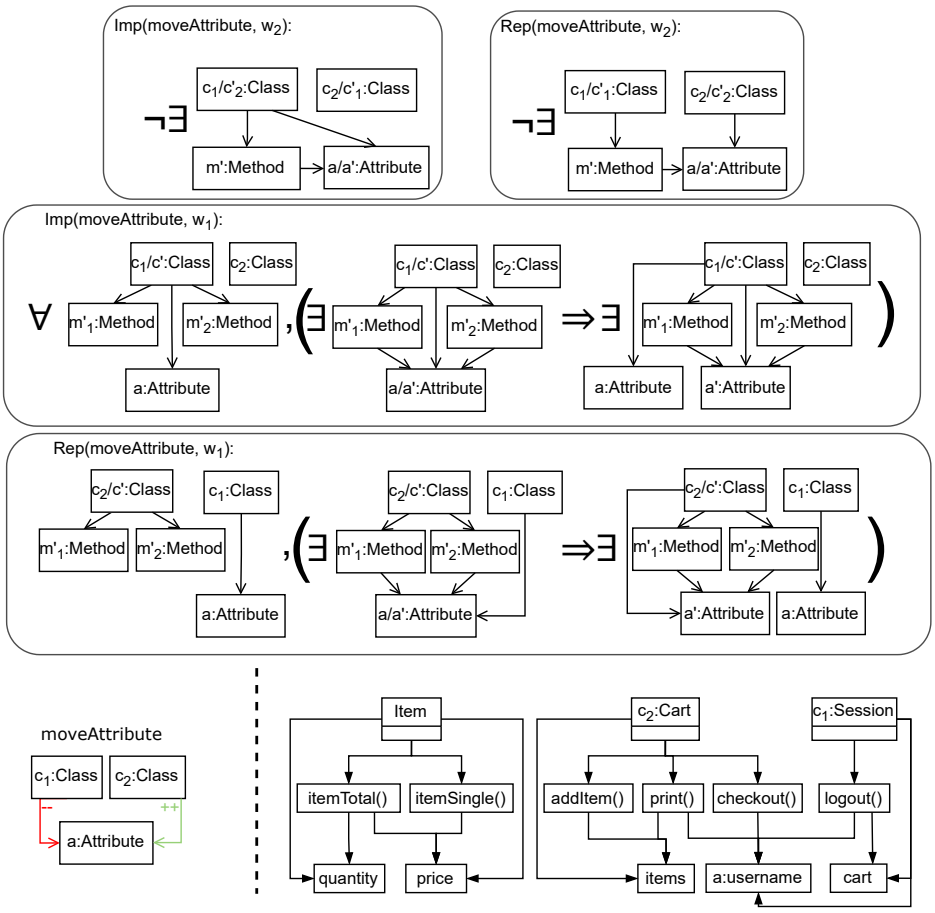

Figure 15: Repair- and impair-indicating application conditions for moveAttribute and w_2, moveAttribute and w_1, and an application of moveAttribute to move username from Session to Cart

Since each application condition is formulated over the LHS of the rule, this gives a look-ahead on the exact gain without actually performing the transformation and reevaluating the number of violations in the resulting graph.

Theorem 5.22. *Given a transformation* $t\colon G \Longrightarrow_{\rho,m} H$ *via a rule* $\rho = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ *and a constraint* c*, then*

$$\mathrm{nv}_H(c) - \mathrm{nv}_G(c) = \sum_{ac \in \mathrm{Imp}(\rho,c)} \mathrm{nv}_m(ac) - \sum_{ac \in \mathrm{Rep}(\rho,c)} \mathrm{nv}_m(ac).$$

Example 5.23. Let us consider the application of moveAttribute to move username from Session to Cart. The rule, match, and application conditions used throughout this example are shown in Figure 15. When evaluating the application condition contained in $\mathrm{Rep}(\mathsf{moveAttribute}, w_1)$ (which signals repairs of w_1), we first check whether the match can be extended by the first graph of the application condition. There are six such extensions in total:

(1) m'_1 is mapped to addItem() and m'_2 is mapped to print(),
(2) m'_1 is mapped to print() and m'_2 is mapped to addItem(),
(3) m'_1 is mapped to addItem() and m'_2 is mapped to checkout(),
(4) m'_1 is mapped to checkout() and m'_2 is mapped to addItem(),
(5) m'_1 is mapped to checkout() and m'_2 is mapped to print(),
(6) m'_1 is mapped to print() and m'_2 is mapped to checkout().

In the next step, we need to check whether these extensions can be further extended by the left graph of the implication, but not by the right graph of the implication. If so, we have found a repair of w_1. For the left part of the implication, we need to check if both methods of the extension are connected to the attribute username. This only applies to extensions 5 and 6 in the list above since both methods print() and checkout() depend on username. Extensions 1–4 do not need to be considered anymore.

In the last step, we need to check if extensions 5 and 6 can be further extended by the right graph of the implication, i.e., we need to check if there is an additional attribute contained in Cart that checkout() and print() are connected to. Since checkout() is not connected to any of the attributes contained in Cart, there is no such extension for 5 and 6. This means that 5 and 6 do not satisfy the implication and the application of moveAttribute will repair two violations of the constraint w_1, i.e., $\sum_{ac\in\mathrm{Rep}(\rho,\mathsf{w}_1)} \mathrm{nv}_m(ac) = 2$. When considering the impair-indicating application condition contained in $\mathrm{Imp}(\mathsf{moveAttribute}, \mathsf{w}_1)$, the match cannot be extended by the first graph of the application condition, because Session only contains one method, i.e., the application condition is satisfied and $\sum_{ac\in\mathrm{Imp}(\rho,\mathsf{w}_1)} \mathrm{nv}_m(ac) = 0$. In total, we get a gain of $\sum_{ac\in\mathrm{Imp}(\rho,\mathsf{w}_1)} \mathrm{nv}_m(ac) - \sum_{ac\in\mathrm{Rep}(\rho,\mathsf{w}_1)} \mathrm{nv}_m(ac) = \mathrm{nv}_H(\mathsf{w}_1) - \mathrm{nv}_G(\mathsf{w}_1) = 0 - 2 = -2$.

Let us now consider the application conditions for moveAttribute and w_2. For the application condition contained in $\mathrm{Rep}(\mathsf{moveAttribute}, \mathsf{w}_2)$, we need to check whether the match can by extended by the first graph of the application condition. There are two such extensions:

(1) m' is mapped to print() and
(2) m' is mapped to checkout().

Both occurrences of the pattern in the application condition signal a repair of w_2, i.e., $\sum_{ac\in\mathrm{Rep}(\rho,\mathsf{w}_2)} \mathrm{nv}_m(ac) = 2$.

For the impairment-indicating application condition $\mathrm{Imp}(\mathsf{moveAttribute}, \mathsf{w}_2)$, the rule match can only be extended by mapping m' to logout(). This occurrence signals an impairment of w_2, i.e., $\sum_{ac\in\mathrm{Imp}(\rho,\mathsf{w}_2)} \mathrm{nv}_m(ac) = 1$. In total, we get an overall gain of $\sum_{ac\in\mathrm{Imp}(\rho,\mathsf{w}_2)} \mathrm{nv}_m(ac) - \sum_{ac\in\mathrm{Rep}(\rho,\mathsf{w}_2)} \mathrm{nv}_m(ac) = \mathrm{nv}_H(\mathsf{w}_2) - \mathrm{nv}_G(\mathsf{w}_2) = 1 - 2 = -1$ for w_2.

In addition, we can predict the overall gain in consistency of a set of constraints and assign weights to each constraint to prioritise certain constraints over others if desired.

Corollary 5.24. *Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$ via some rule ρ at match m, a finite set of constraints $\{c_1, \ldots, c_n\}$, and some weights $(w_1, \ldots, w_n) \in \mathbb{R}^n$, then*

$$\sum_{i=1}^{n} w_i\big(\,\mathrm{nv}_H(c_i) - \mathrm{nv}_G(c_i)\big) = \sum_{i=1}^{n} w_i\Big(\sum_{ac\in\mathrm{Imp}(\rho,c_i)} \mathrm{nv}_m(ac) - \sum_{ac\in\mathrm{Rep}(\rho,c_i)} \mathrm{nv}_m(ac)\Big).$$

Example 5.25. Consider the set of constraints $\{\mathsf{w}_1, \mathsf{w}_2\}$, the weights $w = (1, 1) \in \mathbb{R}^2$ and a transformation via moveAttribute to move username from Session to Cart (see Figure 15). As discussed in Example 5.23, we have $\mathrm{nv}_H(\mathsf{w}_1) - \mathrm{nv}_G(\mathsf{w}_1) = 2$ and $\mathrm{nv}_H(\mathsf{w}_2) - \mathrm{nv}_G(\mathsf{w}_2) = 1$.

This means that the overall consistency of this transformation is increased by 3 since H contains fewer violations than G:

$$\sum_{i=1}^{2} \mathrm{nv}_H(\mathsf{w}_i) - \mathrm{nv}_G(\mathsf{w}_i) = (-2) + (-1) = -3$$

5.4. **Comparison to Consistency Sustaining and Improving Transformations.** To conclude the theoretical part of this paper, we want to compare the newly introduced application conditions with the notions of consistency-sustaining and -improving transformations introduced in [KSTZ22]. As described in Section 4, the number of violations is an extension of the corresponding notion presented in [KSTZ22]. Next, we will recall the notions of consistency-sustaining and -improving transformations and compare them with our approach. We will see that a transformation is consistency-sustaining if the gain in consistency is not negative, i.e., if the consistency does not decrease during a transformation. It is consistency-improving if the gain is positive, i.e., if the consistency increases during a transformation. *Please note that for this comparison, we are assuming that the weight of each constraint is equal to 1.*

Definition 5.26 (Consistency sustaining and improving transformation [KSTZ22])**.** Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$ and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, the transformation t is *consistency sustaining w.r.t. c* if

$$\mathrm{nv}_H(c) \leq \mathrm{nv}_G(c).$$

The transformation t is called *consistency improving w.r.t. c* if

$$\mathrm{nv}_H(c) < \mathrm{nv}_G(c).$$

The result of Theorem 5.22 implies that we can use the application conditions to predict whether a transformation is consistency-sustaining (-improving) or not.

Corollary 5.27. *Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$ and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, the transformation t is consistency sustaining w.r.t. c if and only if*

$$\sum_{ac \in \mathrm{Imp}(\rho,c)} \mathrm{nv}_m(ac) - \sum_{ac \in \mathrm{Rep}(\rho,c)} \mathrm{nv}_m(ac) \leq 0.$$

The transformation t is consistency-improving w.r.t. c if

$$\sum_{ac \in \mathrm{Imp}(\rho,c)} \mathrm{nv}_m(ac) - \sum_{ac \in \mathrm{Rep}(\rho,c)} \mathrm{nv}_m(ac) < 0.$$

Kosiol et al. have also introduced a stricter variant of consistency-sustaining and -improving transformations called *directly consistency-sustaining and (-improving) transformations*. A consistency-sustaining (-improving) transformation is directly consistency-sustaining (-improving) if the transformation does not introduce any violations, i.e., there is no impaired morphism w.r.t. the transformation. Thus, for a transformation to be directly consistency-sustaining it is sufficient that no new violation is introduced.

Definition 5.28 (Directly consistency-sustaining transformation [KSTZ22])**.** Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a transformation $t\colon G \Longrightarrow_{\rho,m} H$ is *directly consistency-sustaining*

w.r.t. c if

$$\forall p\colon P \hookrightarrow G((p \models d \wedge \mathrm{tr}_t \circ p \text{ is total}) \implies \mathrm{tr}_t \circ p \models d) \text{ and}$$
$$\forall p'\colon P \hookrightarrow H(\neg\exists p\colon P \hookrightarrow G(p' = \mathrm{tr}_t \circ p) \implies p' \models d).$$

Note that any morphism $\mathrm{tr}_t \circ p$, where p does not satisfy the first part of the condition, is an impairment of the conclusion according to Definition 5.8, and any morphism p' not satisfying the second part of the condition is an impairment of the premise. For a directly consistency-improving transformation, it is sufficient that at least one violation of the constraint is repaired, i.e., there is at least one repaired morphism.

Definition 5.29 (Directly consistency-improving transformation [KSTZ22])**.** Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a transformation $t\colon G \Longrightarrow_{\rho,m} H$ is *directly consistency-improving* w.r.t. c if t is direct consistency-sustaining w.r.t. c and

$$\exists p\colon P \hookrightarrow G(p \not\models d \wedge p' := \mathrm{tr}_t \circ p \text{ is total } \wedge p \models d) \text{ or}$$
$$\exists p\colon P \hookrightarrow G(p \not\models d \wedge p' := \mathrm{tr}_t \circ p \text{ is not total}).$$

Again, any morphism p that does not satisfy the condition is a repaired morphism according to Definition 5.7.

In [KSTZ22], application conditions were presented that are only sufficient, i.e., in some cases, transformations are prevented even if the transformation would be consistency-sustaining (consistency-improving). Our construction of application conditions in Section 5.3 computes application conditions that prevent transformations if and only if the transformation would not be directly consistency-sustaining (-improving). In particular, this implies that they are the weakest directly consistency-sustaining (-improving) application conditions.

Theorem 5.30. *Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a transformation $t\colon G \Longrightarrow_{\rho,m} H$ is directly consistency-sustaining w.r.t. c if and only if*

$$m \models \bigwedge_{ac \in \mathrm{Imp}(\rho,c)} ac.$$

The transformation t is directly consistency-improving w.r.t. c if and only if

$$m \models \bigwedge_{ac \in \mathrm{Imp}(\rho,c)} ac \wedge \neg\Big(\bigwedge_{ac \in \mathrm{Rep}(\rho,c)} ac\Big).$$

6. Evaluation

To evaluate the practical relevance of our approach, we implemented a greedy graph optimization algorithm for a variant of the CRA case study (cf. Section 2). This was done using the graph transformation tool eMoflon[3], which incrementally computes all matches to a given graph for rules and their application conditions. This is a prerequisite for the efficient implementation of this approach, as calculating all matches from scratch after each rule application can easily become a serious performance bottleneck. Based on the matches provided by eMoflon, our implementation then counts violations of derived application conditions and ranks rule matches w.r.t. the number of constraint violations that are removed or added by the application of the considered rule at the considered match (see Theorem 5.22). Then, the rule application with the highest rank is greedily selected and

[3] https://www.emoflon.org

applied, and the ranking of rule matches is updated. Note that the application conditions are currently not automatically derived, but were designed and implemented by hand in eMoflon based on our formal construction.

Our evaluation was run on a Ryzen 7 3900x with 64 GB RAM on Windows 11 23H2. It is available as a virtual machine[4] with a detailed description of how to reproduce our results. In the following, we first investigate the scalability and effectiveness of our approach. To assess the effectiveness of our approach, we compare it to an ILP-based approach that can find the global optimum of our test scenario[5].

6.1. **Scalability and effectiveness.** Finding all the matches for rules and application conditions can become very expensive if there are many matches to find. The CRA case study is particularly challenging because a feature can be moved from one class to any other class, which means that the number of refactoring steps, and thus the number of possible matches, grows rapidly with an increasing number of classes and features. Therefore, we pose the following research questions: *(RQ1) How does our approach scale with respect to the size of a processed class diagram (graph)?* and *(RQ2) Can our approach reduce the number of violations?*

To answer the first question, we need to examine the two phases of our approach, which are related to how eMoflon works and incrementally provide us with collected matches. First, we measure the time it takes to compute the initial collection of all rule matches and application condition occurrences. Then, we use the application condition occurrences to rank the rule matches. Depending on the size of the model, the initialisation is expected to take longer than applying a rule, incrementally updating eMoflon's internal structures, and ranking the rule matches. Second, we measure the time taken to perform 10 repair steps, where we have to judge which repair to apply next, based on an actually selected repair step. For this, we use the most promising (i.e., highest ranked) rule application, where each repair and each impairment is uniformly weighted with a value of 1.

To investigate the scalability of our approach, we created synthetic class diagrams of varying sizes with increasing numbers of classes, where each class has five methods and five attributes. Each method has two dependencies on attributes of the same class and another three dependencies on attributes of other classes. Having more dependencies means that it is less likely to move features from one class to another, because more features would form a dependency clique within a class.

Figure 16 shows our evaluation results, where the left plot shows the time in seconds for processing models with up to 2,751 nodes. Starting with 276 nodes, the initialisation time is 2.2 s, and performing 10 refactoring steps takes 0.5 s. With 2,751 nodes, this time increases to 65 s for the initialisation and 2.8 s for the 10 steps. Obviously, the initialisation takes 30 times longer for a 10 times larger model, while the incremental updates scale better and take only 5 times longer. The reason for this non-linear increase lies in the structure of our rules, where the number of matches increases rapidly with each new class. For a model with 276 nodes, we collected 600 rule matches and 36,000 application condition matches, while for the largest model with 2,751 nodes, we found 622,500 rule matches and 3,7 million condition matches. So, while it takes 30 times as long to run a model 10 times the size, we found 118

[4]https://www.zenodo.org/records/10727438

[5]This comparison is an extension of our earlier work.

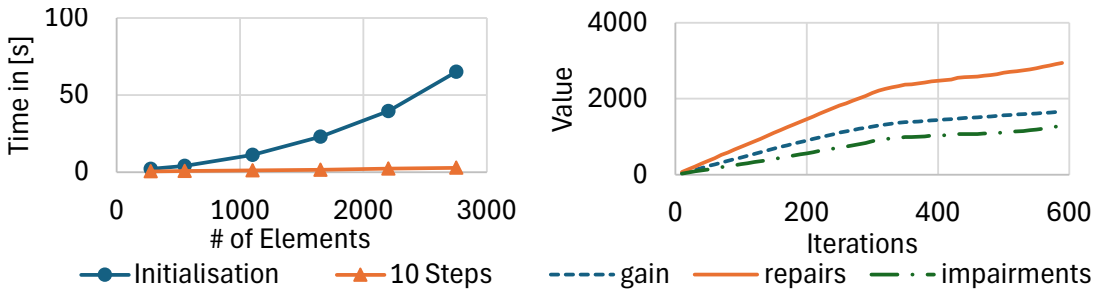

Figure 16: Performance measurements for models with increasing size (left) and consistency improvement of a model with 2201 nodes (right)

times as many matches. This shows that even for a challenging scenario like the CRA use case, our approach scales reasonably well given the number of repair steps available (RQ1).

To answer RQ2, we took a model with 2,201 nodes and measured the aggregated number of impairments and repairs after n iterations (i.e., repair steps) along with their gains in consistency. As before, in each iteration, we chose the rule application with the highest gain and continued until there was no rule application left with a positive gain, i.e., the application of a rule at that point would have no effect or cause more impairments than repairs. The results are shown in Figure 16 on the right. After 589 iterations with a total gain of 1,661, the process terminated with a (local) optimum. Thus, the resulting model contains 1,661 fewer violations than before, leading to a first answer to RQ2.

As shown, our approach can incrementally maintain the necessary ranking information for rule-matches (RQ1) and improve consistency in a relatively small number of iterations (RQ2). This is particularly interesting considering the fact that the search space of all rule matches can grow very rapidly, which is a particularly challenging scenario for our approach.

6.1.1. *Threats to validity.* Currently, we only investigate the CRA case using synthetic class diagrams. To evaluate the general scalability of our approach, more scenarios should be investigated, preferably using real-world data, e.g., extracted from public code repositories. In addition, the application conditions are currently constructed by hand (following our formal construction process), which can be a source of error. In addition, since we only have a look-ahead of 1, there may not always be a good next step to improve consistency, even though a better overall solution exists. Therefore, future work should investigate how our approach performs when the greedy strategy is replaced by another strategy, such as simulated annealing.

6.2. **Comparison with an ILP-based approach.** In this subsection, we want to further investigate our approach and compare it with another solution that calculates an optimal solution using the **G**raph-Based (M)**I**LP **P**roblem **S**pecification (GIPS) tool [EKS22]. GIPS is a framework that combines Graph Transformation (GT) and ILP to define optimal model transformations. We, thereby, want to show that our approach can indeed find the global optimum (at least for smaller models), with the benefit of scaling better for larger models than GIPS.

As a basis for our comparison, we use the test data from the TTC16 [FTW16] CRA case study, for which we have extended our example to also support dependencies between methods. Note that these dependencies were omitted before (cf. Section 2). Another difference between our previous variant and the original CRA case study is that the latter consists only of features that need to be clustered into (newly created) classes. Previously, we assumed a predefined class diagram, in which each feature was already assigned to a class. Thus, the case study represents a more challenging scenario in which we now have to find out how many classes are needed to maximise the ratio between cohesion and coupling. We bridge this gap similar to the CRA solution of Georg Hinkel [Hin16] by creating a new class for each feature and then applying either our approach or GIPS to the resulting model. However, we do not compare our approach to any of the original CRA competition entries because we are not actually implementing the CRA index (the metric that was used in the TTC16), but rather a custom metric that is inspired by it. To understand why, let us first introduce the metric itself.

Our custom metric is the sum of violations of the following two constraints: First, features within the same class must have a dependency between each other, and, second, features within different classes must not have a dependency between each other. The definition of a dependency is the same as in the TTC16 [FTW16] CRA case study. Thus, the resulting value of our metric represents the total number of violations of these two constraints within the system. In contrast, the CRA index is the difference between coupling and cohesion, where both are determined by the number of dependencies divided by the number of all possible dependencies. While the results of the TTC16 show that improving the CRA index does indeed lead to class diagrams with improved coupling and cohesion, it is difficult to determine the coupling's and cohesion's current state from the CRA index alone, which is (usually) not an integer. In general, our approach can only be combined with the CRA index under severe drawbacks. Using our custom metric, an impairment/repair will always have a constant value, e.g., -1 for an impairment and $+1$ for a repair. While performing a refactoring step may inflict new impairments or lead to some repairs, we can track and maintain these changes incrementally to find the next best refactoring step in an efficient way. The CRA index, however, does not come with such a constant value for each impairment/repair, and depending on the previous former actions, an impairment/repair may have a different value due to the CRA index' nature of incorporating all dependencies in a class diagram. This means that, after each refactoring step, we would need to reevaluate all other refactoring steps anew to determine their actual effect on the CRA index. As this does not align with our concept that leverages incremental techniques, we chose this custom metric, which was designed to mimic the CRA index to some degree. However, there are also technical reasons related to GIPS for not using the CRA index, which will be discussed at the end of this chapter.

In the following subsections, we will discuss the modifications to support the TTC16 test data and briefly present the implementation based on GIPS. We then compare the performance of the two approaches in terms of runtime and quality, and discuss how our custom metric relates to the CRA index.

6.2.1. *Reducing violations with application conditions.* Compared to our evaluation in Section 6.1, we have changed the constraints and thus the application conditions derived from them. In particular, we use the refactoring rule moveMethod, assume that the hard constraint h_1 is always satisfied, and consider the weak constraints w_3 and w_4 shown in Figure 17.

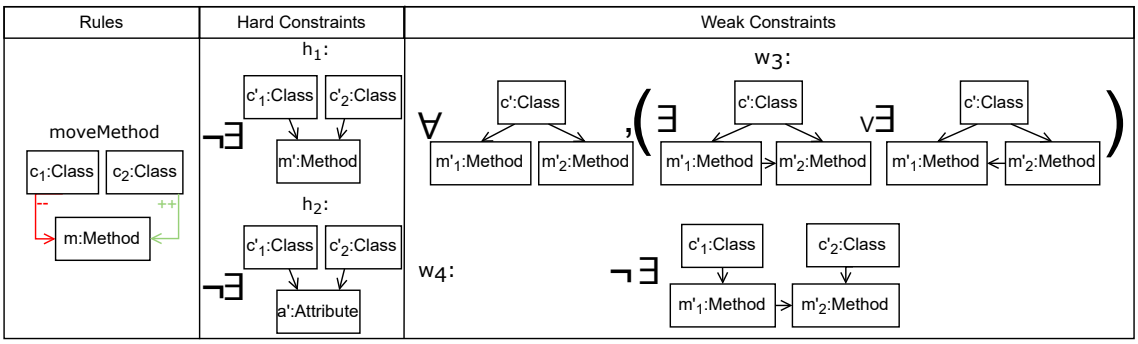

Figure 17: Rules and constraints for the extended evaluation

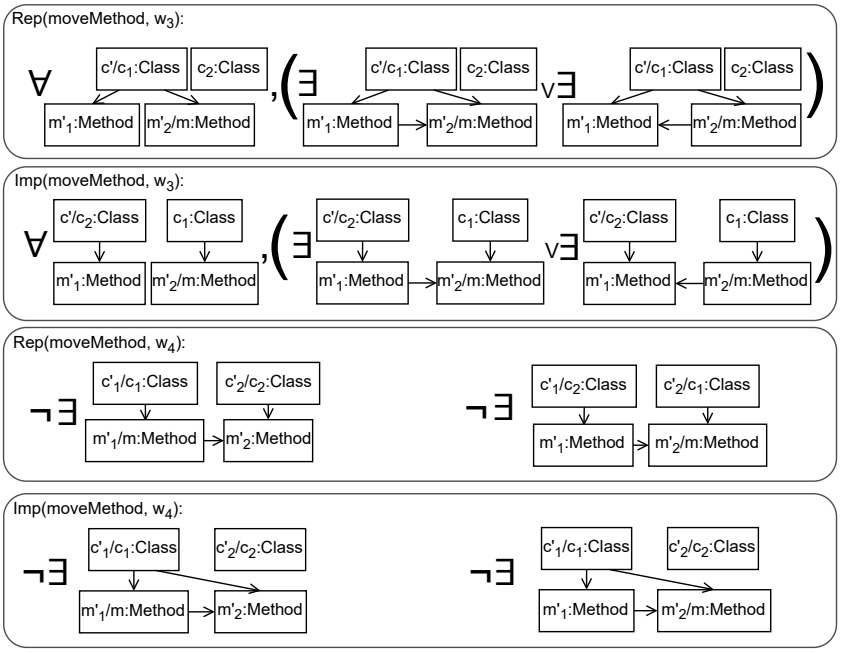

Figure 18: Repair- and impairment-indicating application conditions for the rule moveMethod w.r.t. the constraints w_3 and w_4

These constraints model good code patterns that should be satisfied in any well-designed class diagram. Constraint w_3 requires that for any pair of methods contained in the same class, one method uses the other, or vice versa; w_4 requires that any pair of methods so that one of the methods uses the other, both are contained in the same class. In total, we have four application conditions for constraint repair and another four for constraint impairment (cf. Figure 3, Figure 14, and Figure 18).

The reason for this is that dependencies between two methods have a different edge type than those between a method and an attribute.

Another change to the experiments in Section 6.1 is that we run the experiments multiple times, and additionally allow a certain amount of refactorings at the beginning of each run,

to impair more than to actually repair. In this way, we try to find different (local) optima and pick the best one.

Since there are no classes at the beginning, we create one for each feature. We then start refactoring the model step by step, moving methods and attributes from one class to another until we can no longer improve with the next step.

6.2.2. *Finding an optimal solution with GIPS.* To find an optimal solution to our example problem, we use the GIPS framework [EKS22]. GIPS combines the GT framework with ILP optimization techniques in order to optimize a given input (graph) model. Therefore, a custom Domain-Specific Language (DSL) called **G**raph-Based (M)**I**LP **P**roblem **S**pecification **L**anguage (GIPSL) has been developed that allows ILP constraints to be expressed in a UML/OCL-like fashion and which extends the eMoflon::IBeX-GT language that provides the ability to specify GT rules and graph patterns. In essence, the GIPSL specification contains a set of GT rules with corresponding ILP constraints and an objective, all of which refer to a given metamodel. Based on this specification, the GIPS framework generates an ILP problem generator that is able to translate any (graph) model that conforms to the metamodel into an ILP optimization problem. For a given input model, the ILP solver determines a subset of all GT rule matches for which the corresponding GT rules should be applied. The application of all selected GT rule matches then optimizes the model according to the objective goal defined in the GIPSL specification. Unlike using only GT, this approach provides the ability to specify global constraints on the model and GT rule applications. This allows for inter-rule constraints, for example, if *rule A* is executed on a specific match, *rule B* must also be executed on the same (or another) match.

For our solution to the CRA assignment problem, we generate a number of empty classes in the model to which the methods and attributes can be assigned. We then use GIPSL[6] to model the following behavior. For the assignment itself, our specification contains a set of GT rules that create an assignment edge between a method/attribute and a class. In addition, we need a set of constraints to ensure that a computed solution is valid. Therefore, our GIPSL specification contains two constraints that ensure that each method and attribute is assigned to exactly one class. For the optimization part, we count the number of CRA violations that a specific solution would produce if all corresponding GT rule matches were applied. This sum of violations forms our objective function which motivates the solver to choose the set of assignments that minimizes the number of violations globally. After the ILP solver has determined the optimal subset of GT rule matches to apply, we execute those rules and obtain the solution to the assignment problem.

6.2.3. *Comparison.* In the following, we will compare our approach with the GIPS solution based on up to six input models of varying complexity degree taken from the TTC16 CRA case study. These models consist only of features (half methods and half attributes) with predefined dependencies between them. From model A to F, they come with 9, 18, 35, 80, 160, and 400 features, respectively. As a first research question, we want to know *how our approach scales compared to another solution that solves the same problem using other optimization techniques* **(RQ1)**. Second, while GIPS will always find the global optimum as long as it terminates, this is not necessarily true for our approach. Therefore, we want to

[6]https://github.com/Echtzeitsysteme/gips-examples/blob/main/architecture.cra.gipssolution/src/architecture/cra/gipssolution/Model.gipsl

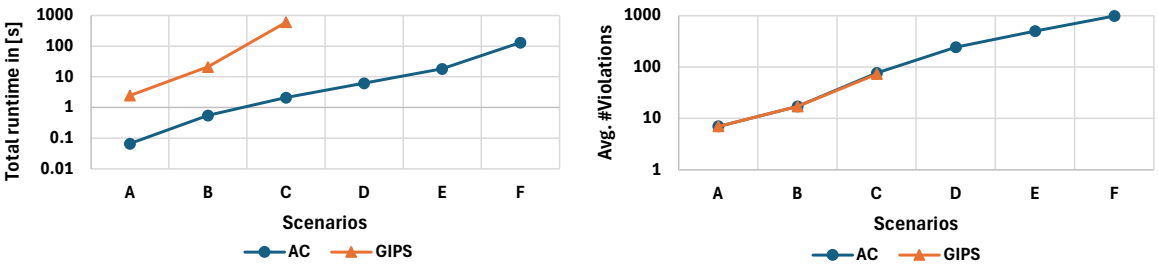

Figure 19: Average performance measurements for increasing model size (left) and average number of total violations (right)

investigate whether, *for the models for which we have a solution from GIPS, our approach is able to find the same solution* **(RQ2)** and *how confident we can be in finding this solution with each run* **(RQ3)**.

To answer these three questions, we evaluated the CRA scenarios on a workstation equipped with an AMD EPYC 7763 with 128 threads and 256 GB RAM using Debian 12, OpenJDK 21.0.4, and Gurobi Optimizer 11.0.3. For both approaches, we measured the total time to compute a solution by repeating each scenario 10 times with GIPS and 200 times with our approach (AC). Figure 19 shows the total time on the left, where we can immediately see that only the smaller three models could be processed with GIPS, after which the memory consumption exceeded 256 GB. The time increases from 2.4 s for A, to 21.3 s for B, and to 613.8 s for C, with a standard deviation of 0.04 s, 2.6 s, and 226.1 s, respectively. In comparison, using application conditions (AC), we see a less steep increase in the runtime, where it takes 132 s on average to find a solution with a standard deviation of less than 2.5 % for all scenarios. For GIPS, doubling the number of features increases the time by about 25-30 times, while using ACs triples/quadruples the time **(RQ1)**.

Figure 19 shows the results of AC and GIPS on the right, showing the average number of violations for each scenario. Obviously, GIPS always returns the optimal solution to the problem, which is 7 remaining violations for A, 17 violations for B, and 73 for C. Interestingly, using AC and picking the best solution from the 200 runs, we always got the globally optimal solution with the same number of remaining violations left **(RQ2)**. Moreover, the standard deviation for the average number of violations never exceeded 2.3 violations, which means that even for larger CRA models, we usually get solutions of a similar quality with respect to our custom metric **(RQ3)**. However, we do not know whether our solutions for D, E, and F are actually globally optimal, because GIPS could not compute this anymore due to the limited amount of memory of our system.

6.2.4. *Threats to validity.* In general, the same threats to validity apply as in Section 6.1.1. However, we changed the greedy algorithm of picking the most promising rule in each run with an alternative that can also pick rules that repair less than they impair at the beginning.

In addition, we want to discuss our reasons for not using the CRA index once more. While it is true that our custom metric is more in line with the idea of using application conditions, we want to emphasise that it would be difficult to actually express incremental changes in the CRA index using our approach. The CRA index is the sum of coupling and cohesion values that are expressed as the ratio of the number of dependencies of a class

divided by the number of all possible dependencies. This value, which is usually not an integer like our violation count, is harder to update incrementally for each refactoring step, and we would have to update it after each iteration for each available refactoring. Also, we cannot express the CRA index using GIPS because the ratio contains divisors that depend on which classes have been assigned what features. Since the divisor is not constant, but depends on the actual solution, it is not possible to express this as an ILP problem.

7. Related Work

The following discusses related work on graph constraint checking and different approaches to graph repair. It also covers ranking of model repair steps and solving the CRA problem.

Graph constraints and application conditions. Habel and Pennmann [HP09] introduced the original process for generating application conditions from nested graph constraints, which are consistency-guaranteeing, meaning that applying a rule with such conditions is guaranteed to produce a graph that is consistent with respect to the given (hard) constraints. In our paper, we extend the binary case of satisfying constraints by ranking rule applications based on how many constraint violations they add or remove w.r.t. a set of weak constraints.

Kosiol et al. [KSTZ22] also count violations but only consider one type of constraint (hard constraints). While they consider constraints in alternating quantifier normal form (i.e., constraints that do not use conjunction or disjunction), our approach supports arbitrary constraints. In [KSTZ22], application conditions are constructed to make rule applications consistency-sustaining, but there is no such construction for consistency-improving rule applications. Moreover, the application conditions are consistency-sustaining in the strict sense that no new constraint violations are allowed.

Since the resulting set of application conditions can be large and thus expensive to evaluate, Nassar et al. [NKAT20] showed that some subconditions can be filtered if they check for cases that cannot occur. We also filter the resulting application conditions, but based on a set of additional hard constraints, e.g., by filtering out conditions that check for features contained in multiple classes simultaneously.

In contrast to all the literature mentioned above, we construct application conditions to rank rule applications along their potential to improve consistency. This means that rule applications are not blocked, even if some application conditions are violated. This approach can be used effectively in a transformation engine like eMoflon, which supports the incremental computation of rule matches.

Incremental graph constraint checking. In [AVS+14], constraints are also checked by incremental graph pattern matching. This paper distinguishes between well-formedness and ill-formedness constraints, the former of which are desirable and the latter of which are to be avoided. Using genetic algorithms with a set of graph-modifying rules as mutators, a graph is searched for that maximises the number of occurrences of well-formedness constraints minus the number of occurrences of ill-formedness constraints. *Compared to our approach, the graph is changed to track consistency and detect violations and repairs, whereas we use a look-ahead to plan the next steps.*

Graph repair. Habel and Sandmann [HS18, SH19, San20, San21] propose a graph repair approach based on constraints in alternating quantifier normal form (i.e., constraints that do not use conjunction or disjunction). They showed that their graph repair algorithm produces a consistent graph for constraints with a nesting level less than or equal to 2, or that end with a condition of the form $\exists(P, \mathsf{true})$. The algorithm derives rules from the constraints and reuses existing graph elements wherever possible. For instance, a violation of w_1 (i.e., a pair of methods contained within the same class that do not depend on a common attribute within that class) would be repaired by creating a dependency between both methods and an arbitrary attribute contained within the same class. However, this approach is not suitable for applications similar to the CRA problem, since dependencies between features cannot be created or deleted without adapting the underlying system. *In contrast to our approach, theirs cannot work with arbitrary rules; instead, the provided rules must instead be equivalent to those derived from constraints.*

The graph repair approach presented in [LKT24] also supports constraints in alternating quantifier normal form, but it uses user-specified rules. Unlike the algorithm of Habel and Sandmann, this approach attempts to create missing structures rather than reusing already existing graph elements. For example, a violation of w_1 would be repaired by creating a new attribute contained in the class, so that both methods depend on it. To support multiple constraints, the authors introduce the concept of 'circular conflict-freeness', where the algorithm searches for an ordering of the considered constraints, $c_1, c_2, \ldots, c_n$, such that repairing constraint c_i does not impair any constraint c_j, with $j < i$. They showed that a graph can be repaired if such an ordering can be found. The algorithm then repairs the constraints in this order, eliminating the need for backtracking, i.e., without repairing constraints that have already been repaired. However, the existence of such an ordering depends heavily on the provided repair rules, which restricts the applicability of this approach. Such an ordering does not exist for the constraints w_1 and w_2 and the rules moveAttribute and moveMethod in our running example, since both rules can impair w_1 while repairing w_2 and vice versa. *Therefore, our approach is applicable to a wider range of scenarios, since it supports arbitrary constraints and repair rules.*

In [NKR17], the authors introduce a graph repair approach for multiplicities, which can be modelled as constraints of the form $\exists(P, \mathsf{true})$ and $\forall(P, \mathsf{false})$. *In contrast to our approach, which supports arbitrary conditions, the set of supported constraints is highly restricted. This makes it unsuitable for complex settings, such as the CRA problem.*

In [SLO21], the authors introduce a logic-based, incremental approach to graph repair that can be used to repair arbitrary nested graph constraints. Given an input model, the approach computes multiple least-changing graph repairs. These repairs are least-changing in the sense that the most recent transformation that introduced an impairment cannot be reverted. For example, if w_2 is impaired by moving a method that depends on an attribute contained in the same class to another class, this approach will not provide a repair that moves the method back into the original class. In general, the approach computes the minimal graphs that satisfy the constraint and contain the input graph, preserving the actions of the most recent transformation that has introduced an impairment. *Compared to our approach, there is no support for using user-specified transformation rules, which can be used to restrict the ways in which a graph can be modelled.* For example, applying the algorithm presented in [SLO21] to the CRA problem may result in graphs where violations of w_2 have been repaired by removing an attribute from its class (i.e., by deleting the edge from a class to an attribute) without assigning it to another class. While the result set

may contain a graph H that can be constructed by using the provided transformation rules, verifying the existence of a sequence of transformations $G \Rightarrow \ldots \Rightarrow H$ (where G is the graph to be repaired) by applying only the provided rules is difficult.

Ranking in model repair. A similar ranking approach for model repair is presented in [KKE19], where impairments are identified as negative side effects and repairs as positive side effects of model repair sequences. All constraints have equal priority. Given a model with a set of violations, all possible repairs are computed and ranked according to their side effects. A repair consists of a sequence of repair actions of limited length, each of which repairs only a single model element or a single property of an element. Rather than being determined in advance, the ranking is established by first executing all computed repair action sequences and then observing their effects on the number of violated constraints. *In contrast, our approach supports arbitrary repair rules and allows multiple repair actions to be performed in one transformation step. Furthermore, we determine the positive and negative effects of all options for applying a given repair rule to all (weak) constraints statically, i.e. without altering the (model) graph for this purpose.*

8. Conclusion

In this paper, we introduce a new dynamic analysis approach that ranks rule matches based on their potential for graph repair, which supports arbitrary nested graph constraints. This potential is computed using application conditions that are automatically derived from a set of nested graph constraints. While some of these conditions indicate repair steps, others detect impairments. We formally show that the gain in consistency of a rule application can indeed be characterized by the difference between the number of violations of repair-indicating and impairment-indicating application conditions. We illustrate and evaluate our approach in the context of the well-known CRA problem and show that even for a worst-case scenario, the performance scales reasonably well. An initial evaluation also shows that using our ranking approach within a greedy algorithm can yield well-acceptable optimization results. In the future, we want to fully automate the ranking of graph transformations based on an automated construction of repair-indicating and impairment-indicating application conditions. To this end, we plan to combine our approach with the work presented in [LKT24] and [LKT25], with the goal of optimising the construction and use of application conditions even further. In addition, we will investigate different strategies besides greedy-based algorithms for optimization of graphs in different scenarios. To further strengthen our ranking approach, which has a look-ahead of 1, it may be advantageous to combine several repair rules into a larger one by composing concurrent rules [EH83, EEPT06].

References

[AVS+14] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 289–300. ACM, 2014. doi:10.1145/2642937.2643005.

[BBL10] Michael Bowman, Lionel C Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering*, 36(6):817–837, 2010. doi:10.1109/TSE.2010.70.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/3-540-31188-2.

[EH83] Hartmut Ehrig and Annegret Habel. Concurrent transformations of graphs and relational structures. In Manfred Nagl and Jürgen Perl, editors, *Proceedings of the WG '83, International Workshop on Graphtheoretic Concepts in Computer Science*, pages 76–88. Universitätsverlag Rudolf Trauner, Linz, 1983.

[Ehr78] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (A survey). In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology, International Workshop, Bad Honnef, Germany, October 30 - November 3, 1978*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1978. doi:10.1007/BFB0025714.

[EKS22] Sebastian Ehmes, Maximilian Kratz, and Andy Schürr. Graph-based specification and automated construction of ilp problems. In Proceedings of the Thirteenth International Workshop on *Graph Computation Models,* Nantes, France, 6th July 2022, volume 374 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–22. Open Publishing Association, 2022. doi:10.4204/EPTCS.374.3.

[FLST24] Lars Fritsche, Alexander Lauer, Andy Schürr, and Gabriele Taentzer. Using application conditions to rank graph transformations for graph repair. In Russ Harmer and Jens Kosiol, editors, *Graph Transformation - 17th International Conference, ICGT 2024, Held as Part of STAF 2024, Enschede, The Netherlands, July 10-11, 2024, Proceedings*, volume 14774 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2024. doi:10.1007/978-3-031-64285-2_8.

[FTW16] Martin Fleck, Javier Troya, and Manuel Wimmer. The class responsibility assignment case. In Antonio García-Domínguez, Filip Krikava, and Louis M. Rose, editors, *Proceedings of the 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 8, 2016*, volume 1758 of *CEUR Workshop Proceedings*, pages 1–8. CEUR-WS.org, 2016. URL: https://ceur-ws.org/Vol-1758/paper1.pdf.

[Hin16] Georg Hinkel. An NMF solution to the class responsibility assignment case. In Antonio García-Domínguez, Filip Krikava, and Louis M. Rose, editors, *Proceedings of the 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 8, 2016*, volume 1758 of *CEUR Workshop Proceedings*, pages 15–20. CEUR-WS.org, 2016. URL: https://ceur-ws.org/Vol-1758/paper3.pdf.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009. doi:10.1017/S0960129508007202.

[HS18] Annegret Habel and Christian Sandmann. Graph repair by graph programs. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, volume 11176 of *Lecture Notes in Computer Science*, pages 431–446. Springer, 2018. doi:10.1007/978-3-030-04771-9_31.

[HT20] Reiko Heckel and Gabriele Taentzer. *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer, 2020. doi:10.1007/978-3-030-43916-3.

[KKE19] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. Detecting and exploring side effects when repairing model inconsistencies. In *Proceedings of the 12th ACM SIGPLAN international conference on software language engineering*, pages 113–126. ACM, 2019. doi:10.1145/3357766.3359546.

[KSTZ22] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. Sustaining and improving graduated graph consistency: A static analysis of graph transformations. *Science of Computer Programming*, 214:102729, 2022. doi:10.1016/J.SCICO.2021.102729.

[LKT24] Alexander Lauer, Jens Kosiol, and Gabriele Taentzer. Empowering model repair: a rule-based approach to graph repair without side effects - extended version. *Innovations in Systems and Software Engineering*, 20(4):597–618, 2024. doi:10.1007/S11334-024-00587-W.

[LKT25] Alexander Lauer, Jens Kosiol, and Gabriele Taentzer. Granular conflict analysis for transformation rules with application conditions. In Jörg Endrullis and Matthias Tichy, editors, *Graph Transformation - 18th International Conference, ICGT 2025, Held as Part of STAF 2025, Koblenz, Germany, June 11-12, 2025, Proceedings*, volume 15720 of *Lecture Notes in Computer Science*, pages 63–90. Springer, 2025. doi:10.1007/978-3-031-94706-3_4.

[MJ14] Hamid Masoud and Saeed Jalili. A clustering-based model for class responsibility assignment problem in object-oriented analysis. *Journal of Systems and Software*, 93:110–131, 2014. doi:10.1016/j.jss.2014.02.053.

[NKAT20] Nebras Nassar, Jens Kosiol, Thorsten Arendt, and Gabriele Taentzer. Constructing optimized constraint-preserving application conditions for model transformation rules. *Journal of Logical and Algebraic Methods in Programming*, 114:100564, 2020. doi:10.1016/J.JLAMP.2020.100564.

[NKR17] Nebras Nassar, Jens Kosiol, and Hendrik Radke. Rule-based repair of emf models: formalization and correctness proof. In *Electronic Pre-Proceedings International Workshop on Graph Computation Models*, 2017.

[Plu05] Detlef Plump. Confluence of graph transformation revisited. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer, 2005. doi:10.1007/11601548_16.

[RAB+18] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. Translating essential OCL invariants to nested graph constraints for generating instances of metamodels. *Science of Computer Programming*, 152:38–62, 2018. doi:10.1016/J.SCICO.2017.08.006.

[Ren04] Arend Rensink. Representing first-order logic using graphs. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335. Springer, 2004. doi:10.1007/978-3-540-30203-2_23.

[San20] Christian Sandmann. Graph repair and its application to meta-modeling. In Berthold Hoffmann and Mark Minas, editors, *Proceedings of the Eleventh International Workshop on Graph Computation Models, GCM@STAF 2020, Online-Workshop, 24th June 2020*, volume 330 of *EPTCS*, pages 13–34, 2020. doi:10.4204/EPTCS.330.2.

[San21] Christian Sandmann. *A Theory on Graph Generation and Graph Repair with Application to Meta-Modeling*. PhD thesis, University of Oldenburg, 2021. URL: http://uol.de/f/2/dept/informatik/download/Promotionen/Sandmann_Dissertation.pdf.

[SH19] Christian Sandmann and Annegret Habel. Rule-based graph repair. In Rachid Echahed and Detlef Plump, editors, *Proceedings Tenth International Workshop on Graph Computation Models, GCM@STAF 2019, Eindhoven, The Netherlands, 17th July 2019*, volume 309 of *EPTCS*, pages 87–104, 2019. doi:10.4204/EPTCS.309.5.

[SLO21] Sven Schneider, Leen Lambers, and Fernando Orejas. A logic-based incremental approach to graph repair featuring delta preservation. *International Journal on Software Tools for Technology Transfer*, 23(3):369–410, 2021. doi:10.1007/S10009-020-00584-X.

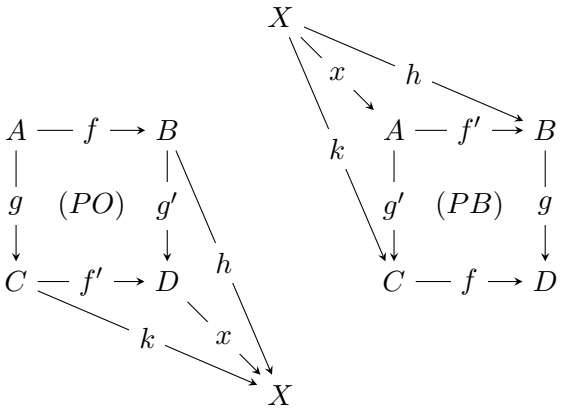

Figure 20: Pushout (on the left) and pullback (on the right)

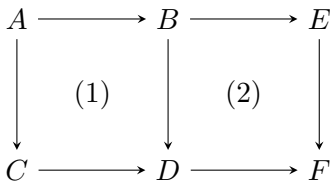

Figure 21: Pushout and Pullback decomposition

Appendix A. Double-pushout approach

In this section, we briefly introduce the double-pushout approach, since we will use some properties of pushouts and pullbacks in our proofs [EEPT06]. We start by introducing pushouts and pullbacks. Intuitively, for graphs, a *pushout* is the smallest gluing of two graphs B and C at some interface A, i.e., we construct a graph D such that A, B and C are subgraphs of D and the embeddings of B and C overlap only at the embedding of A. Intuitively, when constructing a pullback of the graphs B, C, and D so that B and C are subgraphs of D, we are searching for the largest overlap A of B and C, which is a subgraph of D.

Definition A.1 (Pushout and pullback [EEPT06])**.** Given the morphisms $f\colon A \to B$ and $g\colon A \to C$ as shown in the left square in Figure 20. The square (PO) is a *pushout* if it is commutative, i.e., $g' \circ f = f' \circ g$ so that the following universal property is satisfied: For all graphs X and morphisms $h\colon B \to X$ and $k\colon C \to X$ with $k \circ g = h \circ f$, there is a unique morphism $x\colon D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.

Given the morphisms $f\colon C \to D$ and $g\colon B \to D$ as shown in the right square in Figure 20. The square (PB) is a *pullback* if it is commutative, i.e., $g' \circ f = f' \circ g$ so that the following universal property is satisfied: For all graphs X and morphisms $h\colon X \to B$ and $k\colon X \to C$ with $f \circ k = g \circ h$, there is a unique morphism $x\colon X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$.

Note that in the category graphs, a pushout is also a pullback. Before continuing with the definition of transformations using pushouts, we state an important property of pushouts and pullbacks used in our proofs.

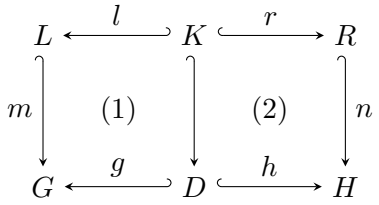

Figure 22: Graph transformation

Lemma A.2 (Properties of pushouts and pullbacks [EEPT06])**.** *For pushouts, the following properties hold:*

(1) Uniqueness: *If the square* (1) *shown in Figure 21 is a pushout, then the graph D is unique up to isomorphism.*
(2) Pushout composition: *If* (1) *and* (2) *shown in Figure 21 are pushouts, then* $(1)+(2)$ *is also a pushout.*
(3) Pushout decomposition: *If* (1) *and* $(1)+(2)$ *shown in Figure 21 are pushouts, then* (2) *is also a pushout.*

For pullbacks, the following properties hold:

(1) Uniqueness: *If the square* (1) *shown in Figure 21 is a pullback, then the graph A is unique up to isomorphism.*
(2) Pullback composition: *If* (1) *and* (2) *shown in Figure 21 are pullbacks, then* $(1)+(2)$ *is also a pullback.*
(3) Pullback decomposition: *If* (2) *and* $(1)+(2)$ *shown in Figure 21 are pullbacks, then* (1) *is also a pullback.*

Definition A.3 (Graph transformation [EEPT06])**.** Given a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ and a morphism $m\colon L \hookrightarrow G$, called the *match*, a *transformation* $t\colon G \Longrightarrow_{\rho,m} H$ is given by the diagram shown in Figure 22 if the squares (1) and (2) are pushouts. The rule ρ is *applicable at* m if (1) is a pushout.

Appendix B. Additional formal results and proofs

This section contains additional formal results and proofs of the lemmas and theorems presented throughout the paper. We start by stating some properties of the Shift and Left operators as presented in Section 5.1.

Lemma B.1 (Correctness of Shift [HP09, Corollary 3])**.** *Given a nested condition c over a graph C and a morphism $p'\colon C \hookrightarrow C'$, for each morphism $p\colon C' \hookrightarrow G$ it holds that*

$$p \models \mathrm{Shift}(p', c) \iff p \circ p' \models c.$$

Lemma B.2 (Correctness of Left [HP09, Theorem 6])**.** *Given a plain rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ and a condition c over R, for each transformation $t\colon G \Longrightarrow_{\rho,m} H$ with comatch n it holds that*

$$n \models c \iff m \models \mathrm{Left}(\rho, c).$$

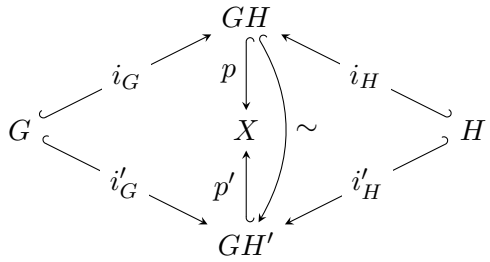

Figure 23: Diagram for the characterisation of equivalence classes of overlaps

The following lemma ensures that the Left operator is stable w.r.t. the number of violations, i.e., if we have an application condition ac formalised for the RHS of a rule ρ, then the number of violations of that application condition w.r.t. the comatch of a transformation is equal to the number of violations of $\text{Left}(ac, \rho)$ w.r.t. the match of this transformation. This ensures that Theorem 5.22 holds if we construct the impairment-indicating application conditions by computing the repair-indicating application conditions of the inverse rule and shift them to the LHS.

Lemma B.3 (Stability of Left w.r.t. the number of violations)**.** *Given a graph P, a transformation $t\colon G \Longrightarrow_{\rho,m} H$ via a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ at a match $m\colon L \hookrightarrow G$ with comatch $n\colon R \hookrightarrow H$, and an application condition $ac = \forall(i_R\colon R \hookrightarrow PR, d')$ constructed via some overlap (i_R, i_P, PR), then*

$$\text{nv}_n(ac) = \text{nv}_m(\text{Left}(\rho, ac)).$$

Proof. (1) First we want to show that $\text{nv}_n(ac) \leq \text{nv}_m(\text{Left}(\rho, ac))$. Given a violation $n'\colon PR \hookrightarrow H \in \text{sv}_n(ac)$, i.e., $n = n' \circ i_R$ and $n' \not\models d'$. By the correctness of the Left operator we have $n' \not\models d' \iff m' \not\models \text{Left}(\rho'^{-1}, d')$ where $\rho' = PR \xhookleftarrow{h} D \xhookrightarrow{g} PL$ is the rule shown in Figure 24. Additionally, by the definition of Left we have $\text{Left}(\rho, ac) = \forall(i_L\colon L \hookrightarrow PL, \text{Left}(\rho'^{-1}, d'))$. By construction, we have $m = i' \circ i_L$ and therefore $m' \in \text{sv}_m(\text{Left}(\rho, ac))$. I.e., each violation $n'\colon PR \hookrightarrow H \in \text{sv}_n(ac)$ corresponds to a violation $m' \in \text{sv}_m(\text{Left}(\rho, ac))$.

For two morphisms $m'_1, m'_2 \in \text{sv}_m(\text{Left}(\rho, ac))$ it holds that $m'_1 \circ i_L = m'_2 \circ i_L$, i.e., m'_1, m'_2 are different in elements preserved by t, i.e., $m'_1 \circ g \neq m'_2 \circ g$. Then $\text{tr}_t \circ m'_1 \circ g \neq \text{tr}_t \circ m'_2 \circ g$ implies that $n'_1 \circ h \neq n'_2 \circ h$ for the corresponding morphisms $n'_1, n'_2 \in \text{sv}_n(ac)$ and we have $n'_1 \neq n'_2$. So $\text{nv}_n(ac) \leq \text{nv}_m(\text{Left}(\rho, ac))$.

(2) Analogously, we can show that each violation $m' \in \text{sv}_m(\text{Left}(\rho, ac))$ corresponds to a violation $n'\colon PR \hookrightarrow H \in \text{sv}_n(ac)$. □

B.1. **Proofs of Section 5.2.**

Proof of Lemma 5.6. (1) Let $(i_G, i_H, GH) \cong (i'_G, i'_H, GH')$ and $p\colon GH \hookrightarrow X$ be a morphism into some graph X, then there is an isomorphism $\sim\colon GH \hookrightarrow GH'$ with $i'_G = \sim \circ i_G$ and $i'_H = \sim \circ i_H$ (see Figure 23). Therefore, for the morphism $p' := p \circ \sim^{-1}\colon GH' \hookrightarrow X$ we

have

$$p' \circ i'_G = p' \circ \sim \circ i_G = p \circ \sim^{-1} \circ \sim \circ i_G = p \circ \circ i_G \text{ and}$$
$$p' \circ i'_H = p' \circ \sim \circ i_H = p \circ \sim^{-1} \circ \sim \circ i_H = p \circ \circ i_H.$$

(2) Given two overlaps (i_G, i_H, GH) and (i'_G, i'_H, GH') and two morphisms $p\colon GH \hookrightarrow X$ and $p'\colon GH' \hookrightarrow X$ into some graph X with

$$p' \circ i'_G = p \circ i_G \text{ and}$$
$$p' \circ i'_H = p \circ i_H.$$

We need to show that $(i_G, i_H, GH) \cong (i'_G, i'_H, GH')$, i.e., that there is an isomorphism $\sim\colon GH \hookrightarrow GH'$ with $i'_G = \sim \circ i_G$ and $i'_H = \sim \circ i_H$. Since $p' \circ i'_G = p \circ i_G$ and $p' \circ i'_H = p \circ i_H$, we have $p(GH) = p'(GH')$. Therefore, the morphism $\sim := p'^{-1} \circ p\colon GH \hookrightarrow GH'$ is total. Analogous, we obtain that $\sim^{-1} = p^{-1} \circ p'\colon GH' \hookrightarrow GH$ is total and

$$\sim \circ \sim^{-1} = p'^{-1} \circ p \circ p^{-1} \circ p' = p'^{-1} \circ p' = \mathrm{id}_{GH'} \text{ and}$$
$$\sim^{-1} \circ \sim = p^{-1} \circ p' \circ p'^{-1} \circ p = p^{-1} \circ p = \mathrm{id}_{GH}.$$

Hence, $\sim$ is an isomorphism and we have

$$\begin{aligned} p' \circ i'_G = p \circ i_G & \quad \Longleftrightarrow \\ i'_G = p'^{-1} \circ p \circ i_G = \sim \circ i_G & \quad \text{and} \\ p' \circ i'_H = p \circ i_H & \quad \Longleftrightarrow \\ i'_H = p'^{-1} \circ p \circ i_H = \sim \circ i_H. & \end{aligned}$$

Therefore, $(i_G, i_H, GH) \cong (i'_G, i'_H, GH')$. □

B.2. **Proofs of Section 5.3.**

Proof of Lemma 5.9. Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$, its inverse transformation $t^{-1}\colon H \Longrightarrow_{\rho^{-1},n} G$, and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$.

(1) Let $p\colon P \hookrightarrow H$ be an impaired morphism w.r.t. t (i.e., $p \not\models d$), then p is either an impair of the premise or an impair of the conclusion:
   (a) If p is an impair of the premise, there is no morphism $p'\colon P \hookrightarrow G$ with $p = \mathrm{tr}_t \circ p'$, i.e., p was introduced by t and is therefore destroyed by t^{-1}. So $\mathrm{tr}_{t^{-1}} \circ p$ is not total and p is a repaired morphism w.r.t. t^{-1}.
   (b) If p is an impair of the conclusion, there is a morphism $p'\colon P \hookrightarrow G$ with $p = \mathrm{tr}_t \circ p'$ and $p' \models d$, i.e., $\mathrm{tr}_{t^{-1}} \circ p$ is total. So p is a repaired morphism w.r.t. t^{-1} since $\mathrm{tr}_{t^{-1}} \circ p = p' \models d$.

(2) Let $p\colon P \hookrightarrow G$ be a repaired morphism w.r.t. t (i.e., $p \not\models d$), then p is either a repair of the premise or a repair of the conclusion.
   (a) If p is a repair of the premise, the proof is analogous to Item 1a.
   (b) If p is a repair of the conclusion, the proof is analogous to Item 1b. □

Proof of Lemma 5.14. We assume that $G \models c$ and that there is a morphism $p\colon P' \hookrightarrow G$ with $p \models c'$. I.e., there is a morphism $q\colon Q' \hookrightarrow G$ with $p = q \circ e'$ and $q \models d$. Since $Q' \not\models c$, there is a morphism $p'\colon P \hookrightarrow Q'$ and hence, there is a morphism $q \circ p'\colon P \hookrightarrow G$. It follows that $G \not\models c$, this is a contradiction. □

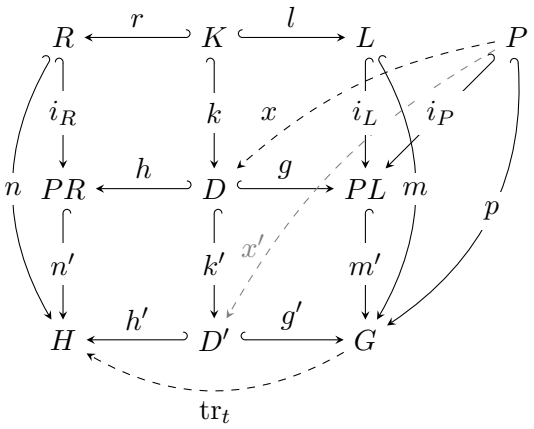

Figure 24: Construction of impairment-indicating and repair-indicating application conditions and shift of overlaps

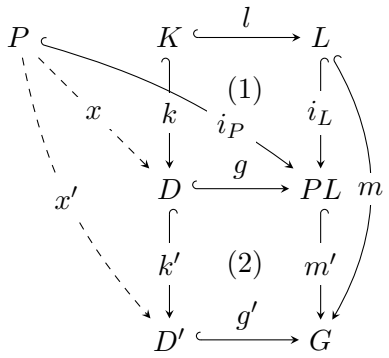

Figure 25: Construction of impairment-indicating and repair-indicating application conditions and shift of overlaps

Proof of Lemma 5.16. This Lemma follows immediately with the fact that $(a \vee b) \implies (a \vee c) \equiv b \implies (a \vee c)$. □

B.3. **Proof of Theorem 5.22.** In this section, we provide the proof of Theorem 5.22 and other theoretical results that are needed for the proof. We begin by showing that the *induced pre-* and *post-conditions* are correct in the sense that they predict whether an occurrence of the premise satisfies the conclusion before the transformation (correctness of the pre-condition) and after the transformation (correctness of the post-condition). For this, we first need the following result.

Lemma B.4. *Given a transformation $t\colon G \Longrightarrow_{m,\rho} H$ via some rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$ at match m, and a morphism $p\colon P \hookrightarrow G$, as shown in Figure 24, then $\mathrm{tr}_t \circ p$ is total if and only if there is a morphism $x'\colon P \hookrightarrow D'$ with $p = g' \circ x'$.*

Proof. (1) If $\mathrm{tr}_t \circ p$ is total, $g'^{-1} \circ p$ is total, i.e., for each $e \in P$ there is an element $e' \in D'$ with $g(e') = p(e)$. Therefore $x' := g'^{-1} \circ p$ satisfies $g \circ x' = p$.

(2) If there is a morphism $x' \colon P \hookrightarrow D'$ with $p = g' \circ p$, the morphism $g'^{-1} \circ p$ is total, since for each $e \in P$ there is an element $e' \in D'$ with $p(e) = g'(e')$. This implies that $\mathrm{tr}_t \circ p$ is also total. □

Lemma B.5 (Correctness of Pre and Post)**.** *Given a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, a transformation $t \colon G \Longrightarrow_{\rho,m} H$ via ρ at match m, a condition d over some graph P, and an overlap $(i_L, i_P, PL) \in \mathrm{OL}(\rho, L)$, for each morphism $m' \colon PL \hookrightarrow G$ with $m = m' \circ i_L$, we have*

(1) *$m' \models \mathrm{Pre}_\rho((i_L, i_P, PL), d)$ if and only if $m' \circ i_P \models d$, and*
(2) *$m' \models \mathrm{Post}_\rho((i_L, i_P, PL), d)$ if and only if ($\mathrm{tr}_t \circ m' \circ i_P$ is total and $\mathrm{tr}_t \circ m' \circ i_P \models d$) or $\mathrm{tr}_t \circ m' \circ i_P$ is not total.*

Proof. (1) As $\mathrm{Pre}_\rho((i_L, i_P, PL), d) = \mathrm{Shift}(i_P, d)$, we have $m' \models \mathrm{Pre}_\rho((i_L, i_P, PL), d)$ if and only if $m' \circ i_P \models d$ (Lemma B.1).

(2) First, we assume that $m' \models \mathrm{Post}_\rho((i_L, i_P, PL), d)$, then the induced transformations of (i_L, i_P, PL) are either parallel dependent or parallel independent:
   (a) If the induced transformations of (i_L, i_P, PL) are parallel dependent, the overlap (i_L, i_P, PL) is contained in $\mathrm{OL}_{\mathrm{dep}}(\rho, P)$, i.e., there is no morphism $x \colon P \hookrightarrow D$ with $g \circ x = i_P$ (Figure 24). To show that $\mathrm{tr}_t \circ m' \circ i_P$ is not total it suffices to show that there is no morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = g' \circ x'$ (Figure 25) as this implies that $g'^{-1} \circ m' \circ i_P$ and therefore especially $\mathrm{tr}_t \circ m' \circ i_P$ is not total (Lemma B.4). Since the squares (1) + (2) and (2) in Figure 25 are pushouts ((1) + (2) is part of the transformation t and (1) is part of the induced transformation of (i_L, i_P, PL)), the square (2) is also a pushout and a pullback as we are in the category of graphs and g and k' are injective. If we assume that there is a morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = g' \circ x'$, there is also a morphism $x \colon P \hookrightarrow D$ such that $x' = k' \circ x$ and especially $i_P = g \circ x$ (universal property of pullbacks). This is a contradiction as the induced transformations of (i_L, i_P, PL) are parallel dependent.
   (b) If the induced transformations of (i_L, i_P, PL) are parallel independent, i.e., there is a morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ and $\mathrm{Post}_\rho((i_L, i_P, PL), d) = \mathrm{Left}(\rho', \mathrm{Shift}(h \circ x, d))$ where $\rho' = PR \xhookleftarrow{h} D \xhookrightarrow{g} PL$ is the induced rule and g, h and x are the morphisms as shown in Figure 24, the morphism m' satisfies $\mathrm{Left}(\rho', \mathrm{Shift}(h \circ x, d))$ if and only if $n' \colon PR \hookrightarrow H$ satisfies $\mathrm{Shift}(h \circ x, d)$ (Lemma B.2). Lemma B.1 implies that $n' \models \mathrm{Shift}(h \circ x, d)$ if and only if $n' \circ h \circ x \colon P \hookrightarrow H$ satisfies d. Therefore, it remains to show that $\mathrm{tr}_t \circ m' \circ i_P$ is total and that $\mathrm{tr}_t \circ m' \circ i_P = n' \circ h \circ x$. As there is the morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ and the square on the bottom right of Figure 24 is a pullback, there is also a morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = g' \circ x'$ (universal property of pullbacks) and Lemma B.4 implies that $\mathrm{tr}_t \circ m' \circ i_P$ is total. We also have

$$\begin{aligned} \mathrm{tr}_t \circ m' \circ i_P &= \mathrm{tr}_t \circ g' \circ k' \circ x \\ &= h' \circ g'^{-1} \circ g' \circ k' \circ x \\ &= h' \circ k' \circ x \\ &= n' \circ h \circ x. \end{aligned}$$

(3) Now, we assume that either $\mathrm{tr}_t \circ m' \circ i_P$ is total and $\mathrm{tr}_t \circ m' \circ i_P \models d$ or $\mathrm{tr}_t \circ m' \circ i_P$ is not total implies that $m' \models \mathrm{Post}_\rho((i_L, i_P, PL), d)$.
   (a) If $\mathrm{tr}_t \circ m' \circ i_P$ is not total, there is no morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = x' \circ g'$ (Lemma B.4) and there is no morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ (universal

property of pullbacks), i.e., the induced transformations of (i_L, i_P, PL) are parallel dependent and $\text{Post}_\rho((i_L, i_P, PL), d) = \mathsf{true}$. So $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$.

(b) If $\text{tr}_t \circ m' \circ i_P$ is total and $\text{tr}_t \circ m' \circ i_P \models d$, there is a morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = g' \circ x'$ and a morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ (Lemma B.4 and the universal property of pullbacks, i.e., the induced transformations of (i_L, i_P, PL) are parallel independent and $\text{Post}_\rho((i_L, i_P, PL), d) = \text{Left}(\rho', \text{Shift}(h \circ x, d))$. The morphism m' satisfies $\text{Left}(\rho', \text{Shift}(h \circ x, d))$ if and only if $n' \models \text{Shift}(h \circ x, d)$ (Lemma B.2). Additionally, $n' \models \text{Shift}(h \circ x, d)$ if and only if $n' \circ h \circ x \models d$ (Lemma B.1). In the previous part of the proof, we have shown that $n' \circ h \circ x = \text{tr}_t \circ m' \circ i_P$. As $\text{tr}_t \circ m' \circ i_P \models d$ we have $n' \circ h \circ c \models d$, $n' \models \text{Shift}(h \circ x, d)$ and $m' \models \text{Left}(\rho', \text{Shift}(h \circ x, d))$ (Lemma B.1 Lemma B.2). □

The following lemma ensures that every violation of a repair-indicating application condition contains a repaired morphism w.r.t. a transformation, and that every repaired morphism w.r.t. a transformation contained in a violation of exactly one repair-indicating application condition.

Lemma B.6. *Given a constraint $c = \forall(e \colon \emptyset \hookrightarrow P, d)$, a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, a transformation $t \colon G \Longrightarrow_{\rho,m} H$ via ρ at match m, then*

(1) *for each application condition $ac \in \text{Rep}(\rho, c)$, constructed via an overlap $[(i_L, i_P, PL)] \in \text{OL}_\sim(\rho, P)$ and for each morphism $m' \colon PL \hookrightarrow G$ contained in $\text{sv}_m(ac)$, $m' \circ i_P$ is a repaired morphism w.r.t. t, and*

(2) *for each repaired morphism $p \colon P \hookrightarrow G$ w.r.t. t, there is exactly one application condition $ac \in \text{Rep}(\rho, c)$ constructed via an overlap $[(i_L, i_P, PL)] \in \text{OL}_\sim(\rho, P)$ so that there is a violation $m' \in \text{sv}_m(ac)$ with $p = m' \circ i_P$.*

Proof. (1) Given a repair indicating application condition $ac \in \text{Rep}(\rho, c)$ constructed via an overlap $[(i_L, i_P, PL)] \in \text{OL}_\sim(\rho, P)$ and a violation $p \colon PL \hookrightarrow G \in \text{sv}_m(ac)$. We need to show that $m' \circ i_P$ is a repaired morphism w.r.t. t. As $m' \not\models \text{Post}_\rho((i_L, i_P, PL), d) \implies \text{Pre}_\rho((i_L, i_P, PL), d)$, we have $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$ and $m' \not\models \text{Pre}_\rho((i_L, i_P, PL), d)$. Let us first consider the second part of the implication: By correctness of the Pre-operator (Lemma B.5), we have $m' \circ i_P \not\models d$. When considering $\text{Post}_\rho((i_L, i_P, PL), d)$ we need to consider whether the induced transformations of (i_L, i_P, PL) are parallel dependent or not:

(a) If the induced transformations of (i_L, i_P, PL) are parallel dependent, there is no morphism $x \colon P \hookrightarrow D$ with $i_P = x \circ g$ (Figure 24), i.e., $\text{Post}_\rho((i_L, i_P, PL), d) = \mathsf{true}$, so $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$. It remains to show that $\text{tr}_t \circ p$ is not total: The universal property of pullbacks implies that there is no morphism $x' \colon P \hookrightarrow D'$ with $m' \circ i_P = g' \circ x'$ (compare Figure 25). So $\text{tr}_t \circ p$ is not total (Lemma B.4) and $p \circ i_P$ is a repair of the premise w.r.t. t.

(b) If the induced transformations of (i_L, i_P, PL) are parallel independent, we have $\text{Post}_\rho((i_L, i_P, PL), d) = \text{Left}(\text{Shift}(h \circ x, d))$, where h and x are the morphisms shown in Figure 24. The existence of x implies that $\text{tr}_t \circ p \circ i_P$ is total (universal property of pullbacks and Lemma B.4). As $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$, Lemma B.5 implies that $\text{tr}_t \circ m' \circ i_P \models d$. In total, m' is a repair of the conclusion w.r.t. t.

(2) Given a repaired morphism $p \colon P \hookrightarrow G$ w.r.t. t, then p is either a repair of the premise or a repair of the conclusion w.r.t. t and there is an overlap $(i_L, i_P, PL) \in \text{OL}(\rho, P)$ and a morphism $m' \colon PL \hookrightarrow P$ with $m = m' \circ i_L$ and $p = m' \circ i_P$. We need to show that $m' \in$

$sv_m(ac)$, i.e., that $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$ and $m' \not\models \text{Pre}_\rho((i_L, i_P, PL), d)$. As p is a repair, we have $p = m' \circ i_P \not\models d$ and Lemma B.5 implies that $m' \models \text{Pre}_\rho((i_L, i_P, PL), d)$. When considering $\text{Post}_\rho((i_L, i_P, PL), d)$, we need to consider whether $\text{tr}_t \circ p$ is total or not:

(a) If $\text{tr}_t \circ p$ is not total, Lemma B.4 implies that there is no morphism $x' \colon P \hookrightarrow D'$ with $p = g' \circ x'$ (compare Figure 24). Therefore, there is also no morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ (universal property of pullbacks), i.e., $\text{Post}_\rho((i_L, i_P, PL), d) = \mathsf{true}$ and $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$. So $m' \in \text{sv}_m(ac)$.

(b) If $\text{tr}_t \circ p$ is total, there is a morphism $x' \colon P \hookrightarrow D'$ with $p = x' \circ g'$ (Lemma B.4) and there is a morphism $x \colon P \hookrightarrow D$ with $i_P = g \circ x$ (universal property of pullbacks). So $\text{Post}_\rho((i_L, i_P, PL), d) = \text{Left}(\rho', \text{Shift}(h \circ x, d))$. As p is a repaired morphism and $\text{tr}_t \circ p$ is total, $\text{tr}_t \circ p = \text{tr}_t \circ m' \circ i_P \models d$. Lemma B.5 implies that $m' \models \text{Post}_\rho((i_L, i_P, PL), d)$ and $m' \in \text{sv}_m(ac)$.

Lastly we have to show that for each repaired morphism $p \colon P \hookrightarrow G$ there is exactly one application condition $ac \in \text{Rep}(\rho, c)$ constructed over an overlap $[(i_L, i_P, PL)] \in \text{OL}_\sim(\rho, P)$ so that there is a violation $m' \in \text{sv}_m(ac)$ with $p = m' \circ i_P$. We assume that there are two such application condition $ac, ac' \in \text{Rep}(\rho, c)$ constructed over overlaps $[(i_L, i_P, PL)]$ and $[(i'_L, i'_P, PL')] \in \text{OL}_\sim(\rho, c)$. There are morphisms $m' \colon PL \hookrightarrow G$ and $m'' \colon PL' \hookrightarrow G$ with

$$p = m' \circ i_P = m'' \circ i'_P \text{ and}$$
$$m = m' \circ i_L = m'' \circ i'_L.$$

Lemma 5.6 implies that $[(i_L, i_P, PL)] \cong [(i'_L, i'_P, PL')]$ and we have $ac = ac'$. □

With the following results, we can extract sets of repaired and impaired morphisms by restricting the violations of each repair- (impairment-)indicating application condition to the embedding of the premise into the overlap graph, this application condition was computed with.

Corollary B.7. *Given a constraint $c = \forall(e \colon \emptyset \hookrightarrow P, d)$, a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, a transformation $t \colon G \Longrightarrow_{\rho,m} H$, then*

$$\dot{\bigcup}_{ac \in \text{Rep}(\rho,c)} \{m' \circ i_P \mid m' \in \text{sv}_m(ac)\} = \{p \colon P \hookrightarrow G \mid p \text{ is a repaired morphism w.r.t. } t\},$$

where $ac = \forall(i_L \colon L \hookrightarrow PL, d')$ is the application condition constructed with the overlap $(i_L, i_P, PL) \in \text{OL}_\sim(\rho, P)$.

Corollary B.8. *Given a constraint $c = \forall(e \colon \emptyset \hookrightarrow P, d)$, a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, a transformation $t \colon G \Longrightarrow_{\rho,m} H$, then*

$$\dot{\bigcup}_{ac \in \text{Imp}(\rho,c)} \{m' \circ i_P \mid m' \in \text{sv}_m(ac)\} = \{p \colon P \hookrightarrow G \mid p \text{ is an impaired morphism w.r.t. } t\},$$

where $ac = \forall(i_L \colon L \hookrightarrow PL, d')$ is the application condition constructed with the overlap $(i_L, i_P, PL) \in \text{OL}_\sim(\rho, P)$.

It remains to show that the number of violations of repair-indicating application conditions is equal to the number of repaired morphisms w.r.t. a transformation and, therefore, the number of violations of impairment-indicating application conditions is equal to the number of impaired morphisms w.r.t. a transformation.

Lemma B.9. *Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, a rule $\rho = L \xhookleftarrow{l} K \xhookrightarrow{r} R$, a transformation $t\colon G \Longrightarrow_{\rho,m} H$ via ρ at match m, and an application condition $ac = \forall(i_L\colon L \hookrightarrow PL, d') \in \mathrm{Rep}(\rho, c)$ constructed using the overlap (i_L, i_P, PL), then*

$$\mathrm{nv}_m(ac) = |\{p \circ i_P\colon P \hookrightarrow G \mid p \in \mathrm{sv}_m(ac)\}|$$

Proof. '$\leq$': Given two morphisms $p, p'\colon PL \hookrightarrow G \in \mathrm{sv}_m(ac)$ with $p \neq p'$. Since i_L and i_P are jointly surjective and $p \circ i_L = m = p' \circ i_L$, we have $p \circ i_P \neq p' \circ i_P$ and

$$\mathrm{nv}_m(ac) \leq |\{p \circ i_P\colon P \hookrightarrow G \mid p \in \mathrm{sv}_m(ac)\}|.$$

'$\geq$': Given two morphisms $p, p' \in |\{pl \circ i_P\colon P \hookrightarrow G \mid pl \in \mathrm{sv}_m(ac)\}|$ with $p \neq p'$, then there are two morphisms $pl, pl'\colon PL \hookrightarrow G \in \mathrm{sv}_m(c)$ with $m = pl \circ i_L = pl' \circ i_L$, $p = pl \circ i_P$ and $p' = pl' \circ i_P$. Since $p = pl \circ i_P \neq pl' \circ i_P = p'$ it follows that $pl \neq pl'$ and we have

$$\mathrm{nv}_m(ac) \geq |\{pl \circ i_P\colon P \hookrightarrow G \mid pl \in \mathrm{sv}_m(ac)\}|.$$ □

We can evaluate the change in consistency by only considering impaired and repaired morphisms.

Lemma B.10. *Given a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$ and a transformation $t\colon G \Longrightarrow_{\rho,m} H$, then*

$$\begin{aligned}\mathrm{nv}_H(c) - \mathrm{nv}_G(c) = &|\{p\colon P \hookrightarrow H \mid p \text{ is impaired w.r.t. } t\}| - \\ &|\{p\colon P \hookrightarrow G \mid p \text{ is repaired w.r.t. } t\}|.\end{aligned}$$

Proof. We can rewrite the set of violations of H as

$$\begin{aligned}\mathrm{sv}_H(c) = &\{p\colon P \hookrightarrow H \mid p \text{ is an impaired morphism w.r.t. } t\} \\ &\cup \{\mathrm{tr}_t \circ p \mid p \in \mathrm{sv}_G(c) \text{ and } p \text{ is not a repaired morphism w.r.t. } t\}.\end{aligned}$$

So we can evaluate $\mathrm{nv}_H(c)$ as

$$\begin{aligned}\mathrm{nv}_H(c) = \mathrm{nv}_G(c) &+ |\{p \mid p \text{ is an impaired morphism w.r.t. } t\}| \\ &- |\{p \mid p \text{ is a repaired morphism w.r.t. } t\}|\end{aligned}$$

and the statement follows immediately. □

With these prerequisites in place, we can now prove Theorem 5.22.

Proof of Theorem 5.22. Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$ and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$, with Lemma B.10 the difference $\mathrm{nv}_H(c) - \mathrm{nv}_G(c)$ can be evaluated by only considering impaired and repaired morphisms. Corollary B.7 and Corollary B.8 show that we can find repaired and impaired morphisms by evaluating the application conditions contained in the sets $\mathrm{Rep}(\rho, c)$ and $\mathrm{Imp}(\rho, c)$, respectively. By using Corollary B.7 and Lemma B.9 we get

$$\begin{aligned}|\{p\colon P \hookrightarrow G \mid p \text{ is a repaired morphism w.r.t. } t\}| &= |\dot{\bigcup_{ac \in \mathrm{Rep}(\rho,c)}} \{p \circ i_P \mid p \in \mathrm{sv}_m(ac)\}| \\ &= \sum_{ac \in \mathrm{Rep}(\rho,c)} |\{p \circ i_P \mid p \in \mathrm{sv}_m(ac)\}| \\ &= \sum_{ac \in \mathrm{Rep}(\rho,c)} \mathrm{nv}_m(ac)\end{aligned}$$

When considering $|\{p\colon P \hookrightarrow G \mid p \text{ is an impaired morphism w.r.t. } t\}|$ we utilize the fact each impaired morphism is a repaired morphism of the inverse transformation $t^{-1}\colon H \Longrightarrow_{\rho^{-1},n} G$ (Lemma 5.9):

$$\begin{aligned}&|\{p\colon P \hookrightarrow H \mid p \text{ is an impaired morphism w.r.t. } t\}|\\ =&|\{p\colon P \hookrightarrow H \mid p \text{ is a repaired morphism w.r.t. } t^{-1}\}|\end{aligned}$$

Analogous to the first part of the proof, we can show that

$$\begin{aligned}|\{p\colon P \hookrightarrow H \mid p \text{ is an impaired morphism w.r.t. } t\}| &= \sum_{ac\in \mathrm{Rep}(\rho^{-1},c)} \mathrm{nv}_n(ac)\\ &= \sum_{ac\in \mathrm{Imp}(\rho,c)} \mathrm{nv}_m(ac)\end{aligned}$$

In total, we obtain

$$\mathrm{nv}_H(c) - \mathrm{nv}_G(c) = \sum_{ac\in \mathrm{Imp}(\rho,c)} \mathrm{nv}_m(ac) - \sum_{ac\in \mathrm{Rep}(\rho,c)} \mathrm{nv}_m(ac) \qquad \square$$

B.4. **Proofs of Section 5.4.**

Proof of Theorem 5.30. Given a transformation $t\colon G \Longrightarrow_{\rho,m} H$ and a constraint $c = \forall(e\colon \emptyset \hookrightarrow P, d)$,

(1) we start by showing that t is direct consistency-sustaining w.r.t. c if and only if $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac$.
   '$\Longrightarrow$': Let t be direct consistency-sustaining w.r.t. c, i.e., there is no impaired morphism w.r.t. t. As Corollary B.8 implies that each violation of an application condition $ac \in \mathrm{Imp}(\rho, c)$ contains an impaired morphism, there is no violation for any ac and $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac$.
   '$\Longleftarrow$': Let $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac$, then Corollary B.8 implies that there are no impaired morphisms w.r.t. t, i.e., t is direct consistency-sustaining w.r.t. c
(2) Now, we show that t is direct consistency-improving w.r.t. c if and only if $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac \wedge \neg(\bigwedge_{ac\in \mathrm{Rep}(\rho,c)} ac)$.
   '$\Longrightarrow$': Let t be direct consistency-improving w.r.t. c. Then t is also directly consistency-sustaining w.r.t. c and the first part of Theorem 5.30 implies that $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac$. Also, there is at least one repaired morphism w.r.t. t and Corollary B.7 implies that this repaired morphism is a violation of an application condition $ac \in \mathrm{Rep}(\rho, c)$. So $m \models \neg(\bigwedge_{ac\in \mathrm{Rep}(\rho,c)} ac)$.
   '$\Longleftarrow$': Let $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac \wedge \neg(\bigwedge_{ac\in \mathrm{Rep}(\rho,c)} ac)$, the first part of Theorem 5.30 implies that $m \models \bigwedge_{ac\in \mathrm{Imp}(\rho,c)} ac$. At least one of the application conditions contained $\mathrm{Rep}(\rho, c)$ is violated. Since each violation of an application condition $ac \in \mathrm{Rep}(\rho, c)$ contains a repaired morphism w.r.t. t (Corollary B.7) t is a direct consistency-improving transformation w.r.t. c. $\square$